Table of Contents

	Page
I C-REF: PREFACE	**1**
II C-REF: THE C LANGUAGE	**5**
1 C-Ref: Introduction	**7**
1.1 C-Ref: Who Defines C?	8
1.2 C-Ref: An Overview of C Programming	9
1.3 C-Ref: Syntax Notation	10
2 C-Ref: Lexical Elements	**11**
2.1 C-Ref: Character Set	11
2.1.1 C-Ref: Execution Character Set	12
2.1.2 C-Ref: Whitespace and Line Termination	12
2.1.3 C-Ref: Character Encodings	13
2.2 C-Ref: Comments	14
2.3 C-Ref: Tokens	15
2.4 C-Ref: Operators and Separators	15
2.5 C-Ref: Identifiers	16
2.5.1 C-Ref: Conventions for Identifiers	17
2.6 C-Ref: Reserved Words	18
2.7 C-Ref: Constants	18
2.7.1 C-Ref: Integer Constants	19
2.7.2 C-Ref: Floating-point Constants	21
2.7.3 C-Ref: Character Constants	23
2.7.4 C-Ref: String Constants	24
2.7.5 C-Ref: Escape Characters	25
2.7.6 C-Ref: Character Escape Codes	25
2.7.7 C-Ref: Numeric Escape Codes	26
3 C-Ref: The C Preprocessor	**29**
3.1 C-Ref: Preprocessor Commands	29
3.2 C-Ref: Preprocessor Lexical Conventions	30
3.3 C-Ref: Definition and Replacement	31
3.3.1 C-Ref: Simple Macro Definitions	31
3.3.2 C-Ref: Defining Macros with Parameters	33
3.3.3 C-Ref: Rescanning of Macro Expressions	35
3.3.4 C-Ref: Predefined Macros	36
3.3.5 C-Ref: Undefining and Redefining Macros	37
3.3.6 C-Ref: Precedence Errors in Macro Expansions	38

3.3.7 C-Ref: Side Effects in Macro Arguments 39
3.3.8 C-Ref: Converting Tokens to Strings 39
3.3.9 C-Ref: Token Merging in Macro Expansions 40
3.3.10 C-Ref: Other Problems 41
3.4 C-Ref: File Inclusion 41
3.5 C-Ref: Conditional Compilation 42
3.5.1 C-Ref: The #if, #else, and #endif Commands 42
3.5.2 C-Ref: The #elif Commands 43
3.5.3 C-Ref: The #ifdef and #ifndef Commands 45
3.5.4 C-Ref: Constant Expressions in Conditional Commands 47
3.5.5 C-Ref: The `defined` Operator 47
3.6 C-Ref: Explicit Line Numbering 48

4 C-Ref: Declarations 49

4.1 C-Ref: Organization of Declarations 50
4.2 C-Ref: Terminology 51
4.2.1 C-Ref: Scope 51
4.2.2 C-Ref: Visibility 52
4.2.3 C-Ref: Forward References 53
4.2.4 C-Ref: Overloading of Names 53
4.2.5 C-Ref: Duplicate Declarations 55
4.2.6 C-Ref: Duplicate Visibility 56
4.2.7 C-Ref: Extent 56
4.2.8 C-Ref: Initial Values 57
4.2.9 C-Ref: External Names 58
4.2.10 C-Ref: Compile-time Objects 58
4.3 C-Ref: Storage Class Specifiers 59
4.3.1 C-Ref: Default Storage Class Specifiers 60
4.3.2 C-Ref: Examples of Storage Class Specifiers 60
4.4 C-Ref: Type Specifiers 62
4.4.1 C-Ref: Default Type Specifiers 62
4.4.2 C-Ref: Missing Declarators 63
4.5 C-Ref: Declarators 64
4.5.1 C-Ref: Simple Declarators 64
4.5.2 C-Ref: Pointer Declarators 65
4.5.3 C-Ref: Array Declarators 65
4.5.4 C-Ref: Function Declarators 67
4.5.5 C-Ref: Composition of Declarators 68
4.6 C-Ref: Initializers 69
4.6.1 C-Ref: Integers 70
4.6.2 C-Ref: Floating-point 70
4.6.3 C-Ref: Pointers 71
4.6.4 C-Ref: Arrays 72
4.6.5 C-Ref: Enumerations 73
4.6.6 C-Ref: Structures 74
4.6.7 C-Ref: Unions 75

4.6.8 C-Ref: Eliding Braces 75
4.7 C-Ref: Implicit Declarations 76
4.8 C-Ref: External Names 76
4.8.1 C-Ref: The Initializer Model 77
4.8.2 C-Ref: The Omitted Storage Class Model 77
4.8.3 C-Ref: The Common Model 77
4.8.4 C-Ref: Mixed Common Model 77
4.8.5 C-Ref: Advice 78
4.8.6 C-Ref: Unreferenced External Declarations 78

5 C-Ref: Types 79

5.1 C-Ref: Type Categories 79
5.2 C-Ref: Integer Types 80
5.2.1 C-Ref: Signed Integer Types 80
5.2.2 C-Ref: Unsigned Integer Types 82
5.2.3 C-Ref: Character Type 83
5.3 C-Ref: Floating-Point Types 86
5.4 C-Ref: Pointer Types 87
5.4.1 C-Ref: Pointer Arithmetic 88
5.4.2 C-Ref: Some Problems with Pointers 89
5.5 C-Ref: Array Types 90
5.5.1 C-Ref: Arrays and Pointers 90
5.5.2 C-Ref: Multidimensional Arrays 91
5.5.3 C-Ref: Array Bounds 91
5.5.4 C-Ref: Operations 92
5.6 C-Ref: Enumeration Types 92
5.7 C-Ref: Structure Types 95
5.7.1 C-Ref: Structure Type References 97
5.7.2 C-Ref: Operations on Structures 98
5.7.3 C-Ref: Components 98
5.7.4 C-Ref: Structure Component Layout 99
5.7.5 C-Ref: Bit Fields 100
5.7.6 C-Ref: Portability Problems 102
5.7.7 C-Ref: Sizes of Structures 102
5.8 C-Ref: Union Types 103
5.8.1 C-Ref: Union Component Layout 104
5.8.2 C-Ref: Sizes of Unions 104
5.8.3 C-Ref: Using Union Types 105
5.9 C-Ref: Function Types 107
5.10 C-Ref: Void 109
5.11 C-Ref: Typedef Names 110
5.11.1 C-Ref: Typedef Names for Function Types 111
5.11.2 C-Ref: Redefining Typedef Names 111
5.11.3 C-Ref: A Note on Implementation of Typedef Names 112
5.12 C-Ref: Type Equivalence 112
5.12.1 C-Ref: More About Array Types 113

5.12.2 C-Ref: Enumeration, Structure, and Union Types 113
5.12.3 C-Ref: More About Typedef Names 113
5.13 C-Ref: Type Names and Abstract Declarators 114

6 C-Ref: Conversions and Representations 117

6.1 C-Ref: Representational Issues 117
6.1.1 C-Ref: Storage Units and Data Sizes 117
6.1.2 C-Ref: Addressing Structure and Byte Ordering 118
6.1.3 C-Ref: Alignment Restrictions 119
6.1.4 C-Ref: Pointer Sizes 121
6.1.5 C-Ref: Difficult Addressing Models 121
6.2 C-Ref: Conversions 122
6.2.1 C-Ref: Representation Changes 122
6.2.2 C-Ref: Trivial Conversions 123
6.2.3 C-Ref: Conversions to Integer Types 123
6.2.4 C-Ref: Conversions to Floating-point Types 125
6.2.5 C-Ref: Conversions to Structure and Union Types 125
6.2.6 C-Ref: Conversions to Enumeration Types 126
6.2.7 C-Ref: Conversions to Pointer Types 126
6.2.8 C-Ref: Conversions to Array and Function Types 126
6.2.9 C-Ref: Conversions to the Void Type 127
6.3 C-Ref: The Usual Conversions 127
6.3.1 C-Ref: The Casting Conversions 127
6.3.2 C-Ref: The Assignment Conversions 127
6.3.3 C-Ref: The Usual Unary Conversions 128
6.3.4 C-Ref: The Usual Binary Conversions 129
6.3.5 C-Ref: The Function Argument Conversions 130
6.3.6 C-Ref: Other Function Conversions 130

7 C-Ref: Expressions 131

7.1 C-Ref: General Comments 131
7.1.1 C-Ref: Objects and LValues 131
7.2 C-Ref: Expressions and Precedence 132
7.2.1 C-Ref: Precedence and Associativity of Operators 132
7.2.2 C-Ref: Overflow and Other Arithmetic Exceptions 134
7.3 C-Ref: Primary Expressions 135
7.3.1 C-Ref: Names 135
7.3.2 C-Ref: Literals 137
7.3.3 C-Ref: Paranthesized Expressions 137
7.4 C-Ref: Postfix Expressions 138
7.4.1 C-Ref: Subscripting Expressions 138
7.4.2 C-Ref: Component Selection 140
7.4.3 C-Ref: Function Calls 141
7.4.4 C-Ref: Postincrement Operator 143
7.4.5 C-Ref: Postdecrement Operator 144
7.5 C-Ref: Unary Expressions 144

7.5.1 C-Ref: Casts 145
7.5.2 C-Ref: Size of Operator 145
7.5.3 C-Ref: Unary Minus 147
7.5.4 C-Ref: Logical Negation 147
7.5.5 C-Ref: Bitwise Negation 147
7.5.6 C-Ref: Address Operator 148
7.5.7 C-Ref: Indirection 149
7.5.8 C-Ref: Preincrement Operator 149
7.5.9 C-Ref: Predecrement Operator 150
7.6 C-Ref: Binary Operator Expressions 151
7.6.1 C-Ref: Multiplicative Operators 151
7.6.2 C-Ref: Additive Operators 153
7.6.3 C-Ref: Shift Operators 155
7.6.4 C-Ref: Relational Operators 157
7.6.5 C-Ref: Equality Operators 158
7.6.6 C-Ref: Bitwise AND Operator 159
7.6.7 C-Ref: Bitwise XOR Operator 160
7.6.8 C-Ref: Bitwise OR Operator 161
7.7 C-Ref: Logical Operator Expressions 167
7.7.1 C-Ref: Logical AND Operator 168
7.7.2 C-Ref: Logical OR Operator 168
7.8 C-Ref: Conditional Expressions 169
7.9 C-Ref: Assignment Expressions 170
7.9.1 C-Ref: Simple Assignment 171
7.9.2 C-Ref: Compound Assignment 172
7.10 C-Ref: Sequential Expressions 173
7.11 C-Ref: Constant Expressions 174
7.12 C-Ref: Order of Evaluation 176
7.13 C-Ref: Discarded Values 178
7.14 C-Ref: Compiler Optimization of Memory Accesses 179

8 C-Ref: Statements 183

8.1 C-Ref: General Syntactic Rules for Statements 183
8.1.1 C-Ref: Semicolons 183
8.1.2 C-Ref: Control Expressions 184
8.2 C-Ref: Expression Statements 184
8.3 C-Ref: Labeled Statements 185
8.4 C-Ref: Compound Statement 186
8.4.1 C-Ref: Declarations Within Compound Statements 186
8.4.2 C-Ref: Use of Compound Statements 187
8.5 C-Ref: Conditional Statement 188
8.5.1 C-Ref: Multiway Conditional Statements 188
8.5.2 C-Ref: The Dangling Else Problem 189
8.6 C-Ref: Iterative Statements 190
8.6.1 C-Ref: While Statement 191
8.6.2 C-Ref: Do Statement 192

8.6.3 C-Ref: For Statement 193
8.6.4 C-Ref: Using the For Statement 194
8.6.5 C-Ref: Multiple Control Variables 197
8.7 C-Ref: Switch Statement; Case and Default Labels 198
8.7.1 C-Ref: Use of Switch Statements 200
8.8 C-Ref: Break and Continue Statements 202
8.8.1 C-Ref: Using `break` and `continue` 203
8.9 C-Ref: Return Statement 205
8.10 C-Ref: Goto Statement and Named Labels 206
8.10.1 C-Ref: Using the `goto` statement 206
8.11 C-Ref: Null Statement 207

9 C-Ref: Functions 209

9.1 C-Ref: Function Definitions 209
9.2 C-Ref: Types of Functions 210
9.3 C-Ref: Formal Parameter Declarations 211
9.4 C-Ref: Adjustments to Parameter Types 212
9.5 C-Ref: Parameter-Passing Conventions 214
9.6 C-Ref: Agreement of Formal and Actual Parameters 215
9.7 C-Ref: Function Return Types 216
9.8 C-Ref: Agreement of Actual and Declared Return Type 216
9.9 C-Ref: Main Programs 217

10 C-Ref: Program Structure 219

10.1 C-Ref: Modularization 219
10.2 C-Ref: Designing the Stack Module 220
10.3 C-Ref: Data Structures 221
10.4 C-Ref: Robustness 221
10.4.1 C-Ref: Stack Example: Conditionally Compiled Debugging Code 222
10.5 C-Ref: Allocating and Deallocating Stacks 223
10.5.1 C-Ref: Stack Example: Allocation of Stacks 224
10.5.2 C-Ref: Stack Example: Deallocation of Stacks 225
10.6 C-Ref: Operations on Stacks 225
10.6.1 C-Ref: Stack Example: Push and Pop Operations 226
10.6.2 C-Ref: Stack Example: Peek Operation 227
10.6.3 C-Ref: Stack Example: Determining Stack Sizes 228
10.7 C-Ref: Packaging the Module 228
10.7.1 C-Ref: Stack Example: Header File (Part 1, Types) 229
10.7.2 C-Ref: Stack Example: Header File (Part 2, Operations) 229

11 C-Ref: Draft Proposed ANSI C 233

11.1 C-Ref: ANSI C Lexical Elements 233
11.1.1 C-Ref: ANSI C Character Sets 233
11.1.2 C-Ref: ANSI C Identifiers 234
11.1.3 C-Ref: ANSI C Reserved Words 234

11.1.4 C-Ref: ANSI C Integer Constants 234
11.1.5 C-Ref: ANSI C Floating Point Constants 235
11.1.6 C-Ref: ANSI C String Constants 236
11.1.7 C-Ref: ANSI C Character Escape Codes 236
11.2 C-Ref: ANSI C Preprocessor 237
11.2.1 C-Ref: ANSI C Lexical Structure 237
11.2.2 C-Ref: ANSI C Stringization and Merging of Tokens 237
11.2.3 C-Ref: ANSI C Predefined Macros 238
11.2.4 C-Ref: ANSI C #include 238
11.2.5 C-Ref: ANSI C Macro Definition and Expansion 239
11.2.6 C-Ref: ANSI C New Commands 239
11.3 C-Ref: ANSI C Declarations 240
11.3.1 C-Ref: ANSI C Scopes and Name Spaces 240
11.3.2 C-Ref: ANSI C Forward References to Structures 241
11.3.3 C-Ref: ANSI C Type Specifiers 241
11.3.4 C-Ref: ANSI C Declarators 242
11.3.5 C-Ref: ANSI C Function Prototypes 243
11.3.6 C-Ref: ANSI C Initializers 246
11.3.7 C-Ref: ANSI C External Names 247
11.4 C-Ref: ANSI C Types 247
11.4.1 C-Ref: ANSI C Integer Types 248
11.4.2 C-Ref: ANSI C Floating-point Types 249
11.4.3 C-Ref: ANSI C const 249
11.4.4 C-Ref: ANSI C volatile 250
11.4.5 C-Ref: ANSI C Generic Pointers 253
11.5 C-Ref: ANSI C Conversions and Representations 254
11.5.1 C-Ref: ANSI C Number Representation 254
11.5.2 C-Ref: ANSI C Assignment Conversions 255
11.5.3 C-Ref: ANSI C The Usual Unary Conversions 255
11.5.4 C-Ref: ANSI C The Usual Binary Conversions 256
11.5.5 C-Ref: ANSI C The Function Argument Conversions 257
11.6 C-Ref: ANSI C Expressions 257
11.6.1 C-Ref: ANSI C Component Selection 257
11.6.2 C-Ref: ANSI C Function Calls 258
11.6.3 C-Ref: ANSI C `sizeof` Operator 258
11.6.4 C-Ref: ANSI C Address Operator 258
11.6.5 C-Ref: ANSI C Unary Plus Operator 258
11.6.6 C-Ref: ANSI C Addition and Subtraction 259
11.6.7 C-Ref: ANSI C Relational Expressions 259
11.6.8 C-Ref: ANSI C Constant Expressions 259
11.7 C-Ref: ANSI C Statements 260
11.8 C-Ref: ANSI C Run-time Library 260

III C-REF: THE C LIBRARIES 263

12 C-Ref: Introduction to the Libraries 265

12.1 C-Ref: Draft Proposed ANSI C Facilities 266
12.1.1 C-Ref: Draft Proposed ANSI C Libraries (Part 1) 267
12.1.2 C-Ref: Draft Proposed ANSI C Libraries (Part 2) 269

13 C-Ref: Standard Language Additions 273

13.1 C-Ref: **NULL, PTRDIFF_T, SIZE_T** 273
13.2 C-Ref: **ERRNO, STRERROR, PERROR** 274
13.3 C-Ref: DATE , FILE , LINE , TIME , STDC 275
13.4 C-Ref: **VARARG, STDARG** 276
13.4.1 C-Ref: Printargs Function in Traditional C 278
13.4.2 C-Ref: Printargs Function in Draft Proposed ANSI C 279

14 C-Ref: Character Processing 281

14.1 C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL** 282
14.2 C-Ref: **ISCSYM, ISCSYMF** 283
14.3 C-Ref: **ISDIGIT, ISODIGIT, ISXDIGIT** 283
14.4 C-Ref: **ISGRAPH, ISPRINT, ISPUNCT** 284
14.5 C-Ref: **ISLOWER, ISUPPER** 284
14.6 C-Ref: **ISSPACE, ISWHITE** 285
14.7 C-Ref: **TOASCII** 285
14.8 C-Ref: **TOINT** 285
14.9 C-Ref: **TOLOWER, TOUPPER** 285

15 C-Ref: String Processing 287

15.1 C-Ref: **STRCAT, STRNCAT** 288
15.2 C-Ref: **STRCMP, STRNCMP** 289
15.3 C-Ref: **STRCPY, STRNCPY** 289
15.4 C-Ref: **STRLEN** 290
15.5 C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS** 290
15.6 C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK** 291
15.7 C-Ref: **STRSTR, STRTOK** 292
15.8 C-Ref: **STRTOD, STRTOL, STRTOUL** 293
15.9 C-Ref: **ATOF, ATOI, ATOL** 295

16 C-Ref: Memory Functions 297

16.1 C-Ref: **MEMCHR** 297
16.2 C-Ref: **MEMCMP, BCMP** 298
16.3 C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY** 298
16.4 C-Ref: **MEMSET, BZERO** 299

17 C-Ref: Input/Output Facilities 301

17.1 C-Ref: EOF 303
17.2 C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN** 303
17.3 C-Ref: **SETBUF, SETVBUF** 304
17.4 C-Ref: **STDIN, STDOUT, STDERR** 306
17.5 C-Ref: **FSEEK, FTELL, REWIND** 306

17.6 C-Ref: **FGETC, GETC, GETCHAR, UNGETC** 307
17.7 C-Ref: **FGETS, GETS** 308
17.8 C-Ref: **FSCANF, SCANF, SSCANF** 309
17.9 C-Ref: **FPUTC, PUTC, PUTCHAR** 318
17.10 C-Ref: **FPUTS, PUTS** 318
17.11 C-Ref: **FPRINTF, PRINTF, SPRINTF** 319
17.11.1 C-Ref: Examples of Output Formatting (Part 1) 330
17.11.2 C-Ref: Examples of Output Formatting (Part 2) 331
17.12 C-Ref: **VFPRINTF, VPRINTF, VSPRINTF** 332
17.13 C-Ref: **FREAD, FWRITE** 333
17.14 C-Ref: **FEOF, FERROR, CLEARERR** 334
17.15 C-Ref: **REMOVE, RENAME** 335
17.16 C-Ref: **TMPFILE, TMPNAM, MKTEMP** 336

18 C-Ref: Storage Allocation 337

18.1 C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC** 337
18.2 C-Ref: **FREE, CFREE** 338
18.3 C-Ref: **REALLOC, RELALLOC** 339

19 C-Ref: Mathematical Functions 341

19.1 C-Ref: **ABS, FABS, LABS** 342
19.2 C-Ref: **DIV, LDIV** 343
19.3 C-Ref: **CEIL, FLOOR, FMOD** 343
19.4 C-Ref: **EXP, LOG, LOG10** 344
19.5 C-Ref: **FREXP, LDEXP, MODF** 344
19.6 C-Ref: **POW, SQRT** 345
19.7 C-Ref: **RAND, SRAND** 346
19.8 C-Ref: **COS, SIN, TAN** 346
19.9 C-Ref: **ACOS, ASIN, ATAN, ATAN2** 347
19.10 C-Ref: **COSH, SINH, TANH** 347

20 C-Ref: Time and Date Functions 349

20.1 C-Ref: **CLOCK, CLOCK_T, CLK_TCK, TIMES** 349
20.2 C-Ref: **TIME, TIME_T** 351
20.3 C-Ref: **ASCTIME, CTIME** 351
20.4 C-Ref: **GMTIME, LOCALTIME, MKTIME** 352
20.5 C-Ref: **DIFFTIME** 353

21 C-Ref: Control Functions 355

21.1 C-Ref: **ASSERT, NDEBUG** 355
21.2 C-Ref: **EXEC, SYSTEM** 356
21.3 C-Ref: **EXIT, ABORT** 357
21.4 C-Ref: **SETJMP, LONGJMP, JMP_BUF** 358
21.5 C-Ref: **ONEXIT, ONEXIT_T** 359
21.6 C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL** 359
21.7 C-Ref: **SLEEP, ALARM** 361

22 C-Ref: Miscellaneous Functions **363**

22.1 C-Ref: **MAIN** 363
22.2 C-Ref: **CTERMID, CUSERID** 364
22.3 C-Ref: **GETCWD, GETWD** 365
22.4 C-Ref: **GETENV, GETLOGIN, GETOPT, PUTENV** 366
22.5 C-Ref: **BSEARCH** 366
22.6 C-Ref: **QSORT** 367

IV C-REF: THE ASCII CHARACTER SET **369**

V C-REF: SYNTAX OF THE C LANGUAGE **377**

PART I.

C-REF: PREFACE

This text is a reference manual for the C programming language. Our aim is to provide a complete discussion of the language, the run-time libraries, and a style of C programming that emphasizes the correctness, portability, and maintainability of C programs. We have focused on full implementations of the language on UNIX systems, although we have refrained from discussing any UNIX-specific features. Where the C language or its run-time library vary among different C implementations we have pointed out the major variations. Finally, we have included a discussion of the Draft Proposed ANSI C language and library, and have indicated in the text where that standard differs from current implementations.

We assume that the reader is, or wants to become, a serious C programmer, one capable of engineering large and complex systems in C. In our view, serious programmers are more concerned with correctness and reliability than with programming speed. Their programs are meant to last a generation, not a weekend. Their programming emphasizes clarity, maintainability, and portability rather than clever tricks and the fewest number of source program lines.

In keeping with a reference text format, we have presented the language in a "bottom-up" order: the lexical structure, the preprocessor, declarations, types, expressions, statements, functions, programs, and the run-time libraries. Although we expect that many of our readers will already understand basic programming concepts and have some experience with C, we have made heavy use of cross-references in the text so that the book can be read beginning at any point.

This book grew out of our effort to write a family of C compilers for a wide range of computers, from micros to supermainframes. We wanted the compilers to be well documented, to provide precise and helpful error diagnostics, and to generate exceptionally efficient object code. A C program that compiles correctly with one compiler must compile correctly under all the others, and a program that executes correctly on one computer must execute correctly on the other computers, insofar as the hardware differences allow. (This is in keeping with the spirit of C programming.)

In spite of C's popularity, and the increasing number of primers and introductory texts on C, we found that there was no description of C precise enough to guide us in designing the new compilers. Similarly, no existing description was precise enough for our programmer/customers, who would be using compilers that analyzed C programs more thoroughly than was the custom. In this text we have been especially sensitive to language features that affect program clarity, object code efficiency, and the portability of programs among different environments, both UNIX and non-UNIX.

Acknowledgments

We wish to thank our colleagues at Tartan Laboratories and Carnegie-Mellon University, who helped in the writing, editing, and production of this book, especially

Sue Broughton and Alex Czajkowski, and also Mady Bauer, Robert Firth, Chris Hanna, Don Lindsay, Joe Newcomer, Kevin Nolish, David Notkin, and Barbara Steele. We also wish to thank our fellow members of the ANSI X3J11 committee on C standardization for their contributions and comments, especially Bill Plauger of Whitesmith's and Larry Rosler of Bell Laboratories. Many people provided useful comments on the first edition of the book, including Larry Breed, Dennis Hamilton, and Ken Harrenstien.

Some of the example programs in this book were inspired by algorithms appearing in the following works, which we recommend to anyone seriously interested in programming.

- Beeler, Michael; Gosper, R. William; and Schroeppel, Richard. *HAKMEM*. AI Memo 239 (Massachusetts Institute of Technology Artificial Intelligence Laboratory, February 1972).

- Bentley, Jon Louis. *Writing Efficient Programs*. (Prentice-Hall, 1982).

- Bentley, Jon Louis. Programming Pearls, monthly column appearing in *Communications of the ACM*. (Association for Computing Machinery, first appearance August 1983).

- Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language* (Prentice-Hall, 1978).

- Knuth, Donald E. *The Art of Computer Programming* (Addison-Wesley). *Volume 1: Fundamental Algorithms* (1968). *Volume 2: Seminumerical Algorithms* (1969, 1981). *Volume 3: Sorting and Searching* (1973).

- Sedgewick, Robert. *Algorithms* (Addison-Wesley, 1983).

We are indebted to the authors of these works for their good ideas.

Sam Harbison
Guy Steele

PART II.

C-REF: THE C LANGUAGE

1. C-Ref: Introduction

C is a member of the "Algol family" of algebraic programming languages, and thus is more similar to languages such as PL/I, Pascal, and Ada, and less similar to BASIC, FORTRAN, or Lisp. A recent collection of papers, *Comparing & Assessing Programming Languages Ada, C, and Pascal* edited by Alan R. Feuer and Narain Gehani (Prentice-Hall, 1984) discusses the similarities and differences found in C, Pascal, and Ada.

The C language was designed by Dennis Ritchie at Bell Laboratories in about 1972, and its ancestry dates from Algol 60 (1960), through Cambridge's CPL (1963), Martin Richards' BCPL (1967) and Ken Thompson's B language (1970) at Bell Labs. Although C is a general-purpose programming language, it has traditionally been used for systems programming. In particular, the popular UNIX operating system is written in C. Now widely available on both UNIX and non-UNIX systems, C is increasingly popular for applications that must be ported to different computers.

C's popularity is due to several factors. First, C provides a fairly complete set of facilities for dealing with a wide variety of applications. It has all the useful data types, including pointers and strings. There is a rich set of operators and modern control structures. C also has a standard run-time library that includes useful functions for input/output, storage allocation, string manipulation, and other purposes.

Second, C programs are efficient. C is a small language, and its data types and operators are closely related to the operations provided directly by most computers. Said another way, there is only a small "semantic gap" between C and the computer hardware.

Third, C programs are generally quite portable across different computing systems. Although C allows the programmer to write nonportable code, the uniformity of most C implementations makes it relatively easy to move applications to different computers and operating systems.

Finally, there is a growing number of C programs and C programmers. The UNIX operating system provides a large set of tools that improve C programming productivity and can serve as starting points for new applications. Because UNIX has been distributed to universities for several years, many computer science students have UNIX experience.

Unfortunately, some of the very characteristics of C that account for its popularity can also pose problems for programmers. For example, C's smallness is due in large part to its lack of "confining rules," but the absence of such rules can lead to error-prone programming habits. To write well-ordered programs, the C programmer often relies on a set of stylistic conventions that are not enforced by the compiler. As another example, to allow C compilers to implement operations efficiently on a variety of computers, the precise meaning of some C operators and types is intentionally unspecified. This can create problems when moving a program to another computer.

In spite of the inelegancies, the bugs, and the confusion that often accompany C, it has withstood the test of time. It remains a language in which the experienced programmer can write quickly and well. Millions of lines of code testify to its usefulness.

1.1. C-Ref: Who Defines C?

Although C tutorials abound, there are no detailed descriptions of the current C language. The information in this book has been compiled from many sources and from personal experience with several C implementations.

The traditional language reference is the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). In fact, it is not uncommon to see references to "Kernighan and Ritchie C" by compiler vendors wanting to emphasize their complete implementations. However, since 1978 the language has evolved; some features have been added and some have been dropped. Usually, a consensus has been reached on these features, although this consensus has not always been documented. Many new implementations of C have added their own variations to the language. When we use the phrase "the original definition of C" in this book, we will mean Kernighan and Ritchie's definition, before the post-1978 changes.

The second source for information on C is the C compilers themselves; you can write a C program and see if it compiles (and, if it does, what code is generated). This approach is almost legitimate for C, because many C compilers are based on the Portable C Compiler (PCC) written at Bell Laboratories. PCC has been retargeted to different computers, and all PCC-derived compilers share a common-language front end. The problem with using PCC as an operational standard is, of course, that PCC has errors like all other compilers, and these errors are often in the gray corners of the language, which is just where clarification would be useful.

In 1982 the American National Standards Institute (ANSI) formed a technical subcommittee on C language standardization, X3J11, to propose a standard for the C language, its run-time libraries, and its compilation and execution environments. The standardization effort brought together a large number of commercial C implementors — including ourselves — and their discussions helped to clarify existing practices as well as to map out the new language. Many parts of this book have benefited from these discussions.

Except for "C-Ref: Draft Proposed ANSI C", this book describes C as it is currently implemented by the major compilers on the larger computers. We do not consider language or implementation subsets found on the smallest microcomputers. Where compilers tend to differ in their implementations, we describe the common variations. Where certain C features tend to lead to bad programming practices, we do not hesitate to suggest a better way to use the language.

1.2. C-Ref: An Overview of C Programming

A C *program* is composed of one or more C *source files*. Each source file contains some part of the entire C program, typically some number of external functions. Source files often have associated with them *header files* that provide declarations for the external functions used in other files. One source file must contain an external function named `main` (see the section "C-Ref: **MAIN**"); by convention this will be the program's entry point.

Each source file is independently processed by a C *compiler*, which translates the C program text into the instructions understood by the computer. It is the compiler which "understands" the C program and analyzes it for correctness. If the programmer has made an error the compiler can detect, the compiler issues an error message and does not complete the translation. Otherwise, the output of the compiler is usually called *object code* or an *object module*.

When all source files are compiled, the object modules are given to a program called the *linker*, which resolves references between the modules and adds some precompiled *library functions* that handle special activities like input/output. Some programming errors, like the failure to define a needed function, are caught by the linker and cause error messages to be generated. The linker is typically not specific to C; each computer system has a standard linker that is used for programs written in many different languages. The linker produces a single *executable program*, which can then be invoked, or "run."

Although all computer systems go through these steps, they may appear different to the programmer. For instance, suppose that a program to be named `prog` consists of the two C source files `proga.c` and `progb.c`. (The "`.c`" in the file names is a common convention for C source files.) The file `proga.c` could contain these lines:

```
void hello()
{
    printf("Hello!\n");
}
```

The header file for `proga.c`, named `proga.h` (also by convention), contains:

```
extern void hello();
```

File `progb.c` contains the main program, which simply calls function `hello`:

```
#include "proga.h"
main()
{
    hello();
}
```

On a UNIX system, compiling, linking, and executing the program takes only two steps:

```
% cc -o prog proga.c progb.c
% prog
```

The first line compiles and links the two source files, adds any standard library

functions needed, and writes the executable program to file `prog`. The second line then executes the program, which prints:

```
Hello!
```

The same sequence on a DEC VAX-11 computer running the VMS operating system might be this:

```
$ cc proga,progb
$ link proga,progb/exe=[prog.exe]
$ run prog
Hello!
```

In this book we will largely ignore the details of linking and running C programs; readers are urged to consult their own computer system and C compiler user documentation. We will concentrate instead on how to write the C programs.

1.3. C-Ref: Syntax Notation

When specifying the C language syntax, terminal symbols are printed in `fixed` type and are to appear in the program exactly as written. Nonterminal symbols are printed in *italic* type; they are spelled beginning with a letter and can be followed by zero or more letters, digits, or hyphens:

expression *argument-list* *declarator2*

Syntactic definitions are introduced by the name of the nonterminal being defined followed by a colon. One or more alternatives then follow on succeeding lines:

character:
  printing-character
  escape-character

When the words "`one of`" follow the colon, this signifies that each of the terminal or nonterminal symbols following on one or more lines is an alternative definition:

digit: one of
  `0 1 2 3 4 5 6 7 8 9`

Optional components of a definition are signified by appending the suffix *opt* to a terminal or nonterminal symbol:

enumeration-constant-definition:
  enumeration-constant *enumeration-initializer*$_{opt}$

initializer:
  expression
  { *initializer-list*, $_{opt}$ }

2. C-Ref: Lexical Elements

This chapter describes the lexical structure of the C language; that is, the characters that may appear in a C source file and how they are collected into lexical units, or *tokens*.

2.1. C-Ref: Character Set

A C source file is a sequence of *characters* selected from a *character set*. A C compiler may use any character set as long as it includes at least the following *standard characters*:

1. the fifty-two uppercase and lowercase alphabetic characters:

   ```
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   a b c d e f g h i j k l m n o p q r s t u v w x y z
   ```

   the ten decimal digits:

   ```
   0  1  2  3  4  5  6  7  8  9
   ```

2. the *blank* or *space* character

3. the twenty-nine graphic characters:

!	exclamation pt.	+	plus	"	double quote
#	number sign	=	equal	{	left brace
%	percent	˜	tilde	}	right brace
^	circumflex	[	left bracket	,	comma
&	ampersand	]	right bracket	.	period
*	asterisk	\	backslash	<	less than
(	left parenthesis	|	vertical bar	>	greater than
)	right paren.	;	semicolon	/	slash
-	hyphen or minus	:	colon	?	question mark
_	underscore	’	apostrophe		

There must also be some way of dividing the source program into lines; this can be done with a character or character sequence or with some mechanism outside the source character set (for example, an end-of-record indication).

Additional characters are sometimes used in C source programs, including:

1. the five formatting characters corresponding to the ASCII characters backspace (BS), horizontal tab (HT), vertical tab (VT), form feed (FF), and carriage return (CR).

2. extra graphic characters not needed in the standard character set, including the characters $ (dollar sign), @(at-sign), and ‘ (accent grave).

The formatting characters are treated as spaces and do not otherwise affect the source program. The extra graphic characters may appear only in comments, character constants, and string constants.

As will be seen from the previous discussion, C has a much larger source character set than do most other programming languages. Draft Proposed ANSI C has defined a set of trigraphs to allow C programs to be written in a restricted character set.

2.1.1. C-Ref: Execution Character Set

The character set interpreted during the execution of a C program is not necessarily the same as the one in which the C program is written. Characters in the execution character set are represented by their equivalents in the source character set or by special character escape sequences that begin with the backslash \ character.

In addition to the standard characters mentioned above, the execution character set must also include:

1. a *null* character that must be encoded as the value 0

2. a *newline* character, which is used as the end-of-line marker

The null character is used to mark the end of strings; the newline character is used to divide character streams into lines during input/output. (It must appear to the programmer as if this newline character were actually present in text streams in the execution environment. However, the run-time library implementation is free to simulate them. For instance, newlines could be converted to end-of-record indications on output, and end-of-record indications could be turned into newlines on input.)

As with the source character set, it is common for the execution character set to include the formatting characters backspace, horizontal tab, vertical tab, form feed, and carriage return. Special escape sequences are provided to represent these characters in the source program.

These source and execution character sets are the same when a C program is compiled and executed on the same computer. However, occasionally programs are cross-compiled; that is, compiled on one computer, the *host*, and executed on another computer (the *target*). When a compiler calculates the compile-time value of a constant expression involving characters, it must use the target computer's encodings, not the more natural (to the compiler writer) source encodings.

2.1.2. C-Ref: Whitespace and Line Termination

In C source programs the blank (space), end-of-line, vertical tab, form feed, and horizontal tab (if present) are known collectively as *whitespace* characters. (Comments, discussed below, are also whitespace.) These characters are ignored except insofar as they are used to separate adjacent tokens or when they appear in character or string constants. Whitespace characters may be used to lay out the C program in a way that is pleasing to a human reader.

The end-of-line character or character sequence marks the end of source program lines. In some implementations, the formatting characters carriage return, form feed, and/or vertical tab additionally terminate source lines and are called *line break* characters. Line termination is important for the recognition of preprocessor control lines. The character following a line break character is considered to be the first character of the next line. If the first character is itself a line break character, then another (empty) line is terminated, and so forth.

Finally, there is a convention in C that a backslash which is the last character on a line has the effect of removing itself and the following end-of-line marker. This convention must be adhered to in preprocessor command lines and within string constants, but many implementations (and Draft Proposed ANSI C) generalize it to apply to any source program line.

When an implementation treats any nonstandard source characters as whitespace or line breaks, it should handle them exactly as it does blanks and end-of-line markers, respectively. Draft Proposed ANSI C suggests that an implementation do this by translating all such characters to some canonical representation as the first action when reading the source program. However, programmers should probably beware of testing this by, for example, expecting a backslash followed by a form feed to be eliminated.

C imposes no limit on the maximum length of lines, although many implementations have a fixed limit, typically in the 100-500 character range. We find that keeping line length under 80 characters facilitates reading a program on a display terminal.

2.1.3. C-Ref: Character Encodings

Each character in a computer's (execution) character set will have some conventional *encoding*; that is, some numerical representation on the computer. This encoding is important because C converts characters to integers, and the values of the integers are the conventional encodings of the characters. All of the standard characters listed earlier must have distinct, nonnegative integer encodings.

A common C programming error is to assume a particular encoding is in use when, in fact, another one holds. For example, the C expression

```
'Z' - 'A' + 1
```

computes one more than the difference between the encodings of Z and A and might be expected to yield the number of characters in the alphabet. Indeed, under the ASCII character set encodings the result is 26, but under the EBCDIC encodings, in which the alphabet is not encoded consecutively, the result is 41.

2.2. C-Ref: Comments

A *comment* in a C program begins with the characters /* and ends with the first subsequent occurrence of the characters */. Comments may contain any number of characters and are always treated as whitespace. For example, the following program contains six legal C comments:

```
void Squares()  /* no arguments */
{
    int i;
    /*
       Loop from 1 to 10,
       printing out the squares
    */
    for (i=1;i<=10;i++)
        printf("%d squared is %d\n",i,i*i);
}
```

The preprocessor also treats comments as whitespace. That is, the preprocessor does not look inside comments for commands or for macro invocations, nor do line breaks inside comments terminate preprocessor commands. For example, the following three `#define` commands all have essentially the same effect.

```
#define  ten   (2*5)

#define  ten /* ten:
                one greater than nine
              */ (2*5)

#define  ten  (2/*/*/*/*/*/5)
```

A few non-UNIX C implementations, including Microsoft and Lattice, implement "nestable comments," in which each occurrence of /* must be balanced by a subsequent */. This implementation has the advantage of allowing a programmer to comment out a large piece of program text without being concerned if the text contains its own comments. However, programs depending on this feature will not be portable, and the same effect can be better achieved using the preprocessor's conditional commands. In particular,

```
#if 0
  ...
#endif
```

will effectively "comment out" any section of a program.

Nestable comments are not standard. However, for a program to be acceptable to both implementations, no comment should contain the character sequence /* inside it.

2.3. C-Ref: Tokens

The characters making up a C program are collected into lexical *tokens* according to the rules presented in the rest of this chapter. There are five classes of tokens: operators, separators, identifiers, reserved words, and constants.

When collecting characters into tokens, the compiler always forms the longest token possible, so that `external` is interpreted as a single identifier rather than as the reserved word `extern` followed by the identifier `al`. Other examples:

`b>x`	is the same as	`b > x`
`b->x`	is the same as	`b -> x`
`b-->x`	is the same as	`b -- > x`
`b--->x`	is the same as	`b -- -> x`

Adjacent tokens may be separated by whitespace characters or comments. To prevent confusion, an identifier, reserved word, integer constant, or floating-point constant must always be so separated from a following identifier, reserved word, integer constant, or floating-point constant. (In a macro body, separating tokens with comments rather than whitespace characters may cause "token merging.")

2.4. C-Ref: Operators and Separators

The simple (one character) operators in C are:

```
!  %  ^  &  *  -  +  =  ~  |  .  <  >  /  ?
```

The compound (multicharacter) operators in C are:

```
->  ++  --  <<  >>  <=   >=   ==  !=  &&  ||
+=  -=  *=  /=  %=  <<=  >>=  &=  ^=  |=
```

The other separator characters are:

```
( )    [ ]    { }    ,   ;   :
```

Strictly speaking, each of the compound assignment operators

```
+=  -=  *=  /=  %=  <<=  >>=  &=  ^=  |=
```

is considered to be two separate tokens that can be separated by whitespace. For example, one may write

```
total + = subtotal;
```

instead of

```
total += subtotal;
```

However, it is much better programming style to write the operators as if they were single tokens, which is how they are treated in Draft Proposed ANSI C.

2.5. C-Ref: Identifiers

An identifier, also called a *name* in C, is a sequence of letters, digits, and underscores. An identifier must not begin with a digit and it must not have the same spelling as a reserved word.

identifier:
  underscore
  letter
  identifier following-character

following-character:
  letter
  underscore
  digit

letter: one of

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

underscore:

```
_
```

digit:

```
0 1 2 3 4 5 6 7 8 9
```

Two identifiers are the same when they are spelled identically, including the case of all letters. That is, the identifiers `abc` and `aBc` are distinct.

The original description of C specified that two identifiers spelled identically up to the first eight characters would be considered the same even if they differed in subsequent characters. For example, the identifiers `countless` and `countlessone` would be considered the same identifier. We consider this to be a misfeature of early implementations; programmers can take note of it but should not rely on it. Draft Proposed ANSI C requires implementations to permit a minimum of 31 significant characters in identifiers, and this limit is met or exceeded by many C implementations. (External names may be subject to more stringent length and character set constraints, with six characters and one letter case being the lower limit.) We favor the use of longer names to improve program clarity and thus reduce errors.

Some compilers permit characters other than those specified above to be used in identifiers. For example, the dollar sign ($) is often allowed in identifiers as either the first or subsequent characters. These extra characters are usually necessary to allow programs to access special non-C library functions provided by some computing systems. Such functions are likely to be nonportable, and the programmer should limit the use of special characters to these cases.

2.5.1. C-Ref: Conventions for Identifiers

Although not part of the C language, there are some conventions in the choice of identifiers which are followed by many C programmers and which may result in programs that are easier to understand and that are more easily ported to different computer systems. Of course, it is more important to be consistent than to follow any one set of conventions slavishly.

It is considered bad style to have distinct identifiers that differ only in the case of their letters, such as `count` and `Count`. In UNIX environments, there has been some tendency to spell preprocessor macro names—especially those that denote numeric constants—with uppercase letters and all other identifiers in lowercase. The following is a typical example.

```
#define TABLESIZE 100
...
int i, squares[TABLESIZE];
for (i=0; i<TABLESIZE; i++)
    squares[i] = i*i;
```

The trend toward longer identifiers may make mixed-case identifiers and the use of underscores more popular, since it is significantly harder to read a name like `averylongidentifer` than `AVeryLongIdentifier` or `a very long identifier`.

An external identifier—any one declared with storage class `extern`—often is subject to additional restrictions on particular computer systems. These identifiers have to be processed by other software, such as debuggers and linkers, which may have their own fixed limits on the lengths of identifiers and which may not distinguish the case of letters. In general, the rule has been to keep external identifiers short (say, six characters) and to not depend on case sensitivity. In UNIX systems, the problem for programmers is slightly worse because the C compilers prefix all of the user's external identifiers with an underscore, _.

When a C compiler permits long internal identifiers but the target computer requires short external names, the preprocessor may be used to hide these short names. In the example below, an external error-handling function has the short and somewhat obscure name `eh73`, but the function is referred to by the more readable name `error handler`. This is done by making `error handler` a preprocessor macro that expands to the name `eh73`.

```
#define error handler  eh73
extern void error handler();
int *p;
...
if (!p) error handler("nil pointer error");
```

A programmer especially concerned with portability might encapsulate all external names in this way.

2.6. C-Ref: Reserved Words

The following identifiers are reserved in the C language and must not be used as program identifiers.

```
auto        else        long        typedef
break       enum        register    union
case        extern      return      unsigned
char        float       short       void
continue    for         sizeof      while
default     goto        static
do          if          struct
double      int         switch
```

The reserved words `enum` and `void` are new since the original description of C, reflecting additions to the language. The former reserved words `entry`, `asm`, and `fortran` are now seldom seen, although `asm` does appear occasionally.

Draft Proposed ANSI C adds the reserved words `const`, `signed`, and `volatile`.

A reserved word may be used as a preprocessor macro name, although doing so is usually poor style. As an example of a reasonable use, the following macro definition could be appropriate when a particular C compiler does not implement the `void` type:

```
#define void int
```

2.7. C-Ref: Constants

The lexical class of constants includes four different kinds of constants: integers, floating-point numbers, characters, and strings.

constant:
- *integer-constant*
- *floating-point-constant*
- *character-constant*
- *string-constant*

Such tokens are called *literals* in other languages, to distinguish them from objects whose values are constant (that is, not changing) but which do not belong to lexically distinct classes. An example of these latter objects in C is enumeration constants, which belong to the lexical class of identifiers. In this book, we use the traditional C terminology of "constant" for both cases.

Every constant is characterized by a *value* and a *type*. The formats of the various kinds of constants are described in the following sections.

2.7.1. C-Ref: Integer Constants

Integer constants may be specified in decimal, octal, or hexadecimal notation.

1. A decimal integer constant consists of a nonempty sequence of digits, the first of which is not `0`.

2. An octal integer constant consists of the digit `0`, followed by a possibly empty sequence of the octal digits `0` through `7`. Originally, C also allowed the digits `8` and `9` in octal constants, but using them was always considered to be bad style.

3. A hexadecimal integer constant consists of the digit `0`, followed by one of the letters x or X, followed by a sequence of hexadecimal digits. The hexadecimal digits are the digits `0` through `9`, plus the characters a through f (or A through F), which have the values 10 through 15, respectively.

There is a question as to whether `"0"` is decimal or octal, but it doesn't matter in practice. Any integer constant may be immediately followed by the one of the letters l or L to indicate a constant of type `long`.

integer-constant:
decimal-constant type-marker$_{opt}$
octal-constant type-marker$_{opt}$
hexadecimal-constant type-marker$_{opt}$

decimal-constant:
nonzero-digit
decimal-constant digit

octal-constant:
`0`
octal-constant octal-digit

hexadecimal-constant:
hex-marker
hexadecimal-constant hex-digit

digit: one of
`0 1 2 3 4 5 6 7 8 9`

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hex-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`A B C D E F a b c d e f`

type-marker: one of
`l L`

hex-marker: one of
`0x 0X`

The value of an integer constant is always nonnegative in the absence of overflow. If there is a preceding minus sign, it is taken to be a unary operator applied to the constant, not part of the constant itself.

The type of integer constants is normally `int`. However, the type will instead be `long` if

1. the value of a decimal constant exceeds the largest positive integer that can be represented in type `int`, or

2. the value of an octal or hexadecimal constant exceeds the largest integer that can be represented in type `unsigned int`, or

3. the constant is terminated by the letter `l` or `L`.

If the value of a decimal constant exceeds the largest integer representable in type `long`, or if the value of an octal or hexadecimal constant exceeds the largest integer representable in type `unsigned long`, the result is unpredictable. Most C compilers will not warn the programmer of the problem and will silently substitute another value for the constant. The programmer should take pains to parameterize large constants so that they can be changed when moving to another computer or compiler. For instance:

```
#define MAXPOSINT 0077777
#define MAXNEGINT 0100000
#define MAXPOSLONG 0x37777777
#define MAXNEGLONG 0x80000000
```

To illustrate some of the subtleties in C's integer constants, assume that for some implementation type `int` uses a 16-bit two's-complement representation, and that type `long` uses a 32-bit two's-complement representation. We list in Table "C-Ref: Integer Constants Table" some interesting integer constants, their true mathematical values, and their types and values. (Parentheses are used to indicate that the value is undefined, but the parenthesized value is likely to be the one used.)

An interesting point to note in this example is that integers greater than 2^{15}-1 but less than 2^{16} will have positive values when written as decimal constants but negative values when written as octal or hexadecimal constants.

In spite of these anomalies, the programmer will rarely be "surprised" by the values of integer constants, at least when the target computer uses the two's-complement representation for integers. (Most computers do.) For the computers that use other integer representations, either the rules for interpreting integer constants or the rules for converting between signed and unsigned integers will have to be adjusted by the implementation.

Draft Proposed ANSI C has extended the syntax of integer constants and has changed the type rules slightly. See the section "C-Ref: ANSI C Integer Constants".

2.7.1.1. C-Ref: Integer Constants Table

2.7.2. C-Ref: Floating-point Constants

Floating-point constants may be written with a decimal point, a signed exponent, or both. A floating-point constant is always interpreted to be in decimal radix.

C Constant	True Value	C Type	C Value
`0`	0	`int`	0
`32767`	2^{15}-1	`int`	32767
`077777`	2^{15}-1	`int`	32767
`32768`	2^{15}	`long`	32768
`0100000`	2^{15}	`(int)`	(-32768)
`65535`	2^{16}-1	`long`	65535
`0xFFFF`	2^{16}-1	`(int)`	(-1)
`65536`	2^{16}	`long`	65536
`0x10000`	2^{16}	`long`	65536
`2147483647`	2^{31}-1	`long`	2147483647
`0x7FFFFFFF`	2^{31}-1	`long`	2147483647
`2147483648`	2^{31}	`(long)`	(-2147483648)
`0x80000000`	2^{31}	`long`	-2147483648
`4294967295`	2^{32}-1	`(long)`	(-1)
`0xFFFFFFFF`	2^{32}-1	`long`	-1
`4294967296`	2^{32}	`(long)`	(0)
`0x100000000`	2^{32}	`(long)`	(0)

floating-constant:
 digit-sequence exponent
 dotted-digits exponent$_{opt}$
exponent:
 `e` *sign-part*$_{opt}$ *digit-sequence*
 `E` *sign-part*$_{opt}$ *digit-sequence*

dotted-digits:
 digit-sequence .
 digit-sequence . *digit-sequence*
 . *digit-sequence*

digit-sequence:
 digit
 digit digit-sequence

digit: one of
 `0 1 2 3 4 5 6 7 8 9`

Examples of floating-point constants include:

```
0.          3e1        3.14159
 .0         1.0e-3     1e-3
1.0         .00034     2e+9
```

The value of a floating-point constant is always nonnegative in the absence of overflow. If there is a preceding minus sign, it is taken to be a unary operator applied to the constant, not part of the constant itself.

The type of a floating-point constant is always `double`. Its value will depend on the precision of the representation of type `double`. If the magnitude of the floating-point constant is too great or too small to be represented, the result is unpredictable. Some compilers will warn the programmer of the problem, but most will silently substitute some other value that can be represented.

Draft Proposed ANSI C has extended the syntax of floating-point constants and the set of floating-point types. See sections "C-Ref: ANSI C Floating Point Constants" and "C-Ref: ANSI C Floating-point Types"

2.7.3. C-Ref: Character Constants

A character constant is written by enclosing a character in apostrophes. A special escape mechanism is provided to write characters that would be inconvenient or impossible to enter directly in the source program.

character-constant:
 ' *character* '

character:
 printing-character
 escape-character

The printing characters include all the characters in the source character set that have equivalents in the target character set, *except* newline, the apostrophe, and the backslash. These excluded characters may be entered as escape characters, as described in the section "C-Ref: Escape Characters".

Character constants have type `int`. Their values are the integer encodings of the corresponding characters in the target character set. Below are some examples of character constants along with their (decimal) values under the ASCII encodings:

`'a'` (97)	`'A'` (65)	`'%'` (37)
`' '` (32)	`'?'` (63)	`'8'` (56)
`'\r'` (13)	`'\0'` (0)	`'\23'` (19)
`'"'` (34)	`'\377'` (255)	`'\\'` (92)

It is good programming style to restrict the "printing characters" to those characters with a graphic representation, including the blank. The formatting characters, in particular, should always be expressed as escape characters. Some compilers may enforce this restriction.

When a character constant appearing in the source program contains a character or escape character for which there is no corresponding character in the execution character set, the resulting value is implementation-defined. When a numeric escape code (section "C-Ref: Numeric Escape Codes") is used in a character constant, it is normal to compute the resulting integer value as if it had been converted from an object of type `char`. For example, if type `char` were implemented as an 8-bit signed type, the character constant `'\377'` would undergo sign extension and thus have the value -1. Draft Proposed ANSI C mandates this interpretation.

Most computers represent integers in a storage area big enough to hold several characters, and so many C compilers allow "multicharacter" constants, such as

'ABC'. The intent is to create an integer value (not a string) from the characters. The use of this feature, if available, can result in portability problems, not only because integers have different sizes on different computers but also because computers differ in their "byte ordering," that is, the order in which characters are packed into words. Given the constant 'ABC', some computers would put 'A' in the low-order bits and others would put 'A' in the high-order bits. Draft Proposed ANSI C permits multicharacter constants but leaves their value up to the implementation.

2.7.4. C-Ref: String Constants

A string constant is a (possibly empty) sequence of characters enclosed in double quotes. The same escape mechanism provided for character constants can be used to express the characters in the string.

string-constant: " *character-sequence*$_{opt}$ "

character-sequence:
character
character-sequence character

In the case of string constants, the printing characters include all the characters in the source character set that have equivalents in the target character set, except newline, the double quote, and the backslash. These excluded characters may be entered as escape characters, as described in section "C-Ref: Escape Characters". A string constant must be contained on one source program line except if the last character on the line is a backslash \ character, in which case the backslash and end-of-line character(s) are ignored. This allows string constants to be written on more than one line. (Some implementations also remove leading whitespace characters from the continuation line, although it is incorrect to do so.) A newline character (i.e., the end of line in the execution character set) may be inserted into a string by putting the escape sequence \n in the string constant; this should not be confused with line continuation within a string constant. Some examples of string constants include:

```
""
"Total expenditures: "
"\""
"Copyright 1982 Tartan Laboratories Incorporated.\
All rights reserved."
"Comments begin with '/*'.\n"
```

For each string constant of n characters there will be at run time a statically allocated block of n+1 characters whose first n characters are initialized with the characters from the string and whose last character is the null character, \0.

The type of a string constant is "array of char," and its value is the n+1 characters. For example, the value of sizeof("abcdef") is 7. However, if the string constant appears anywhere except as an argument to sizeof or as an initializer of a character array, the conversions that are usually applied to arrays come into play, and these conversions change the string from an array of characters to a pointer

to the first character in the string, of type "pointer to char." Thus we can have

```
char *p = "abcdef";
```

Traditionally—at least under UNIX—string constants are placed in read/write storage and no two string constants are ever represented by the same block of storage, even when they contain the same characters. We consider it a bad programming style to modify the contents of a string constant or to depend on distinct copies in storage, and Draft Proposed ANSI C agrees by allowing strings to share storage and to be in read-only memory. However, it is known that some UNIX library routines do modify their string arguments, which are often string constants. When a pointer to a writable string must be used, it is better to initialize an array of characters than to establish a pointer to a string constant.

```
char p1[] = "abcdef";   /* p1[i] will be writable */
char *p2  = "ghijkl";   /* p2[i] may not be writable */
```

2.7.5. C-Ref: Escape Characters

Escape characters can be used in character and string constants to represent characters that would be awkward or impossible to enter in the source program directly. The escape characters come in two varieties: "character escapes," which can be used to represent some particular formatting and special characters, and "numeric escapes," which allow a character to be specified by its numeric encoding.

escape-character:
    \ *escape-code*

escape-code:
    character-escape-code
    numeric-escape-code

character-escape-code: one of
    `n t b r f v \ ' "`

numeric-escape-code:
    octal-digit
    octal-digit octal-digit
    octal-digit octal-digit octal-digit

The meanings of these escapes are discussed in the following sections.

If the character following the backslash is neither an octal digit nor one of the character escape codes listed above, the result should be considered unpredictable, although traditionally the effect is simply that the backslash is ignored.

2.7.6. C-Ref: Character Escape Codes

Character escape codes are used to represent some common special characters in a fashion that is independent of the target computer character set. The characters that may follow the backslash, and their meanings, are listed below.

b backspace
f form feed
n newline
r carriage return
t horizontal tabulate
v vertical tabulate
\ backslash
' single quote
" double quote

To show how the character escapes can be used, here is a small program that counts the number of lines (actually, the number of newline characters) in the input. The function getchar returns the next input character until the end of the input is reached, at which point getchar returns -1 (the standard name for this conventional value is EOF).

```
/* Count the number of lines in the input. */
main()
{
    int next char;          /* Next input character */
    int num lines = 0;      /* Number of newlines seen*/

    while ((next char = getchar()) != EOF)
                                /* For each character */
        if (next char == '\n') /*  if it's a newline */
            ++num lines;        /*   bump the counter.*/

    /* Now print the total. */
    printf("%d lines read.\n",num lines);
}
```

Draft Proposed ANSI C extends the set of character escapes.

2.7.7. C-Ref: Numeric Escape Codes

Numeric escape codes allow any character to be expressed by writing that character as its octal encoding in the target character set. Up to three octal digits may be used to express the encoding, which is sufficient for characters represented by up to nine bits on the target computer. For instance, under the ASCII encodings, the character a may be written as \141 and the character ? as \77. The null character, used to terminate strings, is always written as \0. The value of a numeric escape that does not correspond to a character in the execution character set is implementation-defined. (See the section "C-Ref: Character Constants".)

The following short segment from a communications protocol program illustrates the use of numeric escape codes.

```
{
    char inchar;
    extern char receive();
    extern void reply();
    for (;;) {             /* Repeat "forever" */
       inchar = receive(); /* Receive next character */
       if (inchar == '\0')
           continue;       /* Ignore null characters */
       if (inchar == '\004')
           break;           /* Quit when EOT seen. */
       if (inchar == '\006')
           reply('\006');   /* Reply ACK to ACK. */
       else
           reply('\025');   /* Reply NAK to others. */
    }
}
```

The programmer should be cautious when using numeric escapes for two reasons. First, the syntax for numeric escapes is very delicate; a numeric escape code terminates when three octal digits have been used or when the first character that is not an octal digit is encountered. Therefore, the string `"\0111"` consists of two characters, `\011` and `1`, and the string `"\080"` consists of three characters, `\0`, `8`, and `0`.

The second reason is that the use of any numeric escape in a character or string constant—except for the null character—may make the C program nonportable. Character sets are implementation dependent, and if the specified encoding is not present in the target character set, the result is unpredictable. There is an implicit assumption in some C programs that the target computer will use eight bits to represent a character, so that codes `\0` through `\377` will span the target character set. This assumption is not portable, either. It is always much better to hide such escape codes in macro definitions. See how much clearer the previous example becomes:

```
#define NUL '\0'
#define EOT '\004'
#define ACK '\006'
#define NAK '\025'
{
    char inchar;
    extern char receive();
    extern void reply();
    for (;;) {              /* Repeat "forever" */
       inchar = receive(); /* Receive next character */
       if (inchar == NUL)
           continue;       /* Ignore null characters */
       if (inchar == EOT)
           break;           /* Quit when EOT seen. */
       if (inchar == ACK)
           reply(ACK);      /* Reply ACK to ACK. */
       else
           reply(NAK);      /* Reply NAK to others. */
    }
}
```

Some implementations (and Draft Proposed ANSI C) permit numeric escapes expressed in hexadecimal notation, such as `\x1A` or sometimes `\0x1A`. This is a fairly convenient feature, but is not common under UNIX.

3. C-Ref: The C Preprocessor

The C preprocessor is a simple macro processor that conceptually processes the source text of a C program before the compiler proper parses the source program. In some implementations of C, the preprocessor is actually a separate program that reads the original source file and writes out a new "preprocessed" source file that can then be used as input to the C compiler. (In such implementations, programs containing no preprocessor commands may typically be compiled directly, bypassing the preprocessing step.) In other implementations, a single program performs the preprocessing and compilation in a single pass over the source file, and no intermediate file is necessarily produced.

3.1. C-Ref: Preprocessor Commands

The preprocessor is controlled by special preprocessor command lines, which are lines of the source file beginning with the character `#`. Note that the character `#` has no other use in the C language. Lines that do not contain preprocessor commands are called lines of source program text. The standard preprocessor commands are:

`#define`	Define a preprocessor macro.
`#undef`	Remove a macro definition.
`#include`	Insert text from another file.
`#if`	Conditionally include some text, based on the value of a constant expression.
`#ifdef`	Conditionally include some text, based on whether a macro name is defined.
`#ifndef`	Conditionally include some text, with the sense of the test opposite that of `#ifdef`.
`#else`	Alternatively include some text, if the previous `#if`, `#ifdef`, or `#ifndef` test failed.
`#endif`	Terminate conditional text.
`#line`	Supply a line number for compiler messages.

There are also two recent additions to the preprocessor commands. They are convenient, but not found in all C compilers:

`#elif`	Alternatively include text based on the value of another constant expression.
`defined`	Determine if a name is defined as a preprocessor macro. (This operator can be used in `#if` commands and removes the need for `#ifdef` and `#ifndef`.)

The preprocessor effectively removes all preprocessor command lines from the source file and makes additional transformations on the source file as directed by

the commands, such as expanding macro calls that occur within the source program text. The resulting preprocessed source text should then be a valid C program and must not contain any occurrences of # except as part of the #line command and in string and character constants.

The syntax of preprocessor commands is completely independent of (though in some ways similar to) the syntax of the rest of the C language. For example, it is possible for a macro definition to expand into a syntactically incomplete fragment, as long as the fragment makes sense (that is, is properly completed) in all contexts in which the macro is called.

All the listed commands are supported "C-Ref: Draft Proposed ANSI C", which also adds #pragma and #error.

3.2. C-Ref: Preprocessor Lexical Conventions

The preprocessor does not parse the source text, but it does break it up into tokens for the purpose of locating macro calls. The lexical conventions of the preprocessor are the same as those for the compiler proper: The preprocessor recognizes identifiers, integer constants, floating-point constants, character constants, string constants, and comments. Preprocessor commands are not recognized within character constants, string constants, or comments. (However, in some implementations names of the formal parameters of macros *are* recognized within character constants and string constants in macro bodies. See the section "C-Ref: Defining Macros with Parameters".)

A line beginning with # is treated as a preprocessor command; the name of the command must follow the # character. Some implementations of C require the # character to be the first character on the line; others allow whitespace to precede it. Some implementations allow whitespace (except line breaks) to appear between the # character and the name of the command, and others do not. The modern trend is to allow whitespace before and after #.

The remainder of the line may contain arguments for the command if appropriate. If a preprocessor command takes no arguments, then the remainder of the command line should be empty except perhaps for whitespace characters or comments. Unfortunately, it is not uncommon for implementations to ignore all characters following the expected arguments (if any), thus allowing certain kinds of errors to go unreported. The arguments to preprocessor commands are generally subject to macro replacement.

At least some UNIX implementations—whether by accident or by design we're not sure—expand macros before looking for preprocessor commands. We think this is a very bad idea and have not heard of any attempts to legitimize it. Even implementations that do this are inconsistent because they turn off macro expansion when skipping over text to search for #else or #endif commands.

Within a preprocessor command line, if an end-of-line marker is immediately preceded by \, then the end-of-line and the \ are deleted and the following line is treated as if it were part of the line that ended in \. This means that a preproces-

sor command line can be continued on the following line by ending it with \. It also means that if a line ends with \, the following line will never be treated as a preprocessor command line, even if its first non-whitespace character is #. For example,

```
#define err(flag,msg)  if (flag) \
    printf(msg)
```

is the same as

```
#define err(flag,msg)  if (flag) printf(msg)
```

As explained in the section "C-Ref: Comments", the preprocessor treats comments as whitespace, and line breaks within comments do not terminate preprocessor commands.

3.3. C-Ref: Definition and Replacement

The `#define` preprocessor command causes a name (identifier) to become defined as a macro to the preprocessor. A sequence of tokens, called the *body* of the macro, is associated with the name. When the name of the macro is recognized in the program source text or in the arguments of certain other preprocessor commands, it is treated as a call to that macro; the name is effectively replaced by a copy of the body. If the macro is defined to accept arguments, then the actual arguments following the macro name are substituted for formal parameters in the macro body.

The preprocessor does not distinguish reserved words from other identifiers, and so it is possible, in principle, to use a C reserved word as the name of a preprocessor macro, but to do so is usually bad programming practice. Macro names are never recognized within comments or string constants.

For example, if a macro `sum` with two arguments is defined by

```
#define sum(x,y)  x+y
```

then the preprocessor replaces the source program line

```
result = sum(5,a*b);
```

with

```
result = 5+a*b;
```

3.3.1. C-Ref: Simple Macro Definitions

The `#define` command has two forms, depending on whether or not a left parenthesis '(' immediately follows the name to be defined. The simpler form has no left parenthesis there:

`#define` *name sequence-of-tokens*

A macro defined in this manner takes no arguments. It is invoked merely by mentioning its name. When the name is encountered in the source program text or other appropriate context, the name is replaced by the body (the associated *sequence-of-tokens*).

The simple form of macro is particularly useful for introducing named constants into a program, so that a "magic number" such as the length of a table may be written in exactly one place and then referred to elsewhere by name. This makes it easier to change the number later. For example:

```
#define BLOCK SIZE 0x100
                    /* Size of one disk block. */
#define TRACK SIZE (16*BLOCK SIZE)
                    /* Size of one disk track. */
#define HASH TABLE SIZE 557
                /* Initial size of hash table. */
#define ERRMSG "*** Error %d: %s.\n"
                /* Format for use with printf. */
/* File protection bits,
     as used in system file directories. */
#define READ ACCESS    0100000
                  /* May read the file as data. */
#define WRITE ACCESS   0040000
                /* May alter existing contents. */
#define APPEND ACCESS  0020000
                      /* May add new contents. */
#define EXECUTE ACCESS 0010000
                       /* May execute as code. */
#define DELETE ACCESS  0004000
                       /* May delete the file. */
#define RENAME ACCESS  0002000
                       /* May rename the file. */
#define BACKUP ACCESS  0001000
                   /* May move to backup tape. */
#define ACCESS ACCESS  0000400
                /* May modify the access bits. */
```

The syntax of the `#define` command does *not* require an equal sign or any other special delimiter token after the name being defined. The body starts right after the name. (If the body begins with an alphabetic character, digit, or left parenthesis, then a space is needed to separate the body from the name, of course.)

A typical programming error is to include an extraneous equal sign:

```
/* Probably wrong:     */
#define NUMBER OF TAPE DRIVES = 5
```

This is a legal definition but causes the name `NUMBER OF TAPE DRIVES` to be defined as "= 5" rather than as "5". If one were then to write the source program line

```
count = NUMBER OF TAPE DRIVES;
```

it would be expanded to

```
count = = 5;                /* Illegal. */
```

which is syntactically illegal. Worse yet, one might accidentally write

```
result = count + NUMBER OF TAPE DRIVES;
```

which would be expanded to

```
result = count + = 5;   /* Probably wrong. */
```

which is syntactically legal (because += is a valid compound assignment operator even with embedded whitespace) but almost certainly not what was intended! One could write

```
count NUMBER OF TAPE DRIVES;    /* Ugh! */
```

which would be expanded to

```
count = 5;
```

and therefore assign 5 to count, but the best that can be said for this practice is that it is confusing. The lesson is clear: Be careful not to include an extraneous equal sign in a macro definition. For similar reasons, be careful also not to include an extraneous semicolon:

```
#define NUMBER OF TAPE DRIVES 5;
                /* Probably wrong. */
```

An important use of simple macro definitions is to isolate implementation-dependent restrictions on the names of externally defined functions and variables. An example of this appears in the section "C-Ref: Identifiers".

3.3.2. C-Ref: Defining Macros with Parameters

The more complex form of macro definition declares the names of formal parameters within parentheses, separated by commas:

`#define` *name*(*name1*, *name2*, ..., *namen*) *sequence-of-tokens*

The left parenthesis must immediately follow the name of the macro with no intervening whitespace. (If whitespace separates the left parenthesis from the macro name, the definition is considered to define a macro that takes no arguments and has a body beginning with a left parenthesis.)

The names of the formal parameters must be identifiers, no two the same. There is no requirement that any of the parameter names be mentioned in the body (though normally they all will be mentioned).

A macro defined in this manner takes as many actual arguments as there are formal parameters. The macro is invoked by writing its name, a left parenthesis, then one actual argument token sequence for each formal parameter, then a right parenthesis. The actual argument token sequences are separated by commas. For example:

```
#define product(x,y)  ((x)+(y))
...
    return product(a+3,b);
```

Whitespace may appear between the macro name and the left parenthesis or in the actual arguments.

A macro can be defined to have zero formal parameters:

```
#define         getchar()    getc(stdin)
```

When such a macro is invoked, an empty actual argument list must be provided:

```
while ((c=getchar()) != EOF) ...
```

This kind of macro is useful to simulate a function that takes no arguments.

An actual argument token sequence may contain parentheses if they are properly nested and balanced and may contain commas if each comma appears within a set of parentheses (this restriction prevents confusion with the commas that separate the actual arguments). Parentheses and commas may also appear freely within character-constant and string-constant tokens and are not counted in the balancing of parentheses and the delimiting of actual arguments. For example, the arguments to the `product` macro above could be function calls:

```
int f(), g();
...
    return product( f(a,b), g(a,b) );
```

Braces and subscripting brackets may also appear within macro arguments, but they cannot contain commas and do not have to balance. For example, suppose we define a macro that takes as its argument an arbitrary statement:

```
#define insert(stmt)   stmt
```

The invocation

```
insert(  {a=1; b=1;}   )
```

works properly, but if we change the two assignment statements to a single statement containing two assignment expressions:

```
insert(  {a=1, b=1;}   )
```

then the preprocessor will complain that we have too many macro arguments for `insert`. To fix the problem we would have to write:

```
insert(  {(a=1, b=1);}   )
```

Some (deficient) preprocessor implementations do not permit the actual argument token list to extend across multiple lines unless the lines to be continued end with a \.

When a complex macro call is encountered, the entire macro call is replaced, after parameter processing, by a processed copy of the body. Parameter processing proceeds as follows. Actual argument token strings are associated with the corresponding formal parameter names. A copy of the body is then made in which every occurrence of a formal parameter name is replaced by a copy of the actual argument token sequence associated with it. This copy of the body then replaces the macro call. The entire process of replacing a macro call with the processed copy of its body is called *macro expansion*; the processed copy of the body is called the *expansion* of the macro call.

As an example, consider this macro definition:

```
/* incr: This macro expands into a for statement
   that causes the variable (any lvalue) to take
   on all values from l to h, inclusive.  Note
   that v and h may be evaluated more than once.
*/
#define incr(v,l,h)  \
              for ((v) = (l); (v) <= (h); (v)++)
```

This provides a convenient way to make a loop that just counts from a given value up to (and including) some limit. For example, to print a table of the cubes of the integers from 1 to 20, we could write

```
main()
{
    int j;
    printf(" N  N cubed\n");
    incr(j, 1, 20)
        printf("%2d %6d\n", j, j*j*j);
}
```

The call to the macro `incr` is expanded by the preprocessor to produce this program to be compiled:

```
main()
{
    int j;
    printf(" N  N cubed\n");
    for ((j) = (1); (j) <= (20); (j)++)
        printf("%2d %6d\n", j, j*j*j);
}
```

(The liberal use of parentheses ensures that complicated actual arguments will not confuse the compiler. See the section "C-Ref: Precedence Errors in Macro Expansions".)

3.3.3. C-Ref: Rescanning of Macro Expressions

Once a macro call has been expanded, the scan for macro calls resumes at the *beginning* of the expansion; this is so that names of macros may be recognized within the expansion for the purpose of further macro replacement. Note that macro replacement is not performed on any part of a `#define` command, not even the body, at the time the command itself is processed and the macro name defined. Macro names are recognized within the body only after the body has been expanded for some particular macro call. Macro replacement is also not performed within the actual argument token strings of a complex macro call at the time the macro call is being scanned. Macro names are recognized within actual argument token strings only during the rescanning of the expansion, assuming that the corresponding formal parameter in fact occurred one or more times within the body (thereby causing the actual argument token string to appear one or more times in the expansion). For example, given the following macro definitions:

```
#define plus(x,y) add(y,x)
#define add(x,y) ((x)+(y))
```

The macro invocation

```
plus( plus(a,b), c )
```

is expanded in the following steps:

```
plus( plus(a,b), c )
add( c, plus(a,b) )
((c)+(plus(a,b)))
((c)+(add(b,a)))
((c)+(((b)+(a))))
```

If a macro expands into something that looks like a preprocessor command, that command will *not* be recognized as a command by the preprocessor. For example, the result of

```
/* This example doesn't work as one might think! */
#define GETMATH #include <math.h>
GETMATH
```

is *not* to include the file `math.h` in the program being compiled. The call to the macro `GETMATH` expands into the token sequence

```
# include < math . h >
```

but that token sequence is not recognized as a preprocessor `#include` command; the token sequence is merely passed through and compiled as (erroneous) C code. In general, a line is treated as a preprocessor command line if and only if its first non-whitespace character is `#` *before* any macro replacement has been performed on the line.

Finally, it is possible to write recursive macros whose expansion does not terminate. For example, the macro

```
#define  repeat(x)  x repeat(x)
```

will expand to an infinite sequence of its argument. Most C compilers will not detect this recursion, and will attempt to continue the expansion until they are stopped by some system error.

3.3.4. C-Ref: Predefined Macros

Many implementations of C provide certain special, built-in macros for use by the programmer. For example, the macro `FILE` (spelled with four underscore characters) is often predefined to be the string name of the file being preprocessed, and `LINE` is often predefined to be the line number in the source file being compiled. These macros are useful in certain kinds of error messages:

```
if (n !== m)
  fprintf(stderr,"Internal error: line %d, file %s\n",
          LINE  ,   FILE   );
```

Built-in macros such as these were considered useful enough to add to Draft Proposed ANSI C. These macros may not be redefined or undefined.

Some implementations also routinely define certain macros to communicate information about the environment, such as the type of computer for which the program is being compiled. For example, a compiler targeted to the DEC VAX-11 computer might predefine a macro named `vax` so that the programmer could write

```
#ifdef vax
    Vax-specific-code
#endif
```

to cause certain source code to be compiled only when the target computer is a VAX. Exactly which macros are defined is implementation dependent, although UNIX implementations usually predefine `unix`. Unlike the built-in macros, these macros may be undefined.

3.3.5. C-Ref: Undefining and Redefining Macros

The `#undef` command can be used to make a name be no longer defined:

```
#undef name
```

This command causes the preprocessor to forget any macro definition of *name*. It is not an error to undefine a name that is currently not defined. Once a name has been undefined, it may then be given a completely new definition (using `#define`) without error. Macro replacement is not performed within `#undef` commands.

Implementations of C differ in how they handle an attempt to define a name that is already defined as a macro. Some implementations will discard the old definition, replace it with the new one, and perhaps issue a warning message. A better alternative is to consider the redefinition an error unless the new definition is the same (token-for-token) as the old one. (This is sometimes known as a "benign redefinition.") Programmers can avoid any problem by using `#ifndef` to be sure there is no preexisting definition:

```
#ifndef MAXTABLESIZE
#define MAXTABLESIZE 1000
#endif
```

This style is particularly effective under UNIX (and other similar implementations) because there the programmer may supply his macro definitions when the compiler is invoked. The syntax usually looks like this:

```
cc -c -DMAXTABLESIZE=5000 prog.c
```

A few non-UNIX preprocessor implementations handle `#define` and `#undef` so as to maintain a stack of definitions. When a name is redefined with `#define`, its old definition is pushed onto a stack and then the new definition replaces the old one. When a name is undefined with `#undef`, the current definition is discarded and the most recent previous definition (if any) is popped from the stack to replace it. The following program can be used to determine whether the stack model of macro definitions is in effect. (The redefinition of `MSG` in the second line may cause a compilation error, in which case the stack model is not in effect.)

```
#define MSG "#define/#undef are stacked"
#define MSG "(This definition should never remain)"
#undef MSG
#ifndef MSG
#define MSG "#define/#undef are not stacked"
#endif
int main()
{
  printf("%s\n",MSG);
  return 0;
}
```

Draft Proposed ANSI C does not support the stack model.

3.3.6. C-Ref: Precedence Errors in Macro Expansions

Macros operate purely by textual substitution of tokens. Parsing of the body into declarations, expressions, or statements occurs only after the macro expansion process. This can lead to surprising results if care is not taken. Consider this macro definition:

```
#define SQUARE(x) x*x
```

The idea is that `SQUARE` takes an argument expression and produces a new expression to compute the square of that argument. For example, `SQUARE(5)` expands to `5*5`. However, the expression

```
SQUARE(z+1)
```

expands to

```
z+1*z+1
```

When this expression is parsed, it is interpreted as

```
z+(1*z)+1
```

which will not produce the same result as `(z+1)*(z+1)` unless `z` happens to be zero. It would be somewhat safer to put parentheses around occurrences of the formal parameter in the definition of `SQUARE`:

```
#define SQUARE(x) (x)*(x)
```

Even this definition does not provide complete protection against precedence problems. Consider casting the squared value to a new type:

```
(short) SQUARE(z+1)
```

This would expand to:

```
(short) (z+1)*(z+1)
```

which would then be parsed as

```
((short) (z+1))*(z+1)
```

because a cast has higher precedence than multiplication. The definition can be improved further to avoid this difficulty:

```
#define SQUARE(x) ((x)*(x))
```

As a rule, it is safest always to parenthesize each parameter appearing in the macro body. The entire body, if it is syntactically an expression, should also be parenthesized.

3.3.7. C-Ref: Side Effects in Macro Arguments

Macros can also produce problems with side effects. Consider the macro `SQUARE` shown above and also a function `square` that does (almost) the same thing:

```
int square(x)
  int x;
{
    return x*x;
}
```

The function, unlike the macro, can square only integers, not floating-point numbers. Also, calling the function is likely to be somewhat slower at run time than using the macro. But these differences are less important than the question of side effects. In the program fragment

```
a = 3;
b = square(a++);
```

the variable `b` gets the value `9` and the variable `a` ends up with the value `4`. However, in the superficially similar program fragment

```
a = 3;
b = SQUARE(a++);
```

the variable `b` may very well get the value 12 and the variable `a` may end up with the value 5, because the expansion of the last fragment is

```
a = 3;
b = ((a++)*(a++));
```

(We say that 12 and 5 "may" be the resulting values of `b` and `a` because C implementations are free to evaluate the expression `((a++)*(a++))` in various ways.)

When a function such as `square` is used, the argument expression is evaluated exactly once, so any side effects of the expression occur exactly once. When a macro, such as `SQUARE`, is used, an actual argument may be textually replicated and therefore executed more than once, and side effects may occur more than once. Macros must be used with care to avoid such problems.

3.3.8. C-Ref: Converting Tokens to Strings

In many (but not all) implementations of C, macro formal parameter names, unlike macro calls, *are* recognized within string and character constants. Because the actual argument may have been broken down into tokens, possibly with comments and extraneous whitespace discarded, the substitution of an actual argument token sequence into a string may not result in the precise sequence of characters that appeared in the macro call. In recording the body of a macro, the compiler may make certain simplifications. It may eliminate extraneous whitespace. It may reduce constants to a "canonical form," for example reducing `0000400` to `0400` or to

256. Comments may be treated in one of three ways: replaced by whitespace, replaced by an empty comment, or maintained verbatim. Consider this definition:

```
#define MAKESTRING(x) "x"
```

The result of the call

```
MAKESTRING(a += 1    /* Increment counter. */)
```

might be any one of the following representative samples:

```
"a += 1    /* Increment counter. */"
"a += 1 "
"a += 1 /**/"
"a+=1/* Increment counter. */"
"a+=1"
```

Programs are more likely to be portable if actual arguments that are substituted for formal parameters within character constants and string constants are single tokens. Better yet, inasmuch as some implementations don't handle such substitution at all, the programmer can maximize portability simply by avoiding entirely the use of macro formal parameters within character constants and string constants.

Draft Proposed ANSI C disallows the recognition of macro formal parameters within string constants, but a need to convert tokens to string representations was recognized and a special mechanism was introduced.

3.3.9. C-Ref: Token Merging in Macro Expansions

Although the original definition of C explicitly described macro bodies as being sequences of tokens, not sequences of characters, nevertheless some C compilers expand and rescan macro bodies as if they were character sequences. This becomes apparent primarily in the case where the compiler also handles comments by eliminating them entirely (rather than by replacing them with a space), a situation exploited by some cleverly written programs:

```
#define INC ++
#define TAB internal table
#define INCTAB table of increments
#define CONC(x,y) x/**/y
CONC(INC,TAB)
```

The proper interpretation of the body of `CONC` is as a sequence of the tokens `x` and `y`, separated by a comment or a space. The comment does, in all implementations, serve to separate `x` and `y` so that they are recognized as formal parameters of the macro `CONC`. The call

```
CONC(INC,TAB)
```

ought to expand to the sequence of tokens

```
INC TAB
```

which will in turn expand to

```
++ internal table
```

However, those implementations that simply eliminate comments and that rescan macro bodies as character sequences rather than token sequences will expand the call

```
CONC(INC,TAB)
```

into the character sequence

```
INCTAB
```

and then, in rescanning it, interpret it as the single token `INCTAB`. This will then expand into

```
table of increments
```

which is a very different thing altogether. Because not all implementations treat this the same way, depending on such implicit concatenation of tokens through rescanning may render a program nonportable.

Draft Proposed ANSI C removed the kind of token merging described in this section from the preprocessor, but thought the effect important enough that a new token-concatenation operator was introduced.

3.3.10. C-Ref: Other Problems

Some implementations of C do not perform stringent error checking on macro definitions and calls, including permitting an incomplete token in the macro body to be completed by text appearing after the macro call. For example, the following definition and call

```
#define FIRSTPART "This is a split
...
    printf(FIRSTPART string.");        /* Yuk! */
```

will, after preprocessing, result in compiling the source text

```
    printf("This is a split string.");
```

The lack of error checking by certain implementations does not make clever exploitation of that lack legitimate. Draft Proposed ANSI C reaffirms that macro bodies must be sequences of well-formed tokens.

3.4. C-Ref: File Inclusion

The `#include` preprocessor command causes the entire contents of a specified source text file to be processed as if those contents had appeared in place of the `#include` command. The `#include` command has two forms. If the first non-whitespace character following the command name `#include` is a double quote `"`, then the last non-whitespace character on the command line must also be a double quote. If the first non-whitespace character following the command name `#include` is <, then the last non-whitespace character on the command line must be >. In either case, all the characters between the two delimiters constitute a file name (whose format is implementation dependent).

If the first non-whitespace character following the command name is neither " nor <, then macro replacement is performed on the part of the command line following `#include`; this allows a macro call to expand into a file name (including the appropriate delimiters).

The two forms differ in how the specified file is to be located if the location is not completely specified in the command. The form

`#include` "*filename*"

typically searches for the file first in the same "directory" in which the file containing the `#include` command was found, and then perhaps in other places according to implementation-dependent search rules. However, the form

`#include` <*filename*>

typically does *not* search for the file in the same "directory" in which the file containing the `#include` command was found, but only in certain "standard" places according to implementation-dependent search rules. The general intent is that the "..." form is used to refer to other files written by the user, whereas the <...> form is used to refer to standard "library" files.

In principle an included file may itself contain `#include` commands. The permitted depth of such `#include` nesting is implementation dependent, but most implementations will allow nesting to at least five or six levels.

3.5. C-Ref: Conditional Compilation

The preprocessor conditional commands allow lines of source text to be passed through or eliminated by the preprocessor on the basis of a computed condition.

3.5.1. C-Ref: The #if, #else, and #endif Commands

The following preprocessor commands are used together to allow lines of source text to be conditionally included in or excluded from the compilation: `#if`, `#else`, and `#endif`. They are used in the following way:

```
#if constant-expression
    group-of-lines-1
#else
    group-of-lines-2
#endif
```

The *constant-expression* is subject to macro replacement and must evaluate to a constant arithmetic value. A "group of lines" may contain any number of lines of text of any kind, even other preprocessor command lines, or no lines at all. The `#else` command may be omitted, along with the group of lines following it; this is equivalent to including the `#else` command with an empty group of lines following it. Either group of lines may also contain one or more sets of `#if`-`#else`-`#endif` commands; that is, conditional compilation commands nest properly.

A set of commands such as shown above is processed in such a way that one group of lines will be passed on for compilation and the other group of lines will be discarded. First the *constant-expression* in the `#if` command is evaluated. If its value is not 0, then *group-of-lines-1* is passed through for compilation and *group-of-lines-2* (if present) is discarded. Otherwise, *group-of-lines-1* is discarded; and if there is an `#else` command, then *group-of-lines-2* is passed through; but if there is no `#else` command, then no group of lines is passed through. The constant expressions that may be used in a `#if` command are described in detail in the sections "C-Ref: Constant Expressions in Conditional Commands" and "C-Ref: Constant Expressions".

A group of lines that is discarded is not processed by the preprocessor. Macro replacement is not performed and preprocessor commands are ignored. The one exception is that, within a group of discarded lines, the commands `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif` are recognized for the sole purpose of counting them; this is necessary to maintain the proper nesting of the conditional compilation commands. (This recognition in turn implies that discarded lines are scanned and broken into tokens and that string constants and comments are properly recognized, for example.)

If an undefined macro name appears in the *constant-expression* of `#if` or `#elif` it is replaced by the integer constant 0. This means that the commands "`#ifdef` *name*" and "`#if` *name*" will have the same effect as long as the macro *name*, when defined, has a constant, arithmetic, nonzero value. We think it is much clearer to use `#ifdef` or the `defined` operator in these cases, but even Draft Proposed ANSI C supports the use of `#if`.

3.5.2. C-Ref: The #elif Commands

The `#elif` command is a fairly recent addition to C and is present in a few compilers as well as Draft Proposed ANSI C. It is convenient because it simplifies nested preprocessor conditionals.

The `#elif` command is like a combination of `#else` and `#if`. It is used between `#if` and `#endif` in the same way as `#else` but has a constant expression to evaluate in the same way as `#if`. It is used in the following way:

```
#if constant-expression-1
   group-of-lines-1
#elif constant-expression-2
   group-of-lines-2
#elif constant-expression-3
   group-of-lines-3
   ...
#elif constant-expression-n
   group-of-lines-n
#else
   last-group-of-lines
#endif
```

A set of commands such as shown above are processed in such a way that at most

one group of lines will be passed on for compilation and all other groups of lines will be discarded. First the *constant-expression-1* in the `#if` command is evaluated. If its value is not 0, then *group-of-lines-1* is passed through for compilation and all other groups of lines up to the matching `#endif` are discarded. If the value of the *constant-expression-1* in the `#if` command is 0, then the *constant-expression-2* in the first `#elif` command is evaluated; if that value is not 0, then *group-of-lines-2* is passed through for compilation. In the general case, each *constant-expression-i* is evaluated until one produces a nonzero value; the preprocessor then passes through the group of lines following the command containing the nonzero constant expression, ignores any other constant expressions in the command set, and discards all other groups of lines. If no *constant-expression-i* produces a nonzero value, but there is an `#else` command, then the group of lines following the `#else` command is passed through; but if there is no `#else` command, then no group of lines is passed through. The constant expressions that may be used in a `#elif` command are the same as those used in a `#if` command (see the sections "C-Ref: Constant Expressions in Conditional Commands" and "C-Ref: Constant Expressions").

Within a group of discarded lines, `#elif` commands are recognized in the same way as `#if`, `#else`, and `#endif` commands, for the sole purpose of counting them; this is necessary to maintain the proper nesting of the conditional compilation commands.

Macro replacement is performed within the part of a command line that follows an `#elif` command, so macro calls may be used in the *constant-expression*.

While the `#elif` command is very convenient when it is appropriate, it is not necessary, because anything it accomplishes can be done using only `#if`, `#else`, and `#endif`. For example, this set of commands:

```
#if constant-expression-1
    group-of-lines-1
#elif constant-expression-2
    group-of-lines-2
#elif constant-expression-3
    group-of-lines-3
#else
    last-group-of-lines
#endif
```

can be rewritten in this way:

```
#if constant-expression-1
    group-of-lines-1
#else
#if constant-expression-2
    group-of-lines-2
#else
#if constant-expression-3
    group-of-lines-3
#else
    last-group-of-lines
#endif
#endif
#endif
```

3.5.3. C-Ref: The #ifdef and #ifndef Commands

The `#ifdef` and `#ifndef` commands can be used to test whether a name is defined as a preprocessor macro. A command line of the form

```
#ifdef name
```

is equivalent in meaning to

```
#if 1
```

when *name* has been defined and is equivalent to

```
#if 0
```

when `name` has not been defined or has been undefined with the `#undef` command. The `#ifndef` command has the opposite sense; it is true when the name is not defined and false when it is.

Note that `#ifdef` and `#ifndef` test names only with respect to whether they have been defined by `#define` (or undefined by `#undef`); they take no notice of names appearing in declarations in the C program text to be compiled.

These commands have come to be used in several stylized ways in C programs. First, it is a common practice to implement a preprocessor-time enumeration type by having a set of symbols of which only one is defined. For example, suppose that we wish to use the set of names `VAX`, `PDP11`, `IBM360`, and `CRAY2` to indicate the computer for which the program is being compiled. One might insist that all these names be defined, with one being defined to be `1` and the rest `0`:

```
#define VAX     0
#define PDP11   0
#define IBM360  0
#define CRAY2   1
```

One could then select machine-dependent source code to be compiled in this way:

```
#if VAX
  VAX-dependent code
#endif
#if PDP11
  PDP11-dependent code
#endif
#if IBM360
  IBM360-dependent code
#endif
#if CRAY2
  CRAY2-dependent code
#endif
```

However, the customary method defines only one symbol:

```
#define CRAY2 1
/* All the other symbols are not defined. */
```

Then the conditional commands test whether each symbol is defined:

```
#ifdef VAX
  VAX-dependent code
#endif
#ifdef PDP11
  PDP11-dependent code
#endif
#ifdef IBM360
  IBM360-dependent code
#endif
#ifdef CRAY2
  CRAY2-dependent code
#endif
```

Another use for the `#ifdef` and `#ifndef` commands is in providing default definitions for macros. For example, a library file might provide a definition for a name only if no other definition has been provided:

```
/* Library file <table.h>.
   Maintains an internal table.
 */

#ifndef TABLE SIZE
#define TABLE SIZE 100
#endif
...
static int internal table[TABLE SIZE];
...
```

A program might simply include this file:

```
#include <table.h>
```

in which case the definition of `TABLE SIZE` would be `100`, both within the library

file itself and after the `#include`; or the program might provide an explicit definition first:

```
#define TABLE SIZE 500
#include <table.h>
```

in which case the definition of `TABLE SIZE` would be `500` throughout.

3.5.4. C-Ref: Constant Expressions in Conditional Commands

The constant expressions that may be used in `#if` and `#elif` commands are described in detail in section "C-Ref: Constant Expressions". The value of the constant expression must be determined in exactly the same way as for any other constant expression in the program: The result of evaluating a constant expression must be identical to the result of evaluating the same expression at run time.

If the entire rest of the command line following `#if` or `#elif` is not, after macro replacement, a syntactically legal constant expression, or if any error occurs while determining its value (for example, division by 0), then some implementation-dependent action is taken. Some compilers issue an error message and assume the entire expression has the value 0; that is, the conditional test fails and the following group of lines is discarded. This assumption is made purely for the purpose of continuing the compilation process in order to search for additional errors.

3.5.5. C-Ref: The `defined` Operator

There is one operator, `defined`, that can be used in `#if` and `#elif` expressions but nowhere else in the C language. An expression of the form

`defined` *name*

or

`defined(` *name* `)`

evaluates to 1 if *name* is defined in the preprocessor, and 0 if it is not. This allows one to write

```
#if defined(VAX)
```

instead of

```
#ifdef VAX
```

The `defined` operator is more convenient to use because it is possible to build up complex expressions, such as

```
#if defined(VAX) && !defined(UNIX) && debugging
...
```

The `defined` operator is relatively new to C and is not implemented by all compilers. It has been adopted by Draft Proposed ANSI C.

3.6. C-Ref: Explicit Line Numbering

The `#line` preprocessor command advises the C compiler that the source program was generated by another tool and indicates the correspondence of places in the source program to lines of the original user-written file from which the C source program was produced. The `#line` commands may have one of two forms. The form

`#line` *integer-constant* "*filename*"

indicates that the next source line was derived from line *n* of the original user-written file named by *filename*. The form

`#line` *integer-constant*

indicates that the next source line was derived from line *n* of the original user-written file last mentioned explicitly in a `#line` command.

Macro replacement is performed on the part of the command line following the name `#line` before the command line is interpreted. This allows a macro call to expand into the *integer-constant*, the *filename*, or both.

The information provided by the `#line` command is used purely for the sake of giving more informative error messages. Some tools that generate C source text as output will use `#line` so that error messages can be related to the tool's input file instead of the actual C source file. Some compilers do not implement `#line`, ignoring it when present.

Some implementations of C allow the preprocessor to be used independently of the rest of the compiler. Indeed, sometimes the preprocessor is a separate program that is executed to produce an intermediate file that is then processed by the "real" compiler. In such cases the preprocessor may generate new `#line` commands in the intermediate file; the compiler proper is then expected to recognize these even though it does not recognize any other preprocessor commands. Whether the preprocessor generates `#line` commands is implementation dependent. Similarly, whether the preprocessor passes through, modifies, or eliminates `#line` commands in the input is also implementation dependent.

Older versions of C allow simply "`#`" as a synonym for the `#line` command:

`#` *integer-constant filename*

This syntax is considered obsolete, but many implementations of C continue to support it for the sake of compatibility.

Many implementations allow a `#` on a line by itself to mean a "do nothing" command; the preprocessor eliminates the line and takes no other action.

The `#line` command is not present in some non-UNIX compilers. It is part of Draft Proposed ANSI C, along with the "empty" `#` command.

4. C-Ref: Declarations

To *declare* an identifier in the C language is to associate the identifier with some C object, such as a variable, function, or type. The identifiers that can be declared in C are

- variables
- functions
- types
- type tags
- structure and union components
- enumeration constants
- statement labels
- preprocessor macros

Except for statement labels and preprocessor macros, all identifiers are declared by their appearance in C *declarations*. Variables, functions, and types appear in *declarators* within declarations, and type tags, structure and union components, and enumeration constants are declared in certain kinds of *type specifiers* in declarations. Statement labels are declared by their appearance in a C function, and preprocessor macros are declared by the `#define` preprocessor command.

Declarations in C are difficult to describe for several reasons. First, they involve some unusual syntax that may be confusing to the novice. For example, the declaration

```
int (*f)();
```

does not declare `f` to be some kind of integer but rather a pointer to a function returning an integer.

Second, many of the abstract properties of declarations, such as *scope* and *extent*, are not clearly evident in C's realization. Before jumping into the actual declaration syntax, we will discuss these properties in section "C-Ref: Terminology".

Finally, some aspects of C's declarations are difficult to understand without a knowledge of C's type system, which is described in Chapter "C-Ref: Types".

In particular, discussions of type tags, structure and union component names, and enumeration constants is left to that chapter, although some properties of those declarations will be discussed here for completeness.

4.1. C-Ref: Organization of Declarations

Declarations may appear in several places in a C program, and where they appear affects the properties of the declarations. To give an overview, a *program* consists of a sequence of *top-level declarations* of functions, variables, and other things. Each function has *parameter declarations* and a body; the body in turn may contain *blocks* (compound statements). A block may contain a sequence of *inner declarations*.

The syntax below shows the location of declarations in a C program. (Some of the syntactic alternatives are not relevant to this discussion and have been elided, as indicated by "...".)

program:
    top-level-declaration-list$_{opt}$
top-level-declaration-list:
    top-level-declaration
    top-level-declaration-list   *top-level-declaration*

top-level-declaration:
    initialized-declaration
    function-definition

function-definition:
    declaration-specifiers$_{opt}$ *declarator*   *function-body*

declaration-specifiers:
    storage-class-specifier
    type-specifier
    declaration-specifiers   *storage-class-specifier*
    declaration-specifiers   *type-specifier*

function-body:
    parameter-declaration-list$_{opt}$ *compound-statement*

compound-statement:
    { *inner-declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

inner-declaration-list:
    initialized-declaration-list

statement-list:
    statement
    statement-list   *statement*

statement:
    compound-statement
    ...

parameter-declaration-list:
    declaration-list

declaration-list:
    declaration
    declaration-list   *declaration*

declaration:
    declaration-specifiers   *declarator-list* ;

initialized-declaration:
    declaration-specifiers   *initialized-declarator-list* ;

initialized-declarator-list:
    initialized-declarator
    initialized-declarator-list   , *initialized-declarator*

initialized-declarator:
    declarator   *initializer-part*$_{opt}$

declarator:
    identifier
    ...

initializer-part:
    '=' *initializer*

initializer:
    expression
    ...

As you can see, all declarations except function definitions share the same syntax. In fact, semantic rules will prohibit certain syntactically valid declarations, depending on the location and form of the declaration. These rules will be considered later.

4.2. C-Ref: Terminology

In order to describe the meaning of declarations in a C program, we must establish some terminology.

4.2.1. C-Ref: Scope

The *scope* of a declaration is the region of the C program text over which that declaration is active. A declaration might have as its scope a single compound statement, a function body, or a larger section of the source program.

In C, identifiers may have one of five scopes:

- An identifier declared in a top-level declaration has a scope that extends from its declaration point (section "C-Ref: Forward References") to the end of the source program file.
- An identifier declared in a formal parameter declaration has a scope that extends from its declaration point to the end of the function body.
- An identifier declared at the beginning of a block has a scope that extends from its declaration point to the end of the block.
- A statement label has a scope that encompasses the entire function body in which it appears.
- A preprocessor macro name has a scope that extends from the `#define` command that declares it through the end of the source program file, or until the first `#undef` command that cancels its definition.

Nonpreprocessor identifiers declared within a function or block are often said to have *local scope*. The scope of every identifier is limited to the C source file in which it occurs. However, some identifiers can be declared to be *external*, in which case the declarations of the same identifier in two or more files can be linked in a fashion described in section "C-Ref: External Names". Draft Proposed ANSI C introduces a new kind of scope associated with function prototype declarations.

4.2.2. C-Ref: Visibility

A declaration of an identifier is *visible* in some context if a use of the identifier in that context will be bound to the declaration; that is, the identifier will have an association made with that declaration. A declaration might be visible throughout its scope, but it may also be *hidden* by other declarations whose scope and visibility overlap that of the first declaration. For example, in the following program, the declaration of `foo` as an integer variable is hidden by the inner declaration of `foo` as a floating-point variable. The outer `foo` is hidden only within the body of function `main`.

```
int foo = 10;   /* foo defined at the top level */

main
{
    float foo;  /* this foo hides the outer foo */
    ... sin(foo) ...
}
```

In C, formal parameter declarations can hide top-level declarations, and declarations at the beginning of a block can hide declarations outside the block. For one declaration to hide another, the declared identifiers must be the same, must belong to the same *overloading class*, and must be declared in two distinct scopes, one of which contains the other.

4.2.3. C-Ref: Forward References

Except in a few selected situations, an identifier may not be used before it is declared. To be precise, we define the *declaration point* of an identifier to be the position of the identifier's lexical token in the declaration. Any use of the identifier after the declaration point is permitted. In the example below, an integer, `intsize`, can be initialized to its own size because the use of `intsize` in the initializer comes after the declaration point.

```
static int intsize = sizeof(intsize);
```

When an identifier is used before its declaration point, a *forward reference* to the declaration is said to occur. C permits forward references in two situations. First, a statement label may appear in a `goto` statement before it is defined:

```
    if (error) goto recover;
    ...
  recover:
    CloseFiles();
    ...
```

Second, a structure, union, or enumeration tag may be used before it is declared. That situation is discussed in section "C-Ref: Structure Type References".

Illegal forward references are illustrated in the following example of an attempt to define a self-referential structure with a typedef declaration. In this case, the last occurrence of `cell` on the line is the declaration point, and therefore the use of `cell` within the structure is illegal.

```
typedef struct { int Value; cell *Next; } cell;
```

Closely related to the idea of forward references are *implicit declarations* and *duplicate declarations*.

4.2.4. C-Ref: Overloading of Names

In C and other programming languages, the same identifier may be associated with more than one program entity at a time. When this happens, we say that the name is *overloaded*, and the context in which the name is used determines the association that is in effect. For instance, an identifier might be both the name of a variable and a structure tag. When used in an expression, the variable association is used; when used in a type specifier, the tag association is used.

When a name is overloaded with several associations, each association has its own scope and may be hidden by other declarations independent of other associations. For instance, if an identifier is being used both as a variable and a structure tag, an inner block may redefine the variable association without altering the tag association.

There are five *overloading classes* for names in C. (We sometimes refer to them as *name spaces*.)

1. *Preprocessor macro names*. Because preprocessing logically occurs before compilation, names used by the preprocessor are independent of any other names in a C program.

2. *Statement labels.* Named statement labels are part of statements. Definitions of statement labels are always followed by `:` (and are not part of `case` labels). Uses of statement labels always immediately follow the reserved word `goto`.

3. *Structure, union, and enumeration tags.* These tags are part of structure, union, and enumeration type specifiers and, if present, always immediately follow the reserved words `struct`, `union`, or `enum`.

4. *Component names.* Component names are allocated in name spaces associated with each structure and union type. That is, the same identifier can be a component name in any number of structures or unions at the same time. Definitions of component names always occur within structure or union type specifiers. Uses of component names always immediately follow the selection operators `.` and `->`.

5. *Other names.* All other names fall into an overloading class that includes variables, functions, typedef names, and enumeration constants.

These rules differ slightly from those in the original definition of C. First, the original definition of C put statement labels in the same name space as ordinary identifiers, and enough compilers still follow this rule that the programmer should be aware of it. The problem is that using a single name space can be a source of great confusion, since labels do not obey normal block structure. For instance, in the following example, does the integer declaration of `L` hide the label, or is it an illegal duplicate definition of `L`?

```
{   ...
    goto L;
    ...
    {   int L;
        ...
        {   ...
            L = 10;
            ...
          L:
            ...
        }
    }
}
```

Compilers placing labels in the same name space as variables consider it an illegal duplicate definition.

Second, the original definition of C allocated all structure and union component names from a single name space instead of separate name spaces for each type. Thus, if `x` was a component of one structure type, it couldn't be the member of another structure type. (Actually, there were complicated rules that allowed identifiers to be in more than one structure if their offsets in the structure were identical.) Fortunately, this is now rarely seen in C compilers and has been corrected in the current language definitions.

Finally, the inclusion of structure, union, and enumeration type tags in the same overloading class is according to the current language definition, although the syntax of C is such that they could be in separate name spaces (and some compilers do define them that way).

4.2.5. C-Ref: Duplicate Declarations

It is illegal to make two declarations of the same name (in the same overloading class) in the same block or at the top level. Such declarations are said to *conflict*. In the following example, the two declarations of `howmany` are conflicting but the two declarations of `str` are not (because they are in different name spaces).

```
extern  int    howmany;
extern  char   str[10];
typedef double howmany;
extern  struct str {int a, b;} x;
```

There are two exceptions to the prohibition against duplicate declarations. First, any number of external (*referencing*) declarations for the same name may exist, as long as the declarations assign the same type to the name in each instance. This exception reflects the belief that declaring the same external library function twice should not be illegal.

Second, if an identifier is declared as being external, that declaration may be followed with a *definition* (section "C-Ref: External Names") of the name later in the program, assuming that the definition assigns the same type to the name as the external declaration(s). This exception allows the user to generate legal forward references to variables and functions. For instance, in the following example we define two functions, `f` and `g`, that reference each other. Normally, the use of `f` within `g` would be an illegal forward reference. However, by preceding the definition of `g` with an external declaration of `f`, we give the compiler enough information about `f` to compile `g`. (Without the initial declaration of `f`, a one-pass compiler could not know when compiling `g` that `f` returns a value of type `double` rather than `int`.)

```
extern double f();

double g(x, y)
  double x, y;
{
    ... f(x-y) ...
}

double f(z)
  double z;
{
    ... g(z, z/2.0) ...
}
```

There is a deficiency in this mechanism. Variables that are the subject of forward references from within the same file must be declared `extern`, when they could

otherwise have storage class `static`. Some compilers, when they see an external declaration followed by a static definition of the same name, will guess (correctly or not) what is going on and not generate an external reference at link time.

4.2.6. C-Ref: Duplicate Visibility

Because C's scoping rules specify that a name's scope begins at its declaration point rather than at the head of the block in which it is defined, a situation can arise in which two nonconflicting declarations can be referenced in different parts of the same block.

In the example below there are two variables named `i` referenced in the block labeled `B`—the integer `i` declared in the outer block is used to initialize the variable `j`, and then a floating-point variable `i` is declared, hiding the first `i`.

```
{
    int i = 0;
    ...
  B:{
        int j = i;
        float i = 10.0;
        ...
    }
}
```

The reference to `i` in the initialization of `j` is ambiguous. Which `i` was wanted? Most compilers will do what was (apparently) intended; the first use of `i` in block `B` is bound to the outer definition and the redefinition of `i` then hides the outer definition for the remainder of the block. We consider this usage to be bad programming style; it should be avoided.

4.2.7. C-Ref: Extent

Variables and functions, unlike types, have an existence at run time; that is, they have storage allocated to them. The *extent* of these objects is the period of time that the storage is allocated.

static extent when it is allocated storage at or before the beginning of program execution and the storage remains allocated until program termination. In C, all functions have static extent, as do all variables declared in top-level declarations. Variables declared at the beginning of blocks may have static extent, depending on the declaration.

An object is said to have *local extent* when (in the case of C) it is created upon entry to a block or function and is destroyed upon exit from the block or function. If a variable with local extent has an initializer, the variable is initialized each time it is created. Formal parameters have local extent, and variables declared at the beginning of blocks may have local extent, depending on the declaration. A variable with local extent is often called *automatic* in C.

Finally, it is possible in C to have data objects with *dynamic extent*; that is, objects that are created and destroyed explicitly at the programmer's whim. However, dy-

namic objects must be created through the use of special library routines such as `malloc` and are not viewed as part of the C language itself.

4.2.8. C-Ref: Initial Values

Allocating storage for a variable does not necessarily establish the initial contents of that storage. Most variable declarations in C may have *initializers*, expressions used to set the initial value of a variable at the time that storage is allocated for it. If an initializer is not specified for a variable, its value after allocation is unpredictable.

It is important to remember that a static variable is initialized only once and that it retains its value even when the program is executing outside its scope. In the following example, two variables, `L` and `S`, are declared at the head of a block and both are initialized to 0. Both variables have local scope, but `S` has static extent while `L` has local (automatic) extent. Each time the block is entered, both variables are incremented by one and the new values printed.

```
{
   static int S = 0;
   auto   int L = 0;
   L = L + 1;
   S = S + 1;
   printf("L = %d, S = %d\n", L, S);
}
```

What values will be printed? If the block is executed many times, the output will be this:

```
L = 1, S = 1
L = 1, S = 2
L = 1, S = 3
L = 1, S = 4
...
```

There is one dangerous feature of C's initialization of automatic variables declared at the beginning of blocks. The initialization is guaranteed to occur *only* if the block is entered normally; that is, if control flows into the beginning of the block. Through the use of statement labels and the `goto` statement, it is possible to jump into the middle of a block; if this is done, there is no guarantee that automatic variables will be initialized. The same is true when `case` or `default` labels are used in conjunction with the `switch` statement to cause control to be transferred into a block. In the following example, for instance, the initialization of variable `sum` will not occur when the `goto` statement transfers control to label `L`, causing erroneous behavior.

```
goto L;
...
{
    static int vector[10] = {1,2,3,4,5,6,7,8,9,10};
    int sum = 0;
  L:
    /* Add up elements of "vector". */
    for ( i=1; i<10; i++ ) sum += vector[i];
    printf("sum is %d", sum);
}
```

4.2.9. C-Ref: External Names

A special case of scope and visibility is the *external* variable or function. An external object is treated just like a static object in the file containing its declaration. However, an identifier declared to be external is *exported* to the linker, and if the same identifier is similarly declared in another program file, the linker will ensure that the two files reference the same object (variable or function).

We have discovered that many C compilers violate normal scope and visibility rules when processing external declarations. In this program fragment, for instance, the occurrence of `E` in the second assignment statement should be illegal (undefined), since the scope of the external declaration should not extend beyond the inner block:

```
{
    {
        extern E;
        E = 0;
    }
    E = 1;
}
```

In practice, however, it seems that many C compilers treat the declaration of `E` as if it had occurred at the top level, thus extending over the second assignment. Draft Proposed ANSI C states that external declarations should obey normal scoping rules.

4.2.10. C-Ref: Compile-time Objects

So far the discussion has focused mainly on variables and functions, which have an existence at run time. However, the scope and visibility rules apply equally to identifiers associated with objects that do not necessarily exist at run time: typedef names, type tags, structure and union component names, and enumeration constants. When any of these identifiers are declared, their scope is the same as that of a variable defined at the same location.

4.3. C-Ref: Storage Class Specifiers

We now proceed to examine the pieces of declarations: storage class specifiers, type specifiers, declarators, and initializers.

The storage class specifier in a declaration mainly determines the extent of the object declared. At most one storage class specifier may appear in a declaration. (Although the syntax in section "C-Ref: Organization of Declarations" indicates that storage class specifiers must precede type specifiers, and we think that is good style, the original definition of C allows them to occur in any order.)

storage-class-specifier: one of
`auto extern register static typedef`

The meanings of the storage classes are given below. Note that not all storage classes are permitted in every declaration context.

`auto`
: This storage class specifier is permitted only in declarations of variables at the heads of blocks. It indicates that the variable has local (automatic) extent. (Because this is the default, `auto` is rarely seen in C programs.)

`extern`
: This storage class specifier may appear in declarations of external functions and variables, either at the top level or at the heads of blocks. It indicates that the object declared has static extent and its name is known to the linker. Section "C-Ref: External Names" discusses how to distinguish defining external declarations from referencing external declarations.

`register`
: This storage class specifier may be used for local variables or parameter declarations. It has the same meaning as `auto`, except that it also provides a hint to the compiler that the local variable (or parameter) will be heavily used and should be allocated in a way that minimizes access time. (For instance, it might be allocated to a machine register.)

`static`
: This storage class specifier may appear on declarations of functions or variables. On function definitions, it is used only to specify that the function name is *not* to be exported to the linker. On function declarations, it indicates that the declared function will be defined—with storage class `static`—later in the file. On data declarations, it always signifies a defining declaration that is not exported to the linker. Variables declared with this storage class have static extent (as opposed to local extent, signified by `auto`).

`typedef`
: When this "storage class" appears, it indicates that the declaration is defining a new data type rather than a variable or function. The name of the new data type appears where a variable name would appear in a variable declaration, and the new data type is the type that would have been assigned to the variable name. (See section "C-Ref: Typedef Names".)

The register storage class has some additional restrictions. Only variables of certain types may have this storage class, and the set of permitted types may vary among different computers and C compilers, with type int always permitted. The compiler is permitted to limit the number of register variables in a function. When this limit is reached, further register variables are treated as auto variables. The programmer is advised to stick to one or two such variables. Finally, variables declared register may not have the address operator, &, applied to them. An attempt to do so may elicit an error or may simply cause register to be ignored.

4.3.1. C-Ref: Default Storage Class Specifiers

If no storage class specifier is supplied on a declaration, one will be assumed on the basis of the declaration context:

1. Top-level declarations (and function definitions) are assumed to have storage class extern. (However, extern that is *assumed* and extern that is *stated* can mean different things. See section "C-Ref: External Names".)

2. Parameter declarations do not take any storage class except register. Omitting the storage class means only "not register."

3. For declarations at the head of blocks, extern is assumed for functions and auto is assumed for everything else.

In spite of these rules, it is a good programming practice to supply the storage class extern explicitly when it applies and not allow it to default. On the other hand, omitting auto in variable declarations is a common practice and is considered good style.

4.3.2. C-Ref: Examples of Storage Class Specifiers

The following code implements the heap sort algorithm for sorting the contents of an array. It is beyond the scope of this book to explain how it works. We remark only that the algorithm regards the array as a binary tree such that the two subnodes of element b[k] are elements b[2*k] and b[2*k+1], and that a *heap* is a tree such that every node contains a number that is no smaller than any of the numbers contained by that node's descendants. We exhibit the code here as a practical example of the use of storage class specifiers.

```
/* Heap sort. */

#define SWAP(x, y) (temp = (x), (x) = (y), (y) = temp)
```

```
/* If v[m+1] through v[n] is already in heap form,
   this puts v[m] through v[n] into heap form. */
static void adjust(v, m, n)
    int v[], m;
    register int n;
{
    register int *b, j, k, temp;
    /* Array in C are 0-origin, but heapsort is
       more easily coded and understood in terms
       of 1-origin arrays.  The variable "b"
       effectively remaps the array "v" to be
       1-origin: v[j] is the same as b[j-1]. */
    b = v - 1;
    j = m;
    k = m * 2;
    while (k <= n) {
        if (k < n && b[k] < b[k+1]) ++k;
        if (b[j] < b[k]) SWAP(b[j], b[k]);
        j = k;
        k *= 2;
    }
}

/* Sort v[0]...v[n-1] into increasing order. */
void heapsort(v, n)
    int v[], n;
{
    int *b, j, temp;
    b = v - 1;
    /* Put the array into the form of a heap. */
    for (j = n/2; j > 0; j--)
        adjust(v, j, n);
    /* Repeatedly extract the largest element and
       put it at the end of the unsorted region. */
    for (j = n-1; j > 0; j--) {
        SWAP(b[1], b[j+1]);
        adjust(v, 1, j);
    }
}
```

The main function is `heapsort`; it must be visible to users of the sort package, and so it has the default storage class, namely `extern`. The auxiliary function `adjust` does not need to be externally visible, and so it is declared to be `static`. The speed of the `adjust` function is crucial to the performance of the sort, and so its local variables have been given storage class `register` as a hint to the compiler. The formal parameter `n` is also referred to repeatedly within `adjust`, and so it is also specified with storage class `register`. The other two formal parameters for `adjust`

are referred to only once and are defaulted to "not `register`." The local variables of function `heapsort` are not so important to performance as those in `adjust`; they have been given the default storage class, namely `auto`.

4.4. C-Ref: Type Specifiers

A type specifier provides some of the information about the data type of the program entity being declared. (We say "some" because the declarators in a declaration provide additional type information.) The type specifier may also declare (as a side effect) type tags, structure and union component names, and enumeration constants. Although type specifiers and storage class specifiers can appear in any order in a declaration, it is customary to put the storage class specifiers (if any) first.

type-specifier:
  enumeration-type-specifier
  floating-point-type-specifier
  integer-type-specifier
  structure-type-specifier
  typedef-name
  union-type-specifier
  void-type-specifier

Examples of type specifiers include:

```
void                            union { int a; char b; }
int                             enum {red, blue, green}
unsigned long int               char
my struct type                  float
```

The type specifiers are described in detail in chapter "C-Ref: Types", and we will defer further discussion of particular type specifiers until then. However, there are a few general issues surrounding type specifiers that will be dealt with here.

Draft Proposed ANSI C introduces new type specifiers.

4.4.1. C-Ref: Default Type Specifiers

C allows the type specifier in a variable declaration or function definition to be omitted, in which case it defaults to `int`. One often sees this in function definitions:

```
/* Sort v[0]...v[n-1] into increasing order. */
sort(v, n)
  int v[], n;
{
    ...
}
```

This is bad programming style in modern C. Older compilers did not implement

the `void` type, so a rationale behind omitting the type specifier on function definitions was to indicate to human readers that the function didn't really return a value (although the compiler had to assume that it did). The modern style is to declare those functions with the `void` type:

```
/* Sort v[0]...v[n-1] into increasing order. */
void sort(v, n)
  int v[], n;
{
    ...
}
```

When using a compiler that doesn't implement `void`, it is much nicer to define `void` yourself and then use it explicitly than to omit the type specifier entirely.

```
/* Make "void" be a synonym for "int". */
typedef int void;
```

At least one compiler we know actually reserves the identifier `void` but doesn't implement it. For that compiler, the preprocessor definition

```
#define void int
```

is one of the few cases in which using a reserved word as a macro name is justified.

The C syntax requires declarations to contain either a storage class specifier, a type specifier, or both. This requirement avoids a syntactic ambiguity in the language. If all specifiers were defaulted, the declaration

```
extern int f();
```

would become simply

```
f();
```

which is syntactically equivalent to a statement consisting of a function call. We think that the best style is to always include the type specifier and to allow the storage class specifier to default, at least when it is `auto`.

A final note for LALR(1) grammar aficionados: Both the storage class specifier and the type specifier can be omitted on a function definition, and this is very common in C programs, as in

```
main()
{
    ...
}
```

There is no syntactic ambiguity in this case, because the declarator in a function declaration must be followed by a comma or semicolon, whereas the declarator in a function definition must be followed by a left brace.

4.4.2. C-Ref: Missing Declarators

The following discussion deals with a subtle point of declarations and type specifiers. Type specifiers that are structure, union, or enumeration definitions have a side effect of defining new types. For example, the type specifier

```
struct S { int a, b; }
```

defines a new structure type `S` with components `a` and `b`. The type can be referenced later by using just the specifier

```
struct S
```

When using these specifiers, it makes sense to omit all the declarators from the declaration, so that the whole declaration consists of just a type definition:

```
struct S { int a, b; };
```

The C grammar permits this, and so do all C compilers. However, the grammar also permits some nonsensical variations on this declaration. Most C compilers will not notice these variations, although they are clearly programming errors.

The first variation is omitting the type tag, as in

```
struct { int a, b; };
```

This is clearly nonsensical because without a tag it is impossible to refer to the type later in the program.

The second variation is including a storage class specifier, which will be ignored:

```
static struct S { int a, b; };
```

This may mislead the programmer into thinking that a later declaration of the form

```
struct S x,y;
```

will cause `x` and `y` to have the storage class `static`. It won't.

The final variation is using a type specifier that has no side effects:

```
int ;
```

4.5. C-Ref: Declarators

Declarators introduce the name being declared and also supply additional type information. No other programming language has anything quite like C's declarators.

declarator :
simple-declarator
(declarator)
function-declarator
array-declarator
pointer-declarator

The different kinds of declarators are described below.

4.5.1. C-Ref: Simple Declarators

Simple declarators are used to define variables of arithmetic, enumeration, structure, and union types.

simple-declarator :
    identifier

Suppose that *S* is a type specifier and that *id* is any identifier. Then the declaration

    S id ;

indicates that *id* is of type *S*. The *id* is called a *simple declarator*. For example:

```
int i;              /* i is an integer variable */
float velocity;
          /* velocity is a floating-point variable */
struct S { int a; float b; } a and b;
      /* a and b is a structure of two components */
```

Simple declarators may be used in a declaration when the type specifier supplies all the typing information. This happens for arithmetic, structure, union, enumeration, and void types, and for types represented by `typedef` names. Pointer, array, and function types require the use of more complicated declarators. However, every declarator has in its "middle" an identifier, and we thus say that a declarator "encloses" an identifier.

4.5.2. C-Ref: Pointer Declarators

Pointer declarators are used to declare variables of pointer types.

pointer-declarator :
    * *declarator*

Suppose that *D* is any declarator enclosing the identifier *id*, and that the declaration "*S D*;" indicates that *id* has type "... *S*." Then the declaration

    S **D* ;

indicates that *id* has type "...pointer to *S*." For example, in the following three declarations of `x`, *id* is `x`, *S* is `int`, and "..." is, respectively, "", "array of," and "function returning."

```
int *x;        /* x is a pointer to an integer */
int *x[];      /* x is an array of pointers
                  to integers */
int *x();      /* x is a function returning a
                  pointer to an integer */
```

It's harder to explain than it is to learn.

Draft Proposed ANSI C extends pointer declarators to allow the specification of pointers to "constant" or "volatile" data.

4.5.3. C-Ref: Array Declarators

Array declarators are used to declare objects of array types.

array-declarator :
    declarator [constant-expression $_{opt}$*]*

constant-expression :
    expression

If D is any declarator enclosing the identifier id, and if the declaration "S D;" indicates that id has type "... S," then the declaration

    S (D)[e] ;

indicates that id has type "... array of S." For example, in the following two declarations, id is x, S is int, and "..." is, respectively, "", "pointer to," and "array of."

```
int (x)[];       /* x is an array of integers */
int (*x)[];      /* x is a pointer to an array of
                    integers */
int (x[])[];     /* x is an array of arrays of
                    integers */
```

(The parentheses may often be elided according to the precedence rules in constructing declarators; see section "C-Ref: Composition of Declarators".)

The integer constant expression e, if present, specifies the number of elements in the array. C's arrays are always "0-origin"; that is, the array

```
int A[3];
```

consists of the elements A[0], A[1], and A[2]. The number of elements in an array must be greater than 0, although some popular C compilers do not check this.

As in the example above, higher-dimensioned arrays are declared as "arrays of arrays." For example:

```
int judges scores[10][2];
int checker board[8][8];
```

The length of the array, a constant expression, may be omitted as long as it is not needed to allocate storage. It is not needed when:

1. The object being declared is a formal parameter of a function.

2. The declarator is accompanied by an initializer from which the length of the array can be deduced.

3. The declaration is not a defining occurrence; that is, it is an external declaration that refers to an object defined elsewhere.

An exception to these cases is that the declaration of any n-dimensional array must include the sizes of the last n-1 dimensions so that the accessing algorithm can be determined. For example:

```
static int vector[5];    /* defining occurrence */
char prompt[]="Yes or No?";
                         /* can deduce size */
extern matrix[][10];
                         /* external, but last
                            dimension must be
                            supplied */
```

For more information, see section "C-Ref: Array Types".

4.5.4. C-Ref: Function Declarators

Function declarators are used to declare objects of function types.

function-declarator :
    declarator (parameter-list$_{opt}$*)*

parameter-list :
    identifier
    parameter-list, identifier

If *D* is any declarator enclosing the identifier *id*, and if the declaration "*S D*;" indicates that *id* has type "... *S*," then the declaration

S (*D*) ();

indicates that *id* has type "... function returning *S*." For example, in the following declarations of x, *id* is x, *S* is int, and "..." is, respectively, "", "pointer to," and "array of pointers to."

```
int (x)();     /* x is a function returning
                  an integer */
int (*x)();    /* x is a pointer to a function
                  returning an integer */
int (*x[])();  /* x is an array of pointers to
                  functions returning integers */
```

The parentheses around x in the first declaration may be elided according to the precedence rules in constructing declarators; see section "C-Ref: Composition of Declarators".

The syntax for a function declarator includes an optional parameter list that may be supplied between the open and close parentheses of the declarator. This parameter list is supplied only when defining a function:

```
void f(x, y)
  int x, y;
{
    ...
}
```

The parameter list is omitted in all other cases:

```
extern void f();      /* f is an external function
                         reference */
static void (*f)();   /* f is a pointer, not
                         a function */
```

Draft Proposed ANSI C extends function declarators to provide "function prototypes" in which the argument types are listed explicitly.

4.5.5. C-Ref: Composition of Declarators

Declarators can be composed to form more complicated types, such as "5-element array of pointers to functions returning `int`," which is the type of `ary` in this declaration:

```
int (*ary[5])();
```

The only restriction on declarators is that the resulting type must be a legal one in C. The only types that are *not* legal in C are:

1. Any type involving `void` except "...function returning `void`." (The type specifier `void` is discussed in section "C-Ref: Void".)

2. "Array of function of ...." Arrays may contain pointers to functions, but not functions themselves.

3. "Function returning array of ...." Functions may return pointers to arrays, but not arrays themselves.

4. "Function returning function of ...." Functions may return pointers to other functions, but not the functions themselves.

Draft Proposed ANSI C also allows "pointer to `void`".

When composing declarators, the precedence of the declarator expressions is important. Function and array declarators have higher precedence than pointer declarators, so that "*x()" is equivalent to "*(x())" ("function returning pointer ...") instead of "(*x)()" ("pointer to function returning ..."). Parentheses may be used to group declarators properly.

Here are some sample declarations with the associated types of the enclosed identifiers.

```
int *Sum1();     /* Sum1 is a function returning a
                    pointer to an integer. */

int (*Sum2)();   /* Sum2 is a pointer to a function
                    returning an integer. */

void (*F)();     /* F is a pointer to a function
                    returning no result. */
```

```
void *F();      /* ILLEGAL!  Can't have a pointer
                   to void. */
```

Although declarators can be arbitrarily complex, it is better programming style to factor them into several simpler definitions. That is, rather than writing

```
int *(*(*(*x)())[10])();
```

write instead

```
typedef int *(*print function ptr)();
typedef print function ptr (*digit routines)[10];
digit routines (*x)();
```

(The variable `x` is a pointer to a function returning a pointer to a 10-element array of pointers to functions returning pointers to integers, in case you wondered.)

The rationale behind the syntax of declarators is that they mimic the syntax of a use of the enclosed identifier. For instance, if you see the declaration

```
int *(*x)[4];
```

then the type of the expression

```
*(*x)[i]
```

is `int`.

4.6. C-Ref: Initializers

The declaration of a variable may be accompanied by an initializer that specifies the value the variable should have at the beginning of its lifetime. The full syntax for initializers is

initializer:
  expression
  { *initializer-list* ,$_{opt}$ }

initializer-list:
  initializer
  initializer-list , *initializer*

The optional trailing comma inside the braces does not affect the meaning of the initializer.

The initializers permitted on a particular declaration depend on the type and storage class of the variable to be initialized and on whether the declaration appears at the top level or at the head of a block. In general, the initializer for any variable with static extent must be a constant expression; such initialization happens prior to execution of the C program. This applies to all top-level variable declarations (static and external) and to variable declarations at the heads of blocks that have storage class `static`. Any static variable that does not have an explicit initializer will be initialized to zero.

Initializers for automatic variables may be arbitrary expressions. These variables will always be declared at the heads of blocks, and the compiler will emit code to evaluate the initializer expression and assign the result to the variable upon block entry.

Declarations of formal parameters may not have initializers. The "shape" of an initializer—the brace-enclosed lists of initializers—should match the structure of the variable being initialized. However, there are special rules for abbreviating initializers. The following sections explain the special requirements for each type of variable.

4.6.1. C-Ref: Integers

The form of an initializer for an integer variable is

declarator = *expression*

Any constant expression of integral type may be used to initialize an external or static integer variable. Any expression of arithmetic type (not necessarily constant) may be used to initialize an integer variable with storage class `auto` or `register`. In all cases the usual assignment conversions are applied. For example,

```
static int Count = 4*200;
extern int getchar();

main()
{
    int ch = getchar();
    ...
}
```

The original definition of C specified that the initializer for an integer variable may optionally be surrounded by braces, although such braces are logically unnecessary. We recommend that braces not be used in this situation, but be reserved to indicate aggregate initialization.

4.6.2. C-Ref: Floating-point

The form of an initializer for a floating-point variable is

declarator = *expression*

All C compilers allow the initialization of static or external floating-point variables; the type of the initializer should be floating-point, but some compilers will permit expressions of any arithmetic type. Automatic variables of floating-point types can be initialized with any expression of arithmetic type. The usual assignment conversions are applied in initializing the variable. For example:

```
static void process data(K)
  double K;
{
    static double epsilon = 1.0e-6;
    auto float fudge factor = K*epsilon;
    ...
}
```

The original definition of C specified that the initializer for a floating-point variable may optionally be surrounded by braces, although such braces are logically unnecessary. We recommend that braces not be used in this situation, but be reserved to indicate aggregate initialization.

C compilers generally shy away from performing compile-time floating-point arithmetic, so initializers for static and external floating-point variables should be restricted to floating-point constants, perhaps with a preceding unary minus operator.

4.6.3. C-Ref: Pointers

The form of an initialization of a pointer variable is

declarator = *expression*

Any constant expression of type *PT* ("pointer to *T*") may be used to initialize an external or static variable of type *PT*. Any expression of type *PT* may be used to initialize a local variable of type *PT*.

Constant expressions used as initializers of pointer type *P* may be formed from the following elements.

1. The integer constant `0` yields a null pointer of any type; it is usually referred to by the name `NULL`.

   ```
   #define NULL 0
   double *dp = NULL;
   ```

2. The name of a static or external function of type "function returning *T*" is converted to a constant of type "pointer to function returning *T*."

   ```
   extern int f;
   static int (*fp)() = f;
   ```

3. The name of a static or external array of type "array of *T*" is converted to a constant of type "pointer to *T*."

   ```
   char ary[100];
   char *cp = ary;
   ```

4. The `& operator` applied to the name of a static or external variable of type *T* yields a constant of type "pointer to *T*."

   ```
   static short s;
   auto short *sp = &s;
   ```

5. The `&` operator applied to an external or static array of type "array of *T*," subscripted by a constant expression, yields a constant of type "pointer to *T*."

   ```
   float PowersOfPi[10];
   float *PiSquared = &PowersOfPi[2];
   ```

6. An integer constant cast to a pointer type yields a constant of that pointer type, although this is not portable.

   ```
   long *PSW = (long *) 0xFFFFFFF0;
   ```

   Not all compilers accept casts in constant expressions.

7. A string literal yields a constant of type "pointer to `char`" when it appears as the initializer of a variable of pointer type.

   ```
   char *greeting = "Type <cr> to begin ";
   ```

8. The sum or difference of any expression shown for cases 3 through 7 above and an integer constant expression.

   ```
   static short s;
   auto short *sp = &s + 3, *msp = &s - 3;
   ```

In general, the initializer for a pointer type must evaluate to an integer or to an address plus (or minus) an integer constant. This limitation reflects the capabilities of most linkers.

The original definition of C specified that the initializer for a pointer variable may optionally be surrounded by braces, although such braces are logically unnecessary. We recommend that braces not be used in this situation, but be reserved to indicate aggregate initialization.

4.6.4. C-Ref: Arrays

If I_j (for j = 0, 1, ... , n-1) are each initializers for type T, then

$\{ I_0, I_1, \ldots, I_{n-1} \}$

is an initializer for type "n-element array of T." The initializer I_j is used to initialize element j the array (zero origin). For example:

```
int ary[4] = { 0, 1, 2, 3 };
```

Multidimensional arrays follow the same pattern, with initializers listed by row. (The last subscript varies most rapidly in C.)

```
int ary[4][2][3] =
      { { { 0,  1,  2}, { 3,  4,  5} },
        { { 6,  7,  8}, { 9, 10, 11} },
        { {12, 13, 14}, {15, 16, 17} },
        { {18, 19, 20}, {21, 22, 23} } };
```

Arrays of structures may be initialized analogously:

```
struct {int a; float b;} a[3] = { {1,  2.5},
                                  {2,  3.9},
                                  {0, -4.0} };
```

Static and external arrays may always be initialized in this way. The original definition of C stated that automatic arrays could not be initialized, but some newer compilers are relaxing this restriction. (The compiler generates a sequence of assignments to initialize the elements of the automatic array at run time.) Draft Proposed ANSI C allows the initialization of automatic arrays, but only with constant expressions.

Array initialization has a number of special rules. First, the number of initializers may be less than the number of array elements, in which case the remaining elements are initialized to zero. That is, the initializations

```
int ary[5] = { 1, 2, 3 };
int mat[3][3] = { {1, 2}, {3} };
```

are equivalent to

```
int ary[5] = { 1, 2, 3, 0, 0 };
int mat[3][3] = { {1, 2, 0},
                  {3, 0, 0},
                  {0, 0, 0} };
```

If the number of initializers is greater than the number of elements, the initializer is in error.

Second, the bounds of the array need not be specified, in which case the bounds are derived from the shape of the initializer. For example,

```
int squares[] = { 0, 1, 4, 9 };
```

is the same as

```
int squares[4] = { 0, 1, 4, 9 };
```

Finally, string literals may be also be used to initialize variables of type "array of `char`." In this case, the first element of the array is initialized by the first character in the string, and so forth. Space must be left for the terminating `'\0'`. Thus, the initializations

```
static char x[5] = "ABCD";
static char str[] = "ABCDEF";
```

are the same as

```
static char x[5] = { 'A', 'B', 'C', 'D', '\0' };
static char str[7] = { 'A','B','C','D','E','F','\0' };
```

Finally, a list of strings can be used to initialize an array of character pointers:

```
char *astr[] = { "John", "Bill", "Susan", "Mary" };
```

4.6.5. C-Ref: Enumerations

The form of initializers for variables of enumeration type E is

declarator = *expression*

where the expression is of the same enumeration type E. Some compilers will tolerate braces around the initialization expression, just as for integer initializers, but they are not necessary. We recommend that braces not be used in this situation, but be reserved to indicate aggregate initialization.

An initializer for a static or external variable of type E must be a constant expression of type E; that is, it must be an enumeration constant of type E. An initializer for an automatic or register variable of type E can be any expression of type E, which in practice means that it can be either an enumeration constant or an enumeration variable of the same type. For example:

```
static enum E { a, b, c } x = a;
auto enum E y = x;
```

Section "C-Ref: Enumeration Types" mentions that some compilers treat enumeration types as integer types. Those compilers will allow initializers for enumeration variables to be integer expressions as well as expressions of enumeration types. (More generally, an initializer should be legitimate in a given implementation if an equivalent assignment statement would be; the same conversions are performed.) However, we recommend adhering to the stricter rules as a matter of good style, and using explicit casts where conversions are needed.

4.6.6. C-Ref: Structures

If a structure type T has n components of types T_j (for j = 1, ... , n) and if I_j is an initializer for type T_j then

$\{ I_1, I_2, \ldots, I_n \}$

is an initializer for type T.

Static and external variables of structure types can be initialized, and the component initializers must be legal initializers for static or external variables of the component types. A few implementations of C are deficient in not supporting bit field initialization. Automatic and register variables of structure types cannot be initialized. (They can in Draft Proposed ANSI C, but brace-enclosed initializers must contain only constants.)

```
struct S {int a; char b[5]; double c; };
struct S x = { 1, "abcd", 45.0 };
```

As with array initializers, structure initializers have some special rules. In particular, if there are fewer initializers than there are structure components, the remaining components are initialized to zero. Thus, given the structure declaration

```
struct S1 {int a;
           struct S2 {double b;
                      char c; } b;
           int c[4]; };
```

the initialization

```
struct S1 x = { 1, {4.5} };
```

is the same as

```
struct S1 x = { 1,
                { 4.5, '\0' },
                { 0, 0, 0, 0 }
              };
```

If there are too many initializers for the structure, it is an error.

4.6.7. C-Ref: Unions

The C language does not permit initialization of any variables of union type, on the grounds that there is no obvious way in the language to specify which union component is being initialized.

Some compilers allow the initialization of union variables, treating them as if they were variables of the type of the first component. For instance,

```
enum Greek { alpha, beta, gamma };
union U {
    struct { enum Greek tag; int Size; } I;
    struct { enum Greek tag; float Size; } F;
    };
static union U x = { alpha, 42 };
```

The compilers that permit such initializations may restrict them to static and external variables.

Draft Proposed ANSI C permits union initialization as described. Automatic unions may also be initialized under the same rules as apply to automatic variables of the type of the first component.

The only other types are function types and `void`, neither of which can have initializers.

4.6.8. C-Ref: Eliding Braces

C permits braces to be dropped from initializer lists under certain circumstances, although it is usually clearer to retain them. The general rules are these:

1. If a variable of array or structure type is being initialized, the outermost pair of braces may not be dropped.

2. Otherwise, if an initializer list contains the correct number of elements for the object being initialized, the braces may be dropped.

The most common use of these rules is in dropping inner braces when initializing a multidimensional array:

```
int matrix[2][3] = { 1, 2, 3, 4, 5, 6 };
        /* same as { {1, 2, 3}, {4, 5, 6} } */
```

Many C compilers treat initializer lists very casually, permitting too many or too few braces. We advise keeping initializers simple and using braces to make their structure explicit.

4.7. C-Ref: Implicit Declarations

In C it is permitted to call an external function that has not been declared previously. If the compiler sees an identifier *id* followed by a left parenthesis, and if *id* has not been previously declared, then a declaration is implicitly entered at the top level. For example, consider this program fragment:

```
void process()
{
    ... f(i, j) ...
}
```

If `f` has not been declared, the compiler implicitly inserts a declaration

```
extern int f();
```

immediately before the `process` function definition.

Allowing functions to be declared in this way is hazardous to program portability. In particular, we've been bitten by the following sequence of events. A pointer-returning function, such as `malloc` (section "C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"), is allowed to be implicitly declared as

```
extern int malloc();
```

rather than the correct

```
extern char *malloc();
```

The program works fine because the compiler and computer being used happen to allocate the same size storage to the `int` type as to pointer types, and the compiler automatically converts between integer and pointer types. (This is normal in older C compilers.) One day the program is moved to another computer and compiler, under which pointers occupy four bytes and type `int` only two bytes. When the compiler sees

```
char *p;
...
p = malloc();
```

it generates code to zero-extend the (presumably two-byte) value returned by `malloc` to the four bytes required by the pointer. The compiler issues no warning, and only the low half of the address returned by `malloc` is assigned to `p`. All of a sudden the program doesn't work.

4.8. C-Ref: External Names

An important issue with external names is ensuring consistency among the declarations of the same external name in several files. For instance, what if two declarations of the same external variable specified different initializations? For this and other reasons, it is useful to distinguish a single *defining declaration* of an external name within a group of files. The other declarations of the same name are then considered *referencing declarations*; that is, they reference the defining declaration.

It is a well-known deficiency in C that defining and referencing occurrences of external variable declarations are difficult to distinguish. In general, compilers use one of four models to determine when a top level declaration is a defining occurrence.

4.8.1. C-Ref: The Initializer Model

The presence of an initializer on a top level declaration indicates a defining occurrence; all others are referencing occurrences. There must be a single defining occurrence among all the files in the C program.

4.8.2. C-Ref: The Omitted Storage Class Model

In this scheme, the storage class `extern` must be explicitly included on all referencing declarations and the storage class must be omitted from the single defining declaration for each external variable. The defining declaration can include an initializer but it is not required to do so. (It is illegal to have both an initializer and the storage class `extern` in a declaration.) This solution is probably the most common one, and the one adopted in Draft Proposed ANSI C.

4.8.3. C-Ref: The Common Model

This scheme is so named because it is related to the way multiple references to a FORTRAN COMMON block are merged into a single defining occurrence in some FORTRAN implementations. Both defining and referencing external declarations have storage class `extern`, whether explicitly or by default. Among all the declarations for each external name in all the object files linked together to make the program, only one may have an initializer. At link time, all external declarations for the same identifier (in all C object files) are combined and a single defining occurrence is conjured, not necessarily associated with any particular file. If any declaration specified an initializer, that initializer is used to initialize the data object. (If several declarations did, the results are unpredictable.)

This solution is the most painless for the programmer and the most demanding on system software.

4.8.4. C-Ref: Mixed Common Model

This model is a cross between the "omitted storage class" model and the "common" model. It is used in many versions of UNIX.

1. If `extern` is omitted, and an initializer is present, a definition for the symbol is emitted. Having two or more such definitions among all the files comprising a program results in an error at link time or before.

2. If `extern` is omitted, and no initializer is present, a "common" definition (a la FORTRAN) is emitted. Any number of common definitions of the same identifier may coexist.

3. If `extern` is present, the declaration is taken to be a reference to a name defined elsewhere. It is illegal for such a declaration to have an initializer. If the identifier so declared is never actually used, the compiler will not issue an external reference to the linker.

If no explicit initializer is provided for the external variable, the variable is initialized as if the initializer had been the integer constant 0.

4.8.5. C-Ref: Advice

To remain compatible with the largest number of compilers, we recommend following these rules:

1. Have a single definition point (source file) for each external variable; in the defining declaration, omit the `extern` storage class and include an explicit initializer:

   ```
   int errcnt = 0;
   ```

2. In each source file referencing an external variable defined in another module, use the storage class `extern` and do not supply an explicit initializer:

   ```
   extern int errcnt;
   ```

Independent of the defining/referencing distinction, an external name should always be declared with the same type in all files making up a program. The C compiler cannot verify that declarations in different files are consistent in this fashion, and the punishment for inconsistency is erroneous behavior at run time. The `lint` program, usually supplied with the C compiler in UNIX systems, can check multiple files for consistent declarations.

4.8.6. C-Ref: Unreferenced External Declarations

Although not required by the C language, it is customary for implementations to completely ignore declarations of external variables or functions that are never referenced. For example, if the declaration

```
extern double fft;
```

appears in a program, but the function `fft` is never called and its address is never taken, then no external linkage to the name `fft` is generated. Thus, if the function `fft` exists in some link-time library, it will not be loaded with the program, where it would take up space to no purpose.

5. C-Ref: Types

A *type* is a set of *values* and a set of *operations* on those values. For example, the values of an integer type consist of integers in some specified range, and the operations on those values consist of addition, subtraction, inequality tests, and so forth. The values of a floating-point type include numbers represented differently from integers, and a set of different operations: floating-point addition, subtraction, inequality tests, and so forth.

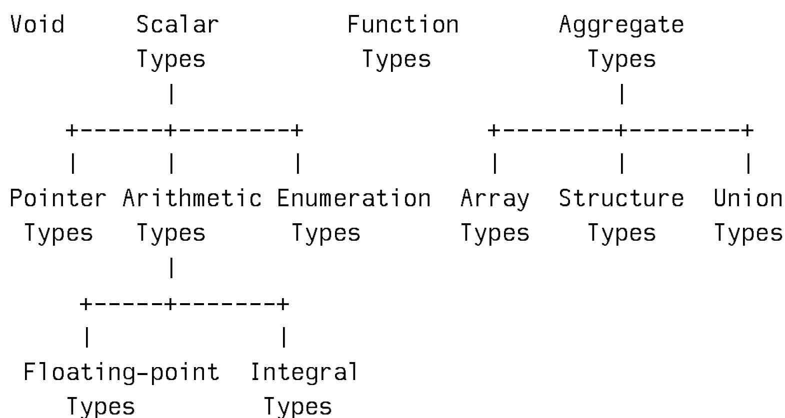
We say a variable or expression "has type *T*" when its values are constrained to the domain of *T*. The types of variables are established by the variable's declaration; the types of expressions are given by the definitions of the expression operators.

The C language provides a large selection of built-in types, including integers of several kinds, floating-point numbers, pointers, enumerations, arrays, structures, unions, and functions. There is also a special "type," `void`, which has no values; it is used to specify functions that return nothing.

It is useful to organize C's types into the categories shown in table "C-Ref: Type Categories". The *void* type has no values and no operations. The term *integral types* includes all forms of integers and characters. The term *arithmetic types* includes the integral and floating-point types. The term *scalar types* includes the arithmetic types, pointer types, and enumeration types. The *function types* are the types "function returning...." *Aggregate types* include arrays, structures, and unions.

All of C's types are discussed in this chapter. For each type, we indicate how objects of the type are declared, the range of values of the type, any restrictions on the size of the type, and what operations are defined on values of the type.

5.1. C-Ref: Type Categories



Draft Proposed ANSI C adds a new floating-point type to the language; otherwise the types are unchanged.

5.2. C-Ref: Integer Types

C provides a larger number of integer types and operators than do most programming languages. The variety reflects the different word lengths and kinds of arithmetic operators found on most computers, thus allowing a close correspondence between C programs and the underlying hardware. Integer types in C are used to represent:

1. signed or unsigned integer values, for which the usual arithmetic and relational operations are provided

2. bit vectors, with the operations AND, OR, XOR, and left and right shifts

3. boolean values, for which zero is considered "false" and all nonzero values are considered "true," with the integer 1 being the canonical "true" value

4. characters, which are represented by their integer encodings on the computer

It is convenient to divide the integer types into three classes: signed types, unsigned types, and characters. Each of these classes has a set of type specifiers that can be used to declare objects of the type.

integer-type-specifier :
    signed-type-specifier
    unsigned-type-specifier
    character-type-specifier

5.2.1. C-Ref: Signed Integer Types

C provides the programmer with three sizes of signed integer types, denoted by the type specifiers `short`, `int`, and `long` in nondecreasing order of size.

signed-type-specifier :
    `short int`*opt*
    `int`
    `long int`*opt*

The specifier `short int` is equivalent to `short` and the specifier `long int` is equivalent to `long`. Here are some examples of typical declarations of signed integers.

```
auto short i, j;
long int l;
static int k;
```

The type `int` may not be shorter than `short` and `long` may not be shorter than `int`. However, it is permitted in principle for `short` and `int` to be the same size or for `int` and `long` to be the same size. On a computer that cannot easily address memory smaller than a word, the implementor might even choose to use a single size for all three types.

Draft Proposed ANSI C allows the programmer to use the new type specifier `signed` to indicate explicitly a signed integer type.

The implementor's selection of representations for signed integer types is determined by what natural representations are provided by the underlying hardware, by what signed and unsigned arithmetic operators are provided, and so on. Many implementations represent characters in 8 bits, short integers in 16 bits, and long integers in 32 bits, with ordinary integers being either 16 or 32 bits wide, whichever leads to greater efficiency. These particular widths technically are not specified by the C language but have become traditional; many programmers expect characters to be 8 bits wide and all other integers to be at least 16 bits wide. Certainly there are many existing C programs that depend on this.

The C programmer must constantly make decisions as to which signed integer type to use for a given purpose. There are three considerations: the range of integer values required by the program, the amount of storage that may be consumed, and the speed of the program. The only way to judge these trade-offs is to check your compiler documentation, but the following general rules seem to apply for the larger computers and for all microprocessors except the small 8-bit ones.

1. The `short` type is likely to be represented in 16 or more bits. If this size is sufficient, `short` may be used for large integer arrays in order to save space. However, because C converts all `short` values to `int` in arithmetic expressions, in most cases it usually doesn't make much sense to use `short` for individual variables.

2. The `long` type is likely to be represented in at least 32 bits. It gives the largest range of signed integer values available. Operations on objects of type `long` are sometimes slower than operations on objects of type `int`, depending on the implementation.

3. The `int` type is traditionally the "standard" integer in C, and operations on it are likely to be efficient. However, it is also the type whose size is least predictable, being represented with 32 bits or more by some compilers and with only 16 bits by others. Because of this, the use of `int` is often a source of portability problems. A good rule of thumb is that `long` is the largest *supported* integer size, while `int` is the largest *efficient* integer size. If efficiency is much more important than portability, then the `int` type may be the better one to use.

The precise range of values representable by a signed integer type depends not only on the number of bits used in the representation but also on the encoding technique. The expectation is that on a computer using two's-complement arithmetic, a signed integer represented with n bits will have a range from -2^{n-1} through $2^{n-1}-1$. On a computer using one's-complement or sign-magnitude representations, however, the lower bound will be $-(2^{n-1}-1)$.

A useful technique for maximizing portability is to define and use your own integer types based on the range of integers needed by your application. Each of your types can then be defined in terms of one of the standard integer types depending

upon the particular computer being used. For example, suppose you are writing an inventory control program and have need of integers that will be part numbers, order quantities, and purchase-order numbers. Depending on the sizes of the numbers and the sizes of the integer types on the computer, you might want to use `short`, `int`, or `long` to represent part numbers. The solution is to use a single definition file, say `invdef.h`, to define your own integer types.

```
/* invdef.h
   Inventory definitions for the XXX computer.
*/
typedef short part number;
typedef int   order quantity;
typedef long  purchase order;
```

The source files that define processing functions would use the types defined by `invdef.h`:

```
#include "invdef.h"

purchase order back order(part, number)
  part number part;
  order quantity number;
/*
  Abstract      Make an order for 'number' units
                of 'part'. Return the purchase
                order number.
*/
{
    ...
}
```

In addition to making the program more readable, this technique makes it possible to adapt to different computers with different integer sizes by changing the definitions in only one file.

5.2.2. C-Ref: Unsigned Integer Types

Unsigned integer types have values that range from 0 to some maximum that depends on the size of the type. This maximum is always one less than a power of two; that is, 2^n-1 where n is the number of bits used to represent the unsigned type.

An unsigned type occupies the same amount of storage as the corresponding signed type, but the bit patterns are interpreted differently. On a two's-complement computer, for instance, a 16-bit word with all bits equal to 1 has the value -1 when treated as a signed integer, and has the value 65,535 when treated as an unsigned integer. (The largest signed integer in the same word would be 32,767.)

The original definition of C provided a single unsigned integer type, `unsigned`. However, some compilers now provide an unsigned type corresponding to each signed integer type described in section "C-Ref: Signed Integer Types". The un-

signed type is specified by preceding the corresponding signed type specifier with the keyword `unsigned`.

unsigned-type-specifier :

`unsigned short int`$_{opt}$

`unsigned int`$_{opt}$

`unsigned long int`$_{opt}$

In each case the keyword `int` is optional and does not affect the meaning of the type specifier. Choosing among the unsigned types involves the same considerations already discussed with respect to the signed integer types.

No matter what representation is used for signed integers, an unsigned integer represented with n bits is always considered to be in straight unsigned binary notation, with values ranging from 0 through 2^n-1. Therefore, the bit pattern for a given unsigned value is predictable and portable, whereas the bit pattern for a given signed value is not predictable and not portable.

Whether an integer is signed or unsigned affects the operations performed on it. All arithmetic operations on unsigned integers behave according to the rules of modular (congruence) arithmetic modulo 2^n. So, for example, adding 1 to the largest value of an unsigned type is guaranteed to produce 0.

Expressions that mix signed and unsigned integers are forced to use unsigned operations. The section "C-Ref: The Usual Binary Conversions" discusses the conversions performed, and the chapter "C-Ref: Expressions" discusses the effect of each operator when its arguments are unsigned. These conversions can be surprising. For example, because unsigned integers are always nonnegative, you would expect that the following test would always be "true":

```
unsigned int u;
...
if (u > -1) ...
```

However, it is always "false!" The (signed) -1 is converted to an unsigned integer before the comparison, yielding the largest unsigned integer, and the value of `u` cannot be greater than that integer.

5.2.3. C-Ref: Character Type

The character type in C is an integral type; that is, values of the type are integers and can be used in integer expressions.

character-type-specifier :

`unsigned`$_{opt}$ `char`

The character type has some special characteristics that set it apart from the normal signed and unsigned types. Its representation is implementation dependent and will depend upon the nature of the character and string processing facilities on the target computer. An array of characters is C's notion of a "string." Typical declarations involving characters are:

```
static char greeting[7] = "Hello\n";
char *prompt = &greeting[0];
char padding character = '\0';
```

One thing that is uncertain about the `char` type is whether it is signed or unsigned. For reasons of efficiency, C compilers are free to treat type `char` in one of three ways:

1. Type `char` may be a normal, signed integral type.

2. Type `char` may be a normal, unsigned integral type.

3. Type `char` may be a "pseudo-unsigned" integral type, that is, it can contain only nonnegative values but it is treated as if it were a signed type when performing the usual unary conversions.

In any case, the values of the characters in the standard character set (section "C-Ref: Character Set") are always guaranteed to have nonnegative values when represented as values of type `char`. The following program will determine the handling of the character type, which is assumed to have an 8-bit representation:

```
int main()
{
  char c = 255;
  if (c/-1 == 1) printf("Type char is signed\n")
  else if (c/-1 == 0)
    printf("Typef char is unsigned\n"):
  else if (c/-1 == -255)
    printf("Type char is pseudo-unsigned\n");
  else printf("?c/-1 == %d\n",c/-1);
  return 0;
}
```

The signedness of characters is an important issue because the standard I/O library routines, which normally return characters from files, return -1 (conventionally named `EOF`) when the end of the file is reached. To guard against unsigned characters, the programmer must always treat these functions as returning values of type `int`. For example, the following program is intended to copy characters from the standard input stream to the standard output stream until an end-of-file indication is returned from `getchar`. The first three definitions are usually supplied in the standard header file `stdio.h`.

```
extern int getchar();
extern void putchar();
#define EOF -1

void copy characters()
{
    char ch;                  /* Incorrect! */
```

```
        while ((ch = getchar()) != EOF)
            putchar(ch);
    }
```

However, this function will not work when char is unsigned or pseudo-unsigned. To see this, assume the char type is represented in 8 bits and the int type in 16 bits, and that two's-complement arithmetic is used. Then, when getchar returns -1, the assignment

```
ch = getchar()
```

assigns the value 255 (the low-order 8 bits of -1) to ch. (Strictly speaking, the conversion is implementation dependent, but this result is usual.) The loop test is then

```
255 != -1
```

If type char is pseudo-unsigned, the above (signed) comparision will evaluate to "true." If type char is unsigned the usual conversions will cause -1 to be converted to an unsigned integer, causing the (unsigned) comparison

```
255 != 65535
```

which still evaluates to "true." Thus, the loop never terminates. Changing the declaration of ch to

```
int ch;
```

makes everything work fine.

A common C programming technique is to define a "pseudocharacter" type to use in these cases. For example:

```
typedef int character;
...
void copy characters()
{
    character ch;
    while ((ch = getchar() ) != EOF)
        putchar(ch);
}
```

Now the reader of the program realizes that ch is logically a character, although it is represented with type int.

An implementation that normally treats characters as signed integers may also provide an "unsigned character" type with the type specifier unsigned char. On the other hand, in an implementation that normally treats characters as unsigned integers, there is no way to specify a signed character; the presumption is that if characters could be conveniently implemented as signed quantities, they would have been.

If characters are normally unsigned, you may wish to use something like the following macro, which simulates sign-extension of 8-bit, two's complement numbers. It maps the integers 128 through 255 to the integers -128 through -1.

```
#define sign char(x)  (((x)^128)-128)
```

Draft Proposed ANSI C allows signed characters to be specified explicitly and requires compilers to implement them correctly, even if it is difficult to do so.

A second area of vagueness about characters is their size. In the above example, we assumed they occupied 8 bits, and this assumption is almost always valid (although you still can't be sure if they range from 0 to 255 or from -128 to 127). However, a few computers may use 9 bits or even 7 bits. This is a problem only when a programmer uses characters (especially arrays of characters) as "very short integers."

It is safe to use character arrays to implement boolean arrays, as in the following example that uses a character array to record whether or not some small integers are prime:

```
static char prime vector[] =
        {0,0,1,1,0,1,0,1,0,0,0,1,0,1,0,0,0,1,0,1};
int is prime(n)
  int n;
{
    if (n > 0 && n < sizeof(prime vector))
        if (prime vector[n])
            printf("Yes.\n");
        else
            printf("No.\n");
    else
        printf("Don't know.\n");
}
```

5.3. C-Ref: Floating-Point Types

C's floating-point numbers (sometimes called "real" numbers) come in two sizes: single and double precision, or `float` and `double`.

floating-type-specifier :
```
float
double
```

The type specifier `long float` is permitted in older implementations as a synonym for `double`, but it was never popular and has been eliminated in Draft Proposed ANSI C.

Here are some typical declarations of objects of floating-point type:

```
double d;
static double pi;
float coefficients[10];
   /* "coefficients" is an array of
      floating-point numbers.  */
```

The use of `float` and `double` is analogous to the use of `short` and `int`. Since in expressions all values of type `float` are converted to `double` before any operations are

performed (see section "C-Ref: The Usual Binary Conversions"), the use of type `float` is mainly restricted to structures and large arrays, for which the savings in storage of the shorter `float` type is important. This may change in the future since Draft Proposed ANSI C does permit arithmetic using type `float`.

C does not dictate the sizes to be used for `float` and `double`. The representations used for floating-point numbers are completely machine dependent. On some computers they may have the same implementation. It is reasonable for the programmer to assume, however, that the precision and range of type `double` is at least as great as for type `float`, and thus that the set of values representable as type `float` is a subset of —possibly the same as —the set of values representable as type `double`.

Most of the arithmetic and logical operations may be applied to floating-point operands. These include arithmetic and logical negation; addition, subtraction, multiplication, and division; relational and equality tests; logical (as opposed to bitwise) AND and OR; assignment; and conversions to and from all the arithmetic types. Chapter "C-Ref: Expressions" discusses the operations in more detail.

There are no requirements about the relative sizes of the floating-point and the integer or pointer types. In the past some C programs have depended on the assumption that the type `double` can accurately represent all values of type `long`; that is, that converting an object of type `long` to type `double` and then back to type `long` results in exactly the original `long` value. While this is likely to be true in many implementations of C, it is not required by the C language definition. For maximum portability of programs, the programmer should avoid depending on this assumption.

Draft Proposed ANSI C adds a third floating-point type to C, `long double`, which is potentially larger than type `double`. It also removes the synonym `long float` for `double`.

5.4. C-Ref: Pointer Types

For any type *T* except `void`, a pointer type "pointer to *T*" may be formed. A value of this type is the address of an object of type *T*. The declaration of pointer types is discussed in section "C-Ref: Pointer Declarators". For example, to declare `ip` as a pointer to an object of type `int` and `cp` as a pointer to an object of type `char`, we write

```
int *ip;
char *cp;
```

Pointers are used heavily in C programs, partly because of C's history as a systems programming language and partly because pointers and arrays are so well integrated (section "C-Ref: Arrays and Pointers"). The two most important operators used in conjunction with pointers are the address operator, `&` (section "C-Ref: Address Operator"), and the indirection operator, `*` (section "C-Ref: Indirection"). In the following example, `ip` is assigned the address of variable `i` (`&i`). After that assignment, the expression `*ip` refers to the variable `i`.

```
int i, j, *ip;
ip = &i;
i = 22;
j = *ip;
  /* j now has the value 22 */
*ip = 17;
  /* i now has the value 17 */
```

Every pointer type has a special value, "pointer to nothing," which is written as the integer constant 0. Standard header files usually define the preprocessor macro name `NULL` to be 0. Recalling that the integer 0 also represents "false" in boolean tests, it is no surprise that pointers may be used in logical comparisons. That is, the statement

```
if (ip) i = *ip;
```

is shorthand for

```
if (ip != NULL) i = *ip;
```

Pointers may also be subscripted as if they were arrays; see section "C-Ref: Arrays and Pointers".

5.4.1. C-Ref: Pointer Arithmetic

A convenient feature of C is its provision for performing arithmetic on pointers. If p is an expression of type "pointer to T" and i is an integer value, then the expression

$$p + i$$

is defined to be a pointer to the ith object of type T beyond the one pointed to by p. Thinking in terms of computer addresses, the integer i must be multiplied by the size of type T and then added to p to arrive at a new address.

As a more concrete example, suppose we are on a computer that is byte addressable and on which the type `int` is allocated four bytes. Let `a` be an array of ten integers that begins at address `0x100000`. Let `ip` be a pointer to an integer, and assign to it the address of the first element of array `a`. Finally, let `i` be an integer variable currently holding the value 6. We now have the following situation:

```
int *ip, i, a[10];
ip = &a[0];
i = 6;
```

What is the value of `ip+i`? Because integers are four bytes long, the expression `ip+i` becomes

```
0x100000 + 4*6
```

or `0x100018` (24 is 18 in hexadecimal radix).

Other operations on pointer types include assignment; subtraction; relational and equality tests; logical (as opposed to bitwise) AND and OR; addition and subtraction of integers; and conversions to and from integers and other pointer types.

5.4.2. C-Ref: Some Problems with Pointers

This section will be of interest primarily to compiler writers and advanced programmers. There is a subtle assumption in C that all pointer types (actually, all addresses) have a uniform representation. For instance, on byte-addressed computers it is usual for all pointers to be simple byte addresses occupying, say, one word. Except for alignment considerations, conversions between pointer types—and between pointers and integers—require no change in representation. When executing C programs on such computers, the C programmer will not usually encounter problems.

On some word-oriented computers, however, pointers to characters must be represented by special "field pointers" or "byte pointers," which have a different format from addresses of larger objects. On the DECSYSTEM-20 computer, for instance, a character pointer may have some high-order bits set in the address—bits that are 0 in other pointer types. As a consequence:

1. Converting from character pointers to other pointer types involves a change of representation.

2. Comparisons of pointers of different types cannot be implemented as simple integer comparisons.

3. Conversions between integers and pointers may produce surprising results.

This problem can be even worse on computers with a capability-based addressing structure.

A relatively common problem with some microprocessors is the presence of both short and long address formats. In order to handle the general case, the C implementor must use the longer and less efficient address form everywhere or must extend the language to allow the programmer to specify when the shorter addresses are being used.

Whatever representations are chosen for pointers, they should satisfy the following criterion, which many C programs heavily depend on (especially those that use the library function `malloc`): If the alignment requirement for type *P* is no more stringent than that for type *Q*, it should always be possible to cast a "pointer to *Q*" to a "pointer to *P*" and back, without losing information. For example, on a computer that requires every object of type `double` to have an address that is a multiple of eight characters and requires every object of type `int` to have an address that is a multiple of four characters, it should be possible to cast a "pointer to `double`" to be a "pointer to `int`" and then back to "pointer to `double`" and get back the original pointer. The most important consequence is that a pointer of type "pointer to character" should be capable of holding the equivalent of a pointer to any other type without loss of information.

The programmer should always use explicit casts when converting between pointer types, and should be especially careful that pointer arguments given to functions have the correct type expected by the function.

5.5. C-Ref: Array Types

If T is any C type except `void` or "function returning...," the array type "array of T" may be declared. Values of this type are sequences of elements of type T. All C arrays are 0-origin; for example, the array declared

```
int A[3];
```

consists of the elements `A[0]`, `A[1]`, and `A[2]`. In the following example, an array of integers (`ints`) and an array of pointers (`ptrs`) are declared, and each of the pointers in `ptrs` is set equal to the address of the corresponding integer in `ints`.

```
int ints[10], *ptrs[10], i;
for (i = 0; i < 10; i++)
   ptrs[i] = &ints[i];
```

The size of an array is always equal to the length of the array in elements multiplied by the size of an element.

5.5.1. C-Ref: Arrays and Pointers

In C there is a close correspondence between types "array of T" and "pointer to T." First, when an array identifier appears in an expression, the type of the identifier is converted from "array of T" to "pointer to T," and the value of the identifier is converted to a pointer to the first element of the array. Thus, in

```
int a[10], *ip;
ip = a;
```

the value `a` is converted to a pointer to the first element of the array. It is exactly as if we had written

```
ip = &a[0];
```

This rule is one of the usual unary conversions. The only exception to this conversion rule is when the array identifier is used as an operand of the `sizeof` operator, in which case `sizeof` returns the size of the entire array, not the size of a pointer to the first array element.

Second, array subscripting is defined in terms of pointer arithmetic. That is, the expression

```
a[i]
```

is defined to be the same as

```
*((a) + (i))
```

which is to say the same as

```
*(&(a)[0] + (i))
```

when `a` is an array. This equivalence means also that pointers may be subscripted; it is up to the programmer to ensure that the pointer is pointing into an appropriate array of elements:

```
double d,*dp;
...
d = dp[4];
```

5.5.2. C-Ref: Multidimensional Arrays

Multidimensional arrays are declared as "arrays of arrays," such as in the declaration

```
int matrix[10][10];
```

which declares `matrix` to be a 10-by-10 element array of `int`. The language places no limit on the number of dimensions an array may have.

Multidimensional arrays are stored such that the last subscript varies most rapidly. That is, the elements of the array

```
int t[2][3];
```

are stored (in increasing addresses) as

```
t[0][0], t[0][1], t[0][2], t[1][0], t[1][1], t[1][2]
```

The conversions of arrays to pointers happens analogously for multidimensional arrays. For example, if `t` is a 2-by-3 array as defined above, then the expression `t[1][2]` is expanded to

```
*(*(t+1)+2)
```

which is evaluated as follows:

`t`	a 2-by-3 array, which is converted immediately to a pointer to (the first) 3-element array
`t+1`	a pointer to (the second) 3-element array
`*(t+1)`	(the second) 3-element array of integers, which is immediately converted to a pointer to (the first) integer (in the second 3-element array)
`*(t+1)+2`	a pointer to (the third) integer (in the second 3-element array)
`*(*(t+1)+2)`	(the third) integer (in the second 3-element array)

In general, any expression A of type "i-by-j-by- ... -by-k array of T" is immediately converted to "pointer to j-by- ... -by-k array of T."

5.5.3. C-Ref: Array Bounds

Any time that storage for an array is allocated, the size of the array must be known. However, because subscripts are not normally checked to lie within declared array bounds, it is possible to omit the size when declaring an external, singly dimensioned array defined in another module or when declaring a singly dimensioned array that is a formal parameter to a function. (See section "C-Ref: Array Declarators".) For instance, the following function, `sum`, returns the sum of the first `n` elements of an external array, `a`, whose bounds are not specified.

```
extern int a[];
```

```
int sum(n)
  int n;
{
    int i, s = 0;
    for (i = 0; i < n; i++)
      s += a[i];
    return s;
}
```

It is common when passing arrays to functions to omit the bounds information on the formal parameter:

```
int sumarray(array,arraylength)
  int arraylength, array[];
{
  ...
}
```

In this example the parameter a could also be declared as "int *a", which would more accurately reflect the implementation but perhaps less clearly indicate the intent.

When multidimensional arrays are used, it is necessary to specify the bounds of all but the first dimension, so that the proper address arithmetic can be calculated.

```
extern int matrix[][10]; /* ?-by-10 array of int */
```

If such bounds are not specified, the declaration is in error.

5.5.4. C-Ref: Operations

The only operation that can be performed directly on an array value is the application of the `sizeof` operator. The array must be bounded. The result of such an operation is the number of storage units occupied by the array. For an n-element array of type T, the result of the `sizeof` operator is always equal to n times the result of `sizeof` applied to the type T.

In all other contexts, such as subscripting, the array value is actually treated as a pointer, and so operations on pointers may be applied to the array value.

5.6. C-Ref: Enumeration Types

Enumeration types are a recent addition to C, and similar concepts occur in other languages, such as Pascal and Ada. Unfortunately, not all C compilers implement enumerations, and those that do are not all consistent in their implementations.

An enumeration type in C is a set of integer values represented by identifiers called *enumeration constants*. The enumeration constants are specified when the type is defined. For example, the declaration

```
enum fish { trout, carp, halibut }
                    my fish, your fish;
```

creates a new enumeration type "enum fish," whose values are trout, carp, and halibut. It also declares two variables of the enumeration type, my fish and your fish, which can be assigned values with the assignments

```
my fish = halibut;
your fish = trout;
```

In addition to assigning values of enumeration types, the programmer can test two values for equality.

Enumeration types are implemented by associating integer values with the enumeration constants, so that the assignment and comparison of values of enumeration types can be implemented as integer assignment and comparison. These integers are normally chosen automatically, but they can be specified by the programmer in the type definition:

```
enum fish { trout=1, carp=0,
            halibut=10 } my fish, your fish;
```

The integers chosen determine the equality and ordering relationships among values of the enumeration type. We will say more about this later.

Here is the general syntax for declaring and using enumeration types:

enumeration-type-specifier :
  enumeration-type-definition
  enumeration-type-reference

enumeration-type-definition :
  enum *enumeration-tag*$_{opt}$
  { *enumeration-definition-list* }

enumeration-type-reference :
  enum *enumeration-tag*

enumeration-tag :
  identifier

enumeration-definition-list :
  enumeration-constant-definition
  enumeration-definition-list , *enumeration-constant-definition*

enumeration-constant-definition :
  enumeration-constant
  enumeration-constant = *expression*

enumeration-constant :
  identifier

Variables or other objects of the enumeration type can be declared in the same

declaration containing the enumeration type definition or in a subsequent declaration that mentions the enumeration type with an "enumeration type reference." For example, the single declaration

```
enum color { red, blue, green, mauve }
 favorite, acceptable, least favorite;
```

is exactly equivalent to the two declarations

```
enum color { red, blue, green, mauve } favorite;
enum color acceptable, least favorite;
```

and to the four declarations

```
enum color { red, blue, green, mauve };
enum color favorite;
enum color acceptable;
enum color least favorite;
```

The enumeration tag, `color`, allows an enumeration type to be referenced after its definition. Although the declaration

```
enum { red, blue, green, mauve }
    favorite, acceptable, least favorite;
```

defines the same type and declares the same variables, the lack of an enumeration tag makes it impossible to introduce more variables of the type in later declarations. Enumeration tags are in the same overloading class as structure and union tags, and their scope is the same as that of a variable declared at the same location in the source program.

Identifiers defined as enumeration constants are members of the same overloading class as variables, functions, and typedef names. Their scope is the same as that of a variable defined at the same location in the source program. In the following example, the declaration of `shepherd` as an enumeration constant hides the previous declaration of the integer variable `shepherd`. However, the declaration of the floating-point variable `collie` will cause a compilation error, because `collie` is already declared in the same scope as an enumeration constant.

```
int shepherd = 12;
{
    enum dog breeds {shepherd, collie};
                    /* Hides outer declaration of
                       the name "shepherd" */
    float collie;   /* Illegal redefinition of
                       the name "collie" */
}
```

The size of an enumeration type is generally same as the size of type `int`, although some implementations leave open the possibility of using `short` or `long` depending on the size of the enumeration type. Integer values are associated with enumeration constants in the following way:

1. An explicit integer value may be associated with an enumeration constant by writing

enumeration-constant = *expression*

in the type definition. The expression must be a constant expression of integral type, although some compilers may also allow expressions involving previously defined enumeration constants, as in

```
enum boys { Bill = 10, John = Bill+2,
            Fred = John+2 };
```

2. The first enumeration constant receives the value 0 if no explicit value is specified.

3. Subsequent enumeration constants without explicit associations receive an integer value one greater than the value associated with the previous enumeration constant.

For example, given the declaration

```
enum sizes { small, medium=10, pretty big, large=20 };
```

the values of `small`, `medium`, `pretty big`, and `large` will be 0, 10, 11, and 20, respectively.

Any signed integer value representable as type `int` may be associated with an enumeration constant. Positive and negative integers may be chosen at random, and it is even possible to associate the same integer with two different enumeration constants. For instance, the following definition is legal

```
enum people { john=1, mary=19, bill=-4, sheila=1 };
```

but then the expression

```
john == sheila
```

is "true," which is not intuitive.

Although the form of an enumeration type definition is suggestive of structure and union types, with strict type checking, Draft Proposed ANSI C and most current implementations mandate that all enumeration types be treated as integer types and that enumerations constants appearing in expressions have type `int`. Thus enumeration types function as little more than ways to name integer constants. As a matter of style, we suggest that programmers treat enumeration types as different from integers and not mix them in integer expressions without using casts.

5.7. C-Ref: Structure Types

The structure types in C are similar to the types known as "records" in other programming languages. They are collections of named *components* (also called "members" or "fields") that can have different types. One way of looking at structures is that they allow the programmer to create *abstract data types*. Structures can be defined to encapsulate related data objects—the values of the abstract type—and functions can be written to manipulate the values—the operations on the type.

For example, a programmer who wanted to implement complex numbers might define a structure `complex` to hold the real and imaginary parts as components `real` and `imag`. The first declaration below defines the new type, and the second declares two variables, `x` and `y`, of that type:

```
struct complex {
    double real;
    double imag;
};
struct complex x, y;
```

A function `new complex` can be written to create a new object of the type. Note that the selection operator `.` is used to access the components of the structure.

```
struct complex  new complex(r, i)
  double r, i;
{
    struct complex new;
    new.real = r;
    new.imag = i;
    return new;
}
```

Operations on the type, such as `complex multiply`, can also be defined:

```
struct complex  complex multiply(a, b)
  struct complex a,b;
{
  struct complex product;
  product.real=(a.real * b.real - a.imag * b.imag);
  product.imag=(a.real * b.imag + a.imag * b.real);
  return product;
}
```

Here is the general syntax for structure declarations:

structure-type-specifier :
    structure-type-definition
    structure-type-reference

structure-type-definition :
    res[struct] structure-tag$_{opt}$ { *field-list* }

structure-type-reference :
    `struct` *structure-tag*

structure-tag :
    identifier

field-list :
    component-declaration
    field-list component-declaration

component-declaration :
　　type-specifier component-declarator-list ;

component-declarator-list :
　　component-declarator
　　component-declarator-list , *component-declarator*

component-declarator :
　　simple-component
　　bit field

simple-component :
　　declarator

bit-field :
　　$declarator_{opt}$: *width*

width :
　　expression

Each structure type definition introduces a new structure type, different from all others. If present in the definition, the structure tag is associated with the new type and can be used in a subsequent structure type reference. For instance, the single declaration

```
struct complex { double real, imag; } x, y;
```

is equivalent to the two declarations

```
struct complex { double real, imag; };
struct complex x, y;
```

5.7.1. C-Ref: Structure Type References

A structure type reference appearing without a previous corresponding structure type definition establishes an "incomplete" type definition whose scope is the innermost enclosing block or, if the reference appears at the top level, the remainder of the program. An incomplete definition may be used only to declare pointers to the type or to declare typedef names as synonyms for the type. That is, incomplete definitions may be used when the size of the structure is not needed.

The most common use for these incomplete definitions is in defining self-referential structure types. In the following example, the definition of structure `P` in the first line also establishes and uses an incomplete definition of structure `Q`, which is then defined in the second line.

```
struct P { struct Q *pq; };
struct Q { struct P *pp; };
```

This feature should be used cautiously. If we modify the program surrounding the two definitions above, they could have a different effect:

```
struct Q { int a, b; };
...
{
    struct P { struct Q *pq; };
    struct Q { struct P *pp; };
    ...
}
```

Structure `P` will now have as its component `pq` a pointer to the structure `Q` defined in the outer block. This is an instance of "duplicate visibility" of tag `Q`. The problem can be avoided by the customary practice of declaring structure types only at the top level of C programs.

Draft Proposed ANSI C is a bit more precise in how incomplete structure definitions are to be handled.

5.7.2. C-Ref: Operations on Structures

The operations provided for structures may vary from compiler to compiler. All C compilers provide the selection operators `.` and `->` on structures, and newer compilers now allow structures to be assigned, to be passed as parameters to functions, and to be returned from functions. (With older compilers, assignment must be done component by component, and only pointers to structures may be passed to and from functions.)

It is not permitted to compare two structures for equality. An object of a structure type is a sequence of components of other types. Because certain data objects may be constrained by the target computer to lie on certain addressing boundaries, a structure object may contain "holes," storage units that do not belong to any component of the structure. The holes would make equality tests implemented as a wholesale bit-by-bit comparison unreliable, and component-by-component equality tests would be too expensive. (Of course, the programmer may write component-by-component equality functions.)

In any situation where it is permitted to apply the unary address operator `&` to a structure to obtain a pointer to the structure, it is also permitted to apply the operator to a component of the structure to obtain a pointer to the component; that is, it is possible for a pointer to point into the middle of a structure. An exception to this rule occurs with components defined to be bit fields. Components defined as bit fields will in general not lie on machine-addressable boundaries, and therefore it may not be possible to form a pointer to a bit field. The C language therefore forbids such pointers.

5.7.3. C-Ref: Components

A component of a structure may have any type except "function returning ..." and `void`. Structures may not contain instances of themselves, although they may contain pointers to instances of themselves. That is,

```
struct S {
    int a;
    struct S next;  /* illegal! */
};
```

is illegal, but

```
struct S {
    int a;
    struct S *next;
};
```

is permitted.

The names of structure components are defined in a special overloading class associated with the structure type. That is, component names within a single structure must be distinct, but they may be the same as component names in other structures and may be the same as variable, function, and type names. For example, consider the following sequence of declarations:

```
int x;
struct A { int x; double y; } y;
struct B { int y; double x; } z;
```

The identifier `x` has three nonconflicting declarations: it is an integer variable, an integer component of structure `A`, and a floating-point component of structure `B`. These three declarations are used, respectively, in the expressions

```
x
y.x
z.x
```

If a structure tag is defined in one of the components, as `T` is in

```
struct S {
    struct T {int a, b; } x;
};
```

the scope of `T` extends to the end of the block in which structure `S` is defined. (If `S` is defined at the top level, so is `T`.)

A historical note: The original definition of C specified that all components in all structures were allocated out of the same overloading class, and therefore no two structures could have components with the same name. (An exception was made when the components had the same type and the same relative position in the structures.) This interpretation is now anachronistic, but you might see it mentioned in older documentation or actually implemented in some old compilers.

5.7.4. C-Ref: Structure Component Layout

Most programmers will be unconcerned with how components are packed into structures. However, C does give the programmer some control over the packing. C compilers are constrained to assign components increasing memory addresses in a strict order, with the first component starting at the beginning address of the structure itself. There is no difference in component layout between the structure

```
struct { int a, b, c; };
```

and the structure

```
struct { int a; int b, c; };
```

Both put a first, b second, and c last. In general, given two pointers *p* and *q* to components within the same structure, $p < q$ will be true if and only if the declaration of the component that *p* points to appears earlier within the declaration of the structure type than the declaration of the component that *q* points to. For example:

```
{
    struct vector3 { int x, y, z; } s;
    int *p, *q, *r;
    ...
    p = &s.x;
    q = &s.y;
    r = &s.z;
    /* At this point (p < q), (q < r),
       and (p < r) are all true. */
    ...
}
```

Holes, or padding, may appear between any two consecutive components in the layout of a structure if necessary to allow proper alignment of components in memory. The bit patterns appearing in such holes are unpredictable, and may differ from structure to structure or over time within a single structure.

5.7.5. C-Ref: Bit Fields

C allows the programmer to pack integer components into spaces smaller than the compiler would ordinarily allow. These integer components are called *bit fields* and are specified by following the component declarator with a colon and a constant integer expression that indicates the width of the field in bits:

```
struct S{
    unsigned  a:4;
    unsigned  b:5, c:7;
};
```

The intent is that bit fields should be packed as tightly as possible in a structure, subject to the rules discussed below.

Bit fields are typically used in machine-dependent programs that must force a data structure to correspond to a fixed hardware representation. The precise manner in which components (and especially bit fields) are packed into a structure is implementation dependent but is predictable for each implementation. The use of bit fields is therefore likely to be nonportable. The programmer should consult the implementation documentation if it is necessary to lay out a structure in memory in some particular fashion, and then verify that the C compiler is indeed packing the components in the way expected.

Here is an example of how bit fields can be used to match the format of a 32-bit virtual address. We assume a right-to-left byte ordering, which is discussed further in section "C-Ref: Addressing Structure and Byte Ordering".

```
/*   Format of 32-bit virtual address
     for the XBQ-43 computer.
+-+-+---------+-----------+----------------------+
|S|x| Segment |   Page    |       Offset         |
+-+-+---------+-----------+----------------------+
31   29        23          15                bit 0
*/
/* Note: the XBQ-43 C compiler packs bit fields
                         from right to left. */
typedef struct {
    unsigned offset     : 16;
    unsigned page       : 8;
    unsigned segment    : 6;
    unsigned            : 1;  /* for future use */
    unsigned supervisor : 1;
  } virtual address;
```

C specifies that no compiler has to allow bit fields of any type except `unsigned`, but some compilers allow signed integer types (that is, sign extension occurs when extracting the contents of the field). Usually, the signedness of bit fields follows the signedness of characters, that is, a bit field of type `int` may actually be implemented as a signed, unsigned, or pseudo-unsigned type. (See section "C-Ref: Character Type".) Use of any type other than `unsigned` for a bit field should therefore be considered even less portable than using bit fields at all.

Compilers are free to impose constraints on the maximum size of a bit field and to specify certain addressing boundaries that bit fields cannot cross. These alignment restrictions are usually related to the natural word size of the target computer. When a field is too long for the computer, the compiler will issue an appropriate error message. When a field would cross a word boundary, it may be moved to the next word.

The grammar indicates that an unnamed bit field may be included in a structure. The intent is to allow a specific amount of padding space to be inserted into the structure. For instance, the structure

```
struct S {
    unsigned a : 4;
    unsigned   : 2;
    unsigned b : 6;
};
```

indicates that component `a` is to be allocated the first four bits of the structure, followed by two bits of padding, followed by the component `b` in six bits. Unnamed bit fields cannot be referenced and their contents at run time are not predictable.

The case of an unnamed bit field of length 0 is a special case. It indicates that the following component should begin on the next boundary appropriate to its type.

("Appropriate" is not specified further.) That is, in the structure

```
struct S {
    unsigned a : 4;
    unsigned   : 0;
    unsigned b : 6;
};
```

the component `b` should begin on a natural addressing boundary following component `a`.

The address operator `&` may not be applied to bit-field components, since many computers cannot address arbitrary-sized fields directly.

5.7.6. C-Ref: Portability Problems

The use of bit fields is likely to be nonportable and therefore should be restricted to situations in which memory is a scarce resource or in which a hardware-defined data structure must be matched exactly. (In the latter case the program will doubtless be nonportable anyway, so using bit fields doesn't make matters any worse.)

There are several ways in which depending on packing strategies is dangerous. First, computers differ on the alignment constraints on data types. For instance, a four-byte integer on some computers must begin on a byte boundary that is a multiple of four, whereas on other computers the integer can (and will) be aligned on the nearest byte boundary.

Second, the restrictions on bit-field widths will be different. Some computers have a 16-bit word size, which limits the maximum size of the field and imposes a boundary that fields cannot cross. Other computers have a 32-bit word size, and so forth.

Third, computers differ in the way fields are packed into a word, that is, in their "byte ordering." On IBM 370-style computers, characters are packed left-to-right into words, from the most significant bit to the least significant bit. On DEC VAX computers and many microprocessors, characters are packed right-to-left, from the least significant bit to the most significant bit.

5.7.7. C-Ref: Sizes of Structures

The size of an object of a structure type is the amount of storage necessary to represent all components of that type, including any unused padding space between or after the components. The rule is that the structure will be padded out to the size the type would occupy as an element of an array of such types. (For any type T, including structures, the size of an n-element array of T is the same as the size of T times n.) Another way of saying this is that the structure must terminate on the same alignment boundary on which it started; that is, if the structure must begin on an even byte boundary, it must also end on an even byte boundary.

For example, on a computer that starts all structures on an address that is a multiple of 4 bytes, the length of the structure

```
struct S {
    char c1;
    char c2;
};
```

will be a multiple of four (probably exactly four), even though only two characters are actually used.

Note that the alignment requirement for a structure type will be at least as stringent as for the component having the most stringent requirements. For example, on a computer that requires all objects of type `double` to have an address that is a multiple of 8 bytes, the length of the structure

```
struct S {
    double value;
    char name[10];
};
```

will probably be 24, even though the components may be allocated contiguously and their total length is 18 units; six extra units of padding are needed at the end to make the size of the structure a multiple of the alignment requirement. (If the padding were not used, then in an array of such structures not all of the structures would have the `value` component aligned properly to a multiple-of-eight address.)

Alignment requirements may cause padding to appear in the middle of a structure. Consider this variation on the previous example in which the two components appear in the other order:

```
struct S {
    char name[10];
    double value;
};
```

The length of this structure will also be 24. After the ten units of storage allocated for the `name` component, six units of padding are required before the `value` component so that the `value` component may be aligned to an address that is a multiple of eight characters relative to the beginning of the structure. Any object of the structure type will be required to have an address that is a multiple of eight, and so the `value` component of such an object will always be properly aligned.

5.8. C-Ref: Union Types

The syntax for defining union types is almost identical to that for defining structure types:

union-type-specifier :
    union-type-definition
    union-type-reference

union-type-definition :

`union` *union-tag*$_{opt}$ { *field-list* }

union-type-reference :
`union` *union-tag*

union-tag :
identifier

The syntax for defining components is the same as that used for structures, except that bit fields are not permitted in unions.

As with structures and enumerations, each union type definition introduces a new union type, different from all others. If present in the definition, the union tag is associated with the new type and can be used in a subsequent union type reference. Forward references and incomplete definitions of union types are permitted with the same rules as structure types.

A component of a union may have any type except "function returning ..." and `void`. Also, unions may not contain instances of themselves, although they may contain pointers to instances of themselves.

As in structures, the names of union components are defined in a special overload class associated with the union type. That is, component names within a single union must be distinct, but they may be the same as component names in other unions and may be the same as variable, function, and type names.

5.8.1. C-Ref: Union Component Layout

Each component of a union type is allocated storage starting at the beginning of the union. An object of a union type will begin on a storage alignment boundary appropriate for any contained component.

In other words, if we have the following union type and object definitions:

```
static union U {
    ...
    int C;
    ...
  } object, *P = &object;
```

then the following equalities hold:

```
(union U *) &(P->C)  ==          P
            &(P->C)  ==  (int *) P
```

Furthermore, these equalities hold no matter what the type of the component `C`.

5.8.2. C-Ref: Sizes of Unions

The size of an object of a union type is the amount of storage necessary to represent the largest component of that type, plus any padding that may be needed at the end to raise the length up to an appropriate alignment boundary. The rule is that the union will be padded out to the size the type would occupy as an element

of an array of such types. (For any type T, including unions, the size of an n-element array of T is the same as the size of T times n.) Another way of saying this is that the structure must terminate on the same alignment boundary on which it started; that is, if the structure had to begin on an even byte boundary, it must end on an even byte boundary.

Note that the alignment requirement for a union type will be at least as stringent as for the component having the most stringent requirements. For example, on a computer that requires all objects of type `double` to have an address that is a multiple of eight characters, the length of the union

```
union U {
    double value;
    char name[10];
};
```

will be 16, even though the size of the longest component is only 10; six extra units of padding are needed to make the size of the union a multiple of the alignment requirement. (If the padding were not used, then in an array of such unions not all of the unions would have the `value` component aligned properly to a multiple-of-eight address.)

5.8.3. C-Ref: Using Union Types

C's union type is somewhat like a "variant record" in other languages. Like structures, unions are defined to have a number of components. Unlike structures, however, a union can hold at most one of its components at a time; the components are conceptually overlaid in the storage allocated for the union.

For example, suppose we want an object that can be *either* an integer or a floating-point number, depending on the situation. We can define union `datum`:

```
union datum {
    int i;
    double d;
};
```

and define a variable of the union type:

```
union datum u;
```

Then, to use the datum to hold an integer, we can say:

```
u.i = 15;
```

To use the datum to hold a floating-point number, we assign to the other component:

```
u.d = 88.9e4;
```

The programmer is responsible for remembering what kind of data is in the union at any one time.

Unions may be used in a portable fashion if certain rules are obeyed. In particular, the only time a component of a union should be referenced is if the last assignment to the union was through the same component. It is *not* portable to assign one union component and then reference another component:

```
union U { long c; double d; } x;
long l;
x.d = 1.0e10;
l = x.c;
```

C provides no way to inquire which component of a union was last assigned. The programmer can encode explicit *data tags* in unions; that is, some indication of which component is stored in the union. The union may be enclosed in a structure that includes a special tag component that is used by programming convention to indicate which component of a union is "active." For example, we might replace the union

```
union widget {
    long count;
    double value;
    char name[30];
  } x;
```

by

```
enum widget tag { count widget,
                  value widget,
                  name widget };

struct WIDGET {
    enum widget tag tag;
    union { long count;
            double value;
            char name[30]; } data;
  } x;

/* Make "widget" a name for "struct WIDGET". */
typedef struct WIDGET widget;
```

To assign to the union, we write either

```
x.tag = count widget;
x.data.count = 10000;
```

or

```
x.tag = value widget;
x.data.value = 3.14159265358979323 84;
```

or

```
x.tag = name widget;
strcpy(x.data.name, "Millard Fillmore");
```

Then we can write a portable function that can discriminate among the possibilities for the union, and call the function without regard to which component was last assigned:

```
/* Print a widget, whatever it contains. */
void print widget(w)
  widget w;
{
    switch(w.tag) {
      case count widget:
        printf("Count %ld\n", w.data.count);
        break;
      case value widget:
        printf("Value %f\n", w.data.value);
        break;
      case name widget:
        printf("Name \"%s\"\n", w.data.name);
        break;
    }
}
```

5.9. C-Ref: Function Types

The type "function returning *T*" is a function type, where *T* may be any type except "array of..." or "function returning...." Said another way, functions may not return arrays or other functions, although they can return pointers to arrays and functions.

Objects of function type may be introduced in only two ways. First, a function *definition* can create a function object, define its parameters and return value, and supply the body of the function. For example, `square` is an object of function type:

```
int square(x)
  int x;
{
    return x*x;
}
```

More information about function definitions is given in section "C-Ref: Function Definitions".

Second, a function *declaration* can introduce an external reference to a function object defined elsewhere, such as in this definition of `square`.

```
extern int square();
```

Most modern implementations allow (forward) declarations of static functions by using the storage class `static` in the declaration:

```
static void printheading();
```

The actual definition of function `printheading` is expected to appear later in the program. However, most UNIX implementations of C also allow forward declarations of static functions to use the `extern` storage class; when the compiler sees the static definition, it avoids issuing any external linkages of the function name:

```
extern void printheading();  /* it isn't extern! */
...
    printheading("Hello");
...
static void printheading(greet)
  char *greet;
{
  ...
}
```

There may be a few deficient implementations for which this programming technique must be used, but it is misleading and should be avoided.

The only operation that can be applied to an expression of function type (aside from the implicit usual unary conversion to "pointer to function") is to call the function. For example, the line

```
extern int f(), (*fp)(), (*apf[])();
```

declares external identifiers `f`, `fp`, and `apf` to have types "function returning `int`," "pointer to function returning `int`," and "array of pointers to functions returning `int`," respectively. These identifiers can be used in function call expressions by writing:

```
int i;
i = f(14);
i = (*fp)(j, k);
i = (*apf[j])(k);
```

When a function is called, certain standard conversions are applied to the actual arguments, but no attempt is made to check the type or number of arguments with the type or number of formal arguments to the function, if known. See section "C-Ref: The Function Argument Conversions".

A function identifier can appear by itself—that is, not in the context of a call—but the identifier is immediately converted to the type "pointer to function returning...." For example,

```
extern int f();
int (*fp)();
fp = f;
```

The only expressions that can yield a value of type "function returning T" are the name of such a function and an indirection expression consisting of the unary indirection operator * applied to an expression of type "pointer to function returning...."

There are only three operations that can be applied to the name of a function:

1. Call the function, by appending to the name a parenthesized, comma-separated list of argument expressions.

2. Use the function name as an actual parameter to another function, in which case a pointer to the function is passed.

3. Assign the function to a variable of the appropriate (pointer) type.

The `sizeof` operator may not be applied to functions.

All the information needed to invoke a function is assumed to be encapsulated in an object of type "pointer to function returning...," which satisfies the normal pointer requirements. Although a pointer to a function is often assumed to point to the function's code in memory, on some computers a function pointer actually points to a block of information needed to invoke the function. Such representation issues are normally invisible to the C programmer and need concern only the compiler implementor.

The important new concept of the "function prototype," which introduces parameter type information into function declarations, is introduced by Draft Proposed ANSI C.

5.10. C-Ref: Void

The type `void` is a recent addition to C. It has no values and no operations and is used mainly as the return type of a function, signifying that the function returns no value.

void-type-specifier :

```
void
```

For example:

```
{
    extern void write line();
    ...
    write line();
    ...
}
```

The `void` type may also be used in a cast expression when it is desired to explicitly discard a value. For example:

```
{
    extern int write line2();  /* returns error
                                  indication */
    ...
    (void) write line2(...);   /* don't check for
                                  error */
    ...
}
```

Casting the return value to void indicates clearly that the programmer knows that `write line2` returns a value but chooses to ignore it.

Draft Proposed ANSI C introduces the type "`void *`", or pointer to `void`, to represent a "universal" data pointer. Traditionally C programmers have used `char *` for this purpose.

5.11. C-Ref: Typedef Names

When a declaration is written whose "storage class" is `typedef`, the type definition facility is invoked.

typedef-name :
identifier

An identifier enclosed in any declarator of the declaration is defined to be a name for a type (a "typedef name"); the type is that which would have been given the identifier if the declaration were a normal variable declaration. For example, in the declaration

```
typedef int *ptr, (*func)();
```

the name `ptr` is defined to be the type "pointer to `int`" and the name `func` is defined to be the type "pointer to function returning `int`."

Once a name has been declared as a type, it may appear anywhere a type specifier is permitted. This is useful because it allows you to create mnemonic abbreviations for complicated types. For example, after writing the above type definitions, we can write

```
ptr link, *indirect link;
func my routine, my vector[10];
```

in which case

`link`	has type "pointer to `int`"
`indirect link`	has type "pointer to pointer to `int`"
`my routine`	has type "pointer to function returning `int`"
`my vector`	has type "10-element array of pointers to functions returning `int`"

Typedef names should not be mixed with other type specifiers. Some implementations permit the following; Draft Proposed ANSI C explicitly forbids it:

```
typedef long int bigint;
unsigned bigint x;     /* probably illegal */
```

Typedef declarations do not introduce new types; the names are always considered to be synonyms for types that could be specified in other ways. For instance, after the declaration

```
typedef struct S { int a; int b; } s1type, s2type;
```

the type specifiers `s1type`, `s2type`, and `struct S` can be used interchangeably to refer to the same type.

C implementations may wish to preserve the type distinctions internally so that debuggers and other associated tools can refer to types by the names used by the programmer.

5.11.1. C-Ref: Typedef Names for Function Types

A function type may be given a typedef name. For example, the DblFunc may be established as a synonym for "function returning double" with this declaration:

```
typedef double DblFunc();
```

Once declared, DblFunc can be used to declare objects of the function type, pointers to the function type, arrays of the function type, and so forth, using the normal rules for composing declarators:

```
extern DblFunc f, *f ptr, *f array[];
```

Abiding by the normal rules of type declarations, the programmer should not declare illegal types, such as a function returning another function:

```
extern DblFunc f ptr();  /* illegal! */
```

The method can be extended to more complex types, such as "function returning pointer to function returning int":

```
typedef int  (*fatfunc())();
extern fatfunc chubby;
```

A fundamental problem arises when trying to use typedef names such as these in function definitions. For example, the following definition of fabs is rejected because it seems to define a function returning another function:

```
typedef double DblFunc();
DblFunc fabs(x)
  double x;
{
  if (x<0.0) return -x; else return x;
}
```

Unfortunately, you can't have the parameter list or omit it! Here is a more complicated example:

```
typedef int  (*fatfunc())();
fatfunc chubby(a,b)
  int a,b;
{
  ...
}
```

In Draft Proposed ANSI C the function typedef name can include prototype information, including the parameter names. However, you cannot inherit the type of a function definition from a typedef name. Thus, the definitions of chubby and fabs are illegal in Draft Proposed ANSI C. We recommend that programmers respect this limitation even if their current C implementation permits the definitions.

5.11.2. C-Ref: Redefining Typedef Names

The language specifies that typedef names may be redefined in inner blocks in the same fashion as ordinary identifiers:

```
typedef int T;
T foo;
...
{
    float T;
    T = 1.0;
    ...
}
```

(One restriction is that the redeclaration cannot omit the type specifiers thinking that the type will default to `int`.) However, some compilers have been known to have problems with such redeclarations, probably because of the pressure `typedef` names put on the C language grammar. We now turn to this problem.

5.11.3. C-Ref: A Note on Implementation of Typedef Names

Allowing ordinary identifiers to be type specifiers makes the C grammar context sensitive, and hence not LALR(1). To see this, consider the program line

```
A ( *B ) ;
```

If `A` has been defined as a typedef name, then the line is a declaration of a variable `B` to be of type "pointer to `A`." (The parentheses surrounding "`*B`" are ignored.) If `A` is not a type name, then the line is a call of the function `A` with the single parameter `*B`. This ambiguity cannot be resolved grammatically.

C compilers based on the UNIX parser-generator YACC—such as the Portable C Compiler—handle this problem by feeding information acquired during semantic analysis back to the lexer. C compilers must do some `typedef` analysis during lexical analysis.

5.12. C-Ref: Type Equivalence

Several times in the text we say that, for instance, two objects must have the same type. What do we mean?

First of all, two pointer or function types are the same if their elements are the same:

1. Two types "pointer to T" and "pointer to S" are the same only if types T and S are the same.

2. Two types "function returning T" and "function returning S" are the same only if types T and S are the same.

Rules for the other types are discussed in the following sections.

In Draft Proposed ANSI C, when function prototypes are used, the parameter names and types become part of the function type. Two function types must agree as to the presence or absence of prototypes and as to the content of the prototypes.

5.12.1. C-Ref: More About Array Types

Two types "n-element array of T" and "m-element array of S" are the same only if types T and S are the same and n=m. A consequence of this is, for instance, that the pointer types "pointer to 10-element array of `int`" and "pointer to 5-element array of `int`" are *not* the same. (Incrementing values of these types will have different effects since the pointer element sizes are different.) However, there are two clarifications to this equivalence rule.

First, in those contexts in which an array's size may be omitted, the size does not participate in computing type equivalence. (If the array is multidimensional, and if the first dimension's size may be omitted, then *only* the first dimension does not participate; the other dimensions must match.) Therefore the two external declarations

```
extern int a[10];
extern int a[];
```

effectively declare `a` to be of the same type.

Second, in many situations a value of type "n-element array of T" is converted to a value of type "pointer to T." In that case, the rules for pointer type equivalence apply; that is, the size of the array becomes irrelevant.

5.12.2. C-Ref: Enumeration, Structure, and Union Types

Each occurrence of a type specifier that is a structure type definition, union type definition, or enumeration type definition introduces a new structure, union, or enumeration type that is neither the same as nor equivalent to any other type.

A type specifier that is a structure, union, or enumeration type *reference* is the same type introduced in the corresponding *definition*. The type tag is used to associate the reference with the definition, and in that sense the tag may be thought of as the name of the type. Thus, the types of `x`, `y`, and `u` below are all different, but the types of `u` and `v` are the same.

```
struct { int a; int b; } x;
struct { int a; int b; } y;
struct S { int a; int b; } u;
struct S v;
```

Historically, tags predate the `typedef` facility in C, which largely supplants them. The only facility provided by tags that cannot be duplicated with type definitions is recursive references within structures and unions:

```
struct S { int data; struct S *next; };
```

5.12.3. C-Ref: More About Typedef Names

Names declared as types in `typedef` definitions are synonyms for types, not new types. Thus, in the following example, the type `my int` is the same as type `int`, and the type `my function` is the same as the type "`float *()`."

```
typedef int my int;
typedef float *my function();
```

In the more complicated example

```
struct S { int a, b; } x;
typedef struct S  t1, t2;
struct S w;
t1 y;
t2 z;
```

the variables `w`, `x`, `y`, and `z` all have the same type.

5.13. C-Ref: Type Names and Abstract Declarators

In two situations in C programming, it is necessary to write the name of a type without declaring an object of that type: when writing cast expressions and when applying the `sizeof` operator applied to a type. In these cases, one uses a *type name* built from an *abstract declarator*. (Don't confuse "type name" with "typedef name" described in section "C-Ref: Typedef Names".)

type-name :
  type-specifier abstract-declarator

abstract-declarator :
  empty-abstract-declarator
  nonempty abstract-declarator

empty-abstract-declarator :

nonempty-abstract-declarator :
  (*nonempty-abstract-declarator*)
  abstract-declarator ( )
  abstract-declarator [$expression_{opt}$]
  * *abstract-declarator*

An abstract declarator resembles a regular declarator in which the enclosed identifier has been replaced by the empty string. Thus, a type name looks like a declaration from which the enclosed identifier has been omitted.

The precedences of the alternatives of the abstract declarator are the same as in the case of normal declarators. However, to resolve an ambiguity, the form

  (*A*)

is permitted only if the abstract declarator *A* is nonempty.

Examples of type names:

`int`	type `int`
`float *`	type "pointer to `float`"

`char (*)()` type "pointer to function returning `char`"

`unsigned *[4]` type "array of four pointers to `unsigned`"

`int (*(*)())()` type "pointer to function returning pointer to function returning `int`"

Type names always appear within the parentheses that form part of the syntax of the cast or `sizeof` operator. If the type specifier in the type name is a structure, union, or enumeration type definition, most implementations will *not* define any included type tag. For example, assume that `struct S` is not defined when the following two statements are encountered:

```
i = sizeof( struct S {int a,b;}); /* OK, but strange */
j = sizeof( struct S );  /* Probably illegal because
                            struct S is still not defined. */
```

We don't think this is a deficiency, but would like to see the implementation issue a warning on the first line.

In Draft Proposed ANSI C, function declarators in type names may have prototype information.

6. C-Ref: Conversions and Representations

Most programming languages are designed to hide from the programmer the details of the language's implementation on a particular computer. For the most part, the C programmer need not be aware of these details, either, although a major attraction of C is that it allows the programmer to go below the abstract language level and expose the underlying representation of programs and data. With this freedom comes a certain amount of risk: some C programmers inadvertently descend below the abstract programming level and build into their programs nonportable assumptions about data representations.

This chapter has three purposes. First, it discusses some characteristics of data and program representations, indicating how the choice of representations can affect a C program. Second, it discusses in some detail the conversion of values of one type to another, emphasizing the characteristics of C that are portable across implementations. Finally, it presents the "usual conversion rules" of C, which are the conversions that happen automatically when expressions are evaluated.

6.1. C-Ref: Representational Issues

This section discusses the representation of programs and data and how the choice of representations can affect C programs and C implementations.

6.1.1. C-Ref: Storage Units and Data Sizes

All data objects in C are represented at run time in the computer's memory in an integral number of abstract *storage units*, or *bytes*. Each storage unit is in turn made up of some fixed number of *bits*, each of which can assume either of two values, typically denoted 0 and 1.

By definition, the *size* of a data object is the number of storage units (bytes) occupied by that data object. A storage unit is taken to be the amount of storage occupied by one character; the size of an object of type `char` is therefore `1`.

Because all data objects of a given type occupy the same amount of storage, we can also refer to the size of a *type* as the number of storage units occupied by an object of that type. The `sizeof` operator may be used to determine the size of a data object or type. We say that a type is "longer" or "larger" than another type if its size is greater. Similarly we say that a type is "shorter" or "smaller" than another type if its size is less. Draft Proposed ANSI C requires certain minimum sizes for the integer and floating-point types and provides implementation-defined header files `limits.h` and `float.h` that define the sizes.

The following program can be used to determine the sizes of the principal C data types:

```
#include<stdio.h>
int main()
{
  printf("\tType sizes:\n");
  printf("char\tshort\tint\tlong\tfloat\tdouble\n");
  printf("%3d\t%3d\t%3d\t%3d\t%3d\t%3d\n",
        sizeof(char), sizeof(short), sizeof(int),
        sizeof(long), sizeof(float), sizeof(double) );
}
```

6.1.2. C-Ref: Addressing Structure and Byte Ordering

The addressing structure of a computer determines how storage pieces of various sizes are named by pointers. The addressing model most natural for C is one in which each character (byte) in the computer's memory can be individually addressed. Computers using this model are called "byte-addressable" computers. The address of a larger piece of storage—one used to hold an integer or a floating-point number, for example—is typically the same as the address of the first character in the larger unit. (The "first" character is the one with the lowest address.)

Even within this simple model, computers differ in their storage "byte order," that is, they differ in which byte of storage they consider to be the "first" one in a larger piece. In "right-to-left" or "little endian" architectures, which includes the PDP-11 and VAX computers and the National Semiconductor 32000 microprocessor, the address of a 32-bit integer is also the address of the low-order byte of the integer. In "left-to-right" or "big endian" architectures, which includes most IBM architectures, the Intel 8086 microprocessor family, and the Motorola 68000 microprocessor family, the address of a 32-bit integer is the address of the high-order byte of the integer. These different conventions are pictured in table "C-Ref: Addressing a 32-Bit Integer At Address A".

6.1.2.1. C-Ref: Addressing a 32-Bit Integer At Address A

```
"Left-to-right"
        high-order          low-order
        +---+---+---+---+---+---+---+---+
        |                               |
        +---+---+---+---+---+---+---+---+
        A       A+1     A+2     A+3     Increasing addresses -->

"Right-to-left"
                        high-order          low-order
                        +---+---+---+---+---+---+---+---+
                        |                               |
                        +---+---+---+---+---+---+---+---+
<-- Increasing addresses        A+3     A+2     A+1     A
```

Programs that assume a particular byte order will not be portable. For example, here is a program that determines a computer's byte ordering by using a union in

a nonportable fashion. The union has the same size as an object of type long and is initialized so that the low-order byte of the union contains a 1 and all other bytes contain zeros. In right-to-left architectures, the character component, Char, of the union will be overlaid on the low-order byte of the long component, Long, whereas in left-to-right architectures Char will be overlaid on the high-order byte of Long:

```
#include <stdio.h>
union { long Long; char Char[sizeof(long)]; } u;
int main()
{
  u.Long = 1;
  if (u.Char[0] == 1)
    printf("Addressing is right-to-left\n");
  else if (u.Char[sizeof(long)-1] == 1)
    printf("Addressing is left-to-right\n");
  else printf("Addressing is strange\n");
  return 0;
}
```

Components of a structure type are allocated in the order of increasing addresses, that is, either left-to-right or right-to-left depending on the byte order of the computer. Because bit fields are also packed following the byte order, it is natural to number the bits in a piece of storage following the same convention. Thus, in a left-to-right computer the most significant (leftmost) bit of a 32-bit integer would be bit number 0 and the least significant bit would be bit number 31. In right-to-left computers the least significant (rightmost) bit would be bit 0, and so forth.

6.1.3. C-Ref: Alignment Restrictions

Some computers allow data objects to reside in storage at any address, regardless of the data's type. Others impose *alignment restrictions* on certain data types, requiring that objects of those types occupy only certain addresses. It is not unusual on a byte-addressed computer, for example, to require that 32-bit (4-byte) integers be located on addresses that are a multiple of four. In this case we say that the "alignment modulus" of those integers is four. Failing to obey the alignment restrictions can result either in a run-time error or in unexpected program behavior. Even when there are no alignment restrictions per se, there may be a performance penalty for using data on unaligned addresses, and therefore a C implementation may align data purely for efficiency.

The C programmer is not normally aware of alignment restrictions because the compiler takes care to place data on the appropriate address boundaries. However, C does give the programmer the ability to violate alignment restrictions by casting pointers to different types.

In general, if the alignment requirement for a type S is at least as stringent as that for a type D (that is, the alignment modulus for S is no smaller than the alignment modulus for D) then converting a "pointer to type S" to a "pointer to type D" is safe. "Safe" here means that the resulting pointer to type D will work

as expected if used to fetch or store an object of type *D*, and that a subsequent conversion back to the original pointer type will recover the original pointer. A corollary to this is that any data pointer can be converted to type `char *` and back safely. (In Draft Proposed ANSI C, the new type "`void *`" is used as the universal data pointer. See the section "C-Ref: ANSI C Generic Pointers". Functions such as `malloc` (section "C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**") which return pointers destined to be cast to various types, always return pointers of type aligned on a boundary suitable for an object of any type.

If the alignment requirement for a type *S* is less stringent than that for type *D*, then the conversion from a "pointer to type *S*" to a "pointer to type *D*" could result in either of two kinds of unexpected behavior. First, an attempt to use the resulting pointer to fetch or store an object of type *D* may cause an error, halting the program. Second, the hardware or implementation may "adjust" the destination pointer to be legal, usually by forcing it back to the nearest previous legal address. A subsequent conversion back to the original pointer type may not recover the original pointer.

The following program deliberately provokes alignment problems by casting pointers from type `char *` to type `long *`. The character pointer is advanced by four 1-byte increments, thus guaranteeing that the `long` pointer will be misaligned. (We assume that type `long` occupies four 8-bit bytes.)

```
#include <stdio.h>
long array[2] = {0x01020304L, 0x05060708L};
int main()
{
  int i;
  char *char p = (char *) &array[0];
  for (i=0; i<=4; i++) {
    printf("*(long *)(char p+%d) = 0x%08x\n",
           i, *(long *)(char p+i) );
  }
  return 0;
}
```

The numbers output by this program will differ depending on whether the computer has left-to-right or right-to-left byte ordering and on whether there are alignment restrictions on type `long`:

1. If there are no alignment restrictions then the output will be:

Heading	*Left-right order*	*Right-left order*
`*(long *)(char p+0) =`	`0x01020304`	`0x01020304`
`*(long *)(char p+1) =`	`0x02030405`	`0x08010203`
`*(long *)(char p+2) =`	`0x03040506`	`0x07080102`
`*(long *)(char p+3) =`	`0x04050607`	`0x06070801`
`*(long *)(char p+4) =`	`0x05060708`	`0x05060708`

2. If there are strict alignment requirements and the hardware or software rejects misaligned addresses then the output might be:

Heading	*Left-right order*	*Right-left order*
`*(long *)(char p+0) =`	`0x01020304`	`0x01020304`
<<error>>	<<error>>	<<error>>

3. If there are strict alignment requirements and the hardware or software adjusts misaligned addresses then the output will be still different. Here's how it might be if the misaligned address is forced to the next lower permissible boundary:

Heading	*Left-right order*	*Right-left order*
`*(long *)(char p+0) =`	`0x01020304`	`0x01020304`
`*(long *)(char p+1) =`	`0x01020304`	`0x01020304`
`*(long *)(char p+2) =`	`0x01020304`	`0x01020304`
`*(long *)(char p+3) =`	`0x01020304`	`0x01020304`
`*(long *)(char p+4) =`	`0x05060708`	`0x05060708`

6.1.4. C-Ref: Pointer Sizes

There is no requirement in C that any of the integral types be large enough to represent a pointer, although C programmers often assume that type `long` is large enough, which it is on most computers.

In many C implementations it happens that pointers have the same size as type `int`. Because `int` is also the default type specifier, some programmers are careless with pointer/integer conversions. In particular, they may omit the return type when declaring or defining functions returning pointers, knowing the type will default to `int`, which is "good enough" because of C's standard conversion rules. This has proved to be a frequent source of portability problems.

Although function pointers are usually no larger than data pointers (at least than character pointers), there are a few computers on which this is not true. For maximum portability, programmers should not use data pointers (such as `char *` , the canonical generic data pointer) to hold pointers to functions. Use a function pointer type that specifies the correct return type, such as `double (*)()`. In Draft Proposed ANSI C there is no generic function pointer type; function pointers must be correctly typed or the results can be unpredictable.

6.1.5. C-Ref: Difficult Addressing Models

Some computers represent data and addresses in forms that are very awkward for C implementations. Major problems can occur when the computer's natural word size is not a multiple of its natural byte size. Suppose—this is a real example—our computer has a 36-bit word and represents characters in 7 bits; each word can hold five characters with one bit remaining unused. All noncharacter datatypes occupy one or more full words. This memory structure will be very difficult for a C implementor, because C programming relies upon the ability to map any data structure onto an array of characters. That is, to copy an object of type T at address A it should be sufficient to copy `sizeof(`T`)` characters beginning at A. The only alternative for the implementor on this computer would be to represent characters using some nonstandard number of bits (for example, 9 or 36) so that they

fit tightly into a word. This representation could have a significant performance penalty.

A second problem occurs on "word-addressed" computers, those whose basic addressable storage unit is larger than a single character. On these computers, there may or may not be a special kind of address, a "byte pointer," that can represent characters within a word. Assuming there is such a byte pointer, it may very well be larger than a pointer to objects of noncharacter types. A C implementor must decide whether to pay the increased overhead of representing all pointers as byte pointers or whether to use the larger format only for objects of type `char *`. The latter decision will force C programmers to be more careful about pointer conversions.

6.2. C-Ref: Conversions

The C language provides for values of one type to be converted to values of other types under several circumstances.

- A cast expression may be used to explicitly convert a value to another type.
- An operand may be implicitly converted to another type in preparation for performing some arithmetic or logical operation.
- An object of one type may be assigned to a location (lvalue) of another type, causing an implicit type conversion.
- An actual argument to a function may be implicitly converted to another type prior to the function call.
- A return value from a function may be implicitly converted to another type prior to the function return.

There are restrictions as to what types a given object may be converted to. Furthermore, the set of conversions that are possible on assignment, for instance, is not the same as the set of conversions that are possible with type casts.

In the following sections we will discuss the set of possible conversions and then discuss which of these conversions are actually performed in each of the circumstances listed above.

6.2.1. C-Ref: Representation Changes

The representation of a data object is the particular pattern of bits in the storage area that holds the object; this pattern distinguishes the value of the object from all other possible values of that type.

A conversion of a value from one type to another may or may not involve a representation change. For instance, whenever the two types have different sizes, a rep-

resentation change has to be made. When integers are converted to a floating-point representation, a representation change is made even if the integer and floating-point type have the same sizes. However, when a value of type `int` is converted to type `unsigned int`, a representation change may not be necessary (it isn't necessary if signed integers are represented in two's-complement form).

Some representation changes are very simple, involving merely discarding of excess bits or padding with extra 0 bits. Other changes may be very complicated, such as conversions between integer and floating-point representations. For each of the conversions discussed in the following sections, we describe the possible representation changes that may be required.

6.2.2. C-Ref: Trivial Conversions

It is always possible to convert a value from a type to another type that is the same as the first type. See section "C-Ref: Type Equivalence" for a discussion of when types are the same. No representation change needs to occur in this case.

Most implementations will refuse to convert structure or union types to themselves, because no conversions to structure or union types are normally permitted.

6.2.3. C-Ref: Conversions to Integer Types

The types that may be converted to integers are the arithmetic types and the pointer types.

From Integer Types The general rule for converting from one integer type to another is that the mathematical value of the result should equal the original mathematical value if that is possible. For example, if an unsigned integer has the value `15` and this value is to be converted to a signed type, the resulting signed value should be `15` also.

If it is not possible to represent the original value of an object of the new type, then there are two cases. If the result type is a signed type, then the conversion is considered to have overflowed and the result value is technically not defined (but see the discussion below). If the result type is an unsigned type, then the result must be that unique value of the result type that is equal (congruent) mod 2^n to the original value, where n is equal to the number of bits used in the representation of the result type.

These rules have a number of interesting consequences. When a signed integer is converted to an unsigned integer of the same size, no change of representation is needed if signed integers are represented using two's-complement notation; that is, the resulting unsigned integer will have the same bit pattern as the original signed integer. On the other hand, if signed integers are represented in some other way, such as with one's-complement or sign-magnitude representation, then a change of representation will be necessary.

When an unsigned integer is converted to a signed integer of the same size, no change of representation is needed if signed integers are represented using two's-complement notation. Technically, this conversion is considered to overflow if the

original value is too large to represent exactly in the signed representation (that is, if the high-order bit of the unsigned number is 1). However, we do not doubt that many programmers and many programs depend on the conversion being performed quietly and with no change of representation to produce a negative number. If signed integers are represented in some other way, such as with one's-complement or sign-magnitude representation, then a change of representation will be necessary. Moreover, the transformation may not be mathematically straightforward. For example, when converting the value 0 there may be a choice of "+0" or "-0" in the result representation, and when converting the unsigned value 2^{n-1} it may not be clear what the result value should be. The best the implementor can do in such cases is to make some rational decision and then document it carefully.

If the destination type is longer than the source type, then the only case in which the source value will not be representable in the result type is when a negative signed value is converted to a longer, unsigned type. In that case, the conversion must necessarily behave as if the source value were first converted to a longer signed type of the same size as the destination type, and then converted to the destination type. For example, since the constant expression `-1` has type `int`,

```
((unsigned long) -1) == ((unsigned long) ((long) -1)))
```

If the destination type is shorter than the source type, and both the original type and the destination type are unsigned, the conversion can be effected simply by discarding excess high-order bits from the original value; the bit pattern of the result representation will be equal to the n low-order bits of the original representation. This same rule of discarding works for converting signed integers in two's-complement form to a shorter unsigned type. The discarding rule is also one of several acceptable methods for converting signed or unsigned integers to a shorter signed type when signed integers are in two's-complement form. Note that this rule will not preserve the sign of the value in case of overflow, but the action on overflow is not officially defined anyway. When signed integers are not represented in two's-complement form, the conversions are necessarily more complicated. While the C language does not require the two's-complement representation for signed integers, it certainly favors that representation, and implementations should use it whenever feasible.

From Floating-point Types The conversion of a floating-point value to an integral value should produce a result that is (if possible) equal in value to the value of the old object. If the floating-point value has a nonzero fractional part, that fraction should be discarded, that is, conversion normally involves truncation of the floating-point value.

The behavior of the conversion is undefined if the floating-point value cannot be represented even approximately in the new type (for example, if its magnitude is much too large, or if a negative floating-point value is converted to an unsigned integer type). The handling of overflow and underflow is left to the discretion of the implementor.

From Pointer Types When the source value is a pointer, the pointer is treated as if it were an unsigned integer of a size equal to the size of the pointer. Then the unsigned integer is converted to the destination type using the rules listed above.

There is one special case: If a null pointer of any type is created through assignment or initialization by the integer constant 0, then the conversion of that null pointer to an integer type must yield the value 0. This requirement may cause a change in representation in some implementations.

C programmers have traditionally assumed that pointers could be converted to type `long` and back without loss of information. Although this is almost always true, it is not required by the language definition. (It is definitely wrong to assume that type `int` is large enough to hold a pointer.) Some computers may have pointer representations (especially for character or function pointers) that are longer than the largest integer type. Programmers needing a "generic" data pointer type should use `char *` instead of `long`. In Draft Proposed ANSI C the pointer type `void *` was invented as a generic data pointer type.

6.2.4. C-Ref: Conversions to Floating-point Types

Only arithmetic types may be converted to floating-point types.

From Floating-point Types When converting from `float` to `double`, the result should have the same value as the original value. (This may be viewed as a restriction on the choice of representations for the floating-point types.)

When converting from `double` to `float`, such that the original value is within the range of values representable as type `float`, the result should be one of the two `float` values closest to the original value. (Whether the original value is rounded up or down is implementation dependent.)

If the original value is outside the range of values representable as type `float`—as when the magnitude is too large or too small for the representation of `float`—the resulting value is undefined, as is the overflow or underflow behavior of the program.

From Integer Types If the integer value is exactly representable in the floating-point type, then the result is the equivalent floating-point value. If the integer value is not exactly representable but is within the range of values representable in the floating-point type, then one of the two closest floating-point values should be chosen as the result. If the integer value is outside the range of values representable in the floating-point type, the result is undefined.

6.2.5. C-Ref: Conversions to Structure and Union Types

An object of a structure or union type T may be converted only to a type that is the same as T (the trivial conversion). There is no change of representation, except that the bit patterns in any unused "holes" in the structure or union are not necessarily preserved.

In fact, most implementations will refuse to convert structure or union types to themselves, because no conversions to structure or union types are normally permitted.

6.2.6. C-Ref: Conversions to Enumeration Types

The rules are the same as for conversions to integral types. Some permissible conversions, such as between enumeration and floating-point types, may be symptoms of a poor programming style.

6.2.7. C-Ref: Conversions to Pointer Types

In general, pointers and integers may be converted to pointer types. There are special circumstances under which an array or a function will be converted to a pointer.

From Pointer Types A null pointer of any type may be converted to any other pointer type and it will still be recognized as a null pointer. There may be a representation change in the conversion.

A value of type "pointer to *S*" may be converted to type "pointer to *D*" for any types *S* and *D*. However, the behavior of the resulting pointer may be affected by any alignment restrictions in the implementation.

From Integer Types The integer constant 0 may always be converted to a pointer type. The conversion may or may not involve a representation change, regardless of the relative sizes of `int` and the pointer type. The result of such a conversion is a "null pointer" that is different from any legal pointer to a data object. Null pointers of different pointer types may have different internal representations in some implementations.

Integers other than the constant 0 may be converted to pointer type, but the result is nonportable. The intent is that the pointer be considered an unsigned integer (of the same size as the pointer) and the standard integer conversions then be applied to take the source type to the destination type.

From Array Types An expression of type "array of *T*" for some type *T* is converted to a value of type "pointer to *T*" by substituting a pointer to the first element of the array for the array itself. This occurs in all contexts except when the array is an argument to `sizeof`.

From Function Types An expression of type "function returning *T*" for some type *T* is converted to a value of type "pointer to function returning *T*" by substituting a pointer to the function for the function itself. The conversion occurs implicitly in all contexts except the function expression in a function call.

6.2.8. C-Ref: Conversions to Array and Function Types

No conversions to array or function types are possible. In particular, it is *not* permissible to convert between array types or between function types:

```
extern int f();
double d;
d = (( double () )f) ();  /* illegal */
d = (double) f();         /* OK */
d = (*(double (*)()) f)();
        /* legal, but will have unexpected results */
```

In the last example, the address of `f` is converted to a pointer to a function returning type `double`; that pointer is then dereferenced and the function called. This is legal, but the resulting value stored in `d` will probably be garbage unless `f` was really defined (contrary to the external declaration above) to have type `double ()`.

6.2.9. C-Ref: Conversions to the Void Type

Any value may be converted to type `void`. Of course, the result of such a conversion cannot be used for anything. Such a conversion may occur only in a context where an expression value will be discarded, such as in an expression statement.

6.3. C-Ref: The Usual Conversions

6.3.1. C-Ref: The Casting Conversions

Any of the conversions discussed earlier in this chapter may be explicitly performed with a type cast without error. Here are some examples of legal cast expressions:

```
int i, *ip;
char c, *cp;
float f;
double d;
enum E { red=1, blue=2, green=3 } color;
...
ip = (int *)  cp;
ip = (int *)  ip;    /* Trivial conversion. */
cp = (char *) ip;
i  = (int)    color;
color = (enum E) ((int) d);
```

6.3.2. C-Ref: The Assignment Conversions

In a simple assignment expression, the types of the expressions on the left and right sides of the assignment operator should be the same. If they are not, an attempt will be made to convert the value on the right side of the assignment to the type on the left side. The conversions that are legal—a subset of the possible conversions—are listed below.

Left Side Type	*Right Side Type*
any arithmetic type	any arithmetic type

any pointer type	the integer constant 0
pointer to T	array of T
pointer to function	function

Attempting any other conversion may elicit a warning or error, but some compilers permit any casting conversion. Draft Proposed ANSI C permits the above conversions but adds some additional restrictions.

6.3.3. C-Ref: The Usual Unary Conversions

The usual unary conversions determine whether and how a single operand is converted before an operation is performed. The conversions are applied automatically to operands of the unary `!`, `-`, `~`, and `*` operators. They are also applied to each of the operands of the binary `<<` and `>>` operators. (Despite the fact that `<<` and `>>` are binary operators, they do not perform the usual binary conversions. They merely perform the usual unary conversions on each operand separately.) Finally, the usual unary conversions are applied to actual arguments in a function call before the call is performed.

The purpose of these conversions is to reduce the large number of arithmetic and pointer types to a smaller number that must be handled by the operators. The conversions are of two kinds. First, arithmetic values of narrow type are widened to a larger type with the same value. For example, `short` integers are widened to type `int` and floating-point numbers of type `float` are widened to type `double`. Second, pointer-like references to arrays and functions are converted into actual pointers.

Original Operand Type	*Converted Type*
`char`, `short`	`int`
`unsigned char`	`unsigned`
`unsigned short`	`unsigned`
`float`	`double`
"array of T"	"pointer to T"
"function returning T"	"pointer to function returning T"

An operand of any other type is unchanged.

The integral conversions listed above are, we believe, the most common in C implementations. Draft Proposed ANSI C has a slightly different set of conversions, which could be used as an alternative even in current C implementations.

Some implementations of C provide an optional compilation mode in which the implicit conversion of type `float` to type `double` is suppressed. This is not compatible with the original C language, but can be very useful if high-quality, efficient numerical software is to be written. (This compilation mode would be compatible with Draft Proposed ANSI C.)

6.3.4. C-Ref: The Usual Binary Conversions

The usual binary conversions determine whether and how operands are converted before a binary or ternary operation is performed. They are applied to the operands of most binary operators and to the second and third operands in a conditional expression. The purpose of these conversions is to reduce the large number of arithmetic and pointer types to a smaller number that must be handled by the operators. When two values must be operated upon in combination, they are first converted to a single common type (and typically the result is also of that same common type).

An operator that performs "the usual binary conversions" on its two operands first performs the usual unary conversions on each of the operands independently (to widen short values and to convert arrays and functions to pointers) and then effectively performs one of the following conversions on the results of the unary conversions, all this being done before the operation itself is executed:

1. If either operand is not of arithmetic type, or if the two operands have the same type, then no additional conversion is performed.

2. Otherwise, if one operand is of type `double`, then the other operand is converted to type `double`.

3. Otherwise, if one operand is of type `unsigned long int`, then the other operand is converted to type `unsigned long int`.

4. Otherwise, if one operand is of type `long int` and the other operand is of type `unsigned int`, then each of the two operands is converted to type `unsigned long int`.

5. Otherwise, if one operand is of type `long int`, then the other operand (which now must be of type `int`) is converted to type `long int`.

6. Otherwise, if one operand is of type `unsigned int`, then the other operand is converted to type `unsigned int`.

7. Otherwise, both operands must be of type `int`, and so no additional conversion is performed.

Rule number 4 is omitted in some implementations, including Draft Proposed ANSI C. When it is omitted, the combination of types `long int` and `unsigned int` result in type `long int` instead of `unsigned long int`. Here is a small program that determines which conversion is in effect:

```
unsigned int UI = -1;
long int LI = 0;
int main()
{
  if (UI < LI) printf("long+unsigned==long\n");
  else printf("long+unsigned==unsigned\n");
  return 0;
}
```

Some implementations of C provide an optional compilation mode in which the implicit conversion of type `float` to type `double` is suppressed. This is not compatible with the original C language, but can be very useful if high-quality, efficient numerical software is to be written. This conversion rule is present in Draft Proposed ANSI C, which also has other subtle changes.

6.3.5. C-Ref: The Function Argument Conversions

When an expression appears as an argument in a function call, the result of the expression is adjusted using the usual unary conversions before being passed as an actual argument.

As described in the previous section, some implementations of C provide an optional compilation mode in which the implicit conversion of type `float` to type `double` is suppressed during the usual unary and binary conversions. When this mode is in effect, the conversion of function arguments of type `float` to type `double` may still be performed. None of the standard C library routines are prepared to accept arguments of type `float`, because they all expect the automatic conversion to `double` to occur.

In Draft Proposed ANSI C, arguments to functions specified with prototypes are converted to the specified parameter types; the "usual" argument conversions are not relevant. When a prototype is not present, the usual conversions are applied, including the promotion of `float` to `double`.

6.3.6. C-Ref: Other Function Conversions

The declared types of the formal parameters of a function, and the type of its return value, are subject to certain adjustments that parallel the function argument conversions. They are discussed in section "C-Ref: Adjustments to Parameter Types".

7. C-Ref: Expressions

C has an unusually rich set of operators that provide access to most of the operations provided by the underlying hardware. This chapter presents the syntax of expressions and describes the function of each of the operators.

7.1. C-Ref: General Comments

7.1.1. C-Ref: Objects and LValues

An *object* is a region of memory that can be examined and stored into. An *lvalue* (pronounced "ell-value") is an expression that refers to an object in such a way that the object may be altered as well as examined. Sometimes we also speak of the result computed by such an expression as being an lvalue. (An expression or result is called an lvalue because it may be used on the left-hand side of an assignment. Similarly, an expression that permits examination but not alteration of a value is sometimes called an *rvalue*, because it can be used on the right-hand side of an assignment.)

Some names are lvalues: names of variables declared to be of arithmetic, pointer, enumeration, structure, or union type. Names of functions, names of arrays, and enumeration constants, on the other hand, are not lvalues. Some operations on non-lvalues can produce lvalues. For example, the name of an array is not an lvalue but references to elements of the array using subscripting expressions are lvalues. (In other words, one cannot modify an entire array variable using assignment, but one can modify individual elements.) Besides names, the following forms of expressions may produce lvalues:

1. Every subscript expression *e*[*k*] is an lvalue, regardless of whether or not the expressions *e* and *k* are lvalues.

2. A parenthesized expression is an lvalue if and only if the contained expression is an lvalue.

3. A direct component selection expression *e*.*name* is an lvalue if and only if the expression *e* is an lvalue.

4. An indirect component selection expression *e*->*name* is always an lvalue, regardless of whether *e* is an lvalue.

5. An indirection expression **e* is always an lvalue, regardless of whether *e* is an lvalue.

No other form of expression can produce an lvalue. In particular, the result of an assignment expression is never an lvalue, and a function call cannot produce an lvalue.

Some operators require certain of their operands to be lvalues:

1. The operand of a unary address operator `&` must be an lvalue.

2. The operand of a unary `++` or `--` operator, whether prefix or postfix, must be an lvalue.

3. The left operand of any assignment operator must be an lvalue.

No other operator requires an lvalue as an operand.

7.2. C-Ref: Expressions and Precedence

7.2.1. C-Ref: Precedence and Associativity of Operators

The grammar for expressions presented in this chapter completely specifies the precedence of operators in C. However, it may be convenient to summarize the rules here.

Each expression operator in C has a precedence level and a rule of associativity. Where parentheses do not explicitly indicate the grouping of operands with operators, it may appear that an operand could be grouped with either of two operators. In such cases the grouping is determined by the rules of precedence and associativity: The operand is grouped with the operator having higher precedence, but if the two operators have the same precedence, the operand is grouped with the left or right operator according to whether the operators are left-associative or right-associative. (All operators having the same precedence level always have the same associativity.) For example, in the expression

```
a * b + c
```

the operand `b` is grouped with the multiplication operator `*`, because `*` has higher precedence than `+`, and so the expression is treated as if it had been written

```
(a * b) + c
```

Similarly, in the expression

```
a += b != c
```

the operand `b` is grouped with the operator `!=`, because `!=` has higher precedence than `+=`, and so the expression is treated as if it had been written

```
a += (b != c)
```

In the case of the expression

```
a - b + c
```

the two operators `-` and `+` have the same precedence and are left-associative, so the operand `b` is grouped with the operator to its left, resulting in the interpretation

```
(a - b) + c
```

rather than

```
a - (b + c)
```

On the other hand, in the case of the expression

```
a = b += c
```

the two operators = and += have the same precedence and are right-associative, so the operand b is grouped with the operator to its right, resulting in the interpretation

```
a = (b += c)
```

The table "C-Ref: C Operators in Order of Precedence" contains a concise list of the C operators in order from the highest to the lowest precedence. The operators are presented in this chapter in decreasing order of precedence.

All of the binary and ternary operators are left-associative except for the conditional and assignment operators, which are right-associative. The unary and postfix operators are sometimes described as being right-associative, but this is needed only to express the idea that an expression such as *x++ is interpreted as *(x++) rather than as (*x)++. We prefer simply to state that the postfix operators have higher precedence than the (prefix) unary operators.

In Draft Proposed ANSI C there is also a unary plus operator and a string concatenation expression (with no explicit operator).

7.2.1.1. C-Ref: C Operators in Order of Precedence

primary and postfix expressions		
16	*names literals*	simple tokens
16	*a*[*k*]	subscripting
16	*f*(...)	function call
16	.	direct selection
16	->	indirect selection
15	++ --	postfix increment/decrement
unary operators		
14	++ --	prefix increment/decrement
14	sizeof	size
14	(*type name*)	casts (type conversion)
14	~	bitwise not
14	!	logical not
14	-	arithmetic negation
14	&	address of
14	*	indirection
binary and ternary operators		
13L	* / %	multiplicative
12L	+ -	additive
11L	<< >>	left and right shift
10L	< > <= >=	relational
9L	== !=	equality/inequality
8L	&	bitwise and

<table>
<tr><td>7L</td><td>^</td><td>bitwise xor</td></tr>
<tr><td>6L</td><td>|</td><td>bitwise or</td></tr>
<tr><td>5L</td><td>&&</td><td>logical and</td></tr>
<tr><td>4L</td><td>||</td><td>logical or</td></tr>
<tr><td>3R</td><td>? :</td><td>conditional (ternary)</td></tr>
<tr><td>2R</td><td>= += -= *= /= %=</td><td>assignment</td></tr>
<tr><td>2R</td><td><<= >>= &= ^= |=</td><td></td></tr>
<tr><td>1L</td><td>,</td><td>sequential evaluation</td></tr>
</table>

The numbers on the left indicate relative precedence; larger numbers indicate higher precedence. "L" indicates left-associative operators and "R" indicates right-associative operators.

7.2.2. C-Ref: Overflow and Other Arithmetic Exceptions

For certain operations in C such as addition and multiplication, it may be that the true mathematical result of the operation cannot be represented as a value of the expected result type (as determined by the usual conversion rules). This condition is called overflow (or, in some cases, underflow).

In general, the C language does not specify the consequences of overflow. One possibility is that an incorrect value (of the correct type) is produced. Another possibility is that program execution is terminated. A third possibility is that some sort of machine-dependent trap or exception occurs that may be detected by the program in some implementation-dependent manner.

For certain other operations, the C language explicitly specifies that the effects are unpredictable for certain operand values or (more stringently) that a value is always produced but the value is unpredictable for certain operand values. If the right-hand operand of the division operator / or the remainder operator, %, is zero, then the effects are unpredictable, as for overflow. If the right-hand operand of a shift operator << or >> is too large or negative, then an unpredictable value is produced.

Traditionally, all implementations of C have ignored the question of signed integer overflow, in the sense that the result is whatever value is produced by the machine instruction used to implement the operation. (Many computers that use a two's-complement representation for signed integers handle overflow of addition and subtraction simply by producing the low-order bits of the true two's-complement result. No doubt many existing C programs depend on this fact, but such code is technically not portable.) Floating-point overflow and underflow is usually handled in whatever convenient way is supported by the machine; if the machine architecture provides more than one way to handle exceptional floating-point conditions, a library function may be provided to give the C programmer access to such options.

For unsigned integers the C language is quite specific on the question of overflow: Every operation on unsigned integers always produces a result value that is congruent mod 2^n to the true mathematical result of the operation (where n is the number of bits used to represent the unsigned result). This amounts to computing

the correct *n* low-order bits of the true result (of the true two's-complement result if the true result is negative, as when subtracting a big unsigned integer from a small one).

As an example, suppose that objects of type `unsigned` are represented using 16 bits; then subtracting the unsigned value `7` from the unsigned value `4` would produce the unsigned value `65533`; that is, 2^{16}-3, because this value is congruent mod 2^{16} to the true mathematical result -3.

An important consequence of this rule is that operations on unsigned integers are guaranteed to be completely portable between two implementations that use representations having the same number of bits; moreover, any implementation can easily simulate the unsigned arithmetic of another implementation using some smaller number of bits.

Despite the explicit rule for the handling of overflow, the operations of division and remainder on unsigned integers nevertheless have unpredictable effects, as for signed integers, when the the right-hand operand is 0.

7.3. C-Ref: Primary Expressions

There are three kinds of primary expressions: names (identifiers), literal constants, and parenthesized expressions.

primary-expression :
  name
  literal
  parenthesized-expression

Function calls, subscript expressions, and component selection expressions were traditionally listed as primary expressions in C, but we have included them in the next section with the postfix expressions.

7.3.1. C-Ref: Names

The value of a name depends on its type. The type of a name is determined by the declaration of that name (if any), as discussed in the chapter "C-Ref: Declarations".

The name of a variable declared to be of arithmetic, pointer, enumeration, structure, or union type evaluates to an object of that type; the name, considered as an expression, is an lvalue.

An enumeration constant evaluates to the integer value associated with that enumeration constant; it is not an lvalue. In the example below, the six color names are enumeration constants. The `switch` statement (described in section "C-Ref: Switch Statement; Case and Default Labels") selects one of six statements to execute based on the value of the variable `color`.

```
typedef enum { red, blue, green, cyan,
               yellow, magenta } colortype;

static colortype complementary(color)
  colortype color;
{
    switch (color) {
      case red:         return cyan;
      case blue:        return yellow;
      case green:       return magenta;
      case cyan:        return red;
      case yellow:      return blue;
      case magenta:     return green;
    }
}
```

The name of an array evaluates to that array; it is not an lvalue. In contexts where the result is subject to the usual conversions, the array value is immediately converted to a pointer to the first object in the array. This occurs in all contexts except as the argument to the `sizeof` operator, in which case the size of the array is returned, rather than the size of a pointer.

```
{
    extern void PrintMatrix();
    int Matrix[10][10], total length, row length;

    total length = sizeof Matrix;
    row length = sizeof Matrix[0];
    PrintMatrix(Matrix);   /* pointer to first
                              element is passed */
}
```

The name of a function evaluates to that function; it is not an lvalue. In contexts where the result is subject to the usual conversions, the function value is immediately converted to a pointer to the function. This occurs in all contexts but two: as the argument to the `sizeof` operator, where a function is illegal, and as the function in a function call expression, in which case the function itself is desired, and not a pointer to it.

```
#include <math.h>
        /* Declares sin and cos functions. */

void PlotFunction(f, x0, x1)
  double (*f)(), x0, x1;
{
    ...
}
```

```
double fn(x)  /* Function to be plotted. */
  double x;
{
    return (sin(x) - 2.0 * cos(x));
}

void main()
{
    PlotFunction(fn, 0.01, 100.0);
                        /* fn becomes a pointer */
}
```

It is not possible for a name, as an expression, to refer to a label, typedef name, structure component name, union component name, structure tag, union tag, or enumeration tag. Names used for those purposes reside in name spaces separate from the names that can be referred to by a name in an expression. Some of these names may be referred to within expressions by means of special constructs. For example, structure and union component names may be referred to using the `.` or -> operators, and typedef names may be used in casts and as an argument to the `sizeof` operator.

7.3.2. C-Ref: Literals

A literal (lexical constant) is a numeric constant, and when evaluated as an expression yields that constant as its value. A literal expression is never an lvalue. See section "C-Ref: Constants" for a discussion of literals and their types and values.

7.3.3. C-Ref: Paranthesized Expressions

A parenthesized expression consists of a left parenthesis, any expression, and then a right parenthesis.

parenthesized-expression :
    (*expression*)

The type of a parenthesized expression is identical to the type of the enclosed expression; no conversions are performed. The value of a parenthesized expression is the value of the enclosed expression, and will be an lvalue if and only if the enclosed expression is an lvalue.

The purpose of the parenthesized expression is simply to delimit the enclosed expression for grouping purposes, either to defeat the default precedence of operators or to make code more readable. For example:

```
x1 = (-b + discriminant)/(2.0 * a)
```

Parentheses do not necessarily force a particular evaluation order. See section "C-Ref: Order of Evaluation".

7.4. C-Ref: Postfix Expressions

There are seven kinds of postfix expressions: subscripting expressions, two forms of component selection (direct and indirect), function calls, postincrement expressions and postdecrement expressions.

postfix-expression :
 primary-expression
 subscript-expression
 component-selection-expression
 function-call
 postincrement-expression
 postdecrement-expression

Function calls, subscript expressions, and component-selection expressions were traditionally listed as primary expressions, but their syntax is more related to the postfix expressions.

7.4.1. C-Ref: Subscripting Expressions

A subscripting expression consists of a postfix expression, a left bracket, an arbitrary expression, and a right bracket. This construction is used for array subscripting, where the postfix expression (commonly an array name) evaluates to a pointer to the beginning of the array and the other expression to an integer offset.

subscript-expression :
 postfix-expression [*expression*]

In C, the expression $e_1[e_2]$ is by definition precisely equivalent to the expression $*((e_1)+(e_2))$. The usual binary conversions are applied to the two operands, and the result is always an lvalue. Note that the operand for the * operator must be a pointer, and the only way that the result of the + operator can be a pointer is for one operand to be a pointer and the other an integer, and therefore it follows that for $e_1[e_2]$ one operand must be a pointer and the other an integer. It is conventional for the first operand to be the pointer.

```
char buffer[100], *bptr = buffer;
int i = 99;
buffer[0] = '\0';
bptr[i] = bptr[0];
```

A consequence of the definition of subscripting is that arrays use 0-origin indexing. In the example above, the first element allocated for the 100-element array `buffer` is referred to as `buffer[0]`, and the last element as `buffer[99]`. The names `buffer` and `bptr` both point to the same place, namely `buffer[0]`, the first element of the `buffer` array, and they can be used in identical ways within subscripting expressions. However, `bptr` is a variable (an lvalue), and thus can be made to point to some other place:

```
bptr = &buffer[6];
```

after which the expression `bptr[-4]` refers to the same place as the expression `buffer[2]`. (This illustrates the fact that negative subscripts make sense in certain circumstances.) An assignment can also make `bptr` point to no place at all:

```
bptr = NULL; /* Store a null pointer into bptr. */
```

On the other hand, the array name `buffer` is not an lvalue and cannot be modified; considered as a pointer, it always points to the same fixed place.

Multidimensional array references are formed by composing subscripting operators, not by putting multiple expressions within the brackets.

```
{
#define SIZE 10
    double matrix[SIZE][SIZE];
    int i, j;
    /* Set up an identity matrix.  Note: this
       method is clear, but is not the fastest
       way to set up such a matrix. */
    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++)
            matrix[i][j] = ((i == j) ? 1.0 : 0.0);
    ...
}
```

It is bad programming style to use a comma expression within the subscripting brackets, because it might mislead a reader familiar with other programming languages to think that it means subscripting of a multidimensional array. For example, the expression

```
commands[k=n+1, 2*k]
```

might appear to be a reference to an element of a two-dimensional array named `commands` with subscript expressions `k=n+1` and `2*k`, whereas its actual interpretation in C is as a reference to a one-dimensional array named `commands` with subscript `2*k` after `k` has been assigned `n+1`. If a comma expression is really needed (and it is hard for us to think of a plausible example), enclose it in parentheses to indicate that it is something unusual:

```
commands[(k=n+1, 2*k)]
```

It is possible to use pointers and casts to refer to a multidimensional array as if it were a one-dimensional array. This may be desirable for reasons of efficiency. It must be kept in mind that arrays in C are stored in row-major order. Here is an example of such trickery:

```
{
#define SIZE 10
    double matrix[SIZE][SIZE];
    int i;
    /* Set up an identity matrix.  Tricky but fast.
       First clear the matrix, treating it as a
       one-dimensional array of length SIZE*SIZE.
     */
    for (i = 0; i < SIZE*SIZE; i++)
        ((double *) matrix)[i] = 0.0;
    /* Now install the 1.0 elements. */
    for (i = 0; i < SIZE*SIZE; i += (SIZE + 1))
        ((double *) matrix)[i] = 1.0;
    ...
}
```

7.4.2. C-Ref: Component Selection

Component selection operators are used to access fields (components) of structure and union types.

component-selection-expression :
  direct-component-selection
  indirect-component-selection

direct-component-selection :
  postfix-expression . *name*

indirect-component-selection :
  postfix-expression -> *name*

A direct component selection expression consists of a postfix expression, a period ., and a name. The postfix expression must be of a structure or union type, and the name must be the name of a component of that type. The value of the selection operation is the named member of the union or structure and is an lvalue if:

1. the expression before the period is an lvalue, and

2. the selected component is not an array type

The first condition is true for all structure and union values except those returned by a function.

Some C implementations, including some UNIX ones, permit structure-returning functions but do not allow the return values of such functions to have components selected from them. We regard this as a deficiency.

An indirect component selection expression consists of a postfix expression, the operator ->, and a name. The value of the postfix expression must be a pointer to a structure or union type, and the name must be the name of a component of that

structure or union type. The value is the named member of the union or structure and is an lvalue unless the member is an array. The expression *e->name* is by definition precisely equivalent to the expression (**e*)*.name*.

```
{
    static struct {float x, y; } Point, *Point ()Ptr;
    /* Set both components of Point to 0.0 in a
       roundabout fashion to demonstrate "->". */
    Point.x = 0.0;           /* Sets x to 0.0 */
    Point ()Ptr = &Point;
    Point ()Ptr->y = 0.0;  /* Sets y to 0.0 */
}
```

7.4.3. C-Ref: Function Calls

A function call consists of a postfix expression (the *function expression*), a left parenthesis, a possibly empty sequence of expressions (the *argument expressions*) separated by commas, and then a right parenthesis.

function-call :
    postfix-expression (*expression-list*$_{opt}$)

expression-list :
    assignment-expression
    expression-list , *assignment-expression*

The type of the function expression must be "function returning *T*" for some type *T*, in which case the result of the function call is of type *T*. The result is not an lvalue. If *T* is `void`, however, then the function call produces no result and may not be used in a context that requires the call to yield a result.

Some implementations of C permit function pointers to be used in function call expressions:

```
int (*f)();
...
f(x,y);          /* call function through pointer */
```

The advantage of this generalization is that it avoids overuse of the clumsy notation `(*f)(x,y)`, and many implementations allow this at least when `f` is the formal parameter of a function. Draft Proposed ANSI C permits function pointers to be used in function calls generally.

The arguments are each converted according to the usual argument conversions, but no other conversions or checks are required of the compiler. In particular, the compiler is not required to issue any warning message or take any other special action if the number of actual arguments does not match the number of formal parameters of the function being called, or if the (converted) type of an actual argument does not match the (promoted) type of the corresponding formal parameter. The reason is that in the general case information about the formal parameters is not available to the compiler, because declarations of external functions do not

necessarily contain any information about the formal parameters. On the other hand, the compiler is not forbidden to issue warnings when the information needed to make such checks happens to be available.

After the actual arguments have been evaluated and converted, they are copied for transmission to the called function; thus, all arguments are passed by value. Within the called function the names of formal parameters are lvalues, but assigning to a formal parameter changes only the value of the formal parameter and has no effect on any actual argument that may happen to be an lvalue. For example, consider this code:

```
/* Computes y to the fourth power. */
double power4(y)
  double y;
{
    y *= y;            /* Square the value of y. */
    return (y * y);    /* Square again. */
}

void main()
{
    double x, z;
    ...
    x = 3.0;
    z = power4(x);
    ...
}
```

The function `power4` uses its formal parameter to hold an intermediate result (the square of the original argument). The motivation for this method is that it requires only two multiplications, whereas computing `y*y*y*y` would require three.

When the function `power4` is called from routine `main`, the assignment to `y` in the function `power4` changes the value of `y` but does not change the value of the variable `x` within `main`. After the assignment to `z`, `z` has the value `81.0` and `x` has the value `3.0` (not `9.0`).

It should be noted, however, that when a pointer is passed as an argument, the pointer itself is copied but the object pointed to is not copied. By using pointers, a function and its caller can cooperate in allowing the called function to modify an object supplied by the caller.

If a function whose return type is not `void` is called in a context where the value of the function would be discarded, the compiler may issue a warning to that effect. The intent to discard the result of the function call may be made explicit by using a cast:

```
{
    ...
    (void) strcat(word, suffix);
                  /* Discard result of strcat. */
}
```

Comma expressions may be arguments to functions if they are enclosed in parentheses so that their parts are not interpreted as separate arguments. We can't think of a plausible reason for doing this, but here is an implausible example:

```
int main()
{
    int i, j;
    f( (i=1, i), (j=1, j)); /* legal, but strange */
    ...
}
```

7.4.4. C-Ref: Postincrement Operator

The postfix operator ++ performs "postincrementation," a side effect-producing operation.

postincrement-expression :
    postfix-expression ++

The operand must be an lvalue and may be of any scalar type. The constant 1 is added to the operand, modifying the operand. The result is the *old* value of the operand, before it was incremented. The result is not an lvalue. The usual binary conversions are performed on the operand and the constant 1 before the addition is performed, and the usual assignment conversions are performed when storing the sum back into the operand. The type of the result is that of the lvalue operand before conversion.

This operation may produce unpredictable effects if overflow occurs and the operand is a signed integer or floating-point number. The result of incrementing the largest representable value of an unsigned type is 0.

If the operand is a pointer, say of type "pointer to *T*" for some type *T*, the effect is to move the pointer forward beyond the object pointed to, as if to move the pointer to the next element within an array of objects of type *T*. (On a byte-addressed computer, this means advancing the pointer by `sizeof(T)` bytes.) However, the value of the expression is the pointer before modification. It is very common to use the postincrement operator when scanning the elements of an array or string; here is an example of counting the number of characters in a string.

```
int strlen(cp)
  char *cp;
{
    int count = 0;
    while (*cp++) count++;
    return count;
}
```

7.4.5. C-Ref: Postdecrement Operator

The postfix operator -- performs "postdecrementation," a side effect-producing operation.

postdecrement-expression :
    postfix-expression --

The operand must be an lvalue and may be of any scalar type. The constant 1 is subtracted from the operand and the result stored back into the lvalue. The result is the *old* value of the operand, before it was decremented, which is not an lvalue. The usual binary conversions are performed on the operand and the constant 1 before the subtraction is performed, and the usual assignment conversions are performed when storing the difference back into the operand. The type of the result is that of the lvalue operand before conversion.

This operation may produce unpredictable effects if overflow occurs and the operand is a signed integer or floating-point number. The result of decrementing the value 0 of an unsigned integer type is the largest representable value of that type.

If the operand is a pointer, say of type "pointer to *T*" for some type *T*, the effect is to move the pointer back over an object of type *T* preceding the one originally pointed to, as if to move the pointer to the previous element within an array of objects of type *T*. (On a byte-addressed computer, this means backing up the pointer by `sizeof(T)` bytes.) However, the value of the expression is the pointer before modification.

7.5. C-Ref: Unary Expressions

There are several kinds of unary expressions.

unary-expression :
    postfix-expression
    cast-expression
    sizeof-expression
    unary-minus-expression
    logical-negation-expression
    bitwise-negation-expression
    address-expression
    indirection-expression
    preincrement-expression
    predecrement-expression

The unary operators have precedence lower than the postfix expressions but higher than all binary and ternary operators. For example, the expression `*x++` is interpreted as `*(x++)`, not as `(*x)++`.

Draft Proposed ANSI C also has a unary plus expression.

7.5.1. C-Ref: Casts

A cast expression consists of a left parenthesis, a type name, a right parenthesis, and an operand expression.

cast-expression :
    (*type-name*) *unary-expression*

The cast causes the operand value to be converted to the type named within the parentheses. Any permissible conversion may be invoked by a cast expression. The result is not an lvalue.

```
extern char *alloc();
struct S *p;
p = (struct S *) alloc(sizeof(struct S));
```

Some implementations of C incorrectly ignore certain casts whose only effect is to make a value "narrower" than normal. For example, suppose that type `unsigned short` is represented in 16 bits and type `unsigned` is represented in 32 bits. Then the value of the expression

```
(unsigned)(unsigned short)0xFFFFFF
```

ought to be `0xFFFF`, because the cast `(unsigned short)` should cause truncation of the value `0xFFFFFF` to 16 bits, and then the cast `(unsigned)` should widen that value back to 32 bits. Deficient compilers fail to implement this truncation effect and generate code that passes the value `0xFFFFFF` through unchanged. Similarly, for the expression

```
(double)(float)3.1415926535897932384
```

deficient compilers do not produce code to reduce the precision of the approximation to pi to that of a `float`, but pass through the double-precision value unchanged. For maximum portability, we advise programmers to truncate values by storing them into variables or performing explicit masking operations (such as with the binary bitwise AND operator `&`) rather than relying on narrowing casts.

7.5.2. C-Ref: Size of Operator

The `sizeof` operator is used to obtain the size of a type or data object.

sizeof-expression :
    `sizeof` (*type-name*)
    `sizeof` *unary-expression*

The `sizeof` expression has two forms: the operator `sizeof` followed by a parenthesized type name, or the operator `sizeof` followed by an operand expression.

Applying the `sizeof` operator to a parenthesized type name yields the size of an object of the specified type; that is, the amount of memory (measured in storage units) that would be occupied by an object of that type. The type name may not name an array type with no explicit length, or a function type, or the type `void`. If the type name is a structure, union, or enumeration type definition (certainly a poor programming style) the effect of the `sizeof` expression should *not* be to create a new type, but only to compute its length.

Applying the sizeof operator to an expression yields the same result as if it had been applied to the name of the type of the expression. The sizeof operator does not of itself cause any of the usual conversions to be applied to the expression in determining its type, but if the expression contains operators that do perform usual conversions, then those conversions are considered when determining the type. For example, applying sizeof to the name of an array produces the total size of the array; this is possible because the sizeof operator does not cause the array to be converted to a pointer first. Likewise, if the variable v is of type short, then sizeof(v) is the same as sizeof(short). However, sizeof(v+0) is the same as sizeof(int), which might not be the same as sizeof(short). This is because the operator + performs the usual conversions, causing the sum of v and 0 to have type int.

The result of applying sizeof to an expression can always be deduced at compile time by examining the types of the objects in the operand expression. The result of sizeof never depends on the particular values of the objects at run time (except insofar as the value of an integer literal may determine whether or not it is considered to be a long constant, but then again the value of a literal is determinable at compile time).

When sizeof is applied to an expression, the expression is analyzed at compile time to determine its type, but the expression itself is not compiled into executable code. This means that any side effects that might be produced by execution of the expression will not take place. For example, execution of the expression

```
k = sizeof(j++)
```

will assign some value to k *but will not increment* j.

Applying the sizeof operator to a structure member that is a bit field produces the size of the declared type of the member; the size of the field in bits does not affect the result. This usage is probably misleading and should be avoided.

The original definition of C did not specify the type of the result of the sizeof operator. We recommend that the result of the sizeof operator be either of type unsigned int or of type unsigned long at the discretion of the implementor. Normally it will be unsigned int unless unsigned int is too small to represent a pointer in the particular implementation, in which case unsigned long must be used. Some implementations of C use int or long int as the type of the result of sizeof, but this is an inferior choice because it may be impossible to represent the size of a very large array, say one that spans more than half the total address space.

There is, on the face of it, a syntactic ambiguity in an expression such as

```
sizeof(long)-2
```

It could be interpreted as three unary operators (unary negation -, the cast (long), and sizeof) to be applied successively to the value 2:

```
sizeof((long)(-2))
```

Alternatively, it could be interpreted as a binary subtraction operator - whose operands are sizeof(long) and 2:

```
(sizeof(long))-2
```

This ambiguous case is resolved quite arbitrarily by declaring that the latter interpretation shall be used.

7.5.3. C-Ref: Unary Minus

The unary operator `-` computes the arithmetic negation of its operand. The operand may be of any arithmetic type. The usual unary conversions are performed on the operand. The result is not an lvalue.

unary-minus-expression :
  `-` *unary-expression*

A unary minus expression "-*e*" may be considered to be a shorthand notation for "`0-(`*e*`)`"; the two expressions in effect always perform the same computation. This computation may produce unpredictable effects if the operand is a signed integer or floating-point number and overflow occurs.

For an unsigned integer operand k, the result is always unsigned and equal to 2^n-k, where `n` is the number of bits used to represent the result. Because the result is unsigned, it can never be negative. This may seem strange, but note, however, that `(-`*x*`)+`*x* is equal to 0 for any unsigned integer *x*. (This identity also holds for any signed integer *x* for which -*x* is well defined.)

7.5.4. C-Ref: Logical Negation

The unary operator `!` computes the logical negation of its operand. The operand may be of any scalar type.

logical-negation-expression :
  `!` *unary-expression*

The usual unary conversions are performed on the operand. The result of the `!` operator is of type `int`; the result is 1 if the operand is zero (null in the case of pointers, 0.0 in the case of floating-point values) and 0 if the operand is not zero (nonnull, not 0.0). The result is not an lvalue. The expression `!(`*x*`)` is identical in meaning to `(`*x*`)==0`.

```
/* Assume that assertion failure accepts a string
   and reports it as a message to the user. */
#define assert(x,s) if (!(x)) assertion failure(s)
...
assert(num cases > 0, "No test cases.");
average = total points/num cases;
...
```

7.5.5. C-Ref: Bitwise Negation

The unary operator `~` computes the bitwise negation of its operand.

bitwise-negation-expression :
  `~` *unary-expression*

The operand may be of any integral type. The usual unary conversions are performed on the operand. Every bit in the binary representation of the result is the inverse of what it was in the (converted) operand. The result is not an lvalue.

```
#define LOW ADDRESS BITS 3L
long address;
...
/* Clear the low-order address bits.  For a 32-bit
   address this performs a bitwise AND of the
   address with 037777777774. However, the use of
   ~ allows the code to work properly
   no matter what the size of the address is.
*/
address &= ~LOW ADDRESS BITS ;
```

Because different implementations may use different representations for signed integers, the result of applying the bitwise NOT operator ~ to signed operands may not be portable. We recommend using ~ only on unsigned operands for portable code.

7.5.6. C-Ref: Address Operator

The unary operator & returns a pointer to its operand, which must be an lvalue.

address-expression :
    & *unary-expression*

If the type of the operand for & is "*T*," then the type of the result is "pointer to *T*." None of the usual conversions are relevant to the & operator, and its result is never an lvalue.

The original description of C specified that it was illegal to apply the unary address operator & to a register variable, and some compilers still enforce this restriction. However, since register is treated only as a hint to the compiler and not a mandatory requirement, it seems appropriate to allow such usage, inasmuch as on some computers the registers really are addressable as if they were memory locations. On the other hand, when the target computer does not have addressable registers, applying & to a register variable may simply defeat the declaration of the variable as register, forcing it to be of class auto instead. We recommend that new compilers take this latter approach and perhaps also issue a warning message. In any case, such usage should be regarded as nonportable.

It is incorrect for the operand expression to be the name of a function or the name of an array, because such a name is not an lvalue. Recall that in most contexts a function value is implicitly converted to a pointer to that function anyway, in exactly the same manner as if & had been used, and an array value is implicitly converted to a pointer to the first element.

```
extern int i, f();
int *ip, (*fp)();
ip = &i;         /* '&' needed */
fp = f;          /* '&' not needed or permitted */
```

Some compilers permit the & operator to be applied to a function or array and consider it to have no effect, but this is confusing at best. Furthermore, Draft Proposed ANSI C assigns a different meaning to the application of & applied to an array, causing older programs that use & in this way to break.

7.5.7. C-Ref: Indirection

The unary operator * performs indirection through a pointer. Thus the & and * operators are each the inverse of the other.

indirection-expression :
  * *unary-expression*

The usual unary conversions are performed on the operand, but the only relevant conversions are from arrays and functions to pointers. The (converted) operand must be a pointer, and the result is an lvalue referring to the object to which the operand points. If the type of the operand is "pointer to *T*," then the type of the result is simply "*T*."

```
int i,*p;
p  = &i;        /* p now points to variable i */
*p = 10;        /* sets value of i to 10 /*
```

The run-time effects of applying the indirection operator to a null pointer are undefined. It may return an unpredictable value, cause a trap, or perform some completely unpredictable action.

7.5.8. C-Ref: Preincrement Operator

The unary operator ++ performs "preincrementation," a side-effect producing operation. (There is also a postfix form of this operator.)

preincrement-expression :
  ++ *unary-expression*

The operand must be an lvalue and may be of any scalar type. The constant 1 is added to the operand and the result stored back in the lvalue. The result is the new (incremented) value of the operand but is not an lvalue. The expression ++(*x*) is identical in meaning to (*x*)+=1. (The operator +=, a compound assignment operator, is described in section "C-Ref: Compound Assignment".) The usual binary conversions are performed on the operand and the constant 1 before the addition is performed, and the usual assignment conversions are performed when storing the sum back into the operand. The type of the result is that of the lvalue operand before conversion.

```
static int uniqueint()
/* uniqueint: Successive calls to this routine
   return the integers 1, 2, ... without check for
   overflow. */
{
    static int count = 0;
    return ++count;
}
```

This operation may produce unpredictable effects if overflow occurs and the operand is a signed integer or floating-point number. The result of incrementing the largest representable value of an unsigned type is 0.

If the operand is a pointer, say of type "pointer to *T*" for some type *T*, the effect is to move the pointer forward beyond the object pointed to, as if to move the pointer to the next object within an array of objects of type *T*. (On a byte-addressed computer, this means advancing the pointer by `sizeof(T)` bytes.)

The expression statements `x++;` and `++x;` are equivalent, with `x++;` seeming to be more popular.

7.5.9. C-Ref: Predecrement Operator

The unary operator `--` performs "predecrementation," a side-effect producing operation. (There is also a postfix form of this operator.)

predecrement-expression :
    `--` *unary-expression*

The operand must be an lvalue and may be of any scalar type. The constant 1 is subtracted from the operand and the result stored back in the lvalue. The result is the new (decremented) value of the operand but is not an lvalue. The expression `--(`*x*`)` is identical in meaning to `(`*x*`)-=1`. (The operator `-=`, a compound assignment operator, is described in section "C-Ref: Compound Assignment".) The usual binary conversions are performed on the operand and the constant 1 before the subtraction is performed, and the usual assignment conversions are performed when storing the difference back into the operand. The type of the result is that of the lvalue operand before conversion.

This operation may produce unpredictable effects if overflow occurs and the operand is a signed integer or floating-point number. The result of decrementing the value 0 of an unsigned integer type is the largest representable value of that type.

If the operand is a pointer, say of type "pointer to *T*" for some type *T*, the effect is to move the pointer back over an object of type *T* preceding the one originally pointed to, as if to move the pointer to the previous element within an array of objects of type *T*. (On a byte-addressed computer, this means backing up the pointer by `sizeof(T)` bytes.)

```
int strrev(s1, s2)
/* strrev: copy string s2 in reversed form into
           string s1.  s2 should be a pointer to
           the first character of a null-terminated
           string.  s1 should point to an area that
           will receive a null-terminated string
           of the same length but with reversed
           contents. */
  char *s1, *s2;
{
    char *p = s1;
    while (*p++);  /* Locate end of first string. */
    --p;           /* Overshot: back up to the null. */
    /* Now copy the characters in reverse order. */
    while (p > s1)
        *s2++ = *--p;
    *s2 = '\0';    /* Terminate the result string. */
}
```

The expression statements `x--;` and `--x;` are equivalent, with `x--;` seeming to be more popular.

7.6. C-Ref: Binary Operator Expressions

A binary operator expression consists of two expressions separated by a binary operator. In order of decreasing precedence, the kinds of binary expressions are:

multiplicative-expression
additive-expression
shift-expression
relational-expression
equality-expression
bitwise-and-expression
bitwise-xor-expression
bitwise-or-expression

All of the binary operators described in this section are left-associative. For example, the operators `*` and `%` have the same level of precedence, and therefore the expression `x*y%z` is treated as `(x*y)%z`, not as `x*(y%z)`; similarly, the expression `x%y*z` is treated as `(x%y)*z`, not as `x%(y*z)`.

For each of the binary operators described in this section, both operands are fully evaluated (but in no particular order) before the operation is performed.

7.6.1. C-Ref: Multiplicative Operators

The three multiplicative operators, `*` (multiplication), `/` (division), and `%` (remainder), have the same precedence and are left-associative.

multiplicative-expression :
  unary-expression
  multiplicative-expression   mult-op   cast-expression

mult-op
  `*  /  %`

Multiplication The binary operator `*` indicates multiplication. Each operand may be of any arithmetic type. The usual binary conversions are performed on the operands, and the type of the result is that of the converted operands. The result is not an lvalue. For integral operands, integer multiplication is performed; for floating-point operands, floating-point multiplication is performed.

The multiplication operator may produce unpredictable effects if overflow occurs and the operands (after conversion) are signed integers or floating-point numbers. If the operands are unsigned integers, the result is congruent mod 2^n to the true mathematical result of the operation (where n is the number of bits used to represent the unsigned result).

The `*` operator is assumed to be commutative and associative, and the compiler is permitted to rearrange an expression with several multiplications, even in the presence of parentheses and without regard to avoiding overflow. For example, the compiler may freely interpret the expression `a*(b*c)*d` as if it were `(c*a)*(b*d)`, subject to the restrictions discussed in section

Division The binary operator `/` indicates division. Each operand may be of any arithmetic type. The usual binary conversions are performed on the operands, and the type of the result is that of the converted operands. The result is not an lvalue.

For floating-point operands, floating-point division is performed.

For integral operands, if the mathematical quotient of the operands is not an exact integer, then the result will be one of the two integers closest to the mathematical quotient of the operands. Of those two integers, the one closer to 0 must be chosen if both operands are positive (that is, division of positive integers is truncating division). Note that this completely specifies the behavior of division for unsigned operands. If either operand is negative, then the choice is left to the discretion of the implementor. For maximum portability, programs should therefore avoid depending on the behavior of the division operator when applied to negative integral operands.

The division operator may produce unpredictable effects if overflow occurs and the operands (after conversion) are signed integers or floating-point numbers. Note that overflow can occur for signed integers represented in two's-complement form if the most negative representable integer is divided by -1; the mathematical result is a positive integer that cannot be represented. Overflow cannot occur if the operands are unsigned integers.

The consequences of division by zero, whether integer or floating-point, are machine dependent.

Remainder The binary operator % computes the remainder when the first operand is divided by the second. Each operand may be of any integral type. The usual binary conversions are performed on the operands, and the type of the result is that of the converted operands. The result is not an lvalue. The library functions div, ldiv, and fmod also compute remainders of integers and floating-point values.

It is always true that (a/b)*b + a%b is equal to a if b is not 0, so the behavior of the remainder operation is coupled to that of integer division. When both operands are positive, the remainder operation will always be equivalent to the mathematical "mod" operation. Note that this completely specifies the behavior of the remainder operation for unsigned operands. If either operand is negative, the behavior will be machine dependent in a manner corresponding to the machine dependence of integer division. For maximum portability, programs should therefore avoid depending on the behavior of the remainder operator when applied to negative integral operands.

The remainder operator may produce unpredictable effects if performing division on the two operands would produce overflow. Note that overflow can occur for signed integers represented in two's-complement form if the most negative representable integer is divided by -1; the mathematical result of the division is a positive integer that cannot be represented, and therefore the results are unpredictable, even though the remainder itself (zero) is representable. Overflow cannot occur if the operands are unsigned integers.

The consequences of taking a remainder with a second operand of zero are machine dependent.

```
/* Compute the greatest common divisor by Euclid's
   algorithm.  The result is the largest integer
   that evenly divides x and y.
 */
unsigned gcd(x, y)
  unsigned x, y;
{
    while ( y != 0 ) {
        unsigned temp = y;
        y = x % y;
        x = temp;
    }
    return x;
}
```

7.6.2. C-Ref: Additive Operators

The two additive operators, + (addition) and - (subtraction), have the same precedence and are left-associative.

additive-expression :
  multiplicative-expression
  additive-expression add-op multiplicative-expression

add-op
  + -

Addition The binary operator + indicates addition. The usual binary conversions are performed on the operands. The operands may both be arithmetic or one may be a pointer and the other an integer. No other operand types are allowed. The result is not an lvalue.

When the operands are arithmetic, the type of the result is that of the converted operands. For integral operands, integer addition is performed; for floating-point operands, floating-point addition is performed.

When adding a pointer *p* and an integer *k*, it is assumed that the object that *p* points to lies within an array of such objects, and the result is a pointer to that object within the presumed array that lies *k* objects away from the one *p* points to. For example, *p*+1 (or 1+*p*) points to the object just after the one *p* points to, and *p*+(-1) (or (-1)+*p*) points to the object just before. It is illegal for *p* to be of type "pointer to function." (Functions cannot be elements of arrays.)

The addition operator may produce unpredictable effects if overflow occurs and the operands (after conversion) are signed integers or floating-point numbers, or if either operand is a pointer. If the operands are both unsigned integers, the result is congruent mod 2^n to the true mathematical result of the operation (where *n* is the number of bits used to represent the unsigned result).

The + operator is assumed to be commutative and associative, and the compiler is permitted to rearrange an expression with several additions, even in the presence of parentheses and without regard to avoiding overflow. For example, the compiler may freely interpret the expression a+(b+c)+d as if it were (c+a)+(b+d), subject to the restrictions discussed in section "C-Ref: Order of Evaluation".

Subtraction The binary operator - indicates subtraction. The usual binary conversions are performed on the operands. The operands may both be arithmetic or may both be of the same pointer type, or the left operand may a pointer and the other an integer. No other operand types are permitted. The result is not an lvalue.

If the operands are both arithmetic, the type of the result is that of the converted operands. For integral operands, integer subtraction is performed; for floating-point operands, floating-point subtraction is performed. Note that the result of subtracting one unsigned integer from another is always unsigned and therefore cannot be negative. However, unsigned numbers always obey such identities as (a+(b-a))==b and (a-(a-b))==b.

Subtraction of an integer from a pointer is analogous to addition of an integer to a pointer. When subtracting an integer *k* from a pointer *p*, it is assumed that the object that *p* points to lies within an array of such objects, and the result is a pointer to that object within the presumed array that lies -*k* objects away from the one *p* points to. For example, *p*-1 points to the object just before the one *p* points

to, and `p-(-1)` points to the object just after. It is illegal for p to be of type "pointer to function." (Functions cannot be elements of arrays.)

Given two pointers p and q of the same type, the difference $p-q$ is an integer k such that adding k to q yields p. The type of the difference may be either `int` or `long`, depending on the implementation. The result is well defined and portable only if the two pointers point to objects in the same array, or at least are aligned as if they did. If either of the pointers is null, the result is undefined. It is illegal for either pointer to be of type "pointer to function."

The subtraction operator may produce unpredictable effects if overflow occurs and the operands (after conversion) are signed integers or floating-point numbers, or if either operand is a pointer. If the operands are both unsigned integers, the result is congruent mod 2^n to the true mathematical result of the operation (where n is the number of bits used to represent the unsigned result).

7.6.3. C-Ref: Shift Operators

The binary operator << indicates shifting to the left and the binary operator >> indicates shifting to the right. Both have the same precedence and are left-associative.

shift-expression :
   additive-expression
   shift-expression   shift-op   additive-expression

shift-op: one of
   <<   >>

Each operand must be of integral type. The usual unary conversions are performed *separately* on each operand (the usual binary conversions are *not* used for the shift operators), and the type of the result is that of the converted left operand. The result is not an lvalue.

The first operand is a quantity to be shifted, and the second operand specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by which operator (<< or >>) is used. The operator << shifts the value of the left operand to the left; excess bits shifted off to the left are discarded, and 0-bits are shifted in from the right. The operator >> shifts the value of the left operand to the right; excess bits shifted off to the right are discarded. The bits shifted in from the left for >> depend on the type of the converted left operand: If it is unsigned, then 0-bits are shifted in from the left; but if it is signed, then at the implementor's option either 0-bits or copies of the leftmost bit of the left operand are shifted in from the left.

Table "C-Ref: Computing the Greatest Common Divisor" shows how unsigned shift operations may be used to compute the greatest common divisor of two integers by the binary algorithm. Although this method is a bit more complicated than the Euclidean algorithm, it may also be faster, because in some implementations of C the remainder operation is rather slow, especially for unsigned operands.

The result value is undefined if the value of the right operand is negative, so specifying a negative shift distance does *not* (necessarily) cause << to shift to the right or >> to shift to the left. The result value is also undefined if the value of the right operand is greater than or equal to the width (in bit positions) of the value of the converted left operand. Note, however, that the right operand may be 0, in which case no shift occurs and the result value is identical to the value of the converted left operand.

The two shift operators have the same precedence and are left-associative. One can exploit this fact to write expressions that are visually pleasing but semantically confusing:

```
b << 4 >> 8
```

If b is a 16-bit quantity, this expression extracts the middle 8 bits. As always, it is better to use parentheses when there is any possibility of confusion:

```
(b << 4) >> 8
```

The original description of C specified that the >> operator with a signed left operand might shift in *either* 0-bits or copies of the leftmost bit of the left operand, at the discretion of the implementor. The intent was to permit the implementor the freedom to implement a right shift efficiently, but the effect has been to discourage any use of >> on a signed left operand. If a new compiler is to use a two's-complement or one's-complement representation for signed integers, we strongly recommend that right shifts on signed integers be performed by replicating the sign bit. This is the most consistent with the definition and use of signed integer types. The programmer can always cast the left operand to an unsigned type to force the shifting in of 0-bits.

Because different implementations may use different representations for signed integers, and because implementations using the same representation may nevertheless differ in their handling of right shifts on signed integers, the result of applying the shift operators << and >> to signed operands may not be portable. We recommend using << and >> only on unsigned operands for portable code.

7.6.3.1. C-Ref: Computing the Greatest Common Divisor

```
/* Compute the greatest common divisor by the so-called
   binary algorithm.  The result is the largest integer
   that evenly divides x and y.  Only subtraction,
   shifts, and bitwise operations are used; the remain-
   der operation (which may be expensive) is not used.
 */
unsigned binary gcd(x, y)
  unsigned x, y;
{
    unsigned temp;
    unsigned common power of two = 0;
```

```
    /* Special cases: if either argument is zero,
       then return the other one. */
    if (x == 0) return y;
    if (y == 0) return x;
    /* Determine the largest power of two that
       divides both x and y. */
    while (((x | y) & 1) == 0) {
        x = x >> 1;  /* One could write "x >>= 1;" */
        y = y >> 1;
        ++common power of two;
    }
    while ((x & 1) == 0)  x = x >> 1;
    while (y) {
       /* At this point x is guaranteed odd,
          and y is nonzero. */
       while ((y & 1) == 0)  y = y >> 1;
       /* At this point both x and y are odd. */
       temp = y;
       if (x > y)  y = x - y;
       else        y = y - x;
       x = temp;
       /* Now x has the old value of y; y was odd, so
          now x is odd.  Now y is even, because it
          was computed as the difference of two odd
          numbers; therefore it will be right-shifted
          at least once on the next iteration. */
    }
    return (x << common power of two);
}
```

7.6.4. C-Ref: Relational Operators

The binary operators <, <=, >, and >= indicate comparison.

> *relational-expression :*
>
> > *shift-expression*
> >
> > *relational-expression   relational-op   shift-expression*
>
> *relational-op*
>
> > <   <=   >   >=

The usual binary conversions are performed on the operands. The operands may both be of arithmetic types or may both be of the same pointer type. The result is always of type `int` and has the value 0 or 1. The result is not an lvalue.

For integral operands, integer comparison is performed (signed or unsigned as appropriate). For floating-point operands, floating-point comparison is performed. For pointer operands, the result depends on the relative locations within the address space of the two objects pointed to; the result is portable only if the objects

pointed to lie within the same array, or at least are aligned as if they did, in which case "greater than" means "having a higher index in the array."

The operator < tests for the relationship "is less than"; <= tests "is less than or equal to"; > tests "is greater than"; and >= tests "is greater than or equal to." The result is 1 if the stated relationship holds for the particular operand values and 0 if the stated relationship does not hold.

The binary relational operators all have the same precedence and are left-associative. It is therefore permitted to write an expression such as 3<x<7. This does not have the meaning it has in usual mathematical notation, however; by left-associativity it is interpreted as (3<x)<7. Because the result of (3<x) is 0 or 1, either of which is less than 7, the result of 3<x<7 is always 1. One must express the meaning of the usual mathematical notation by using a bitwise AND operator, as in 3<x & x<7, or a logical AND operator, as in 3<x && x<7.

The programmer should exercise care when using relational operators on mixed types. A particularly confusing case is this expression:

```
-1 < (unsigned) 0
```

One might think that this expression would always produce 1 (true), because -1 is less than 0. However, the usual binary conversions cause the value -1 to be converted to a (large) unsigned value before the comparison, and such an unsigned value cannot be less than 0. Therefore, the expression always produces 0 (false).

Some implementations permit relational comparisons between pointers and integers, which is disallowed by the language. The implementations may treat the comparisons as signed or unsigned.

7.6.5. C-Ref: Equality Operators

The binary operators == and != indicate comparison.

equality-expression :
  relational-expression
  equality-expression equality-op relational-expression

equality-op
  == !=

In this they are similar to the binary relational operators discussed in section "C-Ref: Relational Operators"; they differ in testing different relationships and in having a different level of precedence. The usual binary conversions are performed on the operands. The result is always of type int and has the value 0 or 1. The result is not an lvalue.

The operands may both be of arithmetic types or may both be of the same pointer type, or one of the operands may be a pointer and the other a constant integer expression with value 0.

For integral operands, integer comparison is performed. For floating-point operands, floating-point comparison is performed. Pointer operands are considered equal if they point to the same object or if they are both null. A pointer is equal to the integer constant 0 if and only if it is a null pointer.

The operator == tests for the relationship "is equal to"; != tests "is not equal to." The result is 1 if the stated relationship holds for the particular operand values and 0 if the stated relationship does not hold.

The programmer should be very careful not to confuse the == operator with the = operator. The == operator performs equality comparison; the = operator performs simple assignment. Several other programming languages use = for equality comparison. As a matter of style, if it is necessary to use an assignment expression in a context that will test the value of the expression against zero, it is best to write "!= 0" explicitly to make the intent clear. For example, consider this code:

```
while (x = next item()) {
    ...
}
```

It is unclear whether this is correct or whether it contains a typographical error that should be corrected to

```
while (x == next item()) {
    ...
}
```

The intent can be made explicitly clear in this manner:

```
while ((x = next item()) != 0) {
    ...
}
```

The binary equality operators both have the same precedence (but lower precedence than <, <=, >, and >=) and are left-associative. It is therefore permitted to write an expression such as x==y==7. This does not have the meaning it has in usual mathematical notation, however; by left-associativity it is interpreted as (x==y)==7. Because the result of (x==y) is 0 or 1, neither of which is equal to 7, the result of x==y==7 is always 0. One must express the meaning of the usual mathematical notation by using a bitwise AND operator, as in x==y & y==7, or a logical AND operator, as in x==y && y==7.

There is a bitwise XOR operator as well as bitwise AND and OR operators, but there is no logical XOR operator to go along with the logical AND and OR operators. The != operator serves the purpose of a logical XOR operator: One may write a<b != c<d for an expression that yields 1 if exactly one of a<b and c<d yields 1, and 0 otherwise. If either of the operands might have a value other than 0 or 1, then the unary ! operator can be applied to both operands: !x != !y yields 1 if exactly one of x and y is nonzero, and yields 0 otherwise. In a similar manner, == serves as a logical equivalence (EQV) operator.

7.6.6. C-Ref: Bitwise AND Operator

The binary operator & indicates the bitwise AND function; it is left-associative.

bitwise-and-expression :
  equality-expression
  bitwise-and-expression & *equality-expression*

The operands must both be integral and the usual binary conversions are per-

formed on the operands. The type of the result is that of the converted operands. The result is not an lvalue.

Each bit of the result is equal to the AND function of the two corresponding bits of the two (converted) operands. The AND function yields a 1-bit if both arguments are 1-bits, and otherwise yields a 0-bit:

a	b	a&b
01100	01010	01000

The `&` operator is commutative and associative, and the compiler is permitted to rearrange an expression with several bitwise AND operators, even in the presence of parentheses. For example, the compiler may freely interpret the expression `a&(b&c)&d` as if it were `(c&a)&(b&d)`, subject to the restrictions discussed in section "C-Ref: Order of Evaluation".

The bitwise AND operator `&` may be used to combine logical (integer `0` or `1`) values, yielding the integer value `1` if both operands are the integer value `1`, and yielding the integer value `0` if either operand is the integer value `0`:

```
if ( a<b & b<c ) ...
```

However, using the logical AND operator `&&` in these cases is both more efficient and safer when the operands are not known to be restricted to the values `0` and `1`. For example, if `a` is 2 and `b` is 4, then `a&b` is 0 (false) whereas `a&&b` is 1 (true). Of course, the operators `&` and `&&` also differ in that the result of `&&` is *always* `0` or `1`, no matter what the values of the operands, while this is not so for `&`.

Because different implementations may use different representations for signed integers, the result of applying the bitwise AND operator `&` to signed operands may not be portable. For portable code we recommend using `&` only on unsigned operands.

7.6.7. C-Ref: Bitwise XOR Operator

The binary operator `^` indicates the bitwise XOR function.

bitwise-xor-expression :
    bitwise-and-expression
    bitwise-xor-expression `^` *bitwise-and-expression*

The operands must both be integral and the usual binary conversions are performed on the operands. The type of the result is that of the converted operands. The result is not an lvalue.

Each bit of the result is equal to the XOR function of the two corresponding bits of the two (converted) operands. The XOR function yields a 1-bit if one argument is a 1-bit and the other is a 0-bit, and yields a 0-bit if both arguments are 1-bits or if both arguments are 0-bits:

a	b	a^b
01100	01010	00110

The `^` operator is commutative and associative, and the compiler is permitted to rearrange an expression with several bitwise XOR operators, even in the presence

of parentheses. For example, the compiler may freely interpret the expression `a^(b^c)^d` as if it were `(c^a)^(b^d)`, subject to the restrictions discussed in section "C-Ref: Order of Evaluation".

Because different implementations may use different representations for signed integers, the result of applying the bitwise XOR operator `^` to signed operands may not be portable. For portable code we recommend using `^` only on unsigned operands or on signed `0`/`1` values such as result from relational operators.

7.6.8. C-Ref: Bitwise OR Operator

The binary operator `|` indicates the bitwise OR function; it is left-associative.

bitwise-or-expression :
bitwise-xor-expression
bitwise-or-expression `|` *bitwise-xor-expression*

The operands must both be integral and the usual binary conversions are performed on the operands. The type of the result is that of the converted operands. The result is not an lvalue.

Each bit of the result is equal to the OR function of the two corresponding bits of the two (converted) operands. The OR function yields a 1-bit if either argument is a 1-bit, and otherwise yields a 0-bit:

a	b	a|b
01100	01010	00110

The bitwise OR operator `|` may be used to combine logical (integer `0` or `1`) values, yielding the integer value `1` if either operand is the integer value `1` and the other operand is the integer value `0` or `1`, and yielding the integer value `0` if both operands are the integer value `0`. For this purpose it differs from the logical OR operator `||` in that both operands for `|` are always fully evaluated, whereas the right operand of `||` is not evaluated if the left operand is nonzero.

Of course, the operators `|` and `||` also differ in that the result of `||` is *always* `0` or `1`, no matter what the values of the operands, while this is not so for `|`.

The `|` operator is commutative and associative, and the compiler is permitted to rearrange an expression with several bitwise OR operators, even in the presence of parentheses. For example, the compiler may freely interpret the expression `a|(b|c)|d` as if it were `(c|a)|(b|d)`, subject to the restrictions discussed in section "C-Ref: Order of Evaluation".

Because different implementations may use different representations for signed integers, the result of applying the bitwise OR operator `|` to signed operands may not be portable. For portable code we recommend using `|` only on unsigned operands or on signed `0`/`1` values such as result from relational operators.

Tables "C-Ref: A Package for Manipulating Sets of Integers (1)", "C-Ref: A Package for Manipulating Sets of Integers (2)", "C-Ref: A Package for Manipulating Sets of Integers (3)", and "C-Ref: A Package for Manipulating Sets of Integers (4)" show a library that defines a "set" package. It uses the bitwise operators to implement sets as bit vectors. Table "C-Ref: A Program for Enumerating Subsets of a Given

Set"shows a program that uses certain facilities in the set package to enumerate and print certain sets. Table "C-Ref: Sample Output From Enumerating Subsets"shows the output produced by that program.

7.6.8.1. C-Ref: A Package for Manipulating Sets of Integers (1)

```
/* A set package, suitable for sets of small integers
   in the range 0 (inclusive) to the number of bits in
   an 'unsigned int' type (exclusive).  Each integer is
   represented by a bit position; if the bit is 1, the
   integer is in the set; if the bit is 0,  the integer
   is not in the set.  The low-order bit represents
   the element 0.
*/

/* Maximum bits per set (implementation dependent). */
#define SET BITS 32

typedef unsigned SET;  /* A type to represent sets. */

/* check: true if i can be a set element. */
#define check(i)       ( ((unsigned) (i)) < SET BITS )

/* emptyset: a set with no elements. */
#define emptyset              ((SET) 0)

/* add: add a single integer to a set. */
#define add(set,i)            ((set) | singleset(i))

/* singleset: return a set with one element in it. */
#define singleset(i)          (((SET) 1) << (i))

/* intersect: return intersection of two sets. */
#define intersect(set1,set2)  ((set1) & (set2))

/* union: return the union of two sets. */
#define union(set1,set2)      ((set1) | (set2))

/* setdiff: symmetric set difference; return a set of
   those elements that appear in either argument set
   but not both. */
#define setdiff(set1,set2)    ((set1) ^ (set2))

/* element: true if integer i is in the set. */
#define element(i,set)        (singleset((i)) & (set))
```

7.6.8.2. C-Ref: A Package for Manipulating Sets of Integers (2)

```
/* forallelements: perform the following statement
   once for every element of the set s, with the
   variable j set to that element.  For example, to
   print all the elements in a set z, just write
        {
            int k;
            forallelements(k, z)
                printf("%d ", k);
        }
 */
#define forallelements(j,s) \
 for ((j)=0; (j)<SET BITS; ++(j)) if (element((j),(s)))

/* cardinality: return the number of elements in x. */
int cardinality(x)
  SET x;
{
    int count = 0;
    /*  At this point one could simply write
                int j;
                forallelements(j, x) ++count;
        which would obviously count all the elements of
        the set. However, the following loop is faster
        (but trickier).
     */
    while (x != emptyset) {
        /* The body of this loop is executed once for
           every 1-bit in the set x. Each time through,
           the smallest remaining element is removed
           from x (and counted).  The trick is that the
           expression (x & -x) yields a set that
           contains  the smallest element in x and no
           others.  This trick exploits properties of
           binary representation and unsigned negation.
         */
        x ^= (x & -x);
        ++count;
    }
    return count;
}
```

7.6.8.3. C-Ref: A Package for Manipulating Sets of Integers (3)

```
/* Produce a set of size n whose elements are the
   integers from 0 to n-1 (inclusive).  This is a bit
   tricky, and exploits the properties of unsigned
   subtraction. */
#define first set of n elements(n)  (SET)((1<<(n))-1)

/* Given a set of n elements, produce a new set of n
   elements.  If you start with the result of
   first set of n elements(k), and then at each step
   apply next set of n elements to the previous result,
   and keep going until a set is obtained containing
   m as a member, you will have obtained sets
   representing all possible ways of choosing k things
   from m things.
 */
SET next set of n elements(x)
  SET x;
{
    /* This code exploits many unusual properties of
       unsigned arithmetic.  As an illustration,
       suppose that the bit pattern 001011001111000
       is given as the argument x. */
    SET smallest = (x & -x);
    /* The value of "smallest" is   000000000001000 */
    SET ripple = x + smallest;
    /* The value of "ripple" is     001011010000000 */
    SET new smallest = (ripple & -ripple);
    /* Now "new smallest" is        000000010000000 */
    SET ones = ((new smallest / smallest) >> 1) - 1;
    /* Now "ones" is                000000000000111 */
    return (ripple | ones);
    /* The returned value is        001011010000111 */
    /* The overall idea is that you find the rightmost
       contiguous group of 1-bits.  Of that group, you
       slide the leftmost 1-bit to the left one place,
       and slide all the others back to the extreme
       right.  (This code was adapted from HAKMEM.) */
}
```

7.6.8.4. C-Ref: A Package for Manipulating Sets of Integers (4)

```
/* Print a set in the form "{1, 2, 3, 4}". */
void printset(z)
  SET z;
{
    int first = 1;
    int e;
    /* Print the elements, with leading punctuation. */
    forallelements(e, z) {
        if (first) printf("{");
        else printf(", ");
        printf("%d", e);
        first = 0;
    }
    /* Take care of the empty set. */
    if (first) printf("{");
    /* Print trailing punctuation. */
    printf("}");
}
```

7.6.8.5. C-Ref: A Program for Enumerating Subsets of a Given Set

```
#define LINE WIDTH 54
```

```
/* Print all the sets of size k having elements less
   than n.  Try to print as many as will fit on each
   line of the output.  Also print the total number of
   such sets; it should equal n!/(k!*(n-k)!) where "!"
   is the factorial symbol (5! = 1*2*3*4*5 = 120). */
void print k of n(k, n)
  int k, n;
{
    int count = 0;
    /* Estimate how wide each printed set will be. */
    int printed set width = k * ((n > 10) ? 4 : 3) + 3;
    int sets per line = LINE WIDTH / printed set width;
    SET z = first set of n elements(k);

    printf("\nAll the size-%d subsets of ", k);
    printset(first set of n elements(n));
    printf(":\n");
    do {                  /* Enumerate all the sets. */
        printset(z);
        if ((++count) % sets per line) printf ("   ");
        else printf("\n");
        z = next set of n elements(z);
    } while ((z != emptyset) && !element(n, z));
    if ((count) % sets per line) printf ("\n");
    printf("The total number of such subsets is %d.\n",
           count);
}

/* The main program merely tries some examples. */
void main()
{
    print k of n(0, 4);
    print k of n(1, 4);
    print k of n(2, 4);
    print k of n(3, 4);
    print k of n(4, 4);
    print k of n(3, 5);
    print k of n(3, 6);
}
```

7.6.8.6. C-Ref: Sample Output From Enumerating Subsets

```
All the size-0 subsets of {0, 1, 2, 3}:
{}
The total number of such sets is 1.
```

```
All the size-1 subsets of {0, 1, 2, 3}:
{0}   {1}   {2}   {3}
The total number of such sets is 4.

All the size-2 subsets of {0, 1, 2, 3}:
{0, 1}   {0, 2}   {1, 2}   {0, 3}   {1, 3}   {2, 3}
The total number of such sets is 6.

All the size-3 subsets of {0, 1, 2, 3}:
{0, 1, 2}   {0, 1, 3}   {0, 2, 3}   {1, 2, 3}
The total number of such sets is 4.

All the size-4 subsets of {0, 1, 2, 3}:
{0, 1, 2, 3}
The total number of such sets is 1.

All the size-3 subsets of {0, 1, 2, 3, 4}:
{0, 1, 2}   {0, 1, 3}   {0, 2, 3}   {1, 2, 3}
{0, 1, 4}   {0, 2, 4}   {1, 2, 4}   {0, 3, 4}
{1, 3, 4}   {2, 3, 4}
The total number of such sets is 10.

All the size-3 subsets of {0, 1, 2, 3, 4, 5}:
{0, 1, 2}   {0, 1, 3}   {0, 2, 3}   {1, 2, 3}
{0, 1, 4}   {0, 2, 4}   {1, 2, 4}   {0, 3, 4}
{1, 3, 4}   {2, 3, 4}   {0, 1, 5}   {0, 2, 5}
{1, 2, 5}   {0, 3, 5}   {1, 3, 5}   {2, 3, 5}
{0, 4, 5}   {1, 4, 5}   {2, 4, 5}   {3, 4, 5}
The total number of such sets is 20.
```

7.7. C-Ref: Logical Operator Expressions

A logical operator expression consists of two expressions separated by one of the logical operators `&&` and `||`. The two operators have different levels of precedence; `&&` has higher precedence than `||`.

Both of the logical operators described in this section are described as being syntactically left-associative, though this doesn't matter much to the programmer because the operators happen to be fully associative semantically and no two operators have the same level of precedence. (Implementors find the fact of syntactic left-associativity useful because it tends to make it easier for relatively simple compilers to produce good code for these operators than right-associativity would.)

For each of the logical operators described in this section, the second operand is *not evaluated at all* if the value of the first operand provides sufficient information to determine the result of the logical operator expression.

7.7.1. C-Ref: Logical AND Operator

The logical AND operator `&&` is called "conditional and" in other programming languages.

> *logical-and-expression :*
>   *bitwise-or-expression*
>   *logical-and-expression* `&&` *bitwise-or-expression*

The logical operator `&&` accepts operands of any scalar type. There is no constraint between the types of the two operands. The type of the result is always `int` and has the value 0 or 1. The result is not an lvalue.

The left operand of `&&` is fully evaluated first. If the left operand is equal to zero (in the sense of the `==` operator), then the right operand is not evaluated and the result value is `0`. If the left operand is not equal to zero, then the right operand is evaluated; if the right operand is equal to zero, then the result value is `0` and otherwise is `1`. For example:

a	b	a&&b
1	0	0
0	-1	0
1	1	
34.5	'\0'	0
&x	&y	1

Unlike the binary bitwise AND operator `&`, the logical operator `&&` guarantees left-to-right conditional evaluation.

7.7.2. C-Ref: Logical OR Operator

The logical OR operator `||` is called "conditional or" in other programming languages.

> *logical-or-expression :*
>   *logical-and-expression*
>   *logical-or-expression* `||` *logical-and-expression*

The logical operator `||` accepts operands of any scalar type. There is no constraint between the types of the two operands. The type of the result is always `int` and has the value 0 or 1. The result is not an lvalue.

The left operand of `||` is fully evaluated first. If the value of the left operand is not equal to zero (in the sense of the `==` operator), then the right operand is not evaluated and the result value is `1`. If the left operand is equal to zero, then the right operand is evaluated; if the right operand is not equal to zero, then the result value is `1` and otherwise is `0`. For example:

a	b	a&&b
0	0	0
1	0	1
0	-1	1
0.0	'\0'	0
34.5	'\0'	1
&x	0	1

Unlike the binary bitwise OR operator `|`, the logical operator `||` guarantees left-to-right conditional evaluation.

7.8. C-Ref: Conditional Expressions

The `?` and `:` operators introduce a conditional expression, which has lower precedence than the binary expressions and differs from them in being right-associative.

> *conditional-expression :*
>   *logical-or-expression*
>   *logical-or-expression* `?` *expression* `:` *conditional-expression*

A conditional expression consists of three expressions, with the first and second expressions separated by `?` and the second and third expressions separated by `:`.

The first operand is used to determine which of the other two operands should be evaluated. The first operand is fully evaluated. If the first operand is not equal to zero (in the sense of the `==` operator), then the second operand is evaluated and the third operand is not evaluated; the result value is the value of the (possibly converted) second operand. If first operand is equal to zero, then the second operand is not evaluated and the third operand is evaluated; the result value is the value of the (possibly converted) third operand. In either case, the result is not an lvalue.

Conditional expressions are right-associative with respect to their first and third operands, so that

```
a ? b : c ? d : e ? f : g
```

is interpreted as

```
a ? b : (c ? d : (e ? f : g))
```

Here is one example where this might be useful:

```
/* Return 1, -1, or 0 if x is positive,
   negative, or zero, respectively. */
int signum(x)
  int x;
{
    return (x > 0) ? 1 : (x < 0) ? -1 : 0;
}
```

Anything more complicated than this is probably better done with one or more `if` statements.

The second operand of a conditional expression may be any expression whatsoever and may use operators that have lower precedence. There is no possibility of confusion, because the tokens ? and : effectively bracket the second operand like parentheses. However, the third operand cannot involve operators of lower precedence without the use of parentheses. As an example, the expression

```
a ? b = c : c = b
```

is not legal. The first assignment is all right, but the second assignment causes a problem. Because the assignment operator has lower precedence than the conditional operator, the expression must be interpreted as

```
(a ? b = c : c) = b
```

However, a conditional expression cannot produce an lvalue, and so the assignment is illegal. When there is any doubt, it is better to use too many parentheses than too few:

```
a ? (b = c) : (c = b)
```

The first operand of a conditional expression may be of any scalar type. There are four possibilities for the second and third operands:

1. They may both be arithmetic. The usual binary conversions are performed on the second and third operands, and the type of the result is the common type to which the operands are converted.

2. They may be pointers of the same type, after application of the usual unary conversions, if necessary, to convert functions and arrays to pointers. The result is a pointer of this same type.

3. They may have identical types (structure, union, or void). The result has this same type.

4. One may be a pointer (after the usual unary conversions) and the other a constant integer expression with value 0. The result is of the same type as the pointer operand.

As a matter of style, it is a good idea to enclose the first operand of a conditional expression in parentheses, but this is not required.

Not all existing implementations of C permit the result of a conditional expression to have structure, union, or void types. New implementations should permit them.

7.9. C-Ref: Assignment Expressions

Assignment expressions consist of two expressions separated by an assignment operator; they are right-associative.

assignment-expression :
    conditional-expression
    unary-expression   assignment-op   assignment-expression

assignment-op
```
=     +=    -=    *=    /=    %=
<<=   >>=   &=    ^=     =
```

The operator `=` is called the *simple* assignment operator; all the others are *compound* assignment operators.

Assignment operators are all of the same level of precedence and are right-associative (all other operators in C that take two operands are left-associative). For example, the expression `x*=y=z` is treated as `x*=(y=z)`, not as `(x*=y)=z`; similarly, the expression `x=y*=z` is treated as `x=(y*=z)`, not as `(x=y)*=z`. The right-associativity of assignment operators allows "multiple assignment expressions" to have the "obvious" interpretation; the expression

```
a = b = c = d + 7
```

is interpreted as

```
a = (b = (c = d + 7))
```

and therefore assigns the value of `d+7` to `c` then to `b` then to `a`.

Every assignment operator requires an lvalue as its left operand and modifies that lvalue by storing a new value into it; the operators are distinguished by how they compute the new value to be stored. The result of an assignment expression is never an lvalue.

7.9.1. C-Ref: Simple Assignment

The simple assignment operator `=` indicates simple assignment. The value of the right operand is stored into the left operand. The two operands may each be of any arithmetic type, in which case the usual assignment conversions are used to convert the right operand to the type of the left operand before assignment. The two operands may also be of the same pointer, structure, or union type. Finally, it is permitted for the left operand to be of any pointer type and the right operand to be the integer constant 0; this has the effect of assigning a null pointer to the pointer lvalue, guaranteed not to point at any object.

The type of the result is equal to the (unconverted) type of the left operand. The result is the value stored into the left operand. The result is not an lvalue.

The simple assignment operator `=` cannot be used to copy the entire contents of one array into another, for two reasons. First, the name of an array is not an lvalue and so cannot appear on the left-hand side of an assignment. Second, the name of an array appearing on the right-hand side of an assignment would be converted (by the usual conversions) to be a pointer to the first element, and so the assignment would copy the pointer, not the contents of the array. The `=` operator can, therefore, be used to copy the address of an array into a pointer variable:

```
{
    int a[20], *p;
    p = a;
    ...
}
```

In this example, a is an array of integers and p is of type "pointer to integer." The assignment causes p to point to (the first element of) the array a.

It is possible to get the effect of copying an entire array by embedding the array within a structure or union, because simple assignment can copy an entire structure or union:

```
struct matrix {double contents[10][10]; };

struct matrix a, b;
...
{
    /* Copy structure containing a 10x10 array. */
    a = b;
    /* Clear the diagonal elements. */
    for (j = 0; j < 10; j++)
        a.contents[j][j] = 0;
}
```

In the original description of C, assignment of structure and union objects was not permitted. Nearly all C compilers now permit structure and union assignment.

7.9.2. C-Ref: Compound Assignment

The compound assignment operators may be informally understood by taking the expression "*a op*= *b*" to be equivalent to "*a* = *a op b*," with the proviso that the expression *a* is evaluated only once. More precisely, the left and right operands of *op*= are evaluated, and the left operand must be an lvalue. The operation indicated by the operator *op* is then applied to the two operand values; this includes any "usual conversions" performed by the operator. The resulting value is then stored into the left operand lvalue, after performing the usual assignment conversions.

For the operators += and -=, the two operands may each be of any arithmetic type. It is also permitted for the left operand to be of pointer type and the right operand to be of integral type.

For the operators *= and /=, the two operands may each be of any arithmetic type.

For the operator %=, the two operands may each be of any integral type.

For the operators <<=, >>=, the two operands may each be of any integral type. For portable code we recommend that only unsigned operands be used as the left operand for each of these operators.

For the operators &=, ^=, and |=, the two operands may each be of any integral type. For portable code we recommend that only unsigned operands be used with these operators.

For the compound assignment operators, as for the simple assignment operator, the type of the result is equal to the (unconverted) type of the left operand. The result is the value stored into the left operand. The result is not an lvalue.

Historical note: In the earliest versions of C, the compound assignment operators were written in the reverse form:

```
=+   =-   =*   =/   =%   =<<   =>>   =&   =^   =|
```

This led to syntactic ambiguities. For example, the expression

```
x=-1
```

might,on the face of it, be interpreted either as

```
x = (-1)
```

or as

```
x =- (1)
```

While the ambiguity was arbitrarily resolved by requiring the latter interpretation, this was found to be quite prone to subtle programming errors in practice. The newer form (+= instead of =+) eliminates these difficulties. A few compilers continue to support the older forms for the sake of compatibility.

7.10. C-Ref: Sequential Expressions

A comma expression consists of two expressions separated by a comma. The comma operator is described here as being syntactically left-associative, though this doesn't matter much to the programmer because the operator happens to be fully associative semantically.

comma-expression :
  assignment-expression
  comma-expression , *assignment-expression*

expression :
  comma-expression

Note that the *comma-expression* is at the top of the expression syntax tree.

The left operand of the comma operator is fully evaluated first. It need not produce any value; if it does produce a value, that value is discarded. The right operand is then evaluated. The type and value of the result of the comma expression are equal to the type and value of the right operand. The result is not an lvalue.

The comma operator is associative, and so one may write a single expression consisting of any number of expressions separated by commas; the subexpressions will be evaluated in order, and the value of the last one will become the value of the entire expression.

In certain contexts the comma character is used for another syntactic purpose. Expressions written within these contexts may not use the comma operator lest ambiguity arise. This restriction can always be circumvented by using parentheses to

enclose the comma operator expression. For example, the expression

```
f(a, b = 5, 2*b, c)
```

is always treated as a call to the function `f` with four arguments. If it is desired to treat the second comma as the comma operator, and to call `f` with three arguments, additional parentheses should be inserted, thus:

```
f(a, (b = 5, 2*b), c)
```

The contexts where the comma operator may not be used because of potential ambiguity include argument expressions in function calls; field-length expressions in structure and union declarator lists; enumeration value expressions in enumeration declarator lists; and initialization expressions in declarations and initializers. Note that the comma character is also used as a separator in preprocessor macro calls.

The comma operator guarantees that its operands will be evaluated in left-to-right order, but other uses of the comma character do not make this guarantee. For example, the argument expressions in a function invocation need not be evaluated in left-to-right order.

The most important application of the comma operator is in `for` statements; it allows several assignment expressions to be combined into a single expression for the purpose of initializing or stepping several variables in a single loop.

7.11. C-Ref: Constant Expressions

In several contexts the C language permits an expression to be written that must evaluate to a constant at compile time (or, in some situations, at link time, but in any case before execution of the program proper). These contexts are:

1. the tested value in the `#if` preprocessor control statement
2. array bounds
3. `case` labels in `switch` statements
4. bit-field lengths in structure declarators
5. explicit enumerator values
6. initializers for static and external variables

Each context imposes slightly different restrictions on what forms of expression are permitted.

In all cases the result of evaluating a constant expression is identical to the result of evaluating the same expression at run time. For the compiler implementor, this consistency requirement is particularly important in the case of cross-compilation, where the computer being used to execute the compiler does not necessarily have the same architecture as the computer used to execute the compiled program. Al-

though most kinds of integer arithmetic can be simulated well enough, floating-point arithmetic—even simply converting a floating-point constant from character form to internal form—may be difficult enough that the implementor may choose to do the evaluation of constant floating-point expressions at run time. (C requires no compile-time floating-point arithmetic capability.)

All constant expressions may contain integer constants and character constants. The binary operators

```
*    /    %    +    -    <<   >>   ==   !=
<    <=   >    >=   &    ^    |    &&   ||
```

may be used, as may the unary operators

```
-    ~    !
```

The conditional operator

```
? :
```

may be used. Parentheses may be freely used for grouping.

Other constants and operations may be permitted in constant expressions according to context.

1. Preprocessor `#if` statements permit the use of the operator `defined`.

2. Array bounds, `case` labels, field lengths, and explicit enumerator values permit the use of enumeration constants, the `sizeof` operator, and casts to integral types. The argument to `sizeof` need not be a constant expression, since only the type of the expression is needed and that type can be determined at compile time.

3. Initializers for static and external variables additionally permit the use of floating-point constants and arbitrary casts. The unary `&` operator is permitted when applied to the name of a static or external object or to the result of subscripting a static or external array by a constant expression. It is also permitted to use the name of a function to refer to its address, or to the name of a static or external array to refer to its address. Semantically, an initializer for a static or external variable must evaluate either to a constant or to the address of a previously declared static or external object plus or minus an integer constant.

The original description of C did not mention that the operator `!` is allowed in constant expressions, but this obviously was an oversight or typographical error. All C implementations should support this operator in constant expressions.

Some compilers do not permit casts of any kind in constant expressions. Others allow casts other than casts to integral types. When writing portable code, it is best for the programmer to avoid casts in constant expressions.

Some compilers permit the comma operator in constant expressions. We don't see the use of it, inasmuch as the left operand of the comma operator is useful only for its side effects, and constant expressions may not have side effects. The syntax for *constant-expression* given in this text in fact permits any form of *expression*, in-

cluding *comma-expression* and *assignment-expression*. The restrictions on constant expressions come from the semantic rules of C, not the syntactic rules.

7.12. C-Ref: Order of Evaluation

In general, the compiler is free to rearrange the order in which an expression is evaluated with the following restrictions.

The rearrangement may consist only of evaluating the arguments of a function call, or the two operands of a binary operator, in some particular order other than the obvious left-to-right order. (The compiler is under no compulsion ever to use left-to-right order for such operators.) There is one additional rule: The binary operators +, *, &, ^, and | are assumed to be completely associative and commutative, and a compiler is permitted to exploit this assumption. For instance, addition is assumed to be commutative and associative, so the compiler is free, for example, to evaluate "(a+b)+(c+d)" as if it were written "(a+d)+(b+c)" (assuming all variables have the same arithmetic type).

The assumption of commutativity and associativity is indeed always true for &, ^, and | on unsigned operands. It may not be true for &, ^, and | on signed operands because of potential problems with certain signed representations. It may not be true for * and + because of the possibility that the order indicated by the expression as written might avoid overflow but another order might not. Nevertheless, the compiler is allowed to exploit the assumption. In such situations the programmer must use assignments to temporary variables to force a particular evaluation order:

```
{
    int temp1, temp2;
    ...
    /* Compute q=(a+b)+(c+d), exactly that way. */
    temp1 = a+b;
    temp2 = c+d;
    q = temp1 + temp2;
}
```

The original description of C placed no restrictions on the exploitation of the assumption of the associativity and commutativity of *, +, &, ^, and |. However, the following restriction seems to be very important and should be observed by every compiler: Any rearrangement of expressions involving these operators must not alter the implicit type conversions of the operands. In the example below, the two assignment statements are not equivalent and the compiler is not free to substitute one for the other, despite the fact that one is obtained from the other "merely by reassociating the additions."

```
x = (1.0 + -3) + (unsigned) 1;   /* Result is -1.0 */
x = 1.0 + (-3 + (unsigned) 1);   /* Result is large */
```

The first assignment is straightforward and produces the expected result. The second produces a large result, because the usual binary conversions cause the signed

value `-3` to be converted to a large unsigned value 2^n-3, where n is the number of bits used to represent an unsigned integer. This is then added to the unsigned value `1`, the result converted to floating-point representation and added to `1.0`, resulting in the value 2^n-3 in a floating-point representation. Now, this result may or may not be what the programmer intended, but the compiler must not confuse the issue further by capriciously rearranging the additions.

When evaluating the actual arguments in a function call, the order in which the arguments are evaluated is not specified; but the program must behave as if it chose one argument, evaluated it fully, then chose another argument, evaluated it fully, and so on, until all arguments were evaluated. That is, the arguments may be evaluated in any order, but their computations may not appear to be interleaved. Consider this example:

```
#define SIZE 100
{
    char *x[10], **p=x;
    ...
    if ( strcmp(*p++, *p++) == 0 ) printf("Same.");
    ...
}
```

The variable `x` is an array of pointers to characters and is to be regarded as an array of strings. The variable `p` is a pointer to a pointer to a character and is to be regarded as a pointer to a string. The purpose of the `if` statement is to determine whether the string pointed to by `p` (call it *s1*) and the next string after that (call it *s2*) one are equal (and, in passing, to step the pointer `p` beyond those two strings in the array). It is, of course, bad programming style to have two side effects on the same variable in the same expression, because the order of the side effects is not defined; but the all-too-clever programmer here has reasoned that the order of the side effects doesn't matter, because the two strings in question may be given to `strcmp` in either order.

If it were permitted for the compiler to evaluate the two expressions in an interleaved manner, it might generate code something like this:

1. Fetch what `p` points to for first argument.

2. Fetch what `p` points to for second argument.

3. Increment `p` for first argument.

4. Increment `p` for second argument.

The net result would be that *s1* would be passed as both arguments to `strcmp` and *s2* would not be passed at all. The restriction against interleaving prohibits such behavior. Our too-clever programmer is therefore justified in believing the program will behave as intended. (However, we would not want to have to maintain that code!)

A similar restriction holds for binary operators: The two operands may be evaluated in either order, but the program must behave as if one of the two operands were evaluated completely before commencement of the evaluation of the other operand.

The original description of C specified that subexpressions may be evaluated in any order and that the arguments to a function may be evaluated in any order. The matter of interleaving was not discussed, nor the question of whether rearranging may alter the implicit type conversions. We advise implementors to adhere rigidly to the restrictions outlined here (which actually are quite sensible and not terribly restrictive). We also advise programmers not to exploit these restrictions too cleverly (as in the example above using `strcmp`). The entire purpose of the restrictions is to make the behavior of a program more understandable.

7.13. C-Ref: Discarded Values

There are three contexts in which an expression can appear but its value is not used:

1. an expression statement

2. the first operand of a comma expression

3. the initialization and incrementation expressions in a `for` statement

In these contexts we say that the expression's value is *discarded*.

When the value of an expression without side effects is discarded, the compiler may presume that an error has been made and issue a warning. Side-effect producing operations include assignment and function calls. For example:

```
{
    extern void f();
    f(x);      /*  These expressions do not   */
    i++;       /*  justify any warning about  */
    a = b;     /*  discarded values.          */
}
```

The compiler may issue a warning message if the main operator of a discarded expression has no side effect. For example, these statements, though legal, may elicit warning messages:

```
{
    extern int g();
    g(x);    /* The call to g may have side effects
                but it also returns a value that is
                discarded. */
    x + 7;   /* Addition has no defined
                side effects. */
    x + (a *= 2);
             /* The expression has a side effect,
                but the last operation to be
                performed, "+", does not, and its
                value is discarded. */
}
```

The programmer may avoid warnings about discarded values by using a cast to type `void` to indicate that the value is purposely being discarded:

```
{
    extern int g();
    (void) g(x);    /* The returned value is
                       purposely discarded. */

    (void)(x + 7);  /* This is pretty silly, but
                       presumably the programmer
                       has a purpose. */
}
```

A compiler that does not implement the `void` type should not issue warnings when the value of a function call is discarded, because it is likely that the function being called is conceptually of type "function returning void," even though the programmer has no way to say this to the compiler.

If a compiler determines that the main operator of a discarded expression has no side effect, it may choose not to generate code for that operator (whereupon its operands become discarded values and may be recursively subjected to the same treatment).

7.14. C-Ref: Compiler Optimization of Memory Accesses

As a general rule, a compiler is free to generate any code equivalent in computational behavior to the program as written. The compiler is explicitly granted certain freedoms to rearrange code, as described in section "C-Ref: Order of Evaluation". It may also generate no code for an expression when the expression has no side effects and its value is discarded, as described in section "C-Ref: Discarded Values".

Some compilers may also reorganize the code in such a way that it does not always refer to memory as many times, or in the same order, as specified in the program. For example, if a certain array element is referred to more than once,

the compiler may cleverly arrange to fetch it only once to gain speed; in effect, it might rewrite this code:

```
{
    int x;
    x = a[j] * a[j] * a[j];
                /* Cube the table entry. */
}
```

causing it to be executed as if it had been written like this:

```
{
    int x;
    register temp;
    temp = a[j];
    x = temp * temp * temp;
                /* Cube the table entry. */
}
```

For most applications, including nearly all portable applications, such optimization techniques are a very good thing, because the speed of a program may be improved by a factor of two or better without altering its effective computational behavior.

However, this may be a problem when writing certain machine-dependent programs in C, such as operating systems. On some computers the status registers of I/O devices may be accessed as if they were memory locations, and accessing such a register may have side effects. The way to read a character from a terminal might be to access a specific "memory location"; every time the location is read, a new character is obtained.

Consider this code to read characters in such a manner:

```
/* Address of the keyboard input register. */
#define KEYBOARD ((char *) 0177614)
...
c1 = *KEYBOARD;   /* Get first character. */
...
c2 = *KEYBOARD;   /* Get next character. */
```

It would be disastrous if this code were to be compiled as if it had been written this way:

```
/* Address of the keyboard input register. */
#define KEYBOARD ((char *) 0177614)
...
temp = *KEYBOARD;
c1 = temp;
...
c2 = temp;
```

A similar difficulty might arise when doing output by writing successive characters to a special "memory location": The compiler might notice that the location is written into and then written into again without being accessed, and therefore cleverly

eliminate the first write operation. This is a good optimization for ordinary memory locations, but disastrous when an I/O register is involved.

We emphasize that this kind of problem does not arise in most applications, and it need not concern most programmers. The programmer who is writing low-level machine-dependent programs, such as operating systems, should carefully study the documentation for the specific C compiler to be used to determine whether problems like this may arise. In Draft Proposed ANSI C, the new declaration modifier `volatile` was introduced to deal with these situations.

8. C-Ref: Statements

The C language provides the usual assortment of statement forms found in most algebraic programming languages, including conditional statements, loops, and the ubiquitous "goto."

statement :
- *expression-statement*
- *labeled-statement*
- *compound-statement*
- *conditional-statement*
- *iterative-statement*
- *switch-statement*
- *break-statement*
- *continue-statement*
- *return-statement*
- *goto-statement*
- *null-statement*

conditional-statement :
- *if-statement*
- *if-else-statement*

iterative-statement :
- *do-statement*
- *while-statement*
- *for-statement*

We describe each of the statements in turn after some general comments about the syntax of statements.

8.1. C-Ref: General Syntactic Rules for Statements

Although C statements will be familiar to programmers used to Algol-like languages, there are a few syntactic differences that are often the cause of confusion and errors.

8.1.1. C-Ref: Semicolons

As in Pascal or Ada, semicolons typically appear between consecutive statements in C. However, in C the semicolon is not a statement separator, but rather simply a part of the syntax of certain statements. The only C statement that does not require a terminating semicolon is the compound statement (or block), which is delimited by braces ({ and }) instead of the more usual `begin` and `end` keywords. For example, the Pascal (or Ada or Algol) statements

```
t := b;
begin b := a end;
a := t;
```

are written in C as

```
t = b;
{ b = a; }
a = t;
```

Note that in C a semicolon follows "b = a" but not }, whereas the situation is reversed in the other languages.

8.1.2. C-Ref: Control Expressions

Another rule for C statements is that "control" expressions, appearing in conditional or iterative statements, must be enclosed in parentheses. These parentheses obviate the need for a keyword following the expression, such as "then" or "do." For example, the Pascal statement

```
if x = y then
    while x = z do
        process(x);
```

is rendered in C as

```
if ( x == y )
    while (x == z)
        process(x);
```

In all cases, if the control expression is 0, it is taken to be "false"; if it is nonzero, it is taken to be "true." More precisely, the type of a control expression *e* must be such that the expression

(*e*) != 0

may be legally evaluated. If the result of this last expression is 1, *e* is said to be nonzero; otherwise, *e* is said to be zero. In practice, this means that *e* may have integral, pointer, or floating-point type. Values of enumeration types may also be permitted depending on the semantics chosen for enumerations.

8.2. C-Ref: Expression Statements

Any expression can be treated as a statement by writing the expression followed by a semicolon.

expression-statement :
    expression ;

The statement is executed by evaluating the expression and then discarding the value, if any.

An expression statement is useful only if evaluation of the expression involves a side effect, such as assigning a value to a variable or performing input or output.

Usually the expression is an assignment, an incrementation or decrementation operation, or a function call. Here are some examples of expression statements:

```
speed = distance / time; /* Assign quotient to speed. */
++event count;           /* Add 1 to event count.     */
printf("Another game?"); /* Call the function printf. */
pattern &= mask;         /* Mask out some bits of
                                         the pattern. */
(x < y) ? ++x : ++y;     /* Increment the smaller of
                                            x and y. */
```

The last example, though legal, might be written more clearly with an `if` statement:

```
if (x < y) ++x; else ++y;
```

The compiler is not obliged to evaluate an expression, or a portion of an expression, that has no side effects and whose result is discarded. This is discussed in more detail in section "C-Ref: Discarded Values".

8.3. C-Ref: Labeled Statements

A label can be used to mark any statement so that control may be transferred to the statement by a `goto` or `switch` statement. There are three kinds of labels. A named label may appear on any statement and is used in conjunction with the `goto` statement. A `case` label or `default` label may appear only on a statement within the body of a `switch` statement.

labeled-statement :
  label : *statement*

label :
  named-label
  case-label
  default-label

A label cannot appear by itself but must always be attached to a statement. If it is desired to place a label by itself, for example at the end of a compound statement, it may be attached to a null statement.

Named labels are discussed further in the description of the `goto` statement. The labels `case` and `default` are discussed further in the description of the `switch` statement.

8.4. C-Ref: Compound Statement

A compound statement—also called a block—consists of a (possibly empty) sequence of declarations followed by a (possibly empty) sequence of statements, all enclosed in braces.

compound-statement :
{ *inner-declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

statement-list :
statement
statement-list statement

A compound statement may appear anywhere a statement does. When the compound statement has no declarations, it just represents a group of statements. When the compound statement has declarations, it brings into existence a new scope.

A compound statement is normally executed by first processing all the declarations one at a time, in sequence, and then executing all the statements one at a time, in sequence. Execution ceases when the last statement has been executed or when control is transferred out of the compound statement through execution of a `goto`, `return`, `continue`, or `break` statement.

It is also possible to jump to a labeled statement within a compound statement by using a `goto` or `switch` statement outside the compound statement. When that happens, storage is allocated for any `auto` or `register` variables declared in the compound statement, but any initialization expressions for those variables are not evaluated and no initialization occurs. Execution then begins at the statement to which control was transferred and continues in sequence until the last statement has been executed or until control is transferred out of the compound statement through execution of a `goto`, `return`, `continue`, or `break` statement.

An unlabeled compound statement used as the body of a `switch` statement cannot be executed normally but only through transfer of control to labeled statements within it. Therefore, initializations of `auto` and `register` variables in such a compound statement never occur and their presence is a priori an error.

8.4.1. C-Ref: Declarations Within Compound Statements

Each identifier declared at the beginning of a compound statement has a scope that extends from its declaration point to the end of the block. It is visible throughout that scope except when hidden by a declaration of the same identifier in an inner block.

An identifier declared at the beginning of a compound statement without a storage class specifier is assumed to have storage class `extern` if the type of the identifier is "function returning...," and is assumed to have storage class `auto` in all other cases. It is illegal for an identifier of function type to have any storage class except `extern` when it is declared at the beginning of a block.

If a variable or function is declared in a compound statement with storage class extern, then no storage is allocated and no initialization expression is permitted. The declaration refers to an external variable or function defined elsewhere, either in the same source file or a different source file.

If a variable is declared in a compound statement with storage class auto or register, then it is effectively reallocated every time the compound statement is entered and deallocated when the compound statement is exited. If there is an initialization expression for the variable, then the expression is reevaluated and the variable reinitialized every time the compound statement is entered normally. (The initialization expression is not evaluated and the variable not initialized when control is passed to a statement within the compound statement via a goto or switch statement from outside.) If there is no initialization expression for the variable, then the value of the variable is initially undefined every time the compound statement is executed; the value of the variable does not carry over from one execution of the compound statement to the next.

If a variable is declared in a compound statement with storage class static, then it is effectively allocated once, prior to program execution, just like any other static variable. If there is an initialization expression for the variable, then the expression is evaluated only once, prior to program execution, and the variable retains its value from one execution of the compound statement to the next.

8.4.2. C-Ref: Use of Compound Statements

Compound statements without declarations are particularly useful as parts of other control statements, so that more than one statement can be executed conditionally or in a loop:

```
if (error seen) {
    ++error count;
    print error message();
}
```

With declarations, compound statements can also introduce additional variables with reduced visibility. This often helps to make a program clearer by restricting the area over which a variable is accessible:

```
if (first time) {
    /* Clear the array. */
    int i;
    for (i = 0; i < 10; i++)
        a[i] = 0;
    /* Reset first-time flag. */
    first time = 0;
}
```

C permits unrestricted jumps into compound statements, but we feel this is bad programming style. In fact, none of the languages Ada, Algol 60, Modula-2, Pascal, or PL/I permit jumps into blocks. The particular danger in C is not having initializations occur. For example, the following code fragment is unlikely to work if the statement labeled L: is jumped to from outside the compound statement, because

the variable sum will not be initialized. Furthermore, it is not possible to tell if any such jump does occur without examining at least the entire body of the enclosing function.

```
{
    extern int a[100];
    int i, sum = 0;
  L:
    for (i = 0; i < 100; i++)
        sum += a[i];
    ...
}
```

8.5. C-Ref: Conditional Statement

There are two forms of conditional statement: with or without an else clause. Each begins with the keyword if, followed by a control expression in parentheses, followed by a statement; there may be appended to this the keyword else and then another statement. Note that C, unlike other programming languages such as Pascal, does not use the keyword then as part of the syntax of its if statement.

conditional-statement :
  if-statement
  if-else-statement

if-statement :
  if (*expression*) *statement*

if-else-statement :
  if (*expression*) *statement* else *statement*

For each form of if statement the expression within parentheses is first evaluated. If this value is nonzero (section "C-Ref: Control Expressions"), then the statement immediately following the parentheses is executed. If the value of the control expression is zero and there is an else clause, then the statement following the keyword else is executed instead; but if the value of the control expression is zero and there is no else clause, then execution continues immediately with the statement following the conditional statement.

8.5.1. C-Ref: Multiway Conditional Statements

A multiway decision can be expressed as a cascaded series of if-else statements, where each if statement but the last has another if statement in its else clause. Such a series looks like this:

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
...
else
    statementn
```

Here is an example of a three-way decision: The function signum returns one of three results depending on its argument.

```
/* Return 1, -1, or 0 if x is positive,
   negative, or zero, respectively. */
int signum(x)
  int x;
{
    if (x > 0) return 1;
    else if (x < 0) return -1;
    else return 0;
}
```

Compare this with the version of signum that uses conditional expressions shown in section "C-Ref: Conditional Expressions".

The switch statement handles the specific kind of multiway decision where the value of an expression is to be compared against a fixed set of constants.

8.5.2. C-Ref: The Dangling Else Problem

An ambiguity arises because a conditional statement may contain another conditional statement: In some situations it may not be apparent to which of several conditional statements an else might belong. Consider this example:

```
/* Warning: this example is indented
                 in a misleading fashion. */
if ((k >= 0) & (k < TABLE SIZE))
    if (table[k] >= 0)
     printf("Entry %d is %d\n", k, table[k]);
  else printf("Error: index %d out of range.\n",k);
```

Inspection of the code might lead one to assume that whoever wrote this code intended the else part to be an alternative to the outer if statement: The error message should be printed when the test

```
(k >= 0) & (k < TABLE SIZE)
```

is false. However, if we change the wording of the last error message to

```
else printf("Error: entry %d is negative.\n", k);
```

then one might assume that the programmer intended the else part to be executed when the test

```
table[k] >= 0
```

is false.

The C language does not require the compiler to interpret the meanings of error messages and make assumptions about the programmer's intentions. Instead, the ambiguity is resolved in an arbitrary but customary way: An `else` part is always assumed to belong to the *innermost* `if` statement possible. From this rule we see that the second interpretation of the code fragment above will work as intended, while the first will not. The first fragment can be made to work as intended by introducing a compound statement:

```
if (k >= 0 & k < TABLE SIZE) {
    if (table[k] >= 0)
        printf("Entry %d is %d\n", k, table[k]);
}
else printf("Error: index %d out of range.\n", k);
```

To reduce confusion, the second interpretation could also use a compound statement:

```
if (k >= 0 & k < TABLE SIZE) {
    if (table[k] >= 0)
        printf("Entry %d is %d\n", k, table[k]);
    else printf("Error: entry %d is negative.\n",k);
}
```

Confusion can be eliminated entirely if braces are always used to surround statements controlled by an `if` statement. However, this conservative rule can clutter a program with unnecessary braces. It seems to us that a good stylistic compromise between confusion and clutter is to use braces with an `if` statement whenever the statement controlled by the `if` is anything but an expression statement.

8.6. C-Ref: Iterative Statements

Three kinds of iteration statements are provided in C.

iterative-statement :
    while-statement
    do-statement
    for-statement

The `while` statement tests an exit condition *before* each execution of a statement. The `do` statement tests an exit condition *after* each execution of a statement. The `for` statement provides a special syntax that is convenient for initializing and updating one or more control variables as well as testing an exit condition. The statement embedded within an iteration statement is sometimes called the *body* of the statement.

8.6.1. C-Ref: While Statement

A `while` statement consists of the keyword `while`, followed by a control expression in parentheses, followed by a statement.

while-statement :
`while` (*expression*) *statement*

Note that C, unlike other programming languages such as Pascal, does not use the keyword "`do`" as part of the syntax of its `while` statement.

The `while` statement is executed by first evaluating the control expression. If the result is not zero, then the statement is executed. The entire process is then repeated, alternately evaluating the expression and then, if the value is not zero, executing the statement. The value of the expression can change from time to time because of side effects in the statement or in the expression itself.

The execution of the `while` statement is complete when the control expression evaluates to zero, or when control is transferred out of the body of the `while` statement by a `return`, `goto`, or `break` statement. Also, the `continue` statement can modify the execution of a `while` statement.

As an example, the following code fragment uses a `while` loop to raise an integer `x` to the power specified by the nonnegative integer `y` (with no checking for overflow):

```
/* Compute x to the power y by repeated squaring. */
{
    int base = x;
    int exponent = y;
    int z = 1;
    while (exponent > 0) {
        if ( exponent % 2 ) /* If exponent is odd,    */
            z *= base;      /*   multiply z by base. */
        base *= base;       /* Square the base.      */
        exponent /= 2;      /* Divide exponent by 2. */
    }
    /* Now z is equal to x raised to the power y. */
}
```

The method used is that of repeated squaring of the base and decoding of the exponent in binary notation to determine when to multiply the base into the result. (To see why this works, note that the `while` loop maintains the invariant condition that `z` times `base` raised to the `exponent` power is equal to `x` raised to the `y` power. When eventually the `exponent` is 0, this condition degenerates to simply `z` equals `x` raised to the `y` power, which is the desired result.)

A `while` loop may usefully have a null statement for its body:

```
while ( *char pointer++ );
```

The character pointer is advanced along by the `++` operator until a null character is found, and it is left pointing to the character after the null. This is a compact idiom for locating the end of a string. (Notice that the test expression depends on

the fact that the postfix operator ++ has higher precedence than the indirection operator *. The test expression is interpreted as `*(char pointer++)`, not as `(*char pointer)++`, which would increment the character pointed to by `char pointer`.)

A variation on this idea uses two pointers to copy a character string from one place to another:

```
while ( *dest pointer++ = *source pointer++ );
```

Characters are copied until the terminating null character is found (and also copied). Of course, in writing this the programmer should have reason to believe that the destination area will be large enough to contain all the characters to be copied.

8.6.2. C-Ref: Do Statement

A `do` statement consists of the keyword `do`, followed by a statement, followed by the keyword `while`, followed by a control expression in parentheses, followed by a semicolon.

do-statement :
  `do` *statement* `while (` *expression* `) ;`

The `do` statement is executed by first executing the embedded statement. Then the control expression is evaluated; if the value is not zero, then the entire process is then repeated, alternately executing the statement, evaluating the control expression, and then, if the value is not zero, repeating the process. Note that the value of the expression can change from time to time because of side effects in the statement or in the expression itself.

The execution of the `do` statement is complete when the control expression evaluates to zero or when control is transferred out of the body of the `while` statement by a `return`, `goto`, or `break` statement. Also, the `continue` statement can modify the execution of a `do` statement.

The `do` statement differs from the `while` statement in that the `do` statement always executes the body at least once, whereas the `while` statement may never execute its body at all.

The C `do` statement is similar in function to what is often called a "repeat-until" statement in other programming languages such as Pascal. The C `do` statement is unusual in that it terminates execution when the control expression is false, whereas a Pascal repeat-until statement terminates if its control expression is true. C is more consistent in this regard: all iteration constructs in C (`while`, `do`, and `for`) terminate when the control expression is false.

As an example of the use of the `do` statement, consider this program fragment that reads and processes characters, halting after a newline character has been processed.

```
int ch;
do {
    ch = getchar();
    process(ch);
} while (ch != '\n');
```

The same effect could have been obtained by moving the computations into the control expression of a `while` statement, but the intent would be less clear:

```
int ch;
while( ch = getchar(ch),
       process(ch),
       ch != '\n' );
```

It is possible to write a `do` statement whose body is a null statement:

`do ; while (`*expression*`);`

It is silly to do so, however, because such a `do` statement is identical in meaning to a `while` statement whose body is a null statement:

`while (`*expression*`);`

8.6.3. C-Ref: For Statement

C's `for` statement is considerably more general than the "increment and test" statements found in most other languages. After explaining the execution of the `for` statement, we give several examples of how it can be used.

for-statement :
  `for` *for-expressions statement*

for-expressions :
  (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$)

A `for` statement consists of the keyword `for`, followed by three expressions separated by semicolons and enclosed in parentheses, followed by a statement. Each of the three expressions within the parentheses is optional and may be independently omitted, but the two semicolons separating them and the parentheses surrounding them are mandatory.

Typically, the first expression is used to initialize a loop variable, the second tests whether the loop should continue or terminate, and the third updates the loop variable (for example, by incrementing it). However, in principle the expressions may be used to perform any computation that is useful within the framework of the `for` control structure.

The `for` statement is executed as follows:

1. If present, the first expression is evaluated and the value is discarded.

2. If present, the second expression is evaluated like a control expression. If the result is zero, then execution of the `for` statement is complete. Otherwise (if the value is not zero or if the second expression was omitted), proceed to step 3.

3. The body of the `for` statement is executed.

4. If present, the third expression is evaluated and the value is discarded.

5. Return to step 2.

The execution of a `for` statement is terminated when the second (control) expression evaluates to zero or when control is transferred outside the `for` statement by a `return`, `goto`, or `break` statement. The execution of a `continue` statement within the body of the `for` statement has the effect of causing a jump to step 4 above.

Stated another way, a `for` loop of the form

```
for (expression1; expression2; expression3) statement
```

is similar (except for the action of the `continue` statement) to

```
{
    expression1;
    while (expression2) {
        statement
        expression3;
    }
}
```

where if *expression1* or *expression3* is not present in the `for` statement, then it is simply omitted in the expansion also, and if *expression2* is not present in the `for` statement, then the constant `1` is used for it in the expansion (and so the `while` loop never terminates due to the control expression becoming zero).

8.6.4. C-Ref: Using the For Statement

The standard way in C to write a loop that "never terminates" (sometimes known as a "do forever" loop) is as a `for` loop with no expressions:

```
for (;;) statement
```

Of course, the loop can still be terminated by a `break`, `goto`, or `return` statement within the body. One can write a "do forever" loop in other ways, such as

```
while (1) statement
```

but the idiom using `for` with no expressions is customary.

Typically, the first expression in a `for` statement is used to initialize a variable, the second expression to test the variable in some way, and the third to modify the variable toward some goal. For example, to print the integers from 0 to 9 and their squares, one might write

```
for (j = 0; j < 10; j++)
    printf("%d  %d\n", j, j*j);
```

Here the first expression initializes `j`; the second expression tests whether it has reached `10` yet (if it has, the loop is terminated); and the third expression increments `j` by 1.

The example of raising an integer to an integer power given above to illustrate the `while` statement can be rewritten using a `for` statement:

```
/* Compute x to the power y by repeated squaring. */
{
    int base = x;
    int exponent;
    int z = 1;
    for (exponent = y; exponent > 0; exponent /= 2) {
        if ( exponent % 2 ) /* If exponent is odd,  */
            z *= base;      /*  multiply z by base. */
        base *= base;       /* Square the base.     */
    }
    /* Now z is equal to x raised to the power y. */
}
```

This form stresses the fact that the loop is controlled by the variable `exponent` as it begins at the value `y` and progresses toward 0 by repeated divisions by 2. Note that the loop variable `exponent` still had to be declared outside the `for` statement. The `for` statement itself does not include the declaration of any variables. A common programming error is to forget to declare a variable such as `i` or `j` used in a `for` statement, only to discover that some other variable named `i` or `j` elsewhere in the program is inadvertently modified by the loop.

The `for` statement need not be used only for counting over integer values. Here is an example of scanning down a linked chain of structures, where the loop variable is a pointer:

```
struct intlist {
    struct intlist *link;
    int data;
};

void print duplicates(p)
struct intlist *p;
{
    for (; p; p = p->link) {
        struct intlist *q;
        for (q = p->link; q; q = q->link)
            if (q->data == p->data) {
                printf("Duplicate data %d", p->data);
                break;
            }
    }
}
```

The structure `intlist` is used to implement a linked list of records, each record containing some data. Given such a linked list, the function `print duplicates` prints the data for every redundant record in the list. A record is considered to be redundant if some other record after it in the list contains the same data. (If several records in the list have the same data, all but the last one are considered

redundant.) The first for statement cleverly (perhaps too cleverly) uses the formal parameter p as its loop variable; it scans down the given list. The loop terminates when a null pointer is encountered. For every record, all the records following it are examined by the inner for statement, which scans a pointer q along the list in the same fashion. If the record pointed to by p is discovered to be redundant, the break statement is used to terminate the inner loop, to prevent the data for p from being printed more than once.

As another example of nested for loops, here is a simple sorting routine that uses the insertion sort algorithm.

```
/* Sort v[0]...v[n-1] into increasing order. */
void insertsort(v, n)
  register int v[], n;
{
    register int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = v[i];
        for (j = i-1; j >= 0 && v[j] > temp; j--)
            v[j+1] = v[j];      v[j+1] = temp;
    }
}
```

The outer for loop counts i up from 1 (inclusive) to n (exclusive). At each step, elements v[0] through v[i-1] have already been sorted, and elements v[i] through v[n-1] remain to be sorted. The inner loop counts j down from i-1, moving elements of the array up one at a time, until the right place to insert v[i] has been found. (That is why this is called "insertion sort.") Notice that the termination test for the inner loop uses the && operator. This prevents the array reference v[j] from being executed when j is less than 0.

Insertion sort is a simple and efficient sorting method for small arrays (for n less than, say, 20) or for arrays that are already almost sorted. It is not a good method for very large unordered arrays, because in the worst case the time to perform the sort is proportional to the square of the number of items to be sorted.

A simple modification to insertion sort can make it astonishingly more efficient by wrapping a third loop around the first two! The following sort routine, using the shell sort algorithm, is similar to one called shell that appeared as an example in the original description of C. The original shell routine was a good example of the use of three different nested for loops in a practical setting. However, we have modified it here in four ways, two of them suggested by Knuth and Sedgewick, to make it faster.

```
/* Sort v[0]...v[n-1] into increasing order. */
void shellsort(v, n)
  register int v[], n;
{
    register int gap, i, j, temp;
```

```
        gap = 1;
        do (gap = 3*gap + 1); while (gap <= n);
        for (gap /= 3; gap > 0; gap /= 3)
            for (i = gap; i < n; i++) {
                temp = v[i];
                for (j=i-gap; (j>=0)&&(v[j]>temp); j-=gap)
                    v[j+gap] = v[j];
                v[j+gap] = temp;
            }
    }
```

In the older version, shell, the value of gap started with n/2, and gap was divided by two each time through the outer loop; in this version, shellsort, gap is initialized by finding the smallest number in the series 1, 4, 13, 40, 121, ... that is not greater than n, and gap is divided by three each time through the outer loop. This is the first improvement. It makes the sort run about 20% to 30% faster on the average. (This is an empirical result; it is not yet completely understood theoretically why this should be so. Experiments also show that one should *not* initialize gap to n and then divide by three each time; such a strategy produces a very poor sorting routine. It is important to start with an element from the series 1, 4, 13, 40, 121, ...)

The second improvement is that the number of assignments is reduced because the inner loop of shellsort contains only one assignment, compared with three assignments in the inner loop of shell.

The third improvement is the introduction of register declarations into shellsort; these make no difference in some implementations of C, but in other implementations these declarations provide a dramatic performance improvement (40% in one case).

The fourth improvement is the use of the void type specifier to indicate explicitly that shellsort returns no value.

Notice that the two inner loops of shellsort are almost identical to the two loops of insertsort; the only change is that the variable gap has replaced the constant 1 in a few places. Despite the fact that shellsort has three nested loops instead of two, experiments show that it executes in time roughly proportional to $n^{1.25}$ instead of n^2.

8.6.5. C-Ref: Multiple Control Variables

Sometimes it is convenient to have more than one variable controlling a for loop. In this connection the comma operator is especially useful, because it can be used to group several assignment expressions into a single expression:

```
/* Returns 1 if the two string arguments
   are equal, 0  otherwise. */
int string equal(s1, s2)
    char s1[], s2[];
{
    char *p1, *p2;
    for (p1=s1, p2=s2; *p1 && *p2; p1++, p2++)
        if (*p1 != *p2) return 0;
    return *p1 == *p2;
}
```

The example function `string equal` accepts two strings and returns 1 if they are equal and 0 otherwise. The `for` statement is used to scan two pointer variables in parallel down the two strings. The expression `p1++, p2++` causes each of the two pointers to be advanced to the next character. If the strings are found to differ at some position, the `return` statement is used to terminate execution of the entire function and return 0. (This is probably a little faster than using `break` to terminate the loop and letting the following `return` redo the comparison.) If a null character is found in either string, as determined by the expression `*p1 && *p2`, then the loop is terminated normally, whereupon the second `return` statement determines whether or not both strings ended with a null character in the same place. (The function would still work correctly if the expression `*p1` were used instead of `*p1 && *p2`. It would also be a bit faster, though not as pleasantly symmetrical.)

8.7. C-Ref: Switch Statement; Case and Default Labels

The `switch` statement is a multiway branch based on the value of a control expression. In use, it is similar to the "case" statement in Pascal or Ada, but it is implemented more like the FORTRAN "computed goto" statement.

switch-statement :
  `switch` (*expression*) *statement*

case-label :
  `case` *constant-expression*

default-label :
  `default`

A `switch` statement consists of the keyword `switch`, followed by a control expression enclosed in parentheses, followed by a statement. The parentheses surrounding the expression are mandatory. The statement embedded within a `switch` statement is sometimes called the *body* of the `switch` statement. The body is usually a compound statement but need not be.

A `case` label consists of the keyword `case` followed by a constant expression. A `default` label consists of the keyword `default`. A `case` label or `default` label is said to *belong* to the innermost `switch` statement that contains it. Any statement within the body of a `switch` statement—or the body itself—may be labeled with a `case` la-

bel or a `default` label. In fact, the same statement may be labeled with several `case` labels and a `default` label.

The `case` and `default` labels that belong to a `switch` statement must satisfy the following rules:

1. All of the `case` labels (if any) must have constant expressions that—after the usual unary conversions—are of the same type as the expression in the `switch` statement.

2. No two `case` labels belonging to the same `switch` statement may have expressions that produce the same value.

3. At most one `default` label may belong to any one `switch` statement.

A `case` label or `default` label is not permitted to appear other than within the body of a `switch` statement.

The control expression of a `switch` statement is subject to the usual unary conversions, but there is some uncertainty in the types permitted for that expression. The original definition of C specified that the type had to be `int`, and this type is always permitted. Compilers that implement enumeration types will generally allow expressions of enumeration types in `switch` statements and enumeration constants in `case` labels. Type `long` may also be permitted in some implementations. However, the use of pointer or floating-point types is not permitted.

A `switch` statement is executed as follows:

1. The control expression is evaluated.

2. If the value of the expression is equal to that of the constant expression in some `case` label belonging to the `switch` statement, then program control is transferred to the point indicated by that `case` label as if by a `goto` statement; the statement labeled by that `case` label is executed next.

3. If the value of the control expression is not equal to any `case` label, but there is a `default` label that belongs to the `switch` statement, then program control is transferred to the point indicated by that `default` label; the statement labeled by the `default` label is executed next.

4. If the value of the control expression is not equal to any `case` label and there is no `default` label, no statement of the body of the `switch` statement is executed; program control is transferred to whatever follows the `switch` statement.

After control is transferred to a `case` or `default` label, execution continues through successive statements, ignoring any additional `case` or `default` labels that are encountered, until the end of the `switch` statement is reached or until control is transferred out of the `switch` statement by a `goto`, `return`, `break`, or `continue` statement.

8.7.1. C-Ref: Use of Switch Statements

The usual style in which the switch statement is used calls for the body to be a compound statement, statements within which are labeled by case and default labels. It should be noted that case and default labels do not themselves alter the flow of program control; execution proceeds unimpeded by such labels. The break statement can be used within the body of a switch statement to terminate its execution.

As an example, consider this program fragment:

```
switch (x) {
    case 1: printf("*");
    case 2: printf("**");
    case 3: printf("***");
    case 4: printf("****");
}
```

If the value of x is 2, then nine asterisks will be printed. The reason for this is that the switch statement transfers control to the case label with the expression 2. The call to printf with argument "**" is executed; next the call to printf with argument "***" is executed; and finally the call to printf with argument "****" is executed. If it is desired to terminate execution of the switch body after a single call to printf in each case, then the break statement should be used:

```
switch (x) {
    case 1: printf("*");
            break;
    case 2: printf("**");
            break;
    case 3: printf("***");
            break;
    case 4: printf("****");
            break;
}
```

While the last break statement in this example is logically unnecessary, it is a good thing to put in as a matter of style. It will help to prevent program errors in the event that a fifth case is later added to the switch statement.

While it is considered good style to use the switch statement in the manner exemplified above, the language definition itself does not require that the body be a compound statement, or that case and default labels appear only at the "top level" of the compound statement, or that case and default labels appear in any particular order or on different statements. Since a switch statement is effectively a multiway computed goto statement, the same stylistic guidelines apply as for goto statements. (See section "C-Ref: Goto Statement and Named Labels".)

Here is an example of how the best intentions can lead to chaos. The intent was to implement this simple program fragment as efficiently as possible:

```
if (prime(x)) process prime(x);
else process composite(x);
```

The function prime was assumed to return 1 if its argument is a prime number and 0 if the argument is a composite number. Program measurements indicated that most of the calls to prime were being made with small integers, so to avoid the overhead of calls to prime the code was changed to this:

```
switch(x) {
    case 2: case 3: case 5: case 7:
        process prime(x);
        break;
    case 4: case 6: case 8: case 9: case 10:
        process composite(x);
        break;
    default:
        if (prime(x)) process prime(x);
        else process composite(x);
        break;
}
```

The final step was to realize that C provided a way to compress this even further:

```
switch (x)
  default:
    if (prime(x))
      case 2: case 3: case 5: case 7:
        process prime(x);
    else
      case 4: case 6: case 8: case 9: case 10:
        process composite(x);
```

This is, frankly, the most bizarre switch statement we have ever seen that still has pretenses to being purposeful. Not only is it unstructured and difficult to understand, but good compilers can generate the same code from the well-structured switch statement above.

We strongly recommend sticking to this simple rule of style for switch statements: The body should always be a compound statement, and all labels belonging to the switch statement should appear on "top level" statements within that compound statement. Furthermore, every case (or default) label but the first should be preceded by one of two things: either a break statement that terminates the code for the previous case or a comment explicitly noting that the previous code is intended to drop in:

```
enum error type {info, warn, error, fatal} errflag;
    ...
    /* Print the appropriate prefix for issuing the */
    /* next error message, and also increment the   */
    /* appropriate counter.                         */
    switch (errflag) {
        case info:
            printf("Info");
            ++info count;
            break;
        case warn:
            printf("Warning");
            ++warn count;
            break;
        case fatal:
            disaster flag = 1;
            printf("Fatal ");
            /* Drops through. */
        case error:
            printf("Error");
            ++error count;
            break;
    }
    print error message();
```

8.8. C-Ref: Break and Continue Statements

The `break` and `continue` statements are used to alter the flow of control inside loops and—in the case of `break`—in `switch` statements. It is better to use these statements than to use the `goto` statement to accomplish the same purpose.

break-statement :
    `break ;`

continue-statement :
    `continue ;`

The `break` statement consists of just the word `break` followed by a semicolon. Execution of a `break` statement causes execution of the smallest enclosing `while`, `do`, `for`, or `switch` statement to be terminated. Program control is immediately transferred to the point just beyond the terminated statement. It is an error for a `break` statement to appear where there is no enclosing iterative or `switch` statement.

The `continue` statement consists of just the word `continue` followed by a semicolon. Execution of a `continue` statement causes execution of the body of the smallest enclosing `while`, `do`, or `for` statement to be terminated. Program control is immediately transferred to the end of the body, and the execution of the affected iterative statement continues from that point with a reevaluation of the loop test (and the

increment expression, in the case of the `for` statement). It is an error for a `continue` statement to appear where there is no enclosing iterative statement.

The `continue` statement, unlike the `break` statement, has no interaction whatever with `switch` statements. A `continue` statement may appear within a `switch` statement, but it will affect only the smallest enclosing iteration statement, not the `switch` statement.

The `break` and `continue` statements can be explained in terms of the `goto` statement. Consider the statements affected by a `break` or `continue` statement:

```
while ( expression ) statement

do statement while ( expression );

for (expression1; expression2; expression3) statement

switch ( expression ) statement
```

Imagine that all such statements were to be rewritten in this manner:

```
{ while (expression) {statement C:;} B:;}

{ do {statement C:;} while (expression); B:;}

{ for (expression1; expression2; expression3) {statement C:;} B:;}

{ switch (expression) statement B:; }
```

where in each case *B* and *C* are labels that appear nowhere else in the enclosing function. Then any occurrence of a `break` statement within the body of any of these statements is equivalent to

```
goto B;
```

and any occurrence of a `continue` statement within the body of any of these statements (except `switch`) is equivalent to

```
goto C;
```

(This assumes that the loop bodies do not contain yet another loop containing the `break` or `continue`.)

8.8.1. C-Ref: Using `break` and `continue`

The `break` statement is frequently used in two very important contexts: to terminate the processing of a particular case within a `switch` statement, and to terminate a loop prematurely. The first use is illustrated in conjunction with `switch` in section "C-Ref: Switch Statement; Case and Default Labels". The second use is illustrated by this example of filling an array with input characters:

```
/* Fill "array" with input characters, stopping
   when the array is full or when the input is
   exhausted.
 */
{
    static char array[100] = {0};
    int i, c;
    for (i = 0; i < 100; i++) {
        c = getchar();
        if (c == EOF)
            break;     /* Quit if end-of-file. */
        array[i] = c;
    }
    /* Now "i" has the actual number of
       characters read. */
}
```

Note how `break` is used to handle the abnormal case. It is generally better style to handle the normal case in the loop test itself.

Most uses of `continue` can be avoided by using a more carefully constructed `if` statement; this usually results in clearer code. Here is an example of poor use of the `continue` statement:

```
extern char command buffer[];
    ...
    for (;;) {
        /* Process all nonempty lines that do
           not start with "#". */
        gets(command buffer));
        if (!command buffer[0]) continue;
        if (command buffer[0] = '#') continue;
        process command();
    }
```

This can be rewritten to make it much more clear that the call to `process command` is conditional:

```
extern command buffer[];
    ...
    for (;;) {
        /* Process all nonempty lines that do
           not start with "#". */
        gets(command buffer));
        if (command buffer[0] &&
                    (command buffer[0] != '#'))
            process command();
    }
```

While `continue` statements are usually not as confusing as `goto` statements, a similar amount of thought should go into the decision to use one. Indiscriminate use of `continue` (or `break`, for that matter) can make programs much more difficult to understand and maintain.

Here is an example of the use of a break statement within a "do forever" loop. The idea is to find the largest element in the array a (whose length is n) as efficiently as possible. It is assumed that the array may be modified temporarily.

```
{
    int temp = a[0];
    register int smallest = a[0];
    register int *ptr = &a[n];
    for (;;) {
        while (*--ptr > smallest);
        if (ptr == &a[0]) break;
        a[0] = smallest = *ptr;
    }
    a[0] = temp;
}
```

The point is that most of the work is done by a very tight while loop. The while loop scans the pointer ptr backwards through the array, skipping elements that are larger than the smallest one found so far. (If the elements are in a random order, then once a reasonably small element has been found, most elements will be larger than that and so will be skipped.) The while loop cannot fall off the front of the array because the smallest element so far is also stored in the first array element. When the while loop is done, if the scan has reached the front of the array, then the break statement terminates the outer loop. Otherwise smallest and a[0] are updated and the while loop is entered again. At the end of the computation, element a[0] is restored to its original value.

Compare the code above with a simpler, more obvious approach:

```
{
    register int smallest = a[0];
    register int j;
    for (j = 1; j < n; ++j)
        if (a[j] < smallest)
            smallest = a[j];
}
```

This version is certainly easier to understand. However, on every iteration of the loop an explicit check (j < n) must be made for falling off the end of the array, as opposed to the implicit check made by the more clever code. Under certain circumstances where efficiency is paramount, the more complicated code may be justified; otherwise, the simpler, clearer loop should be used.

8.9. C-Ref: Return Statement

A return statement is used to terminate the current function, perhaps returning a value.

return-statement :
    return *expression*$_{opt}$;

A `return` statement consists of the keyword `return`, optionally followed by an expression, followed by a semicolon. Execution of a `return` statement causes execution of the current function to be terminated; program control is transferred to the caller of the function at the point immediately following the call.

If program control should "drop off the end" of a function, then the effect is as if a `return` statement with no expression were executed.

If no expression appears in the `return` statement, then no value is returned from the function; if the function was called from a context requiring a value, then the value returned is undefined. If an expression appears in the `return` statement, then it is converted, if necessary, as if by simple assignment, to the type of the return value of the function in which the statement appears.

The rules governing the agreement of the actual value returned with the declared return value in the function definition are discussed in section "C-Ref: Agreement of Actual and Declared Return Type".

8.10. C-Ref: Goto Statement and Named Labels

A `goto` statement may be used to transfer control from any statement in a function to any other statement.

goto-statement :
    `goto` *identifier* `;`

named-label :
    identifier

A `goto` statement consists of the keyword `goto`, followed by an identifier, followed by a semicolon. The identifier must be the same as a named label on some statement within the current function. Execution of the `goto` statement causes an immediate transfer of program control to the point in the function indicated by the label; the statement labeled by the indicated name is executed next.

8.10.1. C-Ref: Using the `goto` statement

C permits a `goto` statement to transfer control to any other statement within a function, but certain kinds of branching can result in confusing programs, and the branching may hinder compiler optimizations.

The following rules should result in a more clear use of the `goto`:

1. Do not branch into the "then" or "else" arm of an if or if-else statement from outside the if or if-else statement.

2. Do not branch from the "then" arm to the "else" arm or back.

3. Do not branch into the body of a `switch` or iteration statement from outside the statement.

4. Do not branch into a compound statement from outside the statement.

Such branches should be avoided not only when using the `goto` statement, but also when placing `case` and `default` labels in a `switch` statement (which, in effect, executes a `goto` statement to get to the appropriate `case` label). Branching into the middle of a compound statement from outside it can be especially confusing, because such a branch bypasses the initialization of any variables declared at the top of the compound statement.

It is good programming style to use the `break`, `continue`, and `return` statements in preference to `goto` whenever possible, and better still to avoid them all by appropriate use of conditional and iteration statements.

Finally, the programmer wanting to produce a C program that executes as rapidly as possible should remember that the presence of *any* label—whether explicit, named labels or implicit labels required by `break` and `continue`—may inhibit compiler optimizations and therefore may slow down the C program.

8.11. C-Ref: Null Statement

The null statement consists of just a semicolon:

null-statement :
```
;
```

It is useful primarily in two situations. First, a null body is often desired for an iterative statement (`while`, `do`, or `for`), as in

```
char *p;
...
while ( *p++ );            /* find the end of the string */
```

The second case is where a label is desired just before the right brace that terminates a compound statement. (A label cannot simply precede the right brace, but must always be attached to a statement.) For example:

```
if (e) {
   ...
   goto L;  /* terminate this arm of the 'if' */
   ...
L:;}
else ...
```

9. C-Ref: Functions

This chapter discusses the definition of functions in C and the rules for the agreement of parameters and return values. Function types and declarations are discussed in section "C-Ref: Function Types". This is an area of the language that has been significantly extended in Draft Proposed ANSI C.

9.1. C-Ref: Function Definitions

A function definition introduces a new function and provides the following information:

1. the type of the value returned by the function, if any

2. the type and number of the formal parameters

3. the visibility of the function outside the file in which it is defined

4. the code that is to be executed when the function is called

Do not confuse function *definitions* with function *declarations*. A function declaration provides access to a function that is defined elsewhere.

The syntax for a function definition is

function-definition :
  declaration-specifiers$_{opt}$ *declarator* *function-body*

function-body :
  parameter-declaration-list$_{opt}$ *compound-statement*

parameter-declaration-list :
  declaration-list

declaration-list :
  declaration
  declaration-list *declaration*

Note that both the storage class specifier and the type specifier may be omitted from the function definition without ambiguity.

The only storage class specifiers that may appear in a function definition are `extern` and `static`. `extern` signifies that the function can be referenced from other files; that is, the function name is exported to the linker. The specifier `static` signifies that the function cannot be referenced from other files; that is, the name is not exported to the linker. If no storage class appears in a function definition, `extern` is assumed.

The storage class does not affect the visibility of the function within the file containing the definition. The function is always visible from the definition point to the end of the file. In particular, it is visible within the body of the function itself. (C allows any function to call itself recursively.)

9.2. C-Ref: Types of Functions

In a function definition, as in a declaration, the type specifier and the declarator together determine the "type" of a function. We will call them the *function specifier*. If no type specifier is present, `int` is assumed.

In a function definition, the declarator and type specifier must together specify a type for the enclosed identifier of "function returning *T*," where *T* is any type (including `void`) except "array of ..." or "function returning ...." In other words, functions may not return arrays or other functions. (However, they may return pointers to arrays or functions.) For example, the following syntactically legal function definition is nonsensical because the type of `f` is "pointer to function returning `int`":

```
int (*f)()
{
    ...
}
```

However, the following definition is legal, because the type of `g` is "function returning *T*" (where *T* is "pointer to array of `int`").

```
int (*g())[]
{
    ...
}
```

Another way of stating the restriction is that the definition must contain a function declarator, "*d*`(...)`," where *d* is the identifier that is the name of the function. If the function has parameters, they must be listed in the function declarator.

Consider the following examples of function specifiers:

`void f()`	`f` is a function with no parameters returning no result.
`int g(x, y)`	`g` is a function taking two parameters named `x` and `y` and returning a value of type `int`.
`int (*h(z))[]`	`h` is a function taking one parameter named `z` and returning a pointer to an array of integers.
`int (*(*d(w))[])()`	`d` is a function taking one parameter named `w` and returning a pointer to an array of pointers to functions returning integers. (Note that only the parameters of `d` are specified, not those of other functions mentioned in the declarator.)

Draft Proposed ANSI C introduces the concept of "function prototypes," which carry more information than function types.

9.3. C-Ref: Formal Parameter Declarations

In function definitions, formal parameters are declared in two parts. As we have just seen, the names of the parameters are listed in the function declarator. In order to supply types for the parameters, the programmer declares each of the parameters (in any order) in the parameter declaration section. For example, to define a function that has three parameters—an integer, a double-precision floating-point number, and a pointer to an integer—the programmer can write:

```
void f(x, y, z)
  int x, *z;
  double y;
{
    ...
}
```

The parameter declaration section may contain declarations of the parameters, and perhaps declarations of types used in the parameter declarations.

The only storage class specifier that may be present in a parameter declaration is `register`, which is a hint to the compiler that the parameter will be used heavily and might better be stored in a register after the function has begun executing. The normal restrictions as to what types of parameters may be marked `register` apply (see section "C-Ref: Storage Class Specifiers".

A parameter may be declared to be of any type except `void` or "function returning ...." Parameters of type "array of *T*" and (sometimes) "function returning *T*" may be declared, but these types are adjusted to be "pointer to *T*" and "pointer to function returning *T*," respectively. The mechanism is discussed in more detail in section "C-Ref: Adjustments to Parameter Types".

It is permissible to include structure, union, or enumeration type definitions in the parameter declaration section, and to include `typedef` definitions. The scope of these definitions extends to the end of the function body. However, the usefulness of these definitions is marginal, and probably bad programming style. To illustrate this, consider the following function definition:

```
int process record(r);
  struct { int a; int b; } *r;
{
    ...
}
```

The inclusion of the `struct` definition as a side effect of the declaration for `r` is permissible but confusing, because no actual parameter could be declared to have that type. (The scope of the structure definition does not extend outside the function.) When we see in another file the code

```
extern struct { int first;
                int second; } *two integers;
process record(two integers);
```

we can guess that the programmer is depending on:

1. the compiler's not checking that the types of the formal and actual parameters match (they don't match)

2. the programmer's being consistent about defining the same structured type in different places so that the the actual parameter's structure matches the formal parameter's structure (this consistency is not guaranteed because the compiler does not necessarily perform this type check)

Programmers who do things like this are living dangerously and invite ridicule from people who have to decipher their programs.

9.4. C-Ref: Adjustments to Parameter Types

C specifies that certain adjustments in the types of function arguments be made to simplify and regularize function arguments. The adjustments are made in two places: on the actual argument types at the point of the function call and on the formal argument types in function definitions.

The adjustments to the actual arguments are listed in section "C-Ref: The Function Argument Conversions". Corresponding to these adjustments, adjustments are made to the types of a function's formal parameters as they appear in the parameter declaration section. In particular, if a formal parameter is declared to be of type `char`, `short`, or `float`, the compiler will expect an actual argument of type `int`, `int`, or `double` (respectively) to be passed to the function. For this reason, formal parameters declared to be of type `char`, `short`, or `float` are implicitly *promoted* to be of type `int`, `int`, and `double`, respectively. This permits many program libraries to be smaller than they would have to be if, for instance, multiple definitions of "square root" had to be provided for each of the argument types `short`, `int`, `float`, `double`, etc.

However, the compiler will ensure that the values of the parameters are appropriate to the declared type. That is, the function

```
void f(c)
  char c;
{
    int i;
    i = c;
    ...
}
```

is implemented as if it were written

```
void f(c)
    int c;
  {
    int i;
    i = (int) (char) c;
    ...
  }
```

(Not all compilers actually implement such explicit narrowing operations on parameters, just as some deficient compilers fail to implement narrowing casts in all cases. For maximum portability, programs should not depend critically on the truncation effects of such narrowing.)

A formal parameter declared to be of type "array of *T*" is treated as if it were declared to be of type "pointer to *T*." Because of the equivalence of pointers and arrays, this change is invisible to the programmer. For example, in the function

```
int sumarray(a, n)
  int a[], n;
{
    int sum=0, i;
    for (i = 0; i < n; i++)
        sum = sum + a[i];
    return sum;
}
```

the parameters `a` and `n` could have been declared as

```
int *a, n;
```

with no other change to the program. Although array names are not usually lvalues, a formal parameter declared to be an array is treated as an lvalue by many compilers.

Formal parameters of type "function returning ..." are not permitted by the language. However, some compilers accept such parameters and implicitly convert them to type "pointer to function returning ..." (sometimes also issuing a warning message). These compilers will also automatically dereference such a parameter when used in a function call. In fact, they do this automatic dereferencing on any expression of type "pointer to function returning ...." For example:

```
extern (*h)();

void f(g)       /* Not a legal C program! */
  void g();
{
    g();        /* This works in some compilers. */
    (*g)();     /* So does this. */
    h();        /* So does this! */
    (*h)();     /* ...and, of course, this. */
}
```

We recommend adhering to the language specification and always declaring parameters to be pointers to functions. Draft Proposed ANSI C does provide automatic dereferencing of "pointer to function returning ...."

9.5. C-Ref: Parameter-Passing Conventions

C provides only call-by-value parameter passing. This means that the values of the actual parameters are conceptually copied into a storage area local to the called function. It is possible to use a formal parameter name as the left side of an assignment, for instance, but in that case only the local copy of the parameter is altered.

If the programmer wants the called function to alter its actual parameters, the addresses of the parameters must be passed explicitly. For example, function `swap` below will not work correctly, because `x` and `y` are passed by value.

```
void swap(x, y)
/* swap: exchange the values of x and y */
/* Incorrect version! */
  int x, y;
{
    int temp;
    temp = x; x = y; y = temp;
}
...
    swap(a, b);  /* Fails to swap a and b. */
```

A correct implementation of the function requires that addresses of the arguments be passed:

```
void swap(x, y)
/* swap - exchange the values of *x and *y */
/* correct version */
  int *x, *y;
{
    int temp;
    temp = *x; *x = *y; *y = temp;
}
...
    swap(&a, &b);  /* Swaps contents of a and b. */
```

The local storage area for parameters is usually implemented on a pushdown stack. However, the order of pushing parameters on the stack is not specified by the language, nor does the language prevent the compiler from passing parameters in registers. It is legal to apply the address operator `&` to a formal parameter name (unless it was declared with storage class `register`), thereby implying that the parameter in question would have to be in addressable storage when the address was taken. (Note that the address of a formal parameter is the address of the copy of the actual parameter, not the address of the actual parameter itself.)

When writing functions that take a variable number of arguments, programmers should use the `varargs` facility in the standard library for maximum portability.

9.6. C-Ref: Agreement of Formal and Actual Parameters

Most modern programming languages such as Pascal and Ada check the agreement of formal and actual parameters to functions; that is, both the number of arguments and the types of the individual arguments must agree. As in FORTRAN, this checking is not performed in C:

1. The syntax of declarations does not provide for a declaration of argument types to functions, and therefore no checking is possible when a function is supplied in another source file.

2. The lack of checking gives programmers some freedom in violating conventions on certain rare occasions, especially in implementing functions that take a variable number of arguments.

For example, in the function `hypotenuse` below, the call on `sqrt` does not generate a warning message, even though the actual parameter is of type `long` whereas the formal parameter is declared to have type `double`. The function will simply return a (probably) incorrect value.

```
double sqrt( x )
  double x;
{
    ...
}

long hypotenuse(x,y)
  long x,y;
{
    return (sqrt(x*x + y*y));
}
```

There is no portable way in C to write a function that accepts a variable number of arguments. Such functions can be written in C—`fprintf` and its variants are examples—but they are not portable. They depend on very specific knowledge of how parameters are passed on the stack, and they still need some way to determine the type and number of arguments. (For example, `fprintf` depends on the format string to indicate the number and types of the arguments.)

Draft Proposed ANSI C corrects many of the deficiencies just noted for functions. In particular, it introduces a mechanism—the function prototype—that permits the declaration of functions with their parameter types, and also formalizes the idea of a function taking a variable number of arguments.

9.7. C-Ref: Function Return Types

A function may be defined to return a value of any type except "array of *T*" or "function returning *T*." These two cases must be handled by returning pointers to the array or function. The actual value, if any, returned by the function is specified by an expression in the `return` statement that causes the function to terminate. If control "falls out the bottom" of a function, it is as if

```
return;
```

had been executed.

The value returned by a function is not an lvalue (the return is "by value"), and therefore a function call cannot appear on the left side of an assignment operator. The language does not specify how the return value is to be transmitted to the calling program.

9.8. C-Ref: Agreement of Actual and Declared Return Type

A return statement with no expression,

```
return;
```

is always permitted, regardless of whether the function has a `void` or nonvoid return type. This rule is to provide backwards compatibility with compilers that do not implement `void`. When a function has a nonvoid return type, and a `return` statement with no arguments is executed, the value actually returned is unpredictable and it is therefore unwise to invoke the function in a context that requires a value. We recommend that this form of `return` be used *only* when the function is declared to have return type `void`.

If a function has a declared return type of `void`, it is an error to supply an expression in any `return` statement in the function. Although supplying a void return value, as in

```
void f()
{
    extern void g();
    ...
    return g();
}
```

would seem to be no more than confusing, many compilers will treat this as an error. It is also an error to call the function in a context that requires a value.

If the function has a declared return type *T* that is not `void`, then the type of any expression appearing in a `return` statement must be convertible to type *T* by assignment, and that conversion in fact happens on return. For instance, in a function with declared return type `int`, the statement

```
return 23.1;
```

is equivalent to

```
return (int) 23.1;
```

which is the same as

```
return 23;
```

With older compilers that do not implement `void`, it is the custom to omit the type specifier on those functions that return no useful value:

```
main()
{
    ...
}
```

9.9. C-Ref: Main Programs

By convention all C programs must define a single external function named `main`. That function will become the entry point of the program, that is, the first function executed when the program is started. Information about parameters usually supplied to `main` is given in the library chapters.

10. C-Ref: Program Structure

In this chapter we will attempt to pull together a number of aspects of software engineering in C. We will do so by developing a complete implementation of a last-in first-out queue, or stack.

10.1. C-Ref: Modularization

We prefer to modularize programs by data types. That is, we think of a program as consisting of a number of *modules*, each of which implements a new, abstract data type by providing objects of the type and operations on the objects. These modules are sometimes called *type managers*, to emphasize that they have control over the internal representation of the types and the implementation of the operations on the types.

A stack can be viewed as such an abstract data type. An object of "stack type" can be imagined (under one implementation) as an array of values and a pointer into that array to mark the "top" of the stack. Operations on the stack include "create a new stack," "push a value onto the stack," "pop a value off the stack," and so forth.

When designing a new abstract data type, the programmer must answer many questions:

1. What functionality is required? What operations will be needed?

2. How often will the operations be invoked? What other performance criteria exist?

3. What implementations might be appropriate? Will the data structures have to be dynamically allocated, or will local or static allocation suffice? Is there an existing module that can be modified to meet the specifications of the new module?

4. How will the data type be used? What information must be exported to users?

5. How can the type be implemented securely and robustly? That is, how can users be prevented from corrupting the internal data structures of the type, and how can the type manager detect improper use of its operations?

6. What functional changes might be required in the future? Who will maintain the module? How does this affect the implementation?

7. What documentation will be needed?

This list could be extended further to include provisions for version control, inter-

nal development reviews, and so forth. However, this should be sufficient to indicate that a well-crafted module can involve much more than a few lines of code.

10.2. C-Ref: Designing the Stack Module

The first thing to do is to sketch out the functional properties of a stack. First, the operations:

- Allocate a new stack.
- Deallocate an old stack.
- Push a value onto the stack.
- Pop a value off the stack.

Experience with stacks has told us that two more operations are often useful:

- Return the top value from the stack without removing it.
- Find out how big the stack currently is and how far it can grow.

Given the operations, we must also ask about the values of the new type:

- What type of elements is the stack to hold?
- How big should the stack be?

Finally, we must worry about handling errors, such as overflow or underflow.

After talking to the potential users of our stacks, we decide the following:

- The stacks will hold values of type `int` (although we suspect that the users will want a different element type later on).
- The stacks will be large and therefore should be dynamically allocated.
- Pushing and popping elements should be fast operations, and users can live with a fixed maximum size for each stack, although different stacks may have different maximums. Therefore, we decide that we can use an array implementation of stacks.
- The stacks will be used in a large product that is likely to have bugs while being completed. Therefore we will include some extra consistency checking that can be removed (for better performance) before the product is shipped.

10.3. C-Ref: Data Structures

We'll begin by making some standard declarations and defining the data structures.

```
#include <stdio.h>
extern char *malloc();

typedef int stack element type;

typedef struct stack struct {
          stack element type *base of stack;
          stack element type *end of stack;
          stack element type *next free element;
        } *stack type;
```

Stacks will be represented by type `stack type`. No user of stacks needs to know how it is implemented, but in fact `stack type` is a pointer to a structure containing three pointer components: `base of stack`, a pointer to the base of an array of elements; `end of stack`, a pointer to the first element beyond the end of the array; and `next free element`, a pointer to the array element that will receive the next pushed value (that is, the first array element following the top element of the stack). Note that we have also introduced the type `stack element type`; by changing the definition of this type we can have stacks that hold other kinds of elements.

The type `stack type` is the "stack header"; the actual data will be held in an array that will also be dynamically allocated.

10.4. C-Ref: Robustness

When the stack module is being developed, or when it is being used in a program that potentially has bugs, we'd like to perform some additional consistency checking on the stack data structure. Our strategy will be to use the preprocessor macro `stack debugging` to control whether special consistency checks are compiled into the stack module. In particular, when `stack debugging` is 1, a special function, `stack check`, is defined. At run time, `stack check` examines the contents of the stack data structure and verifies that the data structure is consistent. When `stack debugging` is 0, `stack check` is defined as a macro with no body, effectively eliminating the checks from the code. The definitions for the debugging information are shown in table "C-Ref: Stack Example: Conditionally Compiled Debugging Code".

Of course, we could have used other strategies for providing the consistency checks. A reasonable alternative to the above scheme would be to provide a run-time test to see if the data structures are to be checked. That is, we would always define the function `stack check`, but would replace

```
#define stack debugging 1
```

with

```
int stack debugging = 1;
```

and then replace all the calls on `stack check` with

```
if (stack debugging) stack check(stack);
```

This scheme has the advantage of not requiring recompilation to turn checks on and off. Its disadvantage is that it involves the test of the variable `stack debugging` on every operation. (This is probably not a significant overhead compared with the overhead of a function call.)

10.4.1. C-Ref: Stack Example: Conditionally Compiled Debugging Code

```
#define stack debugging 1

#if stack debugging

/* interactive debugger (not included here) */
extern void debugger();
```

```
static void stack check(stack)
  stack type stack;
/*
    Check the internal consistency of the 'stack' data
    structure. If inconsistent, print a message and
    invoke a debugger. If consistent, just return.
*/
{
    if (stack == NULL) {
        printf("?Stack is NULL\n");
        debugger();
        return;
    }
    if (stack->base of stack == NULL) {
        printf("?Stack array is NULL\n");
        debugger();
        return;
    }
    if ( (stack->next free element <
                        stack->base of stack) ||
         (stack->next free element >
                        stack->end of stack)
       ) {
        printf("?Stack pointers are invalid.\n");
        debugger();
        return;
    }
    /* Stack is OK */
    return;
}
#else
/* If not debugging, then calls to stack check
   will be quietly eliminated by the preprocessor.
 */
#define stack check(stack)
#endif
```

10.5. C-Ref: Allocating and Deallocating Stacks

We now consider the creation and deletion of new stack objects. The creation code is shown in table "C-Ref: Stack Example: Allocation of Stacks". Note how we check each call to `malloc` to be sure the requested storage was allocated.

10.5.1. C-Ref: Stack Example: Allocation of Stacks

```
stack type stack alloc(size)
  unsigned int size;
/* Create a new stack object with a maximum of "size"
   elements.  Return NULL if insufficient storage is
   available, or if "size" is not greater than zero.
*/
{
    stack type stack;
    unsigned header size =
                    sizeof(struct stack struct);
    unsigned array size =
                    size * sizeof(stack element type);

    if (size <= 0) return NULL;
    stack = (stack type) malloc(header size);
    if (stack == NULL) return NULL;
    stack->base of stack =
        (stack element type *) malloc(array size);
    if (stack->base of stack == NULL) {
        /* Can't get the array, so free the header. */
        free(stack);
        return NULL;
    }
    stack->end of stack = stack->base of stack + size;
    stack->next free element = stack->base of stack;
    return stack;
}
```

The deallocation code (table "C-Ref: Stack Example: Deallocation of Stacks") is very simple, but it is an opportunity to make the type manager a bit more robust. It is always possible for the caller of `stack free` to accidentally use the (deallocated) stack in a subsequent call on the type manager. So that this error may be caught quickly, `stack free` zeros all the internal pointers before freeing the storage. If `stack debugging` is 1, this helps to ensure that `stack check` will fail if given a pointer to the old stack. However, even if `stack debugging` is 0, the null pointers should cause the program to halt more quickly than it would if the pointers were just left dangling. Furthermore, the overhead of zeroing the pointers is small compared with the expected overhead of the storage allocator, so we don't worry about the extra code.

10.5.2. C-Ref: Stack Example: Deallocation of Stacks

```
void stack free(stack)
  stack type stack;
/* Deallocate the given stack and return its
   storage to the heap.
 */
{
    stack check(stack);
    /* Free the data array first. */
    free((char *) stack->base of stack);
    /* Clear the pointers so that "stack check"
       is more likely to fail on a freed stack. */
    stack->base of stack     = NULL;
    stack->next free element = NULL;
    stack->end of stack      = NULL;
    /* Free the header. */
    free((char *) stack);
    return;
}
```

10.6. C-Ref: Operations on Stacks

The operations on stacks are pretty simple (see tables "C-Ref: Stack Example: Push and Pop Operations"and "C-Ref: Stack Example: Peek Operation". We have decided to handle overflow and underflow errors by having the type manager set an error flag through a pointer provided by the caller. An alternative would be for the type manager to export a variable that was used as a status indicator after each operation, or use the standard variable `errno` that is used by the standard C library routines. We think our scheme results in more readable programs, even though it involves a bit of overhead on calls. Notice also that we have used a type `boolean` for the error flags to make clear our intended use:

```
typedef int boolean;
#define TRUE   1
#define FALSE  0
```

10.6.1. C-Ref: Stack Example: Push and Pop Operations

```
void stack push(stack, data, overflow ptr)
  stack type stack;
  stack element type data;
  boolean *overflow ptr;
/*
    Push "data" onto the stack.  If the stack is full,
    set "*overflow ptr" to TRUE and don't do the push.
    Otherwise, set "*overflow ptr" to FALSE.
*/
{
    stack check(stack);
    if (stack->next free element
            >= stack->end of stack) {
        *overflow ptr = TRUE;
    }
    else {
        *overflow ptr = FALSE;
        *(stack->next free element++) = data;
    }
}
/* Dummy value of type "stack element type". */
static stack element type stack element novalue;
stack element type stack pop(stack, underflow ptr)
  stack type stack;
  boolean *underflow ptr;
/*
  If "stack" is empty, set "*underflow ptr" to TRUE and
  return.  Otherwise, set "*underflow ptr" to FALSE and
  remove and return the top stack element.
*/
{
    stack check(stack);
    if (stack->next free element
            <= stack->base of stack) {
        *underflow ptr = TRUE;
        return stack element novalue;
    }
    else {
        *underflow ptr = FALSE;
        return *(--stack->next free element);
    }
}
```

10.6.2. C-Ref: Stack Example: Peek Operation

```
stack element type stack peek(stack, underflow ptr)
    stack type stack;
    boolean *underflow ptr;
/*
   If "stack" is empty, set "*underflow ptr" to TRUE
   and return. Otherwise set "*underflow ptr" to FALSE,
   and return the top stack element.  (Do not remove
   it from the stack.)
*/
{
    stack check(stack);
    if (stack->next free element
            <= stack->base of stack) {
        *underflow ptr = TRUE;
        return stack element novalue;
    }
    else {
        *underflow ptr = FALSE;
        return *(stack->next free element - 1);
    }
}
```

Notice in `stack pop` and `stack peek` the use of the static variable `stack element novalue` as a return value. We really don't want to return anything, because the stack has underflowed. However, we think just writing

```
return;
```

would be confusing, since the return value doesn't match the declared return type of the function. Writing

```
return 0;
```

would be better but depends on the fact that `stack element type` has a 0 value. By defining a variable containing "no return value," we maintain generality with any type `stack element type`.

Finally, table "C-Ref: Stack Example: Determining Stack Sizes" shows a routine that returns the current and maximum sizes of a stack, and some boolean predicates to test whether the stack is empty or full. These predicates are macros for efficiency.

10.6.3. C-Ref: Stack Example: Determining Stack Sizes

```
void stack sizes(stack,
                 current size ptr,
                 allocated size ptr)
  stack type stack;
  unsigned int *current size ptr, *allocated size ptr;
/*
    Set "*current size ptr" to the number of elements
    on the stack, and set "*allocated size ptr" to the
    number of elements the stack can hold.
*/
{
    stack check(stack);
    *current size ptr =
       stack->next free element - stack->base of stack;
    *allocated size ptr =
       stack->end of stack - stack->base of stack;
    return;
}

#define stack isempty(stack) \
          ((stack)->next free element == \
           (stack)->base of stack))

#define stack isfull(stack) \
          ((stack)->next free element == \
           (stack)->end of stack))
```

10.7. C-Ref: Packaging the Module

Now that the data structures and algorithms for the stack module are finished, we can give some thought to the best way to export the module's facilities to the user. We will have to provide declarations of the functions and macros implementing the operations, and we'll have to provide the type `stack type`.

The custom in C is to collect these definitions in a *header file* that can be imported (with `#include`) by users of the module. The header file is also a good place to put some short documentation. Tables "C-Ref: Stack Example: Header File (Part 1, Types)" and "C-Ref: Stack Example: Header File (Part 2, Operations)" show the header file for the stack module, which we have named `stack.h` following normal conventions.

10.7.1. C-Ref: Stack Example: Header File (Part 1, Types)

```
/* stack.h     Definitions for stack-of-integers.
   Typical use:
        #include <stack.h>
        stack type      stack;
        stack element type data;
        unsigned int    current size, maximum size;
        boolean                 overflow, underflow;
        stack = stack alloc(100);
        if (stack==NULL) ...;
        stack push(stack, data, &overflow);
        if (overflow) ...;

        data = stack pop(stack, &underflow);
        if (underflow) ...;
        stack sizes(stack,&current size,&maximum size);
        stack free(stack);
*/
/* Short external names for operations. */
#define stack alloc     stkall
#define         stack free      stkfre
#define stack push      stkpsh
#define stack pop       stkpop
#define stack peek      stkpek
#define stack sizes     stksiz
/* Type of elements in stack. */
typedef  int  stack element type;
/* Boolean error flags. */
typedef  int  boolean;
/* The stack type itself. */
typedef struct stack struct {
          stack element type *base of stack;
          stack element type *end of stack;
          stack element type *next free element;
        } *stack type;
```

10.7.2. C-Ref: Stack Example: Header File (Part 2, Operations)

```
extern stack type stack alloc();
/* Create a new stack for up to "size" elements.
   extern stack type stack alloc(size);
      stack size type size;  */
```

```
extern void stack free();
/* Deallocate a stack.
   extern void stack free(stack);
      stack type stack;  */

extern void stack push();
/* Push "data" onto "stack"; set "*overflow ptr"
   to TRUE if full and otherwise to FALSE.
   extern void stack push(stack, data, overflow ptr)
     stack element type data;
     stack type stack; boolean *overflow ptr; */

extern stack element type stack pop();
/* Pop top element from "stack"; set "*underflow ptr"
   to TRUE if empty and otherwise to FALSE.
   stack element type stack pop(stack, underflow ptr)
      stack type stack;  boolean *underflow ptr;  */

extern stack element type stack peek();
/* Return, but don't pop, top element from "stack";
   set "*underflow ptr" as in stack pop().
   stack element type stack peek(stack, underflow ptr)
      stack type stack; boolean *underflow ptr; */

extern void stack sizes();
/* Return current and maximum sizes of stack.
   void stack sizes(stack, current size ptr,
                           allocated size ptr)
      stack type stack;
      unsigned int *current size ptr,
             *allocated size ptr;  */

/* Predicate: is stack empty? */
#define stack isempty(stack) \
  ((stack)->next free element==(stack)->base of stack))

/* Predicate: is stack full? */
#define stack isfull(stack) \
   ((stack)->next free element==(stack)->end of stack))
```

A couple of points should be mentioned here. First, because of the restrictions on external names, we have defined macros that convert our names (and the names that should be used by clients) to shorter names less likely to conflict in the linker. (We have also been careful with the internal names, such as `stack push`; they all begin with the prefix `stack` to minimize conflicts.) Second, although our principal exported type is `stack type`, we must also export `stack element type` and `boolean`, since we use those types in the definition of `stack type`. Third, although

we consider the implementation of `stack type` to be "private" to our type manager, there is no way to actually hide the implementation from the user of the module. A caller could alter the components of the structure in arbitrary ways, thus corrupting the data. Finally, it is very helpful to include as much documentation in the header file as possible.

All the other code goes into the stack implementation module, `stack.c`, which should begin with the line

```
#include <stack.h>
```

since it will need the same type and macro definitions. To avoid duplication, the declarations for `stack element type`, `stack type`, `boolean`, `stack isempty`, and `stack isfull` may be removed from `stack.c`, since they are now supplied in `stack.h`.

11. C-Ref: Draft Proposed ANSI C

In 1982 the American National Standards Institute (ANSI) formed a technical subcommittee on C language standardization, X3J11, to propose a standard for the C language, its run-time libraries, and its compilation and execution environments. In 1986 the subcommittee released a Proposed American National Standard for Information Systems—Programming Language C, referred to as "Draft Proposed ANSI C" in this book.

In large part, Draft Proposed ANSI C codifies existing practice by C programmers and makes an attempt not to invalidate the large body of C programs. In a few areas the language is extended to overcome specific shortcomings, but in large part the "spirit of C" is preserved. An attempt is made to allow the programmer to write portable programs, but the programmer is not obligated to do so. In many places, the description of C is made more precise.

This chapter summarizes Draft Proposed ANSI C by specifying how it differs from the C language presented in this book. The sections are organized in the same order as the preceding chapters.

11.1. C-Ref: ANSI C Lexical Elements

Specific changes to the lexical structure of C include the introduction of trigraphs so that C programs may be written in a subset of ASCII; the introduction of new reserved words `volatile`, `const`, and `signed`; and extensions to the syntax for literals. Also, the compound assignment operators are now treated as single tokens.

11.1.1. C-Ref: ANSI C Character Sets

A set of trigraphs is included so that C programs may be written using only the ISO 646-1083 Invariant Code Set, a subset of the seven-bit ASCII code set. The trigraphs, introduced by two consecutive question mark characters, are listed below:

Trigraph	is equivalent to	Trigraph	is equivalent to
`??(`	`[`	`??)`	`]`
`??<`	`{`	`??>`	`}`
`??/`	`\`	`??!`	`|`
`??'`	`^`	`??-`	`~`
`??=`	`#`		

The translation of trigraphs in the source program occurs before lexical analysis (tokenization) and even before the recognition of character escapes introduced with a backslash, \. Only these exact nine trigraphs are recognized; all other character sequences (including relatives such as `??&`) should be left untranslated. A new character escape, `\?`, is available to prevent the interpretation of trigraph-like character sequences. For example:

String constant	Trigraph Form
`"What??!"`	`"What?\?!"`
`"The backslash is \\"`	`"The backslash is ??/??/"`

A source program line ending in a backslash (\) is understood to be continued on the next line. This "splicing" conceptually occurs before the lexical analysis of the C program; hence even tokens may be split across lines. Formerly line continuation was permitted in preprocessor command lines and in string constants, although some implementations allowed it elsewhere. In Draft Proposed ANSI C it is a general mechanism, but it is probably useful only in preprocessor lines since string constants may now be concatenated.

11.1.2. C-Ref: ANSI C Identifiers

The Draft Proposed ANSI C standard imposes no limitations on the length of identifiers, and requires that implementations support identifiers with at least 31 significant characters. Implementations must distinguish alphabetic case in identifiers.

Implementations may reduce the significance of external names (down to the first six characters and one alphabetical case) in order to reflect limitations in existing system software.

11.1.3. C-Ref: ANSI C Reserved Words

The `enum` and `void` reserved words, already common in C implementations, are included in Draft Proposed ANSI C. However, `entry`, `fortran`, and `asm` are not included as reserved words.

New type specifiers `const`, `volatile`, and `signed` are added to the reserved word list.

11.1.4. C-Ref: ANSI C Integer Constants

The syntax of constants is extended so that the type of the constants can be better controlled. In addition to the existing `L` suffix to denote type `long`, the suffix `U` is now allowed to denote a constant of unsigned type. The two suffixes may be used together in either order. The new syntax is:

type-marker : (see "C-Ref: Integer Constants")

  long-marker unsigned-marker$_{opt}$
  unsigned-marker long-marker$_{opt}$

long-marker : one of

  `l L`

unsigned-marker : one of

  `u U`

This permits the following new examples of integer constants:

```
100u   34LU   32767ul
```

The type of an integer constant is given by the following rules:

1. An integer constant written with both the *long-marker* and the *unsigned-marker* has type `unsigned long int`.

2. An integer constant written with just the *long-marker* has type `long int` if that type can represent the constant; otherwise it has type `unsigned long int`.

3. An integer constant written with just the *unsigned-marker* has type `unsigned int` if that type can represent the constant; otherwise it has type `unsigned long int`.

4. An octal or hexadecimal integer constant has the first type in the following list that can represent its value: `int`, `unsigned int`, `long int`, `unsigned long int`.

5. A decimal integer constant has the first type in the following list that can represent its value: `int`, `long int`, `unsigned long int`.

The Draft Proposed ANSI C rules differ subtly from those found in most current implementations: under the new rules constants without suffixes may have an unsigned type and therefore force expressions to use unsigned arithmetic. Formerly, the type was always `int` or `long int`. If type `long int` has a 32-bit, two's complement representation, the following program will determine the rules in effect:

```
#define K 0xFFFFFFFF    /* -1 in 32-bit, 2's compl. */
int main()
{
  if (0<K) printf("K is unsigned (ANSI C)\n");
  else printf("K is signed (traditional C)\n");
  return 0;
}
```

11.1.5. C-Ref: ANSI C Floating Point Constants

To accommodate the new type `long double` and the new legitimacy of type `float`, floating-point constants can now have suffixes also: `F` to denote type `float` and `L` to denote type `long double`. (The specifier `long float` has been eliminated.)

floating-constant : (see "C-Ref: Floating-point Constants")
    digit-sequence exponent float-marker$_{opt}$
    dotted-digits exponent$_{opt}$ *float-marker*$_{opt}$

float-marker : one of

```
f F l L
```

The suffix `f` or `F` makes the constant of type `float`; the suffix `L` or `l` makes the constant of type `long double`.

11.1.6. C-Ref: ANSI C String Constants

In Draft Proposed ANSI C adjacent string constants are automatically concatenated, with a single null character appended at the end:

```
static char helptext[]= "Type:\n"
                        "  h for help\n"
                        "  q to quit\n";
```

This makes it unnecessary to resort to the line continuation convention for writing very long string constants.

Two other changes to strings have more impact on existing programs. First, string constants need not be represented distinctly, i.e., two identical string constants may share the same storage. Second, strings need not be modifiable and it is undefined what will happen if the programmer attempts to modify them.

Here is a simple program that discriminates the implementations.

```
char *string1, *string2;
int main() {
  string1 = "abcd";
  string2 = "abcd";
  if (string1==string2) printf("Strings are shared\n");
  else printf("Strings not shared\n");
  string1[0] = '1';     /* May cause runtime error */
  if (*string1=='1') printf("Strings writable\n");
  else printf("Strings not writable[n");
  return 0;
}
```

It is a good practice for Draft Proposed ANSI C implementations to make strings nonwritable when they can share storage.

In Draft Proposed ANSI C the programmer can specify strings that are not to be writable by storing them in arrays of type "`const char []`":

```
const char die message[] = "?Internal Error\n";
```

11.1.7. C-Ref: ANSI C Character Escape Codes

New character escape codes, `\a` ("alert") and `\?` ("question mark"), are added. The code `\a` is typically mapped to a "bell" or other audible signal on the output device (ASCII control-G). The `\?` escape is used to obtain a question mark character in the rare case in which it might be mistaken as part of a trigraph. The new syntax is:

character-escape-code : one of

(see "C-Ref: Character Escape Codes")

```
a  n  t  b  r  f
v  \  '  "  ?
```

As is traditional in C, the quotation mark `"` may appear un-escaped in character constants and the apostrophe `'` may appear un-escaped in string constants.

Numeric escape codes may be written in hexadecimal notation by following \ with the letter x (lower case only) and from one to three hexadecimal digits. The new syntax is:

numeric-escape-code : (see "C-Ref: Character Escape Codes")
  octal-digit
  octal-digit octal-digit
  octal-digit octal-digit octal-digit
  x *hex-digit*
  x *hex-digit hex-digit*
  x *hex-digit hex-digit hex-digit*

11.2. C-Ref: ANSI C Preprocessor

The Draft Proposed ANSI C standard clarifies many of the properties of the preprocessor, especially macro expansion. Because we have tried to do the same in our earlier exposition, we won't repeat it here. However, Draft Proposed ANSI C does extend the preprocessor in several ways: it provides new operators for merging tokens and converting them to strings; it adds several predefined macros; and it adds several new preprocessor commands.

11.2.1. C-Ref: ANSI C Lexical Structure

Whitespace may precede or follow the `#` that starts a preprocessor command. Lines containing no tokens except `#` are ignored.

11.2.2. C-Ref: ANSI C Stringization and Merging of Tokens

Section "C-Ref: Token Merging in Macro Expansions" mentioned that some C preprocessors, perhaps by accident, can merge tokens and insert tokens into strings. There are new mechanisms in Draft Proposed ANSI C to provide the programmer some control over merging tokens and converting macro parameters to strings.

Within a macro definition, the `#` character is recognized as a unary "stringization" operator that must be followed by the name of a macro formal parameter. During macro expansion, the `#` and the formal name are replaced by the corresponding actual argument enclosed in string quotations. For example, after preprocessing and string concatenation the source text

```
#define TEST(a,b) printf( #a "<" #b "=%d\n", (a)<(b) )
TEST(0,0xFFFFFFFF);
```

becomes

```
printf("0<0xFFFFFFFF=%d\n", (0)<(0xFFFFFFFF) );
```

What happens with whitespace during the stringization process is implementation dependent.

Merging of tokens to form new tokens is controlled by the presence of a "merging" operator, `##`, in macro definitions. After all macro replacements have been done,

the two tokens surrounding any ## operator are combined into a single token. (If they do not form a legal token, the result is undefined.) For example, after preprocessing the source text

```
#define TEMP(i)  temp ## i
TEMP(1) = TEMP(2);
```

becomes

```
temp1 = temp2;
```

11.2.3. C-Ref: ANSI C Predefined Macros

Preprocessors for Draft Proposed ANSI C are required to define five special macros that take no arguments. (Each is spelled beginning and ending with two underscore characters.)

`LINE`	The value of the macro is the line number of the current source program line, expressed as a decimal integer constant.
`FILE`	The value of the macro is the name of the current source file, expressed as a string constant.
`DATE`	The value of the macro is the calendar date of the translation, expressed as a string constant of the form `"Mmm dd yyyy"`.
`TIME`	The value of the macro is the calendar time of the translation, expressed as a string constant of the form `"hh:mm:ss"`.
`STDC`	In a conforming implementation of Draft Proposed ANSI C this macro will be defined and have a nonzero value.

None of these predefined macros may be redefined or undefined by the programmer.

11.2.4. C-Ref: ANSI C #include

The `#include` command syntax is:

include-command : (see "C-Ref: File Inclusion")
    `# include` < *x-char-sequence* >
    `# include` " *x-char-sequence* "
    `# include` *identifer*

x-char-sequence :
    x-char
    x-char-sequence x-char

x-char : one of

    "any character legal in the source program
    except the end-of-line character(s)"

To avoid ambiguity, an *x-char-sequence* following a < character may not include a >

character. Similarly, an *x-char-sequence* following a " character may not include another " character.

This syntax acknowledges that the characters making up the file name argument to `#include` do not have to be recognizable as legal C tokens. Even in the second form of `#include`, where the argument looks like a string constant, the *x-char-sequence* should not undergo the usual backslash escape conventions. (Trigraph substitution presumably does take place; either way it will affect what file names may be represented.)

The argument to `#include` may be an identifier only if the identifier is a macro that evaluates to one of the other permitted forms for `#include`. Since macros must expand to a sequence of legal C tokens, this restricts somewhat the file names that may be specified. The tokens resulting from the macro expansion presumably undergo some implementation-defined merging to yield a file name. No search rules are given for files.

11.2.5. C-Ref: ANSI C Macro Definition and Expansion

The new features for stringization and merging of tokens have already been discussed in section "C-Ref: ANSI C Stringization and Merging of Tokens". There are a few other changes to the way macros are handled.

A significant implementation change is that macros appearing in their own expansion must not be reexpanded. This permits a programmer to redefine a function in terms of its old definition:

```
#define sqrt(x)  ((x)<0 ? sqrt(-x) : sqrt(x))
```

Benign redefinition of macros is allowed. That is, a macro may be redefined if the new definition is, token for token, identical to the existing definition.

Lines are not inspected for preprocessing commands after macro expansion, so macro expansion may not introduce new preprocessor commands.

11.2.6. C-Ref: ANSI C New Commands

There are four additional preprocessor commands defined in Draft Proposal ANSI C: `#elif`, `defined`, `#error`, and `#pragma`.

The commands `#elif` and `defined` are described in section "C-Ref: The #elif Commands" as common extensions to C.

The new directive `#error` produces a compile-time error message that will include the string constant that is an argument to `#error`. It is most useful in detecting programmer inconsistencies and violations of constraints during preprocessing:

```
#if defined(A THING) && defined(NOT A THING)
#error "Inconsistent things!"
#endif
```

```
#if SIZE % 256 != 0
#error "SIZE must be a multiple of 256!"
#endif
```

The new directive `#pragma` allows the programmer to supply implementation-defined information to the compiler. No restrictions are placed on the information that follows the `#pragma` command, and implementations should ignore information they do not understand. There is obviously the possibility that two implementations will place inconsistent interpretations on the same information, so it is wise to use `#pragma` conditionally:

```
#if defined(TCC) && defined(  STDC  ) && defined(vax)
#pragma builtin(abs),inline(myfunc)
#endif
```

11.3. C-Ref: ANSI C Declarations

11.3.1. C-Ref: ANSI C Scopes and Name Spaces

Statement labels have their own name space, which is an improvement.

Structure, enumeration, and union type tags share their own name space. The component names for each distinct structure or union type have their own name space, as is the modern convention.

Declarations of external names must obey normal scoping rules. That is, the following two declarations of `X` do not conflict in the source file, although their behavior at run time is undefined because they assign different types to the same external variable. (A good compiler would issue a warning message.)

```
if (test) {
  extern int X;
  return X;
} else {
  extern double X;
  return X;
}
```

Function prototypes introduce a new kind of scope discussed in section "C-Ref: ANSI C Function Prototypes".

Parameter names appearing in function definitions are treated as if they were declared at the top of the function body. The motivation for this is to disallow the accidental hiding of a parameter by a local declaration, which is almost always the result of misplacing the parameter declarations:

```
int f(x,y)
{
  int x,y;      /* An error in Draft Proposed ANSI C */
  ...
}
```

11.3.2. C-Ref: ANSI C Forward References to Structures

The use of a type specifier of the syntactic classes *structure-type-definition* (section "C-Ref: Structure Types") or *union-type-definition* (section "C-Ref: Union Types") introduces the definition of a new type. The scope of the definition (and the type tag, if any) is from the declaration point to the end of the innermost block containing the specifier. The new definition explicitly overrides (hides) any definition of the type tag in an enclosing block.

The use of a type specifier of the syntactic classes *structure-type-reference* (section "C-Ref: Structure Types") or *union-type-reference* (section "C-Ref: Union Types") without a preceding definition in the same or enclosing scope is allowed when the size of the structure is not required, including when declaring:

1. pointers to the structure

2. a typedef name as a synonym for the structure

The use of the specifier introduces an "incomplete" definition of the type and type tag in the innermost block containing the use. For this definition to be completed, a *structure-type-definition* or *union-type-definition* must appear later in the same scope.

As a special case, the occurrence of a *structure-type-reference* or *union-type-reference* in a declaration with no declarators hides any definition of the type tag in any enclosing scope.

As an example, consider the following correct definition of two self-referential structures in an inner block.

```
{
  struct cell ;
  struct header { struct cell *first; /* ... */ };
  struct cell  { struct header *head; /* ... */ };
  ...
}
```

The incomplete definition "`struct cell ;`" in the first line is necessary to hide any definitions of `struct cell` in an enclosing scope. The definition of `struct header` in the second line automatically hides any enclosing definitions, and its use of `struct cell` to define a pointer is legal. The definition of `struct cell` on the third line completes the information about `cell`.

11.3.3. C-Ref: ANSI C Type Specifiers

There are several new type specifiers, including additions to the signed integer types (`signed`), the floating-point types (`long double`), and two type modifiers, `const` and `volatile`. These are all discussed in the following section on types. The relevant change to the type specifier syntax is:

type-specifier : (section "C-Ref: Type Specifiers")
const-type-specifier
enumeration-type-specifier
floating-point-type-specifier
integer-type-specifier
structure-type-specifier
typedef-name
union-type-specifier
void-type-specifier
volatile-type-specifier

11.3.4. C-Ref: ANSI C Declarators

There are several changes to declarators discussed in this section:

1. Pointer declarators may now contain embedded type specifiers.

2. Function declarators may now include parameter specifications, including a "variable number of arguments" specification.

Pointer declarators Type specifiers (particularly the new `const` and `volatile` type specifiers) may be supplied in pointer declarators:

pointer-declarator : (section "C-Ref: Pointer Declarators")
∗ *type-specifier-list*$_{opt}$ *declarator*

type-specifier-list : (new)
type-specifier
type-specifier-list type-specifier

The reason for this change to pointer declarators is to give the programmer the ability to declare a "constant pointer" and also a "pointer to constant data."

Function declarators Parameter type information may now be included in all function declarators. The new syntax is:

function-declarator : (section "C-Ref: Function Declarators")
declarator (*parameter-type-list* ,...$_{opt}$)
declarator (*parameter-list*$_{opt}$)

parameter-type-list :
parameter-declaration
parameter-type-list , *parameter-declaration*

parameter-declaration :
declaration-specifiers declarator
declaration-specifiers abstract-declarator

To avoid an ambiguity between a *parameter-list* and a *parameter-type-list*, it is illegal to have a parameter name that is the same as a visible typedef name. (It would probably be poor style to do so anyway.)

The only storage-class-specifier allowed in a parameter declaration is `register`, which is ignored unless the declaration appears in a function definition. This means that `register` cannot be used in a prototype to alter the calling convention of the function; it can only be used as a hint within the function body.

A function declarator that includes the *parameter-type-list* is said to include a function *prototype*, and that prototype specifies the number and type of the arguments accepted by functions of that type. A function declarator without the *parameter-type-list*, that is, with either a *parameter-list* of identifiers or nothing at all, declares a function type without a prototype, and such function types may be called with arbitrary parameters as is traditional in C. The prototype syntax may be used in declarators for function declarations, function definitions, or function types used within other types. Here are some examples of prototypes:

```
extern double sqrt(double x);
int abs(int i)  { return i<0 ? -i : i ; }
void (*func array[3])(int order,double epsilon) =
        { &f1, &f2, &f3 };
```

The following section discusses prototypes in more detail.

11.3.5. C-Ref: ANSI C Function Prototypes

Function prototypes are one of the significant additions in Draft Proposal ANSI C. They add to the C language the ability to check function arguments for consistency, and in fact to coerce those arguments to the declared types of the formal parameters. The use of function prototypes should increase program readability and decrease errors. The nonprototype form is kept only to accommodate existing programs.

The syntax for function prototypes is given in section "C-Ref: ANSI C Declarators". Prototypes can be distinguished from the traditional function declarators syntactically, and programmers must not mix the two in a single function or the resulting behavior is undefined. It will be clearer if we discuss the two "styles" separately, beginning with a review of the status quo.

In the traditional style, not using prototypes:

- Functions may be declared implicitly by their appearance in a call.
- Arguments to functions undergo the usual argument conversions before the call.
- No checking of the type or number of arguments occurs. All functions potentially take an arbitrary number of arguments with arbitrary types.

In contrast to this, when prototypes are used:

- Functions must be declared explicitly—with a prototype—before any call on them. If there are multiple declarations, they must agree exactly.

- Arguments to functions are converted, as if by assignment, to the declared types of the formal parameters.

- The number and types of arguments must agree with (or be convertible to) the declared types, or else the program is in error. Functions taking a variable number of arguments are designated explicitly.

The rest of this discussion will concern the use of prototypes.

There are three basic kinds of prototypes depending on whether a function takes no arguments, a fixed number of arguments, or a variable number of arguments:

1. A function that takes no arguments must have a *parameter-type-list* consisting of the single type specifier `void`, e.g.,

```
extern int random generator(void);
static void print header(void);
```

2. A function that takes a fixed number of arguments indicates the types of those arguments in the *parameter-type-list*. If the prototype appears in a function declaration, parameter names may be included or not, as desired; they do not affect the prototype. (We think they help in documenting the function.) For example,

```
extern double atan2(double, double);
extern char *strncpy(char *dest,
                     char *src, int count);
```

3. A function that takes a variable number of arguments or arguments of varying types indicates the types of any fixed arguments in the normal way and follows them by the notation ",..." (which is composed of four tokens: a comma and three periods):

```
extern int fprintf( FILE file, char *format,... );
```

   There must be at least one fixed parameter, or else the parameter list cannot be referenced using the standard library facilities from `stdarg.h` (section "C-Ref: **VARARG, STDARG**").

Prototypes may be used in any function declarator, including those used to form more complicated types. For example, the Draft Proposal ANSI C declaration of `signal` (section "C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**") is:

```
void (*signal(int sig,void (*func)(int sig)))(int sig);
```

This declares `signal` to be a function that takes two arguments: `sig`, an integer, and `func`, a pointer to a void function of a single integer argument, `sig`. `signal` returns a pointer of the same type as its second parameter, i.e., a pointer to a void function taking a single integer argument. A clearer way to write the declaration of `signal` is:

```
typedef void (*sig handler)(int sig);
sig handler signal(int sig, sig handler func);
```

It is possible to use prototypes for some declarators and not for others (as long as they do not refer to the same function or function type). If we were to declare `signal2` as

```
typedef void (*sig handler2)();
sig handler2 signal2(int sig, sig handler2 func);
```

then we would lose the prototype on the `sig handler2` function pointer, although `signal2` still has the prototype.

Now we consider function definitions. A function definition necessarily includes a function declarator, and that declarator may be in prototype form or traditional form. For example, the following two definitions of `f` would be approximately equivalent:

```
int f(int i, int j) { /*...*/ } /* prototype form   */
int f(i,j) int i,j; { /*...*/ } /* traditional form */
```

We say they are "approximately" equivalent because there may be a difference in whether a prototype exists or not. These are the rules:

1. A function defined in prototype form simultaneously establishes a prototype for that function. That prototype must agree with any preceding or following declarations of the same function (which also must be in prototype form).

2. A function defined in traditional form does not introduce a prototype, but if a prototype exists because of a previous declaration, the widened parameter declarations in the definition must agree exactly with the prototype and that prototype remains in effect.

The second rule makes it easier to convert existing programs to use prototypes. It is only necessary to introduce a declaration in prototype form or replace existing traditional declarations. The function definition need not be altered.

Agreement of formal parameter types in duplicate prototype declarations must be exact, including array bounds (except the first) and function return types (including prototype information, if present). The parameter names (if present) must also agree. For example, below are listed several pairs of formal parameter types. In no case are the elements of the pair equivalent in the sense required:

```
int                  short
int *                short *
int ()               int (double x)
int (*)[20,20]       int (*)[10,40]
```

Calling functions with prototypes When a function with a prototype is called, the actual arguments of the call must agree in number and type with the prototype formals, or must be assignment compatible with them. The actuals are converted as by assignment to the formal types before the call. Those arguments in the variable part of the prototype formal list (corresponding to the `",..."` portion of the formal list) undergo the usual argument conversions before being passed.

The presence of a prototype does not restrict in any way the implementation of function calls. Some implementations may wish to take advantage of the additional

information provided by prototypes and optimize certain function linkages, but this is not required. One additional freedom is granted implementors: unless a function is defined in prototype form and that prototype specifies a variable number of arguments, the implementation need not use a function linkage that supports a variable number of arguments. In effect this means that any existing functions that take a variable number of arguments must be rewritten to have a prototype before they are compiled by an ANSI C implementation.

Miranda prototypes Finally, Draft Proposed ANSI C specifies that calls on a function without a prototype should have a certain relationship to calls on a function with a prototype. In particular, a function call not within the scope of a prototype should behave exactly as if it was within the scope of prototype whose fixed number of parameter types were exactly the types of the actual arguments after the usual argument conversions are applied. (The term "Miranda prototypes" was coined because the rule effectively states that all function calls have a right to a prototype; if they cannot afford a prototype one will be appointed for them.)

For example, if a compiler for Draft Proposed ANSI C sees the function call

```
process( a, b, c, d );
```

where the types of the actual arguments are

```
short a;  struct {int a,b;} b;  float *c;  float d;
```

then the function call should be implemented the same as if this prototype were in effect:

```
int process(int, struct {int a,b;}, float *, double);
```

(If an explicit return type had been declared traditionally, that return type would be used in the prototype.) Note that this rule does not actually establish a prototype which might affect later calls. Should a second call on `process` appear later in the program:

```
process( x, y, z )
```

where `x`, `y`, and `z` all have type `float`, then that second call must be implemented as if the prototype were

```
int process( double, double, double );
```

Because the linkages may be different in the two calls, run-time confusion can certainly result. However, Draft Proposed ANSI C does not require that functions taking a variable number (or type) of arguments work correctly unless a prototype is used.

11.3.6. C-Ref: ANSI C Initializers

Unions may be initialized. The initializer must be an expression that would be acceptable as an initializer for the first component of the union, e.g.,

```
union U {double d; long q;} x = 0.0;
union V {struct {int a; union U *b;} s;
        struct {union U *b; int a;} t; } y = {0, &x};
```

Automatic array, structure, and union variables may be initialized, but the initializers must be constant expressions that would be acceptable as initializers of static variables. This somewhat arbitrary restriction is believed to make implementation a bit easier and to avoid some pathological initializers.

In the case of automatic structure and union variables, the rule is relaxed to permit general initializer expressions whose type is the same as the variable being initialized. For example, the following initializer is permitted:

```
extern struct S f(), a;
auto struct S  x = (i<0 ? f() : a);
```

Aggregate initializers using brace-enclosed lists of expressions must either fully specify all the levels of braces or must omit all but the single outermost set of braces. This restriction removes some ambiguities in interpreting partially-structured initializers.

When a list of initializer expressions is too short for the aggregate being initialized, the remaining elements are initialized to 0, cast to the appropriate types. That is, the initializations of the two variables below are the same:

```
struct S {int a; float b; char *c; };
struct S x = { 0 };
struct S y = { 0,  0.0F, (char *)0 };
```

Static and global variables that do not have explicit initializers are initialized to zeros, cast to the appropriate type.

The old form of initializers, in which the `=` operator was omitted, is no longer acceptable.

11.3.7. C-Ref: ANSI C External Names

Names declared `extern` have file or block scope according to their position. Later re-declarations of an external name may supply more information.

Draft Proposed ANSI C adopts the "omitted storage class" model (also known as the "strict ref/def" model) for resolving external data definitions. The "mixed common model" (also known as "relaxed ref/def") is recognized as a common extension.

11.4. C-Ref: ANSI C Types

Types have been extended in several ways:

1. Unsigned integer types now officially come in all sizes, as generally assumed in this book.

2. Signed integer types may now be designated with the type specifier `signed`, useful with `char` and bit fields.

3. A new floating-point type, `long double`, has been added. The specifier `long float` is no longer allowed as a synonym for `double`.

4. Two new type specifiers, `const` and `volatile`, have been added. They act as type modifiers rather than as new types.

5. Enumeration types and the `void` type are officially added.

11.4.1. C-Ref: ANSI C Integer Types

The new type specifier `signed` may be used to explicitly request a signed integer type. The corresponding syntax changes are:

signed-type-specifier : (section "C-Ref: Signed Integer Types")

`signed`
`signed`$_{opt}$ `int`
`signed`$_{opt}$ `short int`$_{opt}$
`signed`$_{opt}$ `long int`$_{opt}$

character-type-specifier : (section "C-Ref: Character Type")

`char`
`signed char`
`unsigned char`

Aside from adding some symmetry to the language, the major use for the new `signed` specifier is in conjunction with character types and bit fields. When the `char` type is implemented as an unsigned type—as is allowed—the `signed` specifier forces a signed implementation. This is analogous to using `unsigned` to force an unsigned representation on characters normally implemented as signed. Bit fields, whose signedness usually follows that of type `char`, operate the same way. (As expected, when `signed` is used without other type specifiers, it is equivalent to `signed int`.)

For example, if type `char` uses an 8-bit, two's complement representation, given the declarations

```
unsigned char uc = -1;
signed char sc = -1;
char c = -1;
int i=uc, j=sc, k=c;
```

then `i` must have the value 255 and `j` must have the value -1 in all implementations. However, `k` may have the value 255 or -1, depending on the implementation. Similarly, after the declarations

```
struct S { unsigned ubf:3;
           signed   sbf:3;
           int       bf:3; } x = { -1, -1, -1 };
...
{ int i=x.ubf, j=x.sbf, k=x.bf; ...
```

Then `i` must have the value 7, `j` must have the value -1, and `k` may be either 7 or -1.

Finally, Draft Proposed ANSI C mandates that integer types use binary encodings to permit portable bitwise operations on positive integers.

11.4.2. C-Ref: ANSI C Floating-point Types

A new floating-point type, `long double`, has been added. To avoid confusion, the notation `long float` is no longer allowed as a synonym for `double`. The new syntax is:

floating-type-specifier : (section "C-Ref: Floating-Point Types")
```
float
double
long double
```

Modern computers are increasingly providing floating-point numbers in "single," "double," and "extended" precisions (often represented in 32, 64, and 128 bits, respectively). The `long double` type gives C programs access to the extended precision numbers. Implementors are free to use the same implementation for `double` and `long double`, as they are free to make `double` and `float` the same. The usual conversion rules have been augmented to handle the new floating-point type in an expected manner.

11.4.3. C-Ref: ANSI C const

The new type specifier `const` may be used with other type specifiers—including structure, union, and enumeration type specifiers and `volatile`—or may be used alone, in which case the additional specifier `int` is assumed. The corresponding syntax changes are:

const-type-specifier : (new)
const

The `const` type attribute prevents objects from having their value changed. That is, any object (strictly speaking, any lvalue expression) whose type includes the `const` type specifier may not be assigned to or have its value modified by the `++` or `--` operators:

```
const int ic = 37;
ic = 5;         /* Illegal */
ic++;           /* Illegal */
```

The new syntax for pointer declarators allows the declaration of both "constant pointers" and "pointers to constant data":

```
int * const const pointer;
const int *pointer to const;
```

The syntax may be confusing: constant pointers and constant integers, say, have the type specifier `const` in different locations. The appearance also changes when `typedef` names are used; the constant pointer `const pointer` may also be declared

```
typedef int *int pointer;
const int pointer const pointer;
```

or equivalently

```
int pointer const const pointer;
```

A pointer to constant data may be assigned to, but the object to which it points cannot be. Expressions with this type can also be generated by applying the address operator & to values whose type includes const.

```
const int * pc;  /* pointer to a constant integer */
int *p, i;
const int ic;
pc = p = &i;    /* OK */
pc = &ic;       /* OK */
*p = 5;         /* OK */
*pc = 5;        /* Illegal */
```

Assigning a value of type "pointer to const *T*" to an object of type "pointer to *T*" is allowed only by using an explicit cast. Continuing the previous example:

```
pc = &i;        /* OK */
pc = p;         /* OK */
p = &ic;        /* Illegal */
p = pc;         /* Illegal */
p = (int *)&ic; /* OK */
p = (int *)pc;  /* OK */
```

The language rules for const are not foolproof, that is, they may be bypassed or overridden if the programmer tries hard enough. (For instance, the address of a constant object can be passed to an external function.) However, implementations are permitted to allocate objects whose type includes const in read-only storage, so that attempts to alter constant objects may cause run-time errors, as shown in this program fragment:

```
const int * pc, * p;
const int ic = 0;
pc = &ic;       /* OK */
p = (int *)pc;  /* Legal, but dangerous */
*p = 5;         /* Legal, but may cause
                   run-time error */
```

11.4.4. C-Ref: ANSI C volatile

The new type specifier volatile may be used with other type specifiers—including structure, union, and enumeration type specifiers and const—or may be used alone, in which case the additional specifier int is assumed. The corresponding syntax changes are:

volatile-type-specifier : (new)
    volatile

The volatile type attribute informs the Draft Proposed ANSI C implementation that certain objects can have their values altered in ways not under control of the implementation. That is, any object (strictly speaking, any lvalue expression) whose type includes the volatile type specifier should not participate in optimizations that would increase, decrease, or delay any references to, or modifications of, the object.

To be more precise, Draft Proposed ANSI C introduces the notion of *sequence points* in C programs. A sequence point exists at the completion of all expressions which are not part of a larger expression, that is, at the end of expression statements, statement control expressions, return expressions, and initializers. Additional sequence points are present in function calls immediately after all the arguments are evaluated, in the logical AND and OR expressions, and before the conditional operator (?) and the comma operator (,).

References to and modifications of volatile objects must not be optimized across sequence points, although optimizations between sequence points are permitted. Extra references or modifications can be generated at any time. For example, consider the following program fragment:

```
extern int f();
auto int i,j;
...
i = f(0);
while (i) {
  if (f(j*j)) break;
}
```

If the variable i were not used again during its lifetime, then traditional C implementations would be permitted to rewrite this program fragment as:

```
  if (f(0)) {
    i = j*j;
    while( !f(i) ) ;
  }
```

The first assignment to i was eliminated, and i was reused as a temporary variable to hold j*j, which is evaluated once outside the loop.

If we change the declaration of i and j to

```
auto volatile int i,j;
```

then these optimizations would not be permitted. However, it would be permitted to write the loop as shown below, eliminating one reference to j before the sequence point at the end of the if statement control expression:

```
i = f(0);
while (i) {
  register int temp = j;
  if (f(temp*temp)) break;
}
```

The new syntax for pointer declarators allows the declaration of type "pointer to volatile ...." References to this kind of pointer may be optimized, but references to the object to which it points cannot be. Assigning a value of type "pointer to volatile *T*" to an object of type "pointer to *T*" is allowed only when an explicit cast is used. For example:

```
extern volatile int *  pv, *p;
pv = p;         /* OK */
p = pv;         /* Illegal */
p = (int *)pv;  /* OK */
```

The most common use of volatile is to provide reliable access to special memory locations used by the computer hardware or by asynchronous processes. Consider the following typical example. A computer has three special hardware locations:

Address	Use
0xFFFFFF20	Input data buffer
0xFFFFFF24	Output data buffer
0xFFFFFF28	Control register

The control register and input data buffer can be read by a program but not written; the output buffer can be written but not (usefully) read. The third least significant bit of the control register is called "input available"; it is set to 1 when data has arrived from an external source, and it is set to 0 automatically when that data is read out of the input buffer by the program (after which time the contents of the buffer are undefined until "input available" becomes 1 again). The second least significant bit of the control register is called "output available"; when the external device is ready to accept data, the bit is set to 1. When data are placed in the output buffer by the program the bit is automatically set to 0 and the data are written out. Placing data in the output buffer when the control bit is 0 causes unpredictable results.

The function copy data below copies data from the input to the output until an input value of 0 is seen. The number of characters copied is returned. There is no provision for overflow or other error conditions.

```
typedef unsigned long datatype, control type, counttype;
#define CONTROLLER/
   ((const volatile controltype * const) 0xFFFFFF28)
#define INPUT BUF/
   ((const volatile datatype * const) 0xFFFFFF20)
#define OUTPUT BUF ((volatile datatype * const) 0xFFFFFF24)
#define INPUT READY BIT  0x4
#define OUTPUT READY BIT 0x2
#define input ready   ((*CONTROLLER) & INPUT READY BIT)
#define output ready  ((*CONTROLLER) & OUTPUT READY BIT)
```

```
counttype copy data()
{
  counttype count = 0;
  datatype temp;
  for(;;) {
    while (!input ready) ;  /* Wait for input */
    temp = *INPUT BUF;
    if (temp == 0) return count;
    while (!output ready) ; /* Wait to do output */
    *OUTPUT BUF = temp;
    count++;
  }
}
```

11.4.5. C-Ref: ANSI C Generic Pointers

Type `void *` is added as a "generic pointer." It was introduced to accommodate the need for a uniform pointer representation while allowing an implementation to choose efficient (possibly different) representations for pointers to different base types. Generic pointers cannot be dereferenced with the * or subscripting operators. All other pointer types (except perhaps function pointers) can be converted to type `void *` and back without change; explicit casts should be used for clarity but need not be:

```
void *generic ptr;
int *int ptr;
char *char ptr;
generic ptr = int ptr;          /* OK */
int ptr = generic ptr;          /* OK */
int ptr = char ptr;             /* Illegal */
int ptr = (int *) char ptr;     /* OK */
```

Generic pointers provide additional flexibility in using function prototypes. When a function has a formal parameter that can accept a pointer of any type, the formal should be declared to be of type `void *`. If the formal is declared with any other pointer type, the actual argument must be of the same type since pointer types are no longer assign compatible (section "C-Ref: ANSI C Assignment Conversions"). For example, the `strcpy` facility copies character strings and therefore requires arguments of type `char *`:

```
char *strcpy(char *s1, const char *s2);
```

On the other hand, `memcpy` can take a pointer to any type and so uses `void *`:

```
void *memcpy(void *s1, const void *s2, size t n);
```

11.5. C-Ref: ANSI C Conversions and Representations

11.5.1. C-Ref: ANSI C Number Representation

Draft Proposed ANSI C mandates that implementations provide a standard header file, `limits.h`, that defines certain characteristics of the integer data types. Table "C-Ref: ANSI C Minimum Integer Sizes" lists the standard names that must be defined in `limits.h` and gives for each the minimum (absolute) value that conforming implementations must support. A second header file, `float.h`, defines the characteristics of the types `float`, `double`, and `long double`. Excerpts from that file are listed in table "C-Ref: ANSI C Floating-point Characteristics".

11.5.1.1. C-Ref: ANSI C Minimum Integer Sizes

Name	Minimum	Meaning
CHAR BIT	8	width of char type, in bits
SCHAR MIN	-127	minimum value of signed char
SCHAR MAX	127	maximum value of signed char
UCHAR MAX	255	maximum value of unsigned char
SHRT MIN	-32767	minimum value of short int
SHRT MAX	32767	maximum value of short int
USHRT MAX	65535	maximum value of unsigned short
INT MIN	-32767	minimum value of int
INT MAX	32767	maximum value of int
UINT MAX	65535	maximum value of unsigned int
LONG MIN	-2147483647	minimum value of long int
LONG MAX	2147483647	maximum value of long int
ULONG MAX	4294967295	maximum value of unsigned long
	If type char is signed by default:	
CHAR MIN	SCHAR MIN or 0	minimum value of char
CHAR MAX	SCHAR MAX or UCHAR MAX	maximum value of char
	If type char is unsigned by default:	
CHAR MIN	0	minimum value of char
CHAR MAX	UCHAR MAX	maximum value of char

11.5.1.2. C-Ref: ANSI C Floating-point Characteristics

Name	Minimum	Meaning
FLT RADIX	2	radix of exponent in floating-point representation (all floating-point types)
FLT ROUNDS		>0 if addition rounds, 0 if chops, else unknown (all floating-point types)
FLT EPSILON	1E-5	minimum x>0.0 such that 1.0+x != x
DBL EPSILON	1E-5	
LDBL EPSILON	1E-5	

FLT DIGITS	6	number of decimal digits of precision
DBL DIGITS	6	
LDBL DIGITS	6	
FLT MIN	1E-37	minimum normalized positive number
DBL MIN	1E-37	
LDBL MIN	1E-37	
FLT MAX	1E+37	maximum representable finite number
DBL MAX	1E+37	
LDBL MAX	1E+37	
FLT MIN 10 EXP	-37	minimum x such that 1Ex approximates `FLT MIN`
DBL MIN 10 EXP	-37	
LDBL MIN 10 EXP	-37	
FLT MAX 10 EXP	37	maximum x such that 1Ex approximates `FLT MAX`
DBL MAX 10 EXP	37	
LDBL MAX 10 EXP	37	

11.5.2. C-Ref: ANSI C Assignment Conversions

The rules given in section "C-Ref: The Assignment Conversions" still apply. Conversions between pointer types during assignment—a common extension in many implementations—is not permitted in Draft Proposal ANSI C except when one of the types is `void *` or when (in assignments) the right-hand side is the constant integer 0. In all other cases an explicit cast must be used. A corollary of this rule is that pointer arguments to functions specified with prototypes must match exactly. For example, it is illegal to pass a pointer of type `char *` to a function expecting an argument of type `int *`.

Assignment of types that include `const` and/or `volatile` type specifiers are subject to additional restrictions discussed with those type specifiers.

Draft Proposed ANSI C does not address the conversion of function pointers in any detail. Conversions between function pointers is allowed with explicit casting, but there is an admission that function pointers might have a representation on some computers that does not map to type `void *`. The programmer should avoid conversions between function and data pointers.

11.5.3. C-Ref: ANSI C The Usual Unary Conversions

The usual unary conversions no longer automatically promote type `float` to type `double`, permitting (but not requiring) arithmetic operations directly on values of type `float`. (Implementations are always free to do arithmetic operations using more precision than required.)

The usual unary conversion rules for the integral types are different from those given for traditional implementations:

1. All the integral types shorter than int, including their unsigned varieties, are widened to int. (Many C implementations traditionally widened shorter unsigned types to unsigned; this is a change for them.) Note that types char and short, although usually shorter than int, may not be in some implementations.

2. Those signed integral types that are the same size as int are converted to type int. Those unsigned integral types that are the same size as int are converted to type unsigned int.

3. In implementations that treat characters and bit fields as signed types, the characters and bit fields are widened according to their base type. In implementations that treat characters and bit fields as unsigned values, the characters and bit fields are widened as if their base type was unsigned. (These rules apply if neither signed nor unsigned is specified explicitly in the declaration.)

For example, assume that type char occupies 8 bits, type short occupies 16 bits, and type int occupies 32 bits. Then the usual unary conversions for integers is given in table "C-Ref: ANSI C Usual Conversions in an Example Signed Implementation" for implementations that treat type char and bit fields as signed and unsigned, respectively. Notice that in this example the only difference is in the widening of bit fields of type int. For implementations under which sizeof(short)==sizeof(int), the conversion rules for type short would be the same as those given for type int in the tables.

11.5.3.1. C-Ref: ANSI C Usual Conversions in an Example Signed Implementation

Original Operand Type	*Signed Converted Type*	*Unsigned Converted Type*
char	int	int
short	int	int
int	int	int
unsigned char	int	int
unsigned short	int	int
unsigned int	unsigned	unsigned
char bit field	int	int
short bit field	int	int
int bit field	int	unsigned
unsigned char bit field	int	int
unsigned short bit field	int	int
unsigned int bit field	unsigned	unsigned

11.5.4. C-Ref: ANSI C The Usual Binary Conversions

The usual binary conversions under Draft Proposed ANSI C are listed below. As customary, the usual unary conversions are first performed on each operand separately before these rules are applied.

1. If either operand is not of arithmetic type, or if the two operands have the same type, then no additional conversion is performed.

2. Otherwise, if one operand is of type `long double`, then the other operand is converted to type `long double`.

3. Otherwise, if one operand is of type `double`, then the other operand is converted to type `double`.

4. Otherwise, if one operand is of type `float`, then the other operand is converted to type `float`.

5. Otherwise, if one operand is of type `unsigned long int`, then the other operand is converted to type `unsigned long int`.

6. Otherwise, if one operand is of type `long int`, then the other operand is converted to type `long int`.

7. Otherwise, if one operand is of type `unsigned int`, then the other operand is converted to type `unsigned int`.

8. Otherwise, both operands must be of type `int`, so no additional conversion is performed.

11.5.5. C-Ref: ANSI C The Function Argument Conversions

The usual unary conversions are applied to function arguments whose types are not specified in a function prototype. In addition, arguments of type `float` are converted to type `double`, for compatibility with existing code.

11.6. C-Ref: ANSI C Expressions

Most of the changes to expressions in Draft Proposed ANSI C concern the permitted types of the operands. In general, the rules have been tightened to promote more readable and portable programs.

11.6.1. C-Ref: ANSI C Component Selection

The left operands of `.` and `->` must be of the proper structure, union, or pointer type, except that the null pointer constant `0` may be used in the following stylized way to determine the offset in bytes of a structure component within the structure:

```
#define OFFSET(type,field)  \
        ((size t)(char *)&((type *)0)->field)
```

11.6.2. C-Ref: ANSI C Function Calls

In the function invocation `f(...)`, the expression `f` may have the type "pointer to function...," in which case an automatic dereference of the function pointer is made in order to perform the call.

11.6.3. C-Ref: ANSI C `sizeof` Operator

The result of the `sizeof` operator may be of type `unsigned int` or `unsigned long`. The type chosen by an implementation is defined as `size t` in the standard header file `stddef.h`.

11.6.4. C-Ref: ANSI C Address Operator

The address operator `&` applied to a function results in a value of type "pointer to function...." The address operator `&` applied to an array yields "pointer to array of T" or "pointer to pointer to T". (This is a change from most current implementations, which simply ignore the `&` operator.)

11.6.5. C-Ref: ANSI C Unary Plus Operator

Unary plus is added with special semantics. The new syntax is:

unary-expression : (see "C-Ref: Unary Expressions")
postfix-expression
cast-expression
sizeof-expression
unary-minus-expression
unary-plus-expression
logical-negation-expression
bitwise-negation-expression
address-expression
indirection-expression
preincrement-expression
predecrement-expression

unary-plus-expression : (new)
\+ *unary-expression*

A unary plus expression "+*e*" may be considered to be a shorthand notation for "`0+(`*e*`)`"; the two expressions in effect always perform the same computation.

The unary plus expression has the additional effect of prohibiting any optimizations that would regroup subexpressions of *e* with subexpressions outside *e*. It is to give the programmer the ability to force some evaluation order on expressions that this new operator was introduced into the langauge. (The unary minus operator does not force an evaluation order.) For example, it is well known that normalized floating-point addition does not obey the associative law. That is, the expression `((X+Y)-X)-Y`, evaluated as written, does not necessarily yield the same value as `(X+Y)-(X+Y)`, or zero. In order to prohibit compilers from rearranging the first ex-

pression the C programmer must write

```
+((+(X+Y))-X)-Y
```

11.6.6. C-Ref: ANSI C Addition and Subtraction

When two pointers of the same type are subtracted the result has type `ptrdiff t`, defined in the standard header file `stddef.h`.

Two pointers of type `void *` cannot be subtracted, nor can an integer be added to or subtracted from a pointer of type `void *`.

Draft Proposed ANSI C requires that a pointer to the first element beyond the end of an array be represented well enough to permit subtraction and relational comparisons with a pointer located within the array. (We're not sure this will be possible on some computers with segmented address spaces, when an array spans an entire segment.)

11.6.7. C-Ref: ANSI C Relational Expressions

Relational expressions between pointers of type `void *` are not allowed. However, equality comparisions are permitted in these cases:

1. between values of arithmetic types
2. between pointers of the same type
3. between a pointer of type `void *` and any other pointer
4. between any pointer and the integer constant 0

11.6.8. C-Ref: ANSI C Constant Expressions

The definition of constant expressions has been clarified. Comma operators are not allowed in constant expressions. The difference of the addresses of two members of the same aggregate is a constant expression and may be used as a constant initializer.

The preprocessor is permitted to perform its constant arithmetic using the natural long integer arithmetic of the host computer. Therefore preprocessor commands can't be reliably used to determine the characteristics of the target computer's arithmetic. For example, here is a program fragment that incorrectly attempts to see if type `int` on the target computer is larger than 16 bits:

```
#if 1<<16
  /* Target computer integer has
        more than 16 bits */
...
#endif
```

In fact, the program may only be testing the representation of type `long` on the host computer.

Static initializers may involve arbitrary expressions involving floating-point constants. Draft Proposed ANSI C states that an implementation is free to perform the actual initialization at run time, and so is not required to simulate the target computer's floating-point arithmetic. However, doing this initialization at run time—before any code that accesses the initialized variable could be executed—would be very difficult in general.

C implementations may use the natural floating-point arithmetic on the host computer for constant expressions as long as its precision is at least as great as on the target computer.

11.7. C-Ref: ANSI C Statements

The control expression of a `switch` statement may be of any integer type; it is subject to the usual unary conversions. Constant expressions appearing in `case` labels are cast to the type of the `switch` expression (after its conversion), and are then checked only for a duplication of values among `case` labels.

In a `return` statement, the expression (if any) is converted as if by assignment to the declared return type of the enclosing function. The statement is illegal if such a conversion is not possible.

11.8. C-Ref: ANSI C Run-time Library

The C run-time library, always one of C's major assets, has been standardized along with the language in Draft Proposed ANSI C. The facilities included in the library avoid those functions particular to UNIX. All the library facilities are classified into several groups, each with its own header file. The library facilities are listed by group in tables "C-Ref: Draft Proposed ANSI C Libraries (Part 1)" and "C-Ref: Draft Proposed ANSI C Libraries (Part 2)" in section "C-Ref: Draft Proposed ANSI C Facilities" and are discussed in subsequent chapters.

Library facilities and header files are special in many ways, mostly to protect the integrity of implementations:

1. Library names are in principle reserved, that is, programmers may not define external objects whose names duplicate the names of the standard library. (All names beginning with an underscore are also reserved to implementations.)

2. Library header files or file names may be "built in" to the implementation, although they still must be included for their names to become visible.

3. Programmers may include library header files in any order, any number of times.

The last requirement may force an implementation to use some careful mecha-

nisms to avoid duplicate declarations:

```
/* Header stddef.h */
#ifndef  STDDEF
#define  STDDEF 1
  typedef int ptrdiff t;  /* Don't try to redeclare */
  ...
#endif
```

Draft Proposed ANSI C requires that most "function-like" library facilities really be implemented as functions, so that the programmer can pass their address, say, to another function. However, to allow for more efficiency, the header files may hide the function name with an equivalent macro. Here is a hypothetical declaration of a function `nonzero` that returns 1 if its argument is nonzero and otherwise returns 0. (Note that the cast to `int` is necessary in the macro to simulate the action of a function call in the scope of a prototype.)

```
extern int nonzero( int x );      /* Functional form */
#define nonzero(x) ((int)(x)?1:0) /* Macro form */
```

A programmer requiring the functional form would have to include an explicit `#undef` command to hide any macros:

```
#ifdef nonzero
#undef nonzero
#endif
```

The library functions are mostly written to take arguments whose types are unchanged under the usual argument conversions. This allows them to be called without the help of a prototype, for compatibility with existing implementations. Compatibility cannot be reliably preserved for functions like `fprintf` and `fscanf` that take a variable number of arguments, because Draft Proposed ANSI C requires such functions to be called in the scope of a prototype.

PART III.

C-REF: THE C LIBRARIES

12. C-Ref: Introduction to the Libraries

Many facilities that are used in C programs are not part of the C language as such but are part of "standard libraries" that are written in C itself for use by other C programs. These facilities include:

- signals (exceptions) and other nonlocal control functions
- operations on characters and strings
- computing and printing the time of day
- operations on arbitrary blocks of memory
- mathematical functions
- storage allocation functions
- input and output operations
- communication with the host operating system

Each of these facilities belongs to a particular library. The correct way to use a facility is to have, at the beginning of the user program, a preprocessor `#include` command to include the relevant library declarations; the facility may then be referred to by name within the user program. For example, in order to use the trigonometric function `cos` in a program, the C programmer should put the command

```
#include <math.h>
```

at the start of the program, thus declaring `cos` for subsequent use. Unfortunately, in many implementations, not all facilities are declared in standard header files; some must be declared explicitly by the programmer. In the descriptions that follow, the header file is shown if there commonly is one. (In Draft Proposed ANSI C, the standard header files may be built into the implementation, that is, they may not be files at all. This permits implementations to conform to Draft Proposal ANSI C even if, say, their file systems do not permit periods in file names.)

The word *facility* has been used in order to evade the question of whether an operation is implemented as a function or a preprocessor macro. Most of the facilities are described as if they were functions, but the implementor is usually free to provide an equivalent—but presumably more efficient—macro. With this understanding, we will often continue to use the more natural term "function." To prevent confusion and subtle bugs, however, a facility implemented as a macro should evaluate every argument expression exactly once, just as it would if it were implemented as a function. (In some cases the programmer does care: if an operation is defined as a macro, it can be removed with `#undef`; if it is a function, the function's address can be passed to another function.)

When a library function is invoked and cannot complete an operation successfully, then it may do either or both of two things: return a special value indicating failure or store a nonzero error code into the external variable `errno` (section "C-Ref: **ERRNO, STRERROR, PERROR**"). The actions taken by various facilities are described explicitly below for each individual facility. Error codes and the external variable `errno` are described in section "C-Ref: **ERRNO, STRERROR, PERROR**".

Draft Proposed ANSI C is the first description of C to explicitly include a large standard library that is independent of the host operating system. Older implementations of C provide different sets of facilities, although the libraries provided with UNIX have been a model for many implementations. We have chosen to describe the facilities that fall into any of three groups:

1. the facilities included in Draft Proposed ANSI C

2. the facilities in common use that duplicate facilities in Draft Proposed ANSI C

3. other facilities in common use that are not heavily dependent on the underlying system (e.g., UNIX)

The facilities are divided into groups according to their general purpose. Each facility is described by giving the appropriate header file (if there is one), followed by a typical function or macro declaration of the facility and a longer prose description.

12.1. C-Ref: Draft Proposed ANSI C Facilities

Draft Proposed ANSI C is the first version of C to include a careful description of a set of standard libraries. In the chapters that follow, the library facilities listed should be assumed to be present in both Draft Proposed ANSI C and traditional implementations of C unless comments to the contrary appear. For example, consider this synopsis:

```
#include <header.h>                                    /* ANSI */

int f();

int g(a)                                     /* Non-ANSI form */
void g(a)                                             /* ANSI */
  int a;
```

The description should be understood to mean:

1. The header file `header.h` is provided only in Draft Proposed ANSI C.

2. The function `f` is provided in both Draft Proposed ANSI C and in many traditional implementations.

3. The function g is provided in both implementations, but under Draft Proposal ANSI C it has a void return value whereas in traditional implementations it returns an integer.

The accompanying prose descriptions should also help to clarify the situation.

The grouping of facilities in Draft Proposed ANSI C do not always follow the divisions used in this book. The tables "C-Ref: Draft Proposed ANSI C Libraries (Part 1)" and "C-Ref: Draft Proposed ANSI C Libraries (Part 2)" list the standard facilities as organized in Draft Proposed ANSI C along with references to the descriptions in this book.

12.1.1. C-Ref: Draft Proposed ANSI C Libraries (Part 1)

Built-in Facilities

Facility	*Cross Reference*
LINE	"C-Ref: DATE , FILE , LINE , TIME , STDC "
FILE	"C-Ref: DATE , FILE , LINE , TIME , STDC "
DATE	"C-Ref: DATE , FILE , LINE , TIME , STDC "
TIME	"C-Ref: DATE , FILE , LINE , TIME , STDC "
STDC	"C-Ref: DATE , FILE , LINE , TIME , STDC "
errno	"C-Ref: **ERRNO, STRERROR, PERROR**"
ptrdiff t	"C-Ref: **NULL, PTRDIFF_T, SIZE_T**"
size t	"C-Ref: **NULL, PTRDIFF_T, SIZE_T**"

Diagnostics assert.h

Facility	*Cross Reference*
assert	"C-Ref: **ASSERT, NDEBUG**"

Character Handling ctype.h

Facility	*Cross Reference*
isalnum	"C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**"
isalpha	"C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**"
iscntrl	"C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**"
isdigit	"C-Ref: **ISDIGIT, ISODIGIT, ISXDIGIT**"
isgraph	"C-Ref: **ISGRAPH, ISPRINT, ISPUNCT**"
islower	"C-Ref: **ISLOWER, ISUPPER**"
isprint	"C-Ref: **ISGRAPH, ISPRINT, ISPUNCT**"
ispunct	"C-Ref: **ISGRAPH, ISPRINT, ISPUNCT**"

`isspace`	"C-Ref: **ISSPACE, ISWHITE**"
`isupper`	"C-Ref: **ISLOWER, ISUPPER**"
`isxdigit`	"C-Ref: **ISDIGIT, ISODIGIT, ISXDIGIT**"
`tolower`	"C-Ref: **TOLOWER, TOUPPER**"
`toupper`	"C-Ref: **TOLOWER, TOUPPER**"

Mathematics `math.h`

Facility	*Cross Reference*
`acos`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`asin`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`atan`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`atan2`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`ceil`	"C-Ref: **CEIL, FLOOR, FMOD**"
`cos`	"C-Ref: **COS, SIN, TAN**"
`cosh`	"C-Ref: **COSH, SINH, TANH**"
`exp`	"C-Ref: **EXP, LOG, LOG10**"
`fabs`	"C-Ref: **ABS, FABS, LABS**"
`floor`	"C-Ref: **CEIL, FLOOR, FMOD**"
`fmod`	"C-Ref: **CEIL, FLOOR, FMOD**"
`frexp`	"C-Ref: **FREXP, LDEXP, MODF**"
`ldexp`	"C-Ref: **FREXP, LDEXP, MODF**"
`log`	"C-Ref: **EXP, LOG, LOG10**"
`log10`	"C-Ref: **EXP, LOG, LOG10**"
`modf`	"C-Ref: **FREXP, LDEXP, MODF**"
`pow`	"C-Ref: **POW, SQRT**"
EDOM	"C-Ref: **ERRNO, STRERROR, PERROR**"
`sin`	"C-Ref: **COS, SIN, TAN**"
`sinh`	"C-Ref: **COSH, SINH, TANH**"
`sqrt`	"C-Ref: **POW, SQRT**"
`tan`	"C-Ref: **COS, SIN, TAN**"
`tanh`	"C-Ref: **COSH, SINH, TANH**"
ERANGE	"C-Ref: **ERRNO, STRERROR, PERROR**"

Nonlocal Jumps `setjmp.h`

Facility	*Cross Reference*
longjmp	"C-Ref: **SETJMP, LONGJMP, JMP_BUF**"
setjmp	"C-Ref: **SETJMP, LONGJMP, JMP_BUF**"

Signal Handling `signal.h`

Facility	*Cross Reference*
raise	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"
signal	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"

Variable Arguments `stdarg.h`

Facility	*Cross Reference*
va arg	"C-Ref: **VARARG, STDARG**"
va end	"C-Ref: **VARARG, STDARG**"
va start	"C-Ref: **VARARG, STDARG**"

12.1.2. C-Ref: Draft Proposed ANSI C Libraries (Part 2)

Input/Output `stdio.h`

Facility	*Cross Reference*
clearerr	"C-Ref: **FEOF, FERROR, CLEARERR**"
fclose	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
feof	"C-Ref: **FEOF, FERROR, CLEARERR**"
ferror	"C-Ref: **FEOF, FERROR, CLEARERR**"
fflush	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
fgetc	"C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"
fgets	"C-Ref: **FGETS, GETS**"
fopen	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
fprintf	"C-Ref: **FPRINTF, PRINTF, SPRINTF**"
fputc	"C-Ref: **FPUTC, PUTC, PUTCHAR**"
fputs	"C-Ref: **FPUTS, PUTS**"
fread	"C-Ref: **FREAD, FWRITE**"
freopen	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
fscanf	"C-Ref: **FSCANF, SCANF, SSCANF**"
fseek	"C-Ref: **FSEEK, FTELL, REWIND**"
ftell	"C-Ref: **FSEEK, FTELL, REWIND**"

`fwrite`	"C-Ref: **FREAD, FWRITE**"
`getc`	"C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"
`getchar`	"C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"
`gets`	"C-Ref: **FGETS, GETS**"
`perror`	"C-Ref: **ERRNO, STRERROR, PERROR**"
`printf`	"C-Ref: **FPRINTF, PRINTF, SPRINTF**"
`putc`	"C-Ref: **FPUTC, PUTC, PUTCHAR**"
`putchar`	"C-Ref: **FPUTC, PUTC, PUTCHAR**"
`puts`	"C-Ref: **FPUTS, PUTS**"
`remove`	"C-Ref: **REMOVE, RENAME**"
`rename`	"C-Ref: **REMOVE, RENAME**"
`rewind`	"C-Ref: **FSEEK, FTELL, REWIND**"
`scanf`	"C-Ref: **FSCANF, SCANF, SSCANF**"
`setbuf`	"C-Ref: **SETBUF, SETVBUF**"
`setvbuf`	"C-Ref: **SETBUF, SETVBUF**"
`sprintf`	"C-Ref: **FPRINTF, PRINTF, SPRINTF**"
`sscanf`	"C-Ref: **FSCANF, SCANF, SSCANF**"
`tmpfile`	"C-Ref: **TMPFILE, TMPNAM, MKTEMP**"
`tmpnam`	"C-Ref: **TMPFILE, TMPNAM, MKTEMP**"
`ungetc`	"C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"
`vfprintf`	"C-Ref: **VFPRINTF, VPRINTF, VSPRINTF**"
`vprintf`	"C-Ref: **VFPRINTF, VPRINTF, VSPRINTF**"
`vsprintf`	"C-Ref: **VFPRINTF, VPRINTF, VSPRINTF**"

General Utilities `stdlib.h`

Facility	*Cross Reference*
`abort`	"C-Ref: **EXIT, ABORT**"
`abs`	"C-Ref: **ABS, FABS, LABS**"
`atof`	"C-Ref: **ATOF, ATOI, ATOL**"
`atoi`	"C-Ref: **ATOF, ATOI, ATOL**"
`atol`	"C-Ref: **ATOF, ATOI, ATOL**"
`bsearch`	"C-Ref: **BSEARCH**"
`calloc`	"C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"

`div`	"C-Ref: **DIV, LDIV**"
`exit`	"C-Ref: **EXIT, ABORT**"
`free`	"C-Ref: **FREE, CFREE**"
`getenv`	"C-Ref: **GETENV, GETLOGIN, GETOPT, PUTENV**"
`labs`	"C-Ref: **ABS, FABS, LABS**"
`ldiv`	"C-Ref: **DIV, LDIV**"
`rand`	"C-Ref: **RAND, SRAND**"
`srand`	"C-Ref: **RAND, SRAND**"
`strtod`	"C-Ref: **STRTOD, STRTOL, STRTOUL**"
`strtol`	"C-Ref: **STRTOD, STRTOL, STRTOUL**"
`strtoul`	"C-Ref: **STRTOD, STRTOL, STRTOUL**"
`malloc`	"C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"
`onexit`	"C-Ref: **ONEXIT, ONEXIT_T**"
`qsort`	"C-Ref: **QSORT**"
`realloc`	"C-Ref: **REALLOC, RELALLOC**"
`system`	"C-Ref: **EXEC, SYSTEM**"

String Handling `string.h`

Facility	*Cross Reference*
`memchr`	"C-Ref: **MEMCHR**"
`memcmp`	"C-Ref: **MEMCMP, BCMP**"
`memcpy`	"C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**"
`memmove`	"C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**"
`memset`	"C-Ref: **MEMSET, BZERO**"
`strcat`	"C-Ref: **STRCAT, STRNCAT**"
`strchr`	"C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**"
`strcmp`	"C-Ref: **STRCMP, STRNCMP**"
`strcpy`	"C-Ref: **STRCPY, STRNCPY**"
`strcspn`	"C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**"
`strerror`	"C-Ref: **ERRNO, STRERROR, PERROR**"
`strlen`	"C-Ref: **STRLEN**"
`strncat`	"C-Ref: **STRCAT, STRNCAT**"
`strncmp`	"C-Ref: **STRCMP, STRNCMP**"

`strncpy`	"C-Ref: **STRCPY, STRNCPY**"
`strpbrk`	"C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**"
`strrchr`	"C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**"
`strspn`	"C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**"
`strstr`	"C-Ref: **STRSTR, STRTOK**"
`strtok`	"C-Ref: **STRSTR, STRTOK**"

Date and Time `time.h`

Facility	*Cross Reference*
`asctime`	"C-Ref: **ASCTIME, CTIME**"
`clock`	"C-Ref: **CLOCK, CLOCK_T, CLK_TCK, TIMES**"
`ctime`	"C-Ref: **ASCTIME, CTIME**"
`difftime`	"C-Ref: **DIFFTIME**"
`gmtime`	"C-Ref: **GMTIME, LOCALTIME, MKTIME**"
`localtime`	"C-Ref: **GMTIME, LOCALTIME, MKTIME**"
`mktime`	"C-Ref: **GMTIME, LOCALTIME, MKTIME**"
`time`	"C-Ref: **TIME, TIME_T**"

13. C-Ref: Standard Language Additions

Name	Section
DATE	"C-Ref: DATE , FILE , LINE , TIME , STDC "
errno	"C-Ref: **ERRNO, STRERROR, PERROR**"
FILE	"C-Ref: DATE , FILE , LINE , TIME , STDC "
NULL	"C-Ref: **NULL, PTRDIFF_T, SIZE_T**"
LINE	"C-Ref: DATE , FILE , LINE , TIME , STDC "
perror	"C-Ref: **ERRNO, STRERROR, PERROR**"
ptrdiff t	"C-Ref: **NULL, PTRDIFF_T, SIZE_T**"
size t	"C-Ref: **NULL, PTRDIFF_T, SIZE_T**"
TIME	"C-Ref: DATE , FILE , LINE , TIME , STDC "
stdarg.h	"C-Ref: **VARARG, STDARG**"
STDC	"C-Ref: DATE , FILE , LINE , TIME , STDC "
varargs.h	"C-Ref: **VARARG, STDARG**"

The facilities of this section are closely tied to the C language. They provide some standard definitions and parameterizations that help make C programs more portable.

13.1. C-Ref: NULL, PTRDIFF_T, SIZE_T

```
#include <stddef.h>                                    /* ANSI */

#define NULL  0

typedef ... ptrdiff t;                                 /* ANSI */

typedef ... size t;                                    /* ANSI */
```

The value of the macro NULL is the traditional null pointer constant.

Many implementations define it to be simply the integer constant 0. In Draft Proposed ANSI C the macro is defined in the header file stddef.h; other systems define it in other library header files, such as stdio.h.

Two type definitions are supplied here in Draft Proposed ANSI C. The type ptrdiff t is an implementation-defined signed integral type that is the type of result of subtracting two pointers; existing implementations use int or long for this type. The type size t is the unsigned integral type of the result of the sizeof operator; existing implementations often used the (signed) type int for this type.

13.2. C-Ref: ERRNO, STRERROR, PERROR

```
#include <stddef.h>                                          /* ANSI */

extern int errno;

#include <string.h>                                          /* ANSI */
char *strerror(errnum);                                      /* ANSI */
  int errnum;

#include <stdio.h>                                           /* ANSI */

void perror(s);
  char *s;

extern int sys nerr;                                /* Non-ANSI form */
extern char *sys errlist[];                         /* Non-ANSI form */

#include <math.h>                                            /* ANSI */
#include <errno.h>                                  /* Non-ANSI form */
#define EDOM ...
#define ERANGE ...
```

The external variable `errno` is used to hold implementation-defined error codes from library routines, traditionally defined in the header file `errno.h`. All error codes are positive integers, and library routines never clear `errno`. Therefore the typical way of using `errno` is to clear it before calling a library function and check it afterward:

```
errno = 0;
x = sqrt(y);
if (errno) {
  printf("?sqrt failed, code %d\n", errno);
  x = 0;
}
```

In Draft Proposed ANSI C `errno` need not be a variable; it can be a macro that expands to any modifiable lvalue, such as a dereferenced pointer returned by a function:

```
extern int * errno();
#define errno (* errno())
```

The function `strerror` returns a pointer to an error message string whose contents are undefined; the string is not modifiable and may be overwritten by a subsequent call to the `strerror` function.

The function `perror` prints on the standard error output:

- the argument string `s`,

- a colon, followed by a space,
- a short message concerning the error whose error code is currently in `errno`, and
- a newline.

The error messages corresponding to values of `errno` may also be stored in a vector of string pointers, `sys errlist`, which can be indexed by the value in `errno`. The variable `sys nerr` contains the maximum integer that can be used to index `sys nerr`; this should be checked to ensure that `errno` does not contain a nonstandard error number.

C implementations generally define a standard list of error codes that can be stored in `errno`. Traditional implementations define them in a central place, such as the header file `errno.h`; others may define them in the individual library header files. Some common ones are:

`EDOM`	An argument was not in the domain accepted by a mathematical function. An example of this is giving a negative argument to the `log` function.
`ERANGE`	The result of a mathematical function is out of range; the function has a well-defined mathematical result but cannot be represented because of the limitations of the implementation's floating-point format. An example of this is trying to use the `pow` function to raise a large number to a very large power.

13.3. C-Ref: `DATE` , `FILE` , `LINE` , `TIME` , `STDC`

```
#define   DATE   ...
#define   FILE   ...
#define   LINE   ...
#define   TIME   ...
#define   STDC   ...
```

These identifiers are special macros built into implementations of Draft Proposal ANSI C; some existing existing implementations also define them, especially `FILE` and `LINE` . These macros cannot be redefined or undefined, and no header file is needed to define them.

`DATE` has as its value a string constant representing the date of translation of the source file, e.g., `"Oct 23 1986"`.

`FILE` has as its value a string constant representing the name of the source file.

`LINE` has as its value a decimal integer constant representing the current line number in the source file, i.e., the one containing the use of the macro `LINE` .

`TIME` has as its value a string constant representing the time of day at which translation occured, e.g., `"14:22:00"`.

`STDC` should be defined (with a nonzero value) only by implementations of Draft Proposed ANSI C.

13.4. C-Ref: VARARG, STDARG

```
#include <varargs.h>                                /* Non-ANSI form */
#include <stdarg.h>                                           /* ANSI */

#define va alist ...                                /* Non-ANSI form */
#define va dcl ...                                  /* Non-ANSI form */

typedef ... va list;

void va start( ap )                                 /* Non-ANSI form */
void va start( ap, LastFixedParm )                            /* ANSI */
  va list ap;
  type LastFixedParm;

type va arg(ap, type);
  va list ap;

void va end(ap);
  va list ap;
```

The `varargs` (or `stdargs`) facility gives programmers a portable way to access variable argument lists, as is needed to implement functions such as `fprintf` (implicitly) and `vfprintf` (explictly).

C traditionally placed no restrictions on the way arguments were passed to functions, and programmers consequently made nonportable assumptions based on the behavior of one computer system. Eventually the `varargs` facility arose under UNIX to promote portability, and Draft Proposal ANSI C has adopted a similar facility under the name `stdarg`. The usage of `stdarg` is slightly different from `vararg` because Draft Proposal ANSI C allows a fixed number of parameters to precede the variable part of an argument list, whereas what we will call traditional implementations force the entire argument list to be treated as variable.

The meanings of the defined macros, functions, and types are listed below. This facility is very stylized so as to make the fewest possible assumptions about the underlying implementation.

`va alist` In traditional C, this macro replaces the parameter list in the definition of a function taking a variable number of arguments. It is not used in Draft Proposed ANSI C.

`va dcl` In traditional C, this macro replaces the parameter declarations in the function definition. It should *not* be followed by a

semicolon, to allow for it to be empty. The macro `va dcl` is not used in Draft Proposed ANSI C.

`va list` — This type is used to declare a local state variable, uniformly called `ap` in this exposition, which is used to traverse the parameters.

`va start` — This function (it is described as a function but it must be implemented as a macro) initializes the state variable `ap`, and must be called before any calls to `va arg` or `va end`. In traditional C, `va start` sets the internal pointer in `ap` to point to the first argument passed to the function; in Draft Proposed ANSI C, `va start` takes an additional parameter—the last fixed parameter name—and sets the internal pointer in `ap` to point to the first variable argument passed to the function.

`va arg` — This macro returns the value of the next parameter in the argument list and advances the internal argument pointer (in `ap`) to the next argument (if any). The type of the next argument (after the usual argument conversions) must be specified (as *type*) so that `va arg` can compute its size on the stack. The first call to `va arg` after calling `va start` will return the value of the first variable parameter.

`va end` — This function or macro should be called after all the arguments have been read with `va arg`. It performs any necessary cleanup operations on `ap` and `va alist`.

As an example, the following "traditional C" function, `printargs`, takes a variable number of arguments of different types and prints their values on the standard output. The first argument to `printargs` is an array of integers that indicates the number and types of the following arguments. The array is terminated by a zero element. The meanings of the integers are given in file `printargs.h`:

```
#define INTARG 1
#define DBLARG 2
/* ... */
```

The `printargs` function itself, including a small test program, is shown in table "C-Ref: Printargs Function in Traditional C". The corresponding example for Draft Proposed ANSI C is shown in table "C-Ref: Printargs Function in Draft Proposed ANSI C". The only differences are in the function argument list and in the call to `va start`.

13.4.1. C-Ref: Printargs Function in Traditional C

```
#include <stdio.h>
#include <varargs.h>
#include ”printargs.h”
void printargs( va alist )
  va dcl
{
  va list ap;
  int argtype, *argtypep;
  va start(ap);
  argtypep = va arg(ap, int *);
  while ( (argtype = *argtypep++) != 0 ) {
    switch (argtype) {
      case INTARG:
            printf(”int: %d\n”, va arg(ap, int) );
            break;
      case DBLARG:
            printf(”double: %f\n”, va arg(ap, double) );
            break;
      /* ... */
    }
  }
  va end(ap);
  return;
}

#ifdef TEST
int at[] = { INTARG, DBLARG, INTARG, DBLARG, 0 };
int main()
{
  printargs( &at[0], 1, 2.0, 3, 4.0 );
  return 0;
}
#endif
```

13.4.2. C-Ref: Printargs Function in Draft Proposed ANSI C

```
#include <stdio.h>
#include <stdarg.h>
#include "printargs.h"
void printargs( int *argtypep, ... )
{
  va list ap;
  int argtype;
  va start(ap, argtypep);
  while ( (argtype = *argtypep++) != 0 ) {
    switch (argtype) {
      case INTARG:
             printf("int: %d\n", va arg(ap, int) );
             break;
      case DBLARG:
             printf("double: %f\n", va arg(ap, double) );
             break;
      /* ... */
    }
  }
  va end(ap);
  return;
}
```

14. C-Ref: Character Processing

Name	Section
isalnum	"C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**"
isalpha	"C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**"
isascii	"C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**"
iscntrl	"C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**"
iscsym	"C-Ref: **ISCSYM, ISCSYMF**"
iscsymf	"C-Ref: **ISCSYM, ISCSYMF**"
isdigit	"C-Ref: **ISDIGIT, ISODIGIT, ISXDIGIT**"
isgraph	"C-Ref: **ISGRAPH, ISPRINT, ISPUNCT**"
islower	"C-Ref: **ISLOWER, ISUPPER**"
isodigit	"C-Ref: **ISDIGIT, ISODIGIT, ISXDIGIT**"
isprint	"C-Ref: **ISGRAPH, ISPRINT, ISPUNCT**"
ispunct	"C-Ref: **ISGRAPH, ISPRINT, ISPUNCT**"
isspace	"C-Ref: **ISSPACE, ISWHITE**"
isupper	"C-Ref: **ISLOWER, ISUPPER**"
iswhite	"C-Ref: **ISSPACE, ISWHITE**"
isxdigit	"C-Ref: **ISDIGIT, ISODIGIT, ISXDIGIT**"
toascii	"C-Ref: **TOASCII**"
toint	"C-Ref: **TOINT**"
tolower	"C-Ref: **TOLOWER, TOUPPER**"
tolower	"C-Ref: **TOLOWER, TOUPPER**"
toupper	"C-Ref: **TOLOWER, TOUPPER**"
toupper	"C-Ref: **TOLOWER, TOUPPER**"

The facilities for handling characters are of two kinds: classification and conversion. Every character classification facility has a name beginning with "is" and returns a value of type int that is nonzero if the argument is in the specified class and zero if not. Every character conversion facility has a name beginning with "to" and returns a value of type int representing a character or EOF.

The value EOF (-1) is conventionally used as a value that is "not a real character." For example, fgetc (section "C-Ref: **FGETC, GETC, GETCHAR, UNGETC**") returns EOF when at end-of-file, because there is no "real character" to be read. It must be remembered, however, that the type char may be signed in some implementations, and so EOF is not necessarily distinguishable from a "real character" if

nonstandard character values appear. (Standard character values are always non-negative, even if the type char is signed.) All of the facilities described here operate properly on all values representable as type char or type unsigned char, and also on the value EOF, but are undefined for all other integer values, unless the individual description states otherwise.

The facilities isascii and toascii assume that the standard 128-character ASCII set is used as the implementation's run-time character set, but the rest do not require this assumption and indeed serve to insulate code from the implementation's run-time character set.

A warning: Some implementations of C let the type char be signed and also support a type unsigned char, yet the character-handling facilities fail to operate properly on all values representable by type unsigned char. In some cases the facilities even fail to operate properly on all values representable by type char, but handle only "standard" character values and EOF.

All of the facilities described here are declared by the library header file ctype.h.

14.1. C-Ref: ISALNUM, ISALPHA, ISASCII, ISCNTRL

```
#include <ctype.h>

int isalnum(c)
  char c;

int isalpha(c)
  char c;

int isascii(c)
  int c;

int iscntrl(c)
  char c;
```

The isalnum function returns a nonzero value if c is the code for an alphanumeric character; that is, one of the following:

```
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Otherwise the returned value is zero.

The isalpha function returns a nonzero value if c is the code for an alphabetic character; that is, one of the following:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Otherwise the returned value is zero.

The function `isascii` returns a nonzero value if the value of `c` is between `0` and `0177` (the range of the standard 128-character ASCII character set). Otherwise `isascii` returns zero. Unlike most of of the character classification functions, `isascii` operates properly on any value of type `int`.

The function `iscntrl` returns a nonzero value if `c` is the code for a "control character"; that is, any character that is not a printing character. If the standard 128-character ASCII set is in use, the control characters are those with codes `000` through `037`, and also code `0177`. The `isprint` function is the complementary function, at least for standard ASCII implementations.

14.2. C-Ref: ISCSYM, ISCSYMF

```
#include <ctype.h>
int iscsym(c)                                   /* Berkeley UNIX only */
  char c;

int iscsymf(c)                                  /* Berkeley UNIX only */
  char c;
```

The `iscsym` function returns a nonzero value if `c` is the code for a character that may appear in a C identifier. `iscsymf` returns a nonzero value if `c` is the code for a character that may additionally appear as the first character of an identifier.

The `iscsymf` function will accept at least the 52 upper- and lower-case letters, and the underscore character. `iscsym` will additionally accept at least the ten decimal digits. Other characters may be accepted by these functions as well, depending on the implementation.

14.3. C-Ref: ISDIGIT, ISODIGIT, ISXDIGIT

```
#include <ctype.h>
int isdigit(c)
  char c;

int isodigit(c)                                      /* Berkeley UNIX */
  char c;

int isxdigit(c)
  char c;
```

The `isdigit` function returns a nonzero value if `c` is the code for one of the ten decimal digits. The `isodigit` function returns a nonzero value if `c` is the code for one of the eight octal digits. The `isxdigit` function returns a nonzero value if `c` is the code for one of the 22 hexadecimal digits; that is, one of the following:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
```

14.4. C-Ref: ISGRAPH, ISPRINT, ISPUNCT

```
#include <ctype.h>
int isgraph(c)
  char c;

int isprint(c)
  char c;

int ispunct(c)
  char c;
```

The `isprint` function returns a nonzero value if `c` is the code for a "printing character"; that is, any character that is not a control character. The `isgraph` function returns a nonzero value if `c` is the code for a "graphic character"; that is, any printing character other than space. The `isprint` and `isgraph` functions differ only in how they handle the space character; `isprint` is the opposite of `iscntrl`.

If the standard 128-character ASCII set is in use, the printing characters are those with codes `040` through `0176`; that is, space plus the following:

```
  ! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ ()
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~
```

The graphic characters are the same but space is omitted.

The function `ispunct` returns a nonzero value if `c` is the code for a "punctuation character"; that is, neither a control character nor an alphanumeric character. If the standard 128-character ASCII character set is in use, the punctuation characters are space plus the following:

```
! " # $ % & ' ( ) * + , - . / : ; < = > ?
— [ \ ] ^   ` { | } ~
```

14.5. C-Ref: ISLOWER, ISUPPER

```
#include <ctype.h>
int islower(c)
  char c;

int isupper(c)
  char c;
```

The `islower` function returns a nonzero value if `c` is the code for one of the 26 lower-case letters. The `isupper` function returns a nonzero value if `c` is the code for one of the 26 upper-case letters. Otherwise the returned values are zero.

14.6. C-Ref: ISSPACE, ISWHITE

```
#include <ctype.h>
int isspace(c)
  char c;

int iswhite(c)                                          /* rare */
  char c;
```

The isspace function returns a nonzero value if c is the code for a whitespace character; that is, a space, tab, carriage return, newline, vertical tab, or form feed. If the standard 128-character ASCII set is in use, the whitespace characters are those with codes 011 through 015 and 040. Some implementations of C provide not this exact function but a variant called iswhite.

14.7. C-Ref: TOASCII

```
#include <ctype.h>
int toascii(c)
  int c;
```

The toascii function accepts any integer value and reduces it to the range of valid ASCII characters (codes 0 through 0177) by discarding all but the low-order seven bits of the value. If the argument is already a valid ASCII code, then the result is equal to the argument.

14.8. C-Ref: TOINT

```
#include <ctype.h>
int toint(c)                                            /* rare */
  char c;
```

The toint function returns the "weight" of a hexadecimal digit: 0 through 9 for the characters '0' through '9', respectively, and 10 through 15 for the letters 'a' through 'f' (or 'A' through 'F'), respectively.

14.9. C-Ref: TOLOWER, TOUPPER

```
#include <ctype.h>
int tolower(c)
  char c;

int toupper(c)
  char c;
```

If `c` is an upper-case letter, then `tolower` returns the corresponding lower-case letter. If `c` is a lower-case letter, then `toupper` returns the corresponding upper-case letter.

You should be wary of the value returned by `tolower` when its argument is not an upper-case letter, nor on the value returned by `toupper` when its argument is not a lower-case letter. Draft Proposed ANSI C specifies that other arguments should be returned unchanged, but many current implementations work correctly only when the argument is a letter of the proper case.

Implementations that allow more general arguments to `tolower` and `toupper` may provide faster versions of these— `tolower` and `toupper`—which require the more restrictive arguments and which are correspondingly faster.

15. C-Ref: String Processing

Name	Section
`atof`	"C-Ref: **ATOF, ATOI, ATOL**"
`atoi`	"C-Ref: **ATOF, ATOI, ATOL**"
`atol`	"C-Ref: **ATOF, ATOI, ATOL**"
`strcat`	"C-Ref: **STRCAT, STRNCAT**"
`strchr`	"C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**"
`strcmp`	"C-Ref: **STRCMP, STRNCMP**"
`strcpy`	"C-Ref: **STRCPY, STRNCPY**"
`strcspn`	"C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**"
`strlen`	"C-Ref: **STRLEN**"
`strncat`	"C-Ref: **STRCAT, STRNCAT**"
`strncmp`	"C-Ref: **STRCMP, STRNCMP**"
`strncpy`	"C-Ref: **STRCPY, STRNCPY**"
`strpbrk`	"C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**"
`strpos`	"C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**"
`strrchr`	"C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**"
`strrpbrk`	"C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**"
`strrpos`	"C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**"
`strspn`	"C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**"
`strtod`	"C-Ref: **STRTOD, STRTOL, STRTOUL**"
`strtol`	"C-Ref: **STRTOD, STRTOL, STRTOUL**"
`strtoul`	"C-Ref: **STRTOD, STRTOL, STRTOUL**"

By convention, strings in C are of variable length and are terminated by a null character (that is, `'\0'`). The compiler automatically supplies an extra null character after all string constants, but it is up to the programmer to make sure that strings created in string variables (that is, character arrays) end with a null character.

All the characters in a string, not counting the terminating null character, are together called the *contents* of the string. An empty string contains no characters and is represented by a pointer to a null character. Note that this is *not* the same as a null character pointer (`NULL`), which is a pointer that points to no character at all. When we speak of a "pointer to the first character of a string," we mean a pointer to the terminating null character if the string is empty and to the first character of the contents if the string is not empty.

All of the string-handling facilities described here assume that strings are terminated by a null character. When characters are transferred to a destination string, no test is made for overflow of the destination. It is up to the programmer to make sure that the destination area in memory is large enough to contain the result string, including the terminating null character.

All of the facilities described here are declared by the library header file `string.h`. In Draft Proposed ANSI C, string parameters that are not modified are generally declared to have type `const char *` instead of `char *`; integer arguments or return values that represent the length of a string have type `size t` instead of `int`. For brevity, we do not shown both forms of the functions.

See also the facilities provided by the memory functions (chapter "C-Ref: Memory Functions"), `sprintf` (section "C-Ref: **FPRINTF, PRINTF, SPRINTF**"), and `sscanf` (section "C-Ref: **FSCANF, SCANF, SSCANF**").

15.1. C-Ref: STRCAT, STRNCAT

```
#include <string.h>

char *strcat(s1, s2)
  char *s1, *s2;

char *strncat(s1, s2, n)
  char *s1, *s2;
  int n;
```

The function `strcat` appends the contents of the string `s2` to the end of the string `s1`. The value of `s1` is returned. The null character that terminates `s1` (and perhaps other characters following it in memory) is overwritten with characters from `s2` and a new terminating null character. Characters are copied from `s2` until a null character is encountered in `s2`. The memory area beginning with `s1` is assumed to be large enough to hold both strings.

The `strncat` function appends up to `n` characters from the contents of `s2` to the end of `s1`. If the null character that terminates `s2` is encountered before `n` characters have been copied, then the null character is copied, but no more characters after that are copied. If no null character appears among the first `n` characters of `s2`, then the net effect is that the first `n` characters are copied and then a null character is supplied to terminate the destination string; that is, `n+1` characters in all are written, the first replacing the null character that formerly terminated `s1`. If the value of `n` is zero or negative, then calling this function has no effect.

The results of both functions are unpredictable if the two string arguments overlap in memory.

15.2. C-Ref: STRCMP, STRNCMP

```
#include <string.h>

int strcmp(s1, s2)
  char *s1, *s2;

int strncmp(s1, s2, n)
  char *s1, *s2;
  int n;
```

The function strcmp lexicographically compares the contents of the null-terminated string s1 with the contents of the null-terminated string s2. It returns a value of type int that is less than zero if s1 is less than s2; equal to zero if s1 is equal to s2; and greater than zero if s1 is greater than s2. Therefore, to check only if two strings are equal the programmer negates the return value from strcmp:

```
if (!strcmp(s1,s2)) printf("Strings are equal\n");
else printf("Strings are not equal\n");
```

Two strings are equal if their contents are identical. String s1 is lexicographically less than string s2 under either of two circumstances:

1. The contents of the strings are equal up to some character position, and at that first differing character position the character value from s1 is less than the character value from s2.

2. The string s1 is shorter than the string s2, and the contents of s1 are identical to those of s2 up to length of s1.

The function strncmp is like strcmp except that it compares up to n characters of the null-terminated string s1 with up to n characters of the null-terminated string s2. In comparing the strings, the entire string is used if it contains fewer than n characters, and otherwise the string is treated as if it were n characters long. If the value of n is zero or negative, then both strings are treated as empty and therefore equal, and zero is returned.

The function memcmp (section "C-Ref: **MEMCMP, BCMP**") provides similar functionality to strcmp.

15.3. C-Ref: STRCPY, STRNCPY

```
#include <string.h>

char *strcpy(s1, s2)
  char *s1, *s2;
```

```
char *strncpy(s1, s2, n)
  char *s1, *s2;
  int n;
```

The function strcpy copies the contents of the string s2 to the string s1, overwriting the old contents of s1. The entire contents of s2 are copied, plus the terminating null character, even if s2 is longer than s1. A pointer to the first character of s1 is returned.

The function strncpy copies exactly n characters to s1. It first copies up to n characters from s2. If there are fewer than n characters in s2 before the terminating null character, then null characters are written into s1 as padding until exactly n characters have been written. If there are n or more characters in s2, then only n characters are copied, and so only a truncated copy of s2 is transferred to s1. It follows that the copy in s1 is terminated with a null by strncpy only if the length of s2 (not counting the terminating null) is less than n. (This is true even in Draft Proposed ANSI C.) If the value of n is zero or negative, then calling strncpy function has no effect.

The functions memcpy and memccpy (section "C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**") provide similar functionality to strcpy. The results of both strcpy and strncpy are unpredictable if the two string arguments overlap in memory; the function memmove (section "C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**") is provided in Draft Proposed ANSI C for cases in which overlap may occur.

15.4. C-Ref: STRLEN

```
#include <string.h>

int strlen( string )
  char *string;
```

The function strlen returns the number of characters in s preceding the terminating null character. An empty string has a null character as its first character and therefore the length of an empty string is zero. In Draft Proposed ANSI C the return type of strlen is size t and the argument has type const char *. In some implementations of C this function is called lenstr.

15.5. C-Ref: STRCHR, STRPOS, STRRCHR, STRRPOS

```
#include <string.h>
char *strchr(s, c)
  char *s, c;
```

```
int strpos(s, c)                                  /* Non-ANSI form */
  char *s, c;

char *strrchr(s, c)
  char *s, c;

int strrpos( s, c )                               /* Non-ANSI form */
  char *s, c;
```

The functions in this section all search for a single character `c` within a null-terminated string `s`. In Draft Proposed ANSI C `c` has type `int` and `s` has type `const char *`.

The function `strchr` searches the string `s` for the first occurrence of the character `c`. If the character `c` is found in the string, a pointer to the first occurrence is returned. If the character is not found, a null pointer is returned. The terminating null character is considered to be a part of `s` for the purposes of this search, so searching for a null character will return a pointer to the null character, not a null pointer.

The function `strpos` is like `strchr` except that the position of the first occurrence of `c` is returned (where the first character of `s` is considered to be at position `0`). If the character is not found, the value `-1` is returned. Searching for a null character will return the position of the terminating null character (that is, the length of the string), not the value `-1`.

The function `strrchr` is like `strchr` except that it returns a pointer to the last occurrence of the character `c`. If the character is not found, a null pointer is returned. Searching for a null character will return a pointer to the terminating null character, not a null pointer.

The function `strrpos` is like `strrchr` except that the position of the last occurrence of `c` is returned (where the first character of `s` is considered to be at position `0`). If the character is not found, the value `-1` is returned. Searching for a null character will return the position of the terminating null character (that is, the length of the string), not the value `-1`.

The function `memchr` (section "C-Ref: **MEMCHR**") provides similar functionality to `strchr`. In some implementations of C `strchr` and `strrchr` are called `index` and `rindex`, respectively. Some implementations of C provide the function `scnstr`, which is a variant of `strpos`.

15.6. C-Ref: STRSPN, STRCSPN, STRPBRK, STRRPBRK

```
#include <string.h>

int strspn(s, set)
  char *s, *set;
```

```
int strcspn(s, set)
  char *s, *set;

char *strpbrk(s, set)
  char *s, *set;

char *strrpbrk(s, set)
  char *s, *set;
```

The functions in this section all search a null-terminated string `s` for occurrences of characters specified by whether or not they are included in a second null-terminated string `set`. The second argument is regarded as a set of characters; the order of the characters, or whether there are duplications, does not matter. In Draft Proposed ANSI C both `s` and `set` have type `const char *`, and the return value of `strspn` and `strcspn` has type `size t`.

The function `strspn` searches the string `s` for the first occurrence of a character that is not included in the string `set`, skipping over ("spanning") characters that are in `set`. The value returned is the length of the longest initial segment of `s` that consists of characters found in `set`. If every character of `s` appears in `set`, then the total length of `s` (not counting the terminating null character) is returned. If `set` is an empty string, then the first character of `s` will not be found in it, and so zero will be returned.

The function `strcspn` is like `strspn` except that it searches `s` for the first occurrence of a character that is included in the character set `set`, skipping over characters that are not in `set`.

The function `strpbrk` is like `strcspn` except that it returns a pointer to the first character found from `set` rather than the number of characters skipped over. If no characters from `set` are found, a null pointer is returned.

The function `strrpbrk` is like `strpbrk` except that it returns a pointer to the *last* character from `set` found within `s`. If no character within `s` occurs in `set`, then a null pointer is returned.

In some implementations of C the functions `strspn` and `strcspn` are called `notstr` and `instr`.

15.7. C-Ref: STRSTR, STRTOK

```
#include <string.h>

char *strstr( src, sub )                                   /* ANSI */
  const char * src,* sub;

char *strtok( str, set)
  char *str,*set;
```

The function strstr is new in Draft Proposed ANSI C. It locates the first occurrence of the string sub in the string src and returns a pointer to the beginning of the first occurrence. If sub does not occur in src, a null pointer is returned.

The function strtok may be used to separate a string str into tokens separated by characters from the string set. A call is made on strtok for each token, possibly changing the value of set in successive calls. The first call includes the string str; subsequent calls pass a null pointer as the first argument, directing strtok to continue from the end of the previous token.

More precisely, if str is not null then strtok first skips over all characters in str that are also in set. If all the characters of str occur in set then strtok returns a null pointer and an *internal state pointer* is set to a null pointer. Otherwise, the internal state pointer is set to point to the first character of str not in set and execution continues as if str had been null.

If str is null and the internal state pointer is null then strtok returns a null pointer and the internal state pointer is unchanged. If str is null but the internal state pointer is not null, then the function searches beginning at the internal state pointer for the first character contained in set. If such a character is found, the character is overwritten with '\0', strtok returns the value of the internal state pointer, and the internal state pointer is adjusted to point to the character immediately following inserted null character. If no such character is found, strtok returns the value of the internal state pointer and the internal state pointer is set to null.

15.8. C-Ref: STRTOD, STRTOL, STRTOUL

```
#include <stdlib.h>                                    /* ANSI */

double strtod( str, ptr )
  char *str, **ptr;

long strtol( str, ptr, base )
  char *str, **ptr;
  int base;

unsigned long strtoul( str, ptr, base )                 /* ANSI */
  char *str, **ptr;
  int base;
```

These functions, found in Draft Proposed ANSI C and System V UNIX, convert strings to numbers. In each case, str points to the string to be converted and ptr (if not NULL) is set by the functions to point to the first character in str immediately following the converted part of the string. If str begins with whitespace characters (as defined by the isspace function) they are skipped.

The function strtod expects the number to be converted to consist of:

1. an optional plus or minus sign,
2. a sequence of decimal digits possibly containing a single decimal point, and
3. an optional exponent part, consisting of the letter `e` or `E`, an optional sign, and a sequence of decimal digits.

The longest sequence of characters matching this model is converted to a floating-point number of type `double`, which is returned. Stated a different way, `strtod` accepts numbers beginning with an optional sign and then having the syntax of *decimal-constant*, *octal-constant*, or *floating-constant*. Numbers having the format of *octal-constant* are still treated as decimal numbers, and in no case are trailing *type-markers* recognized as part of the number.

If no conversion is possible because the string does not match the expected number model (or is empty), zero is returned, `ptr` (if not `NULL`) is set to the value of `str`, and `errno` is set to `ERANGE`. If the number converted would cause overflow, `HUGE VAL` (with the correct sign) is returned; if the number converted would cause underflow, zero is returned. In either case, `errno` is set to `ERANGE`. (According to this definition, an illegal number is indistinguishable from one that causes underflow, except perhaps by the value set in `*ptr`. Some implementations may set `errno` to `EDOM` when the string does not match the number model.)

The functions `strtol` and `strtoul` convert the initial portion of the argument string to an integer of type `long int` or `unsigned long int`, respectively. The expected format of the number—which changes with the value of `base`, the expected radix—is the same in both cases except that an optional sign may preceed the number in the case of `strtol` but not in the case of `strtoul`.

If `base` is zero, the number should have the format of a *decimal-constant*, *octal-constant*, or *hexadecimal-constant*. The number's radix is deduced from its format. If the value of `base` is between 2 and 36, inclusive, the number must consist of a nonzero sequence of letters and digits representing an integer in the specified base. The letters `a` through `z` (or `A` through `Z`) represent the values `10` through `36`, respectively. Only those letters representing values less than `base` are permitted. (As a special case, if `base` is 16 then the number may begin with `0x` or `0X`, which is ignored.)

If no conversion can be performed, the functions return zero, `ptr` (if not `NULL`) is set to the value of `str`, and `errno` is set to `ERANGE`. If the number to be converted would cause an overflow, then `strtol` returns `LONG MAX` or `LONG MIN` (depending on the sign of the result in the case of `strtol`) or `ULONG MAX` (in the case of `strtoul`) and `errno` is set to ERANGE.

15.9. C-Ref: ATOF, ATOI, ATOL

```
#include <stdlib.h>                                              /* ANSI */

double atof( str )
  char *str;

int atoi( str )
  char *str;

long atol( str )
  char *str;
```

These functions, which convert strings to numbers, are found in many UNIX implementations. In Draft Proposed ANSI C they are present for compatibility but the functions `strtod`, `strtol`, and `strtoul` are preferred. They may be defined in terms of the more general functions:

```
extern double strtod();
extern long   strtol();
extern unsigned long strtoul();

double atof(str)
  char *str;
{
  return strtod(str, (char **)NULL);
}

int atoi(str)
  char *str;
{
  return (int) strtol(str, (char **) NULL, 10);
}

long atol(str)
  char *str;
{
  return strtol(str, (char **) NULL, 10);
}
```

16. C-Ref: Memory Functions

Name	Section
bcmp	"C-Ref: **MEMCMP, BCMP**"
bcpy	"C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**"
bzero	"C-Ref: **MEMSET, BZERO**"
memchr	"C-Ref: **MEMCHR**"
memcmp	"C-Ref: **MEMCMP, BCMP**"
memcpy	"C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**"
memccpy	"C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**"
memmove	"C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**"
memset	"C-Ref: **MEMSET, BZERO**"

The facilities in this section give the C programmer efficient ways to copy, compare, and set blocks of memory. In Draft Proposed ANSI C these functions are considered part of the string functions and are declared in the library header file `string.h`. In many other implementations they are declared in the header file `memory.h`.

16.1. C-Ref: MEMCHR

```
#include <string.h>                                  /* ANSI */
#include <memory.h>                         /* Non-ANSI form */

char *memchr( ptr, val, len )
  char *ptr;
  int val, len;
```

The function `memchr` searches for the value `val` in each of the first `len` characters beginning at `ptr`. It returns a pointer to the first character containing `val`, if any, or returns a null pointer if no such character is found. In Draft Proposed ANSI C, `memchr` has return type `void *`, its first argument has type `const void *`, and the argument `len` has type `size t`. See also `strchr` (section "C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**").

16.2. C-Ref: MEMCMP, BCMP

```
#include <string.h>                                      /* ANSI */
#include <memory.h>                              /* Non-ANSI form */

int memcmp( ptr1, ptr2, len )
  char *ptr1, *ptr2;
  int len;

int bcmp( ptr1, ptr2, len )                      /* Berkeley UNIX */
  char *ptr1, *ptr2;
  int len;
```

The function `memcmp` compares the first `len` characters beginning at `ptr1` with the first `len` character beginning at `ptr2`. If the first string of characters is lexicographically less than the second, `memcmp` returns a negative integer. If the first string of characters is lexicographically greater than the second, `memcmp` returns a positive integer. Otherwise `memcmp` returns 0. In Draft Proposed ANSI C, the arguments `ptr1` and `ptr2` have type `void *`, and the argument `len` has type `size t`.

The function `bcmp`, found on some implementations, also compares two strings of characters, but returns 0 if they are the same and nonzero otherwise. No comparison for less or greater is made. See also `strcmp` (section "C-Ref: **STRCMP, STRNCMP**").

16.3. C-Ref: MEMCPY, MEMCCPY, MEMMOVE, BCPY

```
#include <string.h>                                      /* ANSI */
#include <memory.h>                              /* Non-ANSI form */

char *memcpy( dest, src, len )
  char *dest, *src;
  int len;

char * memccpy(dest, src, val, len)              /* Non-ANSI form */
  char *dest, *src;
  int val, len;

void * memmove(dest, src, len)                           /* ANSI */
  void *dest;
  const void *src;
  size t len;

char *bcopy( src, dest, len )
  char *dest, *src;
  int len;
```

The function memcpy copies len characters from src to dest and returns the value of src. If the source and destination areas overlap, the results will be unpredictable. In Draft Proposed ANSI C, memcpy has type void *, its first argument has type const void *, and the argument len has type size t.

The function memmove, found in Draft Proposed ANSI C, differs from memcpy only in that it is guaranteed to work for overlapping memory regions. That is, the effect is as if the source area were first copied to a separate temporary area and then copied back to the destination area. (Some implementations define memcpy to have these semantics.)

The function memccpy, found in some implementations, also copies len characters from src to dest, but it will stop immediately after copying a character whose value is val. When all len characters are copied memccpy returns a null pointer; otherwise it returns a pointer to the character following the copy of val in dest.

The function bcopy, found on some implementations, works like memcpy but the source and destination operands are reversed. See also strcpy (section "C-Ref: **STRCPY, STRNCPY**").

16.4. C-Ref: MEMSET, BZERO

```
#include <string.h>                                    /* ANSI */
#include <memory.h>                           /* Non-ANSI form */

char *memset( ptr, val, len )
  char *ptr;
  int val, len;

void bzero( ptr, len )                      /* Berkeley UNIX */
  char *ptr;
  int len;
```

The function memset copies the value val into each of len characters beginning at ptr. It returns the value of ptr. In Draft Proposed ANSI C, memset has type void *, its first argument has type const void *, and the argument len has type size t.

The more restricted function bzero copies the value 0 into each of len characters beginning at ptr; it is found in some UNIX implementations.

17. C-Ref: Input/Output Facilities

Name	Section
`clearerr`	"C-Ref: **FEOF, FERROR, CLEARERR**"
`feof`	"C-Ref: **FEOF, FERROR, CLEARERR**"
`ferror`	"C-Ref: **FEOF, FERROR, CLEARERR**"
`fflush`	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
`fgetc`	"C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"
`fgets`	"C-Ref: **FGETS, GETS**"
`fclose`	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
`fopen`	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
`fputc`	"C-Ref: **FPUTC, PUTC, PUTCHAR**"
`fputs`	"C-Ref: **FPUTS, PUTS**"
`fprintf`	"C-Ref: **FPRINTF, PRINTF, SPRINTF**"
`fread`	"C-Ref: **FREAD, FWRITE**"
`freopen`	"C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"
`fscanf`	"C-Ref: **FSCANF, SCANF, SSCANF**"
`fseek`	"C-Ref: **FSEEK, FTELL, REWIND**"
`ftell`	"C-Ref: **FSEEK, FTELL, REWIND**"
`fwrite`	"C-Ref: **FREAD, FWRITE**"
`getc`	"C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"
`getchar`	"C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"
`gets`	"C-Ref: **FGETS, GETS**"
`mktemp`	"C-Ref: **TMPFILE, TMPNAM, MKTEMP**"
`perror`	"C-Ref: **ERRNO, STRERROR, PERROR**"
`printf`	"C-Ref: **FPRINTF, PRINTF, SPRINTF**"
`putc`	"C-Ref: **FPUTC, PUTC, PUTCHAR**"
`putchar`	"C-Ref: **FPUTC, PUTC, PUTCHAR**"
`puts`	"C-Ref: **FPUTS, PUTS**"
`remove`	"C-Ref: **REMOVE, RENAME**"
`rename`	"C-Ref: **REMOVE, RENAME**"
`rewind`	"C-Ref: **FSEEK, FTELL, REWIND**"
`scanf`	"C-Ref: **FSCANF, SCANF, SSCANF**"

`setbuf` "C-Ref: **SETBUF, SETVBUF**"

`setvbuf` "C-Ref: **SETBUF, SETVBUF**"

`sprintf` "C-Ref: **FPRINTF, PRINTF, SPRINTF**"

`sscanf` "C-Ref: **FSCANF, SCANF, SSCANF**"

`stderr` "C-Ref: **STDIN, STDOUT, STDERR**"

`stdin` "C-Ref: **STDIN, STDOUT, STDERR**"

`stdout` "C-Ref: **STDIN, STDOUT, STDERR**"

`tmpfile` "C-Ref: **TMPFILE, TMPNAM, MKTEMP**"

`tmpnam` "C-Ref: **TMPFILE, TMPNAM, MKTEMP**"

`ungetc` "C-Ref: **FGETC, GETC, GETCHAR, UNGETC**"

`vfprintf` "C-Ref: **VFPRINTF, VPRINTF, VSPRINTF**"

`vprintf` "C-Ref: **VFPRINTF, VPRINTF, VSPRINTF**"

`vsprintf` "C-Ref: **VFPRINTF, VPRINTF, VSPRINTF**"

C has a very rich and useful set of I/O facilities based on the concept of a *stream*, which may be a file or some other source or consumer of data, including a terminal or other physical device. The data type `FILE` (defined in `stdio.h` along with the rest of the I/O facilities) holds information about a stream. A *file pointer*, an object of type `FILE *`, is created by calling `fopen` and is used as an argument to most of the I/O facilities described in this chapter.

Among the information included in a `FILE` object is the current position within the stream, pointers to any associated buffers, and indications whether an error or end-of-file has occurred. Streams are normally buffered unless they are associated with physical devices. The programmer has some control over buffering with the `setvbuf` facility, but in general streams can be implemented very efficiently and the programmer should not have to worry about performance.

There are two general forms of streams: text and binary. A text stream consists of a sequence of characters divided into lines; each line consists of zero or more characters followed by (and including) a newline character '`\n`'. Text streams are portable when they consist only of complete lines made from characters from the standard character set. The hardware and software components underlying a particular C run-time library implementation may have different representations for text files (especially for the end-of-line indication) but the run-time library must map those representations into the standard one. Draft Proposed ANSI C requires implementations to support text stream lines of at least 254 characters including the terminating newline.

Binary streams are sequences of data values of type `char`. Because any C data value may be mapped onto an array of values of type `char`, binary streams can transparently record internal data.

17.1. C-Ref: EOF

```
#include <stdio.h>

#define EOF (-1)
```

The value EOF is conventionally used as a value that is "not a real character." For example, fgetc (section "C-Ref: **FGETC, GETC, GETCHAR, UNGETC**") returns EOF when at end-of-file, because there is no "real character" to be read. It must be remembered, however, that the type char may be signed in some implementations, and so EOF is not necessarily distinguishable from a "real character" if nonstandard character values appear. (Standard character values are always nonnegative, however, even if the type char is signed.) When the value EOF is read, it is best to use the feof facility (section "C-Ref: **FEOF, FERROR, CLEARERR**") to determine whether end-of-file has indeed been encountered.

17.2. C-Ref: FOPEN, FCLOSE, FFLUSH, FREOPEN

```
#include <stdio.h>

FILE *fopen(filename, type)
  char *filename, *type;

int fclose(stream)
  FILE *stream;

int fflush(stream)
  FILE *stream;

FILE *freopen(filename, type, stream)
  char *filename, *type;
  FILE *stream;
```

The function fopen takes as arguments a file name and a type; each is specified as a character string. The file name is used in an implementation-specified manner to identify or create a file. A stream is associated with the file in a manner indicated by the type argument. A pointer of type FILE * is returned to identify the stream for other input/output operations. If any error is detected, fopen stores an error code into errno and returns a null pointer. The number of streams that may be open simultaneously is not specified; in Draft Proposal ANSI C it is given by the value of the macro SYS OPEN, which must be at least eight.

The function fclose takes a stream, which should be open for input or output. The stream is closed in an appropriate and orderly fashion, including any necessary emptying and freeing of internal data buffers. The function fclose returns EOF if an error is detected; otherwise, it returns zero.

The function fflush takes an output stream as its argument and causes any internal buffers to be emptied to the destination device. The stream remains open. If any error is detected, fflush returns EOF; otherwise, it returns 0. fflush is typically used only in exceptional circumstances; fclose and exit normally may be relied upon to take care of flushing output buffers.

The function freopen takes a file name, a type, and an open stream. It first tries to close stream as if by a call to fclose, but any error while doing so is ignored. Then, filename and type are used to open a new file as if by a call to fopen, except that the new stream is associated with stream rather than getting a new value of type FILE *. The function freopen returns stream if it is successful; otherwise (if the new open fails) a null pointer is returned. One of the main uses of freopen is to reassociate one of the standard input/output streams stdin, stdout, and stderr with another file.

The following values are permitted for the type specification in fopen and freopen:

"r"	Open an existing file for input.
"w"	Create a new file, or truncate an existing one, for output.
"a"	Create a new file, or append to an existing one, for output.
"r+"	Open an existing file for update (both reading and writing), starting at the beginning of the file.
"w+"	Create a new file, or truncate an existing one, for update.
"a+"	Create a new file, or append to an existing one, for update.

When a file is opened for update ('+' is present in the type string), the resulting stream may be used for both input and output. However, an output operation may not be followed by an input operation without an intervening call to fseek, rewind, or fflush and an input operation may not be followed by an output operation without an intervening call to fseek, rewind, or fflush or an input operation that encounters end-of-file.

Draft Proposed ANSI C allows any of the types listed above to be followed by the character b to indicate a "binary" (as opposed to "text") stream is to be created. (The distinction under UNIX was blurred because both kinds of files are handled the same; other operating systems are not so lucky.) Draft Proposed ANSI C also allows any of the "update" file types to assume binary mode.

17.3. C-Ref: SETBUF, SETVBUF

```
#include <stdio.h>
```

```
int setvbuf( stream, buf, type, size )
  FILE *stream;
  char *buf;
  int type, size;

void setbuf( stream, buf )
  FILE *stream;
  char *buf;

/* Buffer size */
#define BUFSIZ ...
/* Values for 'type' in setvbuf */
#define  IOFBF ...
#define  IOLBF ...
#define  IONBF ...
```

These functions allow the programmer to control the buffering strategy for streams in those rare instances in which the default buffering is unsatisfactory. The functions must be called after a stream is opened and before any data is read or written.

The function `setvbuf` is the more general function. The first argument is the steam being controlled; the second (if not `NULL`) is a character array to use in place of the automatically-generated buffer; `type` specifies the type of buffering, and `size` specifies the buffer size. The function returns zero if it is successful and nonzero if the arguments are improper or the request cannot be satisfied.

If `type` is `IOFBF`, the stream is fully buffered; if `type` is `IOLBF`, the buffer is flushed when a newline character is written or when the buffer is full; if `type` is `IONBF`, the stream is unbuffered. If buffering is requested and if `buf` is not a null pointer, then the array specified by `buf` should be `size` bytes long and will be used in place of the automatically-generated buffers. The constant `BUFSIZ` is an "appropriate" value for the buffer size. In Draft Proposed ANSI C the parameter `size` has type `size t`.

The function `setbuf` is a simplified form of `setvbuf`. The call

```
setbuf(stream,buf)
```

is equivalent to the expression

```
((buf==NULL) ?
    (void) setvbuf(stream,NULL, IONBF,0) :
    (void) setvbuf(stream,buf, IOFBF,BUFSIZ))
```

17.4. C-Ref: STDIN, STDOUT, STDERR

```
#include <stdio.h>
extern FILE *stderr;
extern FILE *stdin;
extern FILE *stdout;
```

The external variables stdin, stdout, and stderr are initialized prior to the start of an application program to certain standard text streams. stdin is initialized to an input stream that is the "normal input" to the program. Similarly, stdout is initialized to an output stream that is to receive the "normal output" from the program, and stderr is initialized to an output stream that is to receive error messages and other unexpected output from the program. In an interactive environment, all three streams are typically associated with the terminal used to start the program.

UNIX systems in particular provide convenient ways to associate these standard streams with files or other programs, making them very powerful when used according to certain standard conventions.

17.5. C-Ref: FSEEK, FTELL, REWIND

```
#include <stdio.h>

/* Seek codes for 'wherefrom' in fseek. */
#define SEEK SET  0                                          /* ANSI */
#define SEEK CUR  1                                          /* ANSI */
#define SEEK END  2                                          /* ANSI */

int fseek(stream, offset, wherefrom)
  FILE *stream;
  long int offset;
  int wherefrom;

long int ftell(stream)
  FILE *stream;

void rewind(stream)
  FILE *stream;
```

The function fseek allows random access within a file. The first argument must be a stream that is open for input or output. The second two arguments specify a file position: offset is a signed integer specifying a number of characters, and wherefrom is a "seek code" indicating from what point in the file the offset should be measured. The stream is positioned as requested and fseek returns zero if successful or a nonzero value if an error occurs. Any end-of-file indication is cleared and any effect of ungetc is undone. Draft Proposal ANSI C defines the constants SEEK SET, SEEK CUR, and SEEK END to represent the values of wherefrom; other implementations simply use the integer values specified.

The new position of the file is:

- `offset` characters from the beginning of the file if `wherefrom` is `SEEK SET` (0);
- `offset` characters from the current file position if `wherefrom` is `SEEK CUR` (1); or
- `offset` characters from the end of the file if `wherefrom` is `SEEK END` (2).

If `wherefrom` is `SEEK END` and `offset` is positive, the file is extended the indicated amount from its end with unspecified contents.

The function `fseek` is usually applied to binary streams; this more limited set of calls is permitted (portably) for text streams:

```
fseek(stream,0L,SEEK SET)       /* Position at beginning */
fseek(stream,0L,SEEK CUR)       /* No effect */
fseek(stream,0L,SEEK END)       /* Position at end */
fseek(stream,ftell-pos,SEEK SET)
                /* Position at previous location */
```

The value *ftell-pos* must be a value previously returned by `ftell` for *stream*. Any effects of a call to `ungetc` are also undone by a call to `fseek`.

The function `ftell` takes a stream that is open for input or output and returns the position in the stream in the form of a value suitable for the second argument to `fseek`. Using `fseek` on a saved result of `ftell` will result in resetting the position of the stream to the place in the file at which `ftell` had been called. For binary files, the value returned will be the number of characters preceeding the current file position. For text files, the value returned is implementation-defined (usable in `fseek`), but the value `0L` must be used to represent the beginning of the file. If `ftell` encounters an error, it returns `-1L` and sets `errno` to an implementation-defined value. Since `-1L` may also be a valid file position, `errno` must be checked to confirm the error.

The function `rewind` resets a stream to its beginning. The call `rewind(stream)` is equivalent to `fseek(stream,0L,SEEK SET)`.

17.6. C-Ref: FGETC, GETC, GETCHAR, UNGETC

```
#include <stdio.h>

int fgetc(stream)
  FILE *stream;

int getc(stream)
  FILE *stream;

int getchar()
```

```
int ungetc(c, stream)
  char c;
  FILE *stream;
```

The function fgetc takes as its argument an input stream. It reads the next character from the stream and returns it as a value of type int. Successive calls to fgetc will return successive characters from the input stream. If an error occurs or if the stream is at end-of-file, fgetc returns EOF. The feof facility should be used in this case to determine whether end-of-file has really been reached.

The function getc is identical to fgetc, except that getc is usually implemented as a macro for efficiency. The argument expression should not have any side effects, because it may be evaluated more than once.

The function getchar reads the next character from the standard input stream stdin and returns it as a value of type int. The call getchar() has the same effect as the call getc(stdin). Like getc, getchar is often implemented as a macro.

The function ungetc causes the character c to be pushed back onto the specified input stream. That character will be returned by the next call to fgetc, getc, or getchar on that stream. ungetc returns c when the character is successfully pushed back, EOF if the attempt fails.

One character of pushback is guaranteed provided the stream is buffered and at least one character has been read from the stream since then last fseek, fopen, or freopen operation on the stream. An attempt to push the value EOF back onto the stream as a character has no effect on the stream and returns EOF. A call to fseek or freopen erases all memory of pushed-back characters from the stream.

The function ungetc is useful for implementing input-scanning operations such as scanf. A program can "peek ahead" at the next input character by reading it and then putting it back if it is unsuitable.

17.7. C-Ref: FGETS, GETS

```
#include <stdio.h>

char *fgets(s, n, stream)
  char *s;
  int n;
  FILE *stream;

char *gets(s)
  char *s;
```

The function fgets takes three arguments: a pointer s to the beginning of a character array, a count n, and an input stream. Characters are read from the input stream into s until:

1. A newline is seen.

2. End-of-file is reached.

3. n-1 characters have been read without encountering end-of-file or a newline character.

A terminating null character is then appended to the array after the characters read. If the input is terminated because a newline was seen, the newline character will be stored in the array just before the terminating null character. The argument s is returned upon successful completion.

If end-of-file is encountered before any characters have been read from the stream, then fgets returns a null pointer and the contents of the array s are unchanged. If an error occurs during the input operation, then fgets returns a null pointer and the contents of the array s are indeterminate. The feof facility (section "C-Ref: **FEOF, FERROR, CLEARERR**") should be used to determine whether end-of-file has really been reached when NULL is returned.

The function gets reads characters from the standard input stdin into the character array s. However, unlike fputs, when the input is terminated by a newline character gets discards the newline and does *not* put into s. The use of gets can be dangerous because it is always possible for the input length to exceed the storage available in the character array. The function fgets is safer because no more than n characters will ever be placed in s.

17.8. C-Ref: FSCANF, SCANF, SSCANF

```
#include <stdio.h>

int fscanf(stream, format, additional arguments )
  FILE *stream;
  char *format;

int scanf(format, additional arguments )
  char *format;

int sscanf(s, format, additional arguments )
  char *s, *format;
```

The function fscanf parses formatted input text, reading characters from the stream specified as the first argument and converting sequences of characters according to the control string format. Additional arguments may be required, depending on the contents of the control string. Each argument after the control string must be a pointer; converted values read from the input stream are stored into the locations pointed to by the pointers.

The functions scanf and sscanf are like fscanf. In the case of scanf characters are read from the standard input stream stdin. In the case of sscanf characters are

read from the string s. When sscanf attempts to read beyond the end of the string s it operates as fscanf and scanf do when end-of-file is reached.

The input operation may terminate prematurely because the input stream reaches end-of-file or because there is a conflict between the control string and a character read from the input stream. The value returned by these functions is the number of successful assignments performed before termination of the operation for either reason. If the input reaches end-of-file before any conflict or assignment is performed, then the functions return EOF.

The control string is a picture of the expected form of the input. One may think of fscanf as performing a simple matching operation between the control string and the input stream. The contents of the control string may be divided into three categories:

1. Whitespace characters. A whitespace character in the control string causes whitespace characters to be read and discarded. The first input character encountered that is not a whitespace character remains as the next character to be read from the input stream. Note that if several consecutive whitespace characters appear in the control string, the effect is the same as if only one had appeared. Thus any sequence of consecutive whitespace characters in the control string will match any sequence of consecutive whitespace characters, possibly of different length, from the input stream.

2. Conversion specifications. A conversion specification begins with a percent sign %; the remainder of the syntax for conversion specifications is described in detail below. The number of characters read from the input stream depends on the conversion operation. As a rule of thumb, a conversion operation processes characters until: (a) end-of-file is reached, (b) a whitespace character or other inappropriate character is encountered, or (c) the number of characters read for the conversion operation equals the explicitly specified maximum field width. A conversion operation may cause an assignment to a location indicated by the next pointer argument to be used.

3. Other characters. Any character other than a whitespace character or a percent sign must match the next character of the input stream. If it does not match, a conflict has occurred; the fscanf operation is terminated, and the conflicting input character remains in the input stream to be read by the next input operation on that stream.

There should be exactly the right number of pointer arguments, each of exactly the right type, to satisfy the conversion specifications in the control string; otherwise, the results are unpredictable. If any conversion specification is malformed, then the effects are unpredictable.

A conversion specification begins with a percent sign %. After the percent sign, the following conversion specification elements should appear in this order:

1. An optional *assignment suppression flag*, written as an asterisk *. If this is present for a conversion operation that normally performs an assignment,

then characters are read and processed from the input stream in the usual way for that operation, but no assignment is performed and no pointer argument is consumed.

2. An optional *maximum field width*, expressed as an unsigned decimal integer; that is, as a nonempty sequence of decimal digits. This width must not be zero; it must be a positive number.

3. An optional *size specification*, expressed as the character `h`, meaning `short`, or as the character `l` (lowercase letter L), meaning `long`. The letter `h` may be used with the `d`, `u`, `o`, or `x` operation to indicate assignment to a location of type `short` or `unsigned short`. The letter `l` may be used with the `d`, `u`, `o`, or `x` operation to indicate assignment to a location of type `long` or `unsigned long`. The letter `l` may be used with the `e`, `f`, or `g` operation to indicate assignment to a location of type `double` rather than type `float`.

4. A required *conversion operation*, expressed (with one exception) as a single character: `c`, `d`, `e`, `E`, `f`, `g`, `G`, `o`, `s`, `u`, `x`, `X`, `%`, or '`[`'. (Draft Proposal ANSI C adds `i`, `n`, and `p`.) The exception is the `[` operation, which causes all following characters up to the next `]` to be part of the conversion specification.

The conversion specifications for `fscanf` are similar in syntax and meaning to those for `fprintf`. In a few cases it is possible to use the same format control string for both `fscanf` and `fprintf`. However, there are certain differences. The '`[`' conversion operation is peculiar to `fscanf`. On the other hand, `fscanf` does not admit any precision specification of the kind accepted by `fprintf`, nor any of the flag characters `-`, `+`, space, `0`, and `#` that are accepted by `fprintf`. For `fprintf`, an explicitly specified field width is a minimum; for `fscanf`, it is a maximum. Whereas `fprintf` allows a field width to be specified by a computed argument, indicated by using an asterisk for the field width, `fscanf` uses the asterisk for another purpose, namely assignment suppression; this is perhaps the most glaring inconsistency of all. It is best to regard the control string syntax for `fprintf` and `fscanf` as being only vaguely similar; do not use the documentation for one as a guide to the other.

The conversion operations are very complicated. Brief descriptions of each operation are given here first, for convenient reference; detailed explanations then follow.

`d` — Signed decimal conversion is performed. A value of type `int`, `short`, or `long` is assigned, depending on the size specification.

`i` — (Draft Proposed ANSI C) Signed, based integer conversion is performed. A value of type `int`, `short`, or `long` is assigned, depending on the size specification. The format of the integer expected is that of a C *integer-constant* except that a following *type-marker* may not appear. (See section "C-Ref: Integer Constants".)

u
: Unsigned decimal conversion is performed. A value of type unsigned, unsigned short, or unsigned long is assigned, depending on the size specification.

o
: Unsigned octal conversion is performed. A value of type unsigned, unsigned short, or unsigned long is assigned, depending on the size specification.

x, X
: Unsigned hexadecimal conversion is performed. A value of type unsigned, unsigned short, or unsigned long is assigned, depending on the size specification. The x and X operations are completely identical; either will accept all of the characters 0123456789abcdefABCDEF as valid hexadecimal digits. A prefix 0x or 0X may appear in the input but is not required.

c
: One or more characters are read and assigned, as many as specified by the field width. No terminating null character is appended.

s
: A whitespace-delimited string is read and assigned and an extra terminating null character is appended.

p
: (Draft Proposed ANSI C) A pointer value—such as produced by the %p conversion in fprintf—is converted assigned to the argument, which must have type void **. Pointer values typically lose their meanings between program executions.

n
: (Draft Proposed ANSI C) No conversions occur. Rather, the number of characters read from the input stream so far is written out through the argument, which must have type int *. This operation does not increment the assignment count returned by fscanf.

f, e, E, g, G
: Signed decimal floating-point conversion is performed. A value of type float or double is assigned, depending on the size specification. These operations are all identical.

%
: A single percent sign is expected in the input. No pointer argument is consumed.

[
: Input characters are scanned over and assigned. The characters following the '[' in the control string up to the next '[' indicate what characters may be scanned over. The scanned characters are stored as a string, and an extra terminating null character is appended.

Detailed explanations of how each conversion operation is performed follow.

d
: Signed decimal conversion is performed. One pointer argument is consumed; it must be of type int * (if no size specification is present), type short * (if an h size specification is present), or type long * (if an l size specification is present).

First, any leading whitespace characters are skipped; they are not counted toward the maximum field width. An optional sign (+ or -) may be present, followed by some number of decimal digits (possibly none). Characters are read until end-of-file is reached, until a nondigit character is seen (in which case that character remains unread), or (if a field width was specified) until the maximum number of characters has been read. The characters read are then interpreted as a signed decimal number and converted to a signed integer value. If no digits are read, the value is zero. If the value expressed by the input is too large to be represented as a signed integer of the appropriate size, then an unpredictable value results.

The value is assigned to the location indicated by the next pointer argument unless the assignment suppression flag * is present in the conversion specification.

`u` Unsigned decimal conversion is performed. One pointer argument is consumed; it must be of type `unsigned *` (if no size specification is present), type `unsigned short *` (if an `h` size specification is present), or type `unsigned long *` (if an `l` size specification is present).

First, any leading whitespace characters are skipped; they are not counted toward the maximum field width. Then decimal digits (possibly none) are read. Characters are read until end-of-file is reached, until a nondigit character is seen (in which case that character remains unread), or (if a field width was specified) until the maximum number of characters has been read. The characters read are then interpreted as a decimal number and converted to an unsigned integer value. If no digits are read, the value is zero. If the value expressed by the input is too large to be represented as an unsigned integer of the appropriate size, then an unpredictable value results.

The value is assigned to the location indicated by the next pointer argument unless the assignment suppression flag * is present in the conversion specification.

`o` Unsigned octal conversion is performed. One pointer argument is consumed; it must be of type `unsigned *` (if no size specification is present), type `unsigned short *` (if an `h` size specification is present), or type `unsigned long *` (if an `l` size specification is present).

First, any leading whitespace characters are skipped; they are not counted toward the maximum field width. Then octal digits (possibly none) are read. Characters are read until end-of-file is reached, until a nondigit character is seen (in which case that character remains unread), or (if a field width was specified) until the maximum number of characters has been read. If the

digit `8` or `9` is seen in the input, the result is unpredictable.

The characters read are interpreted as an octal number (regardless of whether there was a leading `0` in the input) and converted to an unsigned integer value. If no digits are read, the value is zero. If the value expressed by the input is too large to be represented as an unsigned integer of the appropriate size, then an unpredictable value results.

The value is assigned to the location indicated by the next pointer argument unless the assignment suppression flag `*` is present in the conversion specification.

`x`, `X` Unsigned hexadecimal conversion is performed. One pointer argument is consumed; it must be of type `unsigned *` (if no size specification is present), type `unsigned short *` (if an `h` size specification is present), or type `unsigned long *` (if an `l` size specification is present).

First, any leading whitespace characters are skipped; they are not counted toward the maximum field width. If the next character is `0` and the one after that is `x` or `X`, then they are counted toward the maximum field width but are otherwise ignored. Then hexadecimal digits (possibly none) are read. Characters are read until end-of-file is reached, until a character not in the set `0123456789abcdefABCDEF` is seen (in which case that character remains unread), or (if a field width was specified) until the maximum number of characters has been read.

The characters read are interpreted as a hexadecimal number (regardless of whether there was a leading `0x` or `0X` in the input) and converted to an unsigned integer value. If no digits are read, the value is zero. If the value expressed by the input is too large to be represented as an unsigned integer of the appropriate size, then an unpredictable value results.

The value is assigned to the location indicated by the next pointer argument unless the assignment suppression flag `*` is present in the conversion specification.

The `x` and `X` operations are completely identical; either will accept all of the characters `0123456789abcdefABCDEF` as valid hexadecimal digits. Compare these to the `x` and `X` conversion operations for `fprintf`.

`c` One or more characters are read. One pointer argument is consumed; it must be of type `char *`. The `c` conversion operation does *not* skip over initial whitespace characters.

If no field width is specified, then exactly one character is read (unless the input stream is at end-of-file, in which case the conversion operation fails). The character value is assigned to

the location indicated by the next pointer argument unless the assignment suppression flag * is present in the conversion specification.

If a field width is specified, then the pointer argument is assumed to point to the beginning of an array, and the field width specifies the number of characters to be read; the conversion operation fails if end-of-file is encountered before that many characters have been read. The characters read are stored into successive locations of the array unless the assignment suppression flag * is present in the conversion specification. No extra terminating null is appended to the characters that are read. Size specification is not relevant to the `c` conversion operation.

`s` — A string is read. One pointer argument is consumed; it must be of type `char *`.

First, any leading whitespace characters are skipped; they are not counted toward the maximum field width. Characters are read until end-of-file is reached, until a whitespace character is seen (in which case that character remains unread), or (if a field width was specified) until the maximum number of characters has been read. If end-of-file is encountered before any nonwhitespace character is seen, the conversion operation is considered to have failed.

The pointer argument is assumed to point to the beginning of an array of characters. The characters read are stored into successive locations of the array unless the assignment suppression flag * is present in the conversion specification. If assignment is not suppressed then an extra terminating null is appended to the stored characters. Size specification is not relevant to the `s` conversion operation.

The `s` conversion operation can be dangerous if no maximum field width is specified because it is always possible for the input length to exceed the storage available in the character array.

The `s` operation with an explicit field width differs from the `c` operation with an explicit field width. The `c` operation does not skip over whitespace characters, and will read exactly as many characters as were specified unless end-of-file is encountered; the `s` operation skips over initial whitespace characters, will be terminated by a whitespace character after reading in some number of characters that are not whitespace, and will append a null character to the stored characters.

`f, e, E, g, G` — Signed decimal floating-point conversion is performed. One pointer argument is consumed; it must be of type `float *` (if

no size specification is present), or type `double *` (if an `l` size specification is present).

First, any leading whitespace characters are skipped; they are not counted toward the maximum field width. The expected input format consists of an optional sign, zero or more decimal digits, an optional decimal point, zero or more decimal digits, and then possibly the letter `e` or `E`, which may be followed by an optional sign and then zero or more decimal digits.

The characters read are interpreted as a floating-point number representation and converted to a floating-point number of the specified size. If no digits are read, or at least no digits are read before the letter `e` or `E` is seen, then the value is zero. If no digits are seen after the letter `e` or `E`, then the exponent part of the decimal representation is assumed to be zero. If the value expressed by the input is too large or too small to be represented as a floating-point number of the appropriate size, then an unpredictable value results. If the value expressed by the input is not too large or too small but nevertheless cannot be represented exactly as a floating-point number of the appropriate size, then some form of rounding or truncation occurs.

The value is assigned to the location indicated by the next pointer argument unless the assignment suppression flag `*` is present in the conversion specification.

The `f`, `e`, `E`, `g`, and `G` conversion operations are completely identical; any one of them will accept any style of floating-point representation, with or without an exponent part.

`%` A single percent sign is expected in the input. Because a percent sign is used to indicate the beginning of a conversion specification, it is necessary to write two of them in order to have one matched. No pointer argument is consumed. The assignment suppression flag, field width, and size specification are not relevant to the `%` conversion operation.

`[` A string is read. One pointer argument is consumed; it must be of type `char *`.

The '`[`' conversion operation does *not* skip over initial whitespace characters. The conversion specification indicates exactly what characters may be read as a part of the input field. The '`[`' must be followed in the control string by more characters, terminated by ']'. All the characters up to the ']' are part of the conversion specification. If the character immediately following the '`[`' is the circumflex ^, it has a special meaning as a negation flag; all other characters between the '`[`' and ']' , including ^ if it does not immediately follow the initial '`[`', are treated alike. The characters in the control string between the

initial '[' and the terminating ']' , other than the negation flag if it is present, are regarded as a set in the mathematical sense. Note that '[' may be in the set, but ']' cannot be. Moreover, ^ can be in the set only if a negation flag is present or if some other character precedes the ^; it is impossible to have a set consisting only of ^ without a negation flag.

Characters are read until end-of-file is reached, until a terminating character is seen (in which case that character remains unread), or (if a field width was specified) until the maximum number of characters has been read. If a negation flag is not present, then a character is a terminating character if it is *not* in the set; if a negation flag is present, then a character is a terminating character if it *is* in the set. Looking at the other side of the coin, if no negation flag is present then only characters in the set are read, but if a negation flag is present then only characters not in the set are read.

The pointer argument is assumed to point to the beginning of an array of characters. The characters read are stored into successive locations of the array unless the assignment suppression flag * is present in the conversion specification. If assignment is not suppressed then an extra terminating null is appended to the stored characters. Size specification is not relevant to the '[' conversion operation.

Like the s conversion, the '[' conversion operation can be dangerous if no maximum field width is specified because it is always possible for the input length to exceed the storage available in the character array.

Note that none of the conversion operations normally skips over trailing whitespace characters as a matter of course. Trailing whitespace characters (such as the newline that terminates a line of input) will remain unread unless explicitly matched in the control string. However, doing this may be tricky because a whitespace character in the control string will attempt to match many whitespace character in the input, resulting in an attempt to read beyond a newline character.

It is not possible to determine directly whether matches of literal character in the control string succeed or fail. It is also not possible to determine directly whether conversion operations involving suppressed assignments succeed or fail. The value returned by these functions reflects only the number of successful assignments performed.

17.9. C-Ref: FPUTC, PUTC, PUTCHAR

```
#include <stdio.h>

int fputc(c, stream)
  char c;
  FILE *stream;

int putc(c, stream)
  char c;
  FILE *stream;

int putchar(c)
  char c;
```

The function fputc takes as arguments a character value and an output stream. It writes the character to the stream and also returns the character as a value of type int. Successive calls to fputc will write the given characters successively to the output stream. If an error occurs, fputc returns EOF instead of the character that was to have been written.

The function putc operates exactly like fputc, but it is usually implemented as a macro. The argument expression must not have any side effects because they may be evaluated more than once.

The function putchar writes a character to the standard output stream stdout. Like putc, putchar is usually implemented as a macro and is quite efficient. The call putchar(c) is equivalent to putc(c,stdout).

17.10. C-Ref: FPUTS, PUTS

```
#include <stdio.h>

int fputs(s, stream)
  char *s;
  FILE *stream;

int puts(s)
  char *s;
```

The function fputs takes as arguments a null-terminated string and an output stream. It writes to the stream all the characters of the string, not including the terminating null character. If an error occurs, fputs returns EOF; otherwise, it returns some other value. (Draft Proposal ANSI C specifies that fputs and puts return a nonzero value when an error occurs and zero otherwise, which is slightly inconsistent with fputc and putchar.

The function puts is like fputs except:

1. The characters are always written to the stream stdout.

2. After the characters in s are written out, an additional newline character is written (regardless whether s contained a newline character).

The functions fputs and puts are often implemented like this:

```
int fputs( s, stream )
  char *s;
  FILE *stream;
{
  int ch, retval;       /* Error? */
  while ( ch = *s++ )
    last return val = fputc( ch, stream );
  return retval;
}

int puts( s )
  char *s;
{
  int ch;
  while( ch = *s++ ) putchar( ch );
  return putchar( '\n', stream );
}
```

Note that the implementation of fputs has an error; when the argument s is the empty string the return value from fputs is indeterminate. This error actually appears in several versions of UNIX, and programmers might beware of that boundary case.

17.11. C-Ref: FPRINTF, PRINTF, SPRINTF

```
#include <stdio.h>

int fprintf(stream, format, additional arguments )
  FILE *stream;
  char *format;

int printf(format, additional arguments )
  char *format;

int sprintf(s, format, additional arguments )
  char *s, *format;
```

The function fprintf performs output formatting, sending the output to the stream specified as the first argument. The second argument is a format control string. Additional arguments may be required, depending on the contents of the control string. A series of output characters is generated as directed by the control string; these characters are sent to the specified stream.

The functions `printf` and `sprintf` are related to `fprintf`. `printf` sends the characters to the standard output stream `stdout`, whereas `sprintf` causes the output characters to be stored into the string buffer `s`. In the case of `sprintf`, a final null character is output to `s` after all characters specified by the control string have been output. (It is the programmer's responsibility to ensure that the destination string area is large enough to contain the output generated by the formatting operation.)

The value returned by these functions is `EOF` if an error occurred during the output operation; otherwise, the result is some value other than `EOF`. In Draft Proposed ANSI C and most current implementations the functions return the number of characters sent to the output stream if no error occurs. In the case of `sprintf`, the count does not include the terminating null character. (Draft Proposed ANSI C allows these functions to return any negative value if an error occurs.)

The control string is simply text to be copied verbatim, except that the string may contain *conversion specifications*. A conversion specification may call for the processing of some number of additional arguments, resulting in a formatted conversion operation that generates output characters not explicitly contained in the control string. There should be exactly the right number of arguments, each of exactly the right type, to satisfy the conversion specifications in the control string; otherwise, the results are unpredictable. If any conversion specification is malformed then the effects are unpredictable.

The sequence of characters output for a conversion specification may be conceptually divided into three elements: the *converted value* proper, which reflects the value of the converted argument; the *prefix*, which, if present, is typically a sign or a space; and the *padding*, which is a sequence of spaces or zero digits added if necessary to increase the width of the output sequence to a specified minimum. The prefix always precedes the converted value. Depending on the conversion specification, the padding may precede the prefix, separate the prefix from the converted value, or follow the converted value.

A conversion specification begins with a percent sign `%`. Following the percent sign, the following conversion specification elements should appear in the following order:

1. Zero or more optional *flag characters*, which modify the meaning of the main conversion operation.

`-`	Left-justify within the field width rather than right-justify.
`0`	Use `0` for the pad character rather than space.
`+`	Always produce a sign, either `+` or `-`.
space	Always produce either the sign `-` or a space.

#	Use a variant of the main conversion operation.

The effects of the flag characters are described in more detail below.

2. An optional *minimum field width*, expressed as an decimal integer constant; that is, as a nonempty sequence of decimal digits. The sequence of digits should not begin with a zero digit (which can be confused with a flag). The field width may also be specified by an asterisk *, in which case an argument is consumed; the argument must be of type `int`, and its value specifies the minimum field width. If the converted value results in fewer characters than the specified field width then pad characters (spaces or zeros) are used to pad the value to the specified width. If the converted value results in more characters than the specified field width, the field is expanded to accommodate it.

3. An optional *precision* specification, expressed as a period '.' followed by an optional decimal integer. If the period appears but the integer is missing, the integer is assumed to be zero, which usually has a different effect from omitting the entire precision specification. The precision may also be specified by an asterisk * following the period, in which case an argument is consumed; the argument must be of type `int`, and its value specifies the precision. The result of specifying a negative precision is unpredictable. The precision specification is used to control the number of digits to be printed for a numeric conversion, and is described in detail in conjunction with the main conversion operations.

4. An optional *long size* specification, expressed as the character `l` (lowercase letter L), which in conjunction with the conversion operations `d`, `o`, `u`, `x`, and `X` indicates that the argument is `long`. (Draft Proposed ANSI C uses the `L` (uppercase letter L) prefix with `e`, `E`, `f`, `F`, `g`, and `G` to indicate that the argument has type `long double`; it also allows an `h` prefix with `d`, `o`, `u`, `x`, and `X` to indicate that the argument is a `short` integer type; this is only for uniformity with `scanf` formats since the argument will have been converted to a non-`short` type.)

5. A required *conversion operation*, expressed as a single character: `c`, `d`, `e`, `E`, `f`, `g`, `G`, `o`, `s`, `u`, `x`, `X`, or `%`. (Draft Proposed ANSI C also adds `i`, `p`, and `n`.)

If either the field width or the precision is specified with an asterisk, then the argument consumed by the asterisk precedes any arguments consumed by the main conversion operation. If both the field width and the precision are specified with asterisks, then the field-width argument precedes the precision argument. The general rule is that situations in conversion specifications that consume arguments are paired with arguments in the obvious strict left-to-right order.

The flag characters and their meanings are as follows:

-	If a minus-sign flag is present, then the converted value will be left-justified within the field; that is, any padding will be placed to the right of the converted value. If no minus sign is

present, then the converted value will be right-justified within the field; that is, any padding will be placed to the left of the converted value. This flag is relevant only when an explicit minimum field width is specified and the converted value is smaller than that minimum width.

`0` — If a zero-digit flag is present, then `0` will be used as the pad character if padding is to be placed to the left of the converted value. If no zero-digit flag is present, then space will be used as the pad character. (Space is always used as the pad character if padding is to be placed to the right of the converted value; that is, if the `-` flag character is present.) The `0` flag is relevant only when an explicit minimum field width is specified and the converted value is smaller than that minimum width. Furthermore, this flag is superceded by the precision specification that can be present in integer conversions.

`+` — If a plus-sign flag is present, then the result of a signed conversion will always begin with a sign; that is, an explicit `+` will precede a converted positive value. (Negative values are always preceded by `-` regardless of whether a plus-sign flag is specified.) This flag is relevant only for the conversion operations `d`, `e`, `E`, `f`, `g`, and `G`.

space — If a space flag is present and the first character in the converted value resulting from a signed conversion is not a sign (`+` or `-`), then a space will be added before the converted value. The adding of this space on the left is independent of any padding that may be placed to the left or right under control of the minus-sign flag character. If both the space and plus-sign flags appear in a single conversion specification, the space flag is effectively ignored because the plus-sign flag ensures that the converted value will always begin with a sign. This flag is relevant only for the conversion operations `d`, `e`, `E`, `f`, `g`, and `G`.

`#` — If a number-sign flag is present, then an alternate form of the main conversion operation is used. This flag is relevant only for the conversion operations `e`, `E`, `f`, `g`, `G`, `o`, `x`, and `X`. The modifications implied by the number-sign flag are described in conjunction with the relevant main conversion operations.

The conversion operations are very complicated. Brief descriptions of each operation are given here first, for convenient reference; detailed explanations then follow.

`d, i` — Signed decimal conversion from type `int` or `long`.

`u` — Unsigned decimal conversion from type `unsigned` or `unsigned long`.

`o`	Unsigned octal conversion from type `unsigned` or `unsigned long`.
`x`, `X`	Unsigned hexadecimal conversion from type `unsigned` or `unsigned long`. The `x` operation uses `0123456789abcdef` as digits, whereas the `X` operation uses `0123456789ABCDEF`.
`c`	The argument is printed as a character.
`s`	The argument is printed as a string.
`p`	The argument must have type `void *` and is printed in an implementation-defined way (Draft Proposed ANSI C).
`n`	The argument must have type `int *`; into that integer is written the number of characters output so far by this call to `fprintf` (Draft Proposal ANSI C).
`f`	Signed decimal floating-point conversion is performed. The output is in the form `[-]ddd.dddd`, loosely speaking. The precision specifies the number of digits to be printed after the decimal point.
`e`, `E`	Signed decimal floating-point conversion is performed. The output is in the form `[-]d.dddddde+dd` or `[-]d.ddddddE+dd`, loosely speaking. One digit appears before the decimal point; the precision specifies the number of digits to be printed after the decimal point.
`g`, `G`	Signed decimal floating-point conversion is performed. Loosely speaking, if the value to be printed is not too large or too small, then `f` format is used; otherwise, `e` or `E` format is used. The general idea is to use whichever format will require less space. The precision specifies the number of significant digits to be printed.
`%`	A single percent sign is printed.

Detailed explanations of how each conversion operation is performed follow. Each operation computes and prints a prefix and a converted value, in that order. Either the prefix or the converted value may be empty. Additional padding may be printed before or after the prefix/value pair.

The padding is handled in the same way for each conversion operator. If the prefix and converted value together are shorter than the minimum field width, then enough padding is added to increase the width to the specified minimum. If the `-` flag is present, then space is the pad character, and the padding follows the converted value. If the `-` flag is not present but the `0` flag is present, then `0` is the pad character, and the padding is placed between the prefix and the converted value. If neither the `-` flag nor the `0` flag is present, then space is the pad character, and the padding precedes the prefix.

Here are the detailed explanations of how the individual conversion operators compute a prefix and converted value.

d, i

Signed decimal conversion is performed. One argument is consumed, which should be of type int (or type long int if the l [long size] specification is present). (The i operator is present in Draft Proposed ANSI C only and is primarily useful in fscanf. It is recognized by fprintf for uniformity, where it is identical to the d operator.)

The converted value consists of a sequence of decimal digits that represents the absolute value of the argument. This sequence is as short as possible but not shorter than the specified precision. The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros (see below). If the precision is 1, then the converted value will not have a leading 0 unless the argument is 0, in which case a single 0 is output. If the precision is 0 and the argument is 0, then the converted value is empty (the null string). If no precision is specified, then a precision of 1 is assumed.

The prefix is computed as follows. If the argument is negative, the prefix is a minus sign. If the argument is nonnegative and the + flag is specified, then the prefix is a plus sign. If the argument is nonnegative, the space flag is specified, and the + flag is not specified, then the prefix is a space. Otherwise, the prefix is empty. The # flag is not relevant to the d conversion operation.

u

Unsigned decimal conversion is performed. One argument is consumed, which should be of type unsigned (or type unsigned long if the l [long size] specification is present).

The converted value consists of a sequence of decimal digits that represents the value of the argument. This sequence is as short as possible but not shorter than the specified precision. The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros (see below). If the precision is 1, then the converted value will not have a leading 0 unless the argument is 0, in which case a single 0 is output. If the precision is 0 and the argument is 0, then the converted value is empty (the null string). If no precision is specified, then a precision of 1 is assumed. The prefix is always empty.

The +, space, and # flags are not relevant to the u conversion operation.

o

Unsigned octal conversion is performed. One argument is consumed, which should be of type unsigned (or type unsigned long if the l [long size] specification is present).

The converted value consists of a sequence of decimal digits that represents the value of the argument. This sequence is as short as possible but not shorter than the specified precision. The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros (see below). If the precision is 1, then the converted value will not have a leading `0` unless the argument is 0, in which case a single `0` is output. If the precision is 0 and the argument is 0, then the converted value is empty (the null string). If no precision is specified, then a precision of 1 is assumed. If the # flag is present, then the prefix is `0`. If the # flag is not present, then the prefix is empty. The + and space flags are not relevant to the `o` conversion operation.

`x`, `X`

Unsigned hexadecimal conversion is performed. One argument is consumed, which should be of type `unsigned` (or type `unsigned long` if the `l` [long size] specification is present).

The converted value consists of a sequence of decimal digits that represents the value of the argument. This sequence is as short as possible but not shorter than the specified precision. The `x` operation uses `0123456789abcdef` as digits, whereas the `X` operation uses `0123456789ABCDEF`. The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros (see below). If the precision is 1, then the converted value will not have a leading `0` unless the argument is 0, in which case a single `0` is output. If the precision is 0 and the argument is 0, then the converted value is empty (the null string). If no precision is specified, then a precision of 1 is assumed.

If the # flag is present, then the prefix is `0x` (for the `x` operation) or `0X` (for the `X` operation). If the # flag is not present, then the prefix is empty. The + and space flags are not relevant to the `x` conversion operation.

`c`

The argument is printed as a character. One argument is consumed, which should be of type `int` or `unsigned`. The value of this integer argument must be a valid encoding of some character; if it is not a valid character code, then the results are unpredictable. The converted value is the single character specified by the argument. The prefix is always empty.

The +, space, and # flags, the precision specification, and the `l` (long size) specification are not relevant to the `c` conversion operation.

`s`

The argument is printed as a string. One argument is consumed, which should be of type "pointer to `char`." If no preci-

sion specification is given, then the converted value is the sequence of characters in the string argument up to but not including the terminating null character. If a precision specification *p* is given, then the converted value is the first *p* characters of the string, or up to but not including the terminating null character, whichever is shorter. The prefix is always empty.

The +, space, and # flags and the `l` (long size) specification are not relevant to the `s` conversion operation.

`p` The argument must have type `void *` and it is printed in an implementation-defined format. This conversion operator is found in Draft Proposed ANSI C but is otherwise not common.

`n` The argument must have type `int *`. Instead of outputting characters, this conversion operator causes the number of characters output so far to be written into the designated integer. This conversion operator is found in Draft Proposal ANSI C but is otherwise not common.

`f` Signed decimal floating-point conversion is performed. One argument is consumed, which should be of type `double`. (Note that if an argument of type `float` is given, it is converted to type `double` by the usual function argument conversions before `printf` ever sees it, so it does work to use `%f` to print a number of type `float`.)

The converted value consists of a sequence of decimal digits, possibly with an embedded decimal point, that represents the approximate absolute value of the argument. At least one digit appears before the decimal point, but no more than are necessary to represent the value (that is, no extraneous leading zeros are produced). The precision specifies the number of digits to appear after the decimal point. If the precision is 0, then no digits appear after the decimal point; moreover, the decimal point itself also does not appear unless the # flag is present. If no precision is specified, then a precision of 6 is assumed.

If the floating-point value cannot be represented exactly in the number of digits produced, then the converted value should be the result of rounding the exact floating-point value to the number of decimal places produced. (However, some implementations do not perform correct rounding in all cases.)

The prefix is computed as follows. If the argument is negative, the prefix is a minus sign. If the argument is nonnegative and the + flag is specified, then the prefix is a plus sign. If the argument is nonnegative, the space flag is specified, and the + flag is not specified, then the prefix is a space. Otherwise, the prefix is empty. The `l` (long size) specification is not relevant to the `f` conversion operation.

e, E

Signed decimal floating-point conversion is performed. One argument is consumed, which should be of type `double`. (An argument of type `float` is permitted, as for the `f` conversion.)

The converted value consists of a decimal digit, then possibly a decimal point and more decimal digits, then the letter `e` (for the `e` operation) or `E` (for the `E` operation), then a plus sign or minus sign, then finally at least two more decimal digits. The part before the letter `e` or `E` represents a value between 1 (inclusive) and 10 (exclusive). The part after the letter `e` or `E` represents an exponent value as a signed decimal integer. The value of the first part, multiplied by 10 raised to the value of the second part, is approximately equal to the absolute value of the argument.

Exactly one digit appears before the decimal point. The precision specifies the number of digits to appear after the decimal point; the total number of digits printed before the letter `e` or `E` is therefore one greater than the specified precision. (Compare this to the `g` and `G` conversion operations, where if `e` format is used the number of digits printed is exactly equal to the precision.) If the precision is 0, then no digits appear after the decimal point; moreover, the decimal point itself also does not appear unless the `#` flag is present. If no precision is specified, then a precision of 6 is assumed.

The exponent is printed always with a sign and always with at least two decimal digits. If more than two digits are needed to represent the exponent, then as few as are necessary are printed.

If the floating-point value cannot be represented exactly in the number of digits produced, then the converted value should be the result of rounding the exact floating-point value to the number of decimal places produced. (However, some implementations do not perform correct rounding in all cases.)

The prefix is computed as follows. If the argument is negative, the prefix is a minus sign. If the argument is nonnegative and the `+` flag is specified, then the prefix is a plus sign. If the argument is nonnegative, the space flag is specified, and the `+` flag is not specified, then the prefix is a space. Otherwise, the prefix is empty.

The `l` (long size) specification is not relevant to the `e` conversion operation.

g, G

Signed decimal floating-point conversion is performed. One argument is consumed, which should be of type `double`. (An argument of type `float` is permitted, as for the `f`, `e`, and `E` conversions.) Only the `g` conversion operator is discussed below; the `G`

operation is identical except that wherever g uses e conversion, G uses E conversion. If the specified precision is less than 1, then a precision of 1 is used. If no precision is specified, then a precision of 6 is assumed.

The conversion process may be explained as follows. Let p be the precision to be used for the g conversion operation. Produce a converted value as if for e conversion operator, using a precision for the e conversion equal to p-1 and assuming the # flag to be present (regardless of whether it is present in the g conversion specification). Let the exponent value (the part of the converted value after the letter e) be called n. If n is greater than p or less than -3, then the converted value from the e conversion is used as the converted value for the g conversion. (Some implementations use a lower bound of -4 instead of -3.) If, however, n is between -3 and p (inclusive), then a new converted value is produced by using f conversion with a precision equal to p-n and assuming the # flag to be present (regardless of whether it is present in the g conversion specification).

If the # flag is not present in the g conversion specification, the converted value is then modified by stripping off trailing zeros in the following manner. If the converted value resulted from e conversion, then call the character just before the letter e the "last fraction character"; if the converted value resulted from f conversion, then call the last character in the converted value the "last fraction character." Then the rule is that if the last fraction character is 0 then that character is discarded and the stripping process is repeated; if the last fraction character is . then that character is discarded and the stripping process is terminated; otherwise, no character is discarded and the stripping process is terminated. The effect is to discard zero digits that do not contribute significantly to the value, and then to discard the decimal point if no digits immediately follow it. Note that only g conversion performs this process of discarding trailing zeros; f and e conversions never do this.

The prefix is computed as follows. If the argument is negative, the prefix is a minus sign. If the argument is nonnegative and the + flag is specified, then the prefix is a plus sign. If the argument is nonnegative, the space flag is specified, and the + flag is not specified, then the prefix is a space. Otherwise, the prefix is empty.

The l (long size) specification is not relevant to the g conversion operation.

% A single percent sign is printed. Because a percent sign is used to indicate the beginning of a conversion specification, it is necessary to write two of them in order to have one printed.

No arguments are consumed. The converted value is the single character `%`. The prefix is always empty.

Note that padding is performed for this conversion operation just as for any other conversion operation; for example, the conversion specification `%05%` will print `0000%`; that is, a percent sign right-justified with `0` padding in a field of width 5.

The `+`, space, and `#` flags, the precision specification, and the `l` (long size) specification are not relevant to the `%` conversion operation.

The tables "C-Ref: Examples of Output Formatting (Part 1)" and "C-Ref: Examples of Output Formatting (Part 2)" give some examples of how the various conversion operations work and how the various flags affect them. For example, the third row of Table "C-Ref: Examples of Output Formatting (Part 1)" was generated by this call to `printf`:

```
printf("%6s|%#5d|%#5o|%#5x|%#7.2f|%#10.2e|%#10.4g|\n",
       "%#", 45, 45, 45, 12.678, 12.678, 12.678);
```

The other rows were generated by similar calls differing only in which combination of flags was used. The third row of Table "C-Ref: Examples of Output Formatting (Part 2)" was generated by this call to `printf`:

```
printf("%6s|%#5s|%#5c|%#5%|%#7.2f|%#10.2e|%#10.4g|\n",
       "%#", "zap", '*', -3.4567, -3.4567, -3.4567);
```

The examples in these tables by no means illustrate all of the interesting effects that can be obtained with `printf`.

17.11.1. C-Ref: Examples of Output Formatting (Part 1)

```
Value     45    45    45  12.678     12.678     12.678
Operation 5d    5o    5x    7.2f      10.2e      10.4g
Flags
     %|   45|   55|   2d|  12.68|  1.27e+01|     12.68|
    %0|00045|00055|0002d|0012.68|001.27e+01|0000012.68|
    %#|   45|  055| 0x2d|  12.68|  1.27e+01|     12.68|
   %#0|00045|00055|0x02d|0012.68|001.27e+01|0000012.68|
    % |   45|   55|   2d|  12.68|  1.27e+01|     12.68|
   % 0| 0045|00055|0002d| 012.68| 01.27e+01| 000012.68|
   % #|   45|  055| 0x2d|  12.68|  1.27e+01|     12.68|
  % #0| 0045|00055|0x02d| 012.68| 01.27e+01| 000012.68|
    %+|  +45|   55|   2d| +12.68| +1.27e+01|    +12.68|
   %+0|+0045|00055|0002d|+012.68|+01.27e+01|+000012.68|
   %+#|  +45|  055| 0x2d| +12.68| +1.27e+01|    +12.68|
  %+#0|+0045|00055|0x02d|+012.68|+01.27e+01|+000012.68|
   %+ |  +45|   55|   2d| +12.68| +1.27e+01|    +12.68|
  %+ 0|+0045|00055|0002d|+012.68|+01.27e+01|+000012.68|
  %+ #|  +45|  055| 0x2d| +12.68| +1.27e+01|    +12.68|
 %+ #0|+0045|00055|0x02d|+012.68|+01.27e+01|+000012.68|
    %-|45   |55   |2d   |12.68  |1.27e+01  |12.68     |
   %-0|45   |55   |2d   |12.68  |1.27e+01  |12.68     |
   %-#|45   |055  |0x2d |12.68  |1.27e+01  |12.68     |
  %-#0|45   |055  |0x2d |12.68  |1.27e+01  |12.68     |
   %- | 45  |55   |2d   | 12.68 | 1.27e+01 | 12.68    |
  %- 0| 45  |55   |2d   | 12.68 | 1.27e+01 | 12.68    |
  %- #| 45  |055  |0x2d | 12.68 | 1.27e+01 | 12.68    |
 %- #0| 45  |055  |0x2d | 12.68 | 1.27e+01 | 12.68    |
   %-+|+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
  %-+0|+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
  %-+#|+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
 %-+#0|+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
  %-+ |+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
 %-+ 0|+45  |55   |2d   |+12.68 |+1.27e+01 |+12.68    |
 %-+ #|+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
%-+ #0|+45  |055  |0x2d |+12.68 |+1.27e+01 |+12.68    |
```

This table demonstrates the effects of the flag characters on various conversion operations. Across the top of the table are shown the values printed for each column (`45` for the first three columns, and `12.678` for the last three) and the conversion operation, plus width and precision, used to print it. Down the left side are shown all possible combinations of the five flag characters.

17.11.2. C-Ref: Examples of Output Formatting (Part 2)

```
Value  "zap"   '*'  none -3.4567    -3.4567    -3.4567
Operation 5s    5c    5%    7.2f      10.2e      10.4g
Flags
    %|  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
   %0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
   %#|  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
  %#0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
   % |  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
  % 0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
  % #|  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
 % #0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
   %+|  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
  %+0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
  %+#|  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
 %+#0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
  %+ |  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
 %+ 0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
 %+ #|  zap|    *|    %|  -3.46| -3.46E+00|    -3.457|
%+ #0|00zap|0000*|0000%|-003.46|-03.46E+00|-00003.457|
   %-|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
  %-0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
  %-#|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
 %-#0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
  %- |zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
 %- 0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
 %- #|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
%- #0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
  %-+|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
 %-+0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
 %-+#|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
%-+#0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
 %-+ |zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
%-+ 0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
%-+ #|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
%-+ #0|zap  |*    |%    |-3.46  |-3.46E+00 |-3.457    |
```

This table demonstrates the effects of the flag characters on various conversion operations. Across the top of the table are shown the values printed for each column (the string `"zap"` for the first column; the character `'*'` for the second; none for the third, which uses the `%` conversion operation; and and `-3.4567` for the last three) and the conversion operation, plus width and precision, used to print it. Down the left side are shown all possible combinations of the five flag characters.

17.12. C-Ref: VFPRINTF, VPRINTF, VSPRINTF

```
#include <varargs.h>                                    /* Non-ANSI form */
#include <stdarg.h>                                              /* ANSI */
#include <stdio.h>

int vfprintf(stream, format, arg)
  FILE *stream;
  char *format;
  va list arg;

int vprintf(format, arg)
  char *format;
  va list arg;

int vsprintf(s, format, arg)
  char *s, *format;
  va list arg;
```

The functions `vfprintf`, `vprintf`, and `vscanf` are the same as the functions `fprintf`, `printf`, and `sprintf`, respectively, except that the extra arguments are given as a variable argument list as defined by the `vararg` facility (section "C-Ref: **VARARG, STDARG**"). They were adopted into Draft Proposal ANSI C from System V UNIX. These functions are useful when the programmer wants to define his own variable-argument functions that use the formatted output facilities. The functions do not invoke the `va end` facility.

As an example, suppose the programmer wanted to write a general function, `trace`, that printed the name of a function and its arguments. Any function to be traced would begin with a call to `trace` of the form:

```
trace(name,format,parm1,parm2,...,parmN)
```

where `name` is the name of the function being called and `format` is a format string suitable for printing the argument values `parm1`, `parm2`, ..., `parmN`. For example, here is what a function `f` would look like with a call to `trace`:

```
int f(x,y)
  int x;
  double y;
{
  trace("f","x=%d, y=%f", x, y);
 ...
}
```

A possible implementation of `trace` is given below:

```
#include <varargs.h>
#include <stdio.h>
extern global trace enabled;
```

```
void trace(va alist)
  va dcl
{
  va list args;
  char *name;
  char *format;

  if (global trace enabled) {
    va start(args);
    name = va arg(args,char*);
    format = va arg(args,char *);
    fprintf(stderr,"--> entering %s(", name);
    vfprintf(stderr, format, args);
    fprintf(stderr,")\n");
    va end(args);
  }
}
```

17.13. C-Ref: FREAD, FWRITE

```
#include <stdio.h>

int fread(ptr, element size, count, stream)
  char *ptr;
  unsigned element size;
  int count;
  FILE *stream;

int fwrite(ptr, element size, count, stream)
  char *ptr;
  unsigned element size;
  int count;
  FILE *stream;
```

The functions `fread` and `fwrite` perform input and output, respectively, to binary files. In both cases, `stream` is the input or output stream and `ptr` is a pointer to an array of `count` elements, each of which is `element size` characters long. In Draft Proposed ANSI C, the parameter `ptr` has type `void *` and the parameters `element size` and `count` have type `size t`.

The function `fread` reads up to `count` elements of the indicated size from the input stream into the specified array. The actual number of items read is returned by `fread`; it may be less than `count` if end-of-file is encountered. If an error is encountered, zero is returned. The `feof` or `ferror` facilities may be used to determine whether an error or an immediate end-of-file caused zero to be returned. If either

count or element size is zero, no data is transferred and zero is returned. As a simple example, the following program reads an input file containing objects of a structure type and prints the number of such objects read. The program depends on the exit function to close the input file.

```
/* Count the number of elements
   of type "struct S" in file "in.dat" */
#include <stdio.h>
static char *FileName = "in.dat";
struct S { int a,b; double d; char str[103]; };

main()
{
  struct S buffer;
  int items read = 0;
  FILE *in file = fopen(FileName,"r");
  if (in file == NULL)
    { fprintf(stderr,"?Couldn't open %s\n",FileName);
      exit(1); }
  while (fread((char *) &buffer,
               sizeof(struct S), 1, in file) == 1)
    items read++;
  if (ferror(in file))
   { fprintf(stderr,"?Read error, file %s record %d\n",
             FileName,items read+1); exit(1); }
  printf("Finished; %d elements read\n",items read);
  exit(0);
}
```

The function fwrite writes count elements of size element size from the specified array. The actual number of items written is returned by fwrite; it will be the same as count unless an error occurs. To read or write blocks of data in random order, use the functions fseek and ftell.

17.14. C-Ref: FEOF, FERROR, CLEARERR

```
#include <stdio.h>

int feof(stream)
  FILE *stream;

int ferror(stream)
  FILE *stream;

void clearerr(stream)
  FILE *stream;
```

The function `feof` takes takes as its argument an input stream. If end-of-file has been detected while reading from the input stream, then a nonzero value is returned; otherwise, zero is returned. Note that even if there are no more characters in the stream to be read, `feof` will not signal end-of-file unless and until an attempt is made to read "past" the last character. The function is normally used after an input operation has signaled a failure.

The function `ferror` returns the error status of a stream. If an error has occurred while reading from or writing to the stream, then `ferror` returns a nonzero value; otherwise, zero is returned. Once an error has occurred for a given stream, repeated calls to `ferror` will continue to report an error unless `clearerr` is used to explicitly reset the error indication. Closing the stream, as with `fclose`, will also reset the error indication.

The function `clearerr` resets any error indication on the specified stream; subsequent calls on `ferror` will report that no error has occurred for that stream unless and until another error occurs.

17.15. C-Ref: REMOVE, RENAME

```
#include <stdio.h>                                    /* ANSI */

int remove(filename)                                  /* ANSI */
  char *filename;

int rename(oldname,newname)
  char *oldname, *newname;
```

The `remove` function removes or deletes the named file; it returns zero if the operation succeeds and a nonzero value if it does not. The string pointed to by `filename` is not altered. Implementations may differ in the details of what "remove" or "delete" actually mean, but it should not be possible for a program to open a file it has deleted. If the file is open or if the file does not exist the action of `remove` is implementation-defined. (In the latter case we think the function should do nothing and return zero.) This function is not present in the C library of most UNIX systems.

The `rename` function changes the name of `oldfile` to `newfile`; it returns zero if the operation succeeds and a nonzero value if it does not. The strings pointed to by `oldname` and `newname` are not altered. If `oldfile` is open, if `oldfile` does not exist, or if `newfile` already exists then the action of `rename` is implementation-defined.

17.16. C-Ref: TMPFILE, TMPNAM, MKTEMP

```
#include <stdio.h>

FILE *tmpfile()

char *tmpnam(buf)
  char *buf;

#define L tmpnam  n
#define TMP MAX  m                                        /* ANSI */

char *mktemp(buf)                                /* Non-ANSI form */
  char *buf;
```

The function `tmpfile` creates a new file and opens it for output using `fopen` mode `"w+"` (`"w+b"` in Draft Proposed ANSI C). `tmpfile` returns a file pointer for the new file, or a null pointer if the operation fails. The intent is that the new file be used only during the current program's execution, and it is automatically deleted upon program termination. After writing data to the file, the programmer can use the `rewind` function to reposition the file at its beginning for reading.

The function `tmpnam` is used to create new file names that do not conflict with any other file names currently in use; the programmer can then open a new file with that name using the full generality of `fopen`. The files so created are not "temporary"; they are not deleted automatically upon program termination. If `buf` is `NULL`, `tmpnam` returns a pointer to the new file name string; the string may be altered by subsequent calls to `tmpnam`. If `buf` is not `NULL` it must point to an array of not less than `L tmpnam` characters; `tmpnam` will copy the new file name string into that array and return `buf`. If `tmpnam` fails, it returns a null pointer. Draft Proposal ANSI C defines the value `TMP MAX` to be the number of successive calls to `tmpnam` that will generate unique names.

The function `mktemp` is like `tmpnam` except that `buf` (the "template") must point to a string with six trailing 'X' characters, which will be overwritten with other letters or digits to form a unique file name. The value `buf` is returned. Successive calls to `mktemp` should specify different templates to ensure unique names.

An example of a common but poor programming practice in C is to write:

```
ptr = fopen(mktemp("/tmp/abcXXXXXX"),"w+");
```

This idiom depends upon being able to alter the characters of the string constant, since that is what `mktemp` will do. The programmer also loses the ability to reference the file name string. It is better and no less efficient to write:

```
char filename[]="/tmp/abcXXXXXX";
ptr = fopen(mktemp(filename),"w+");
```

18. C-Ref: Storage Allocation

Name	Section
calloc	"C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"
cfree	"C-Ref: **FREE, CFREE**"
clalloc	"C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"
free	"C-Ref: **FREE, CFREE**"
malloc	"C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"
mlalloc	"C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"
realloc	"C-Ref: **REALLOC, RELALLOC**"
relalloc	"C-Ref: **REALLOC, RELALLOC**"

The storage allocation facilities provide a simple form of heap memory management that allows a program to repeatedly request allocation of a "fresh" region of memory and perhaps later to deallocate such a region when it is no longer needed. Explicitly deallocated regions are recycled by the storage manager for satisfaction of further allocation requests.

When a region of memory is allocated in response to a request, a pointer to the region is returned to the caller. This pointer will be of type `char *` (`void *` in Draft Proposed ANSI C) but is guaranteed to be properly aligned for any data type. The caller may then use a cast operator to convert this pointer to another pointer type.

In Draft Proposed ANSI C the facilities described in this section are declared in the header file `stdlib.h`. In other C implementations there is typically no associated header file and the programmer must declare the facilities himself.

18.1. C-Ref: MALLOC, CALLOC, MLALLOC, CLALLOC

```
#include <stdlib.h>                                    /* ANSI */

char *malloc(size)
  unsigned size;

char *calloc(elt count, elt size)
  unsigned elt count, elt size;

char *mlalloc(size)                           /* Non-ANSI form */
  unsigned long size;

char *clalloc(elt count, elt size)            /* Non-ANSI form */
  unsigned long elt count, elt size;
```

The function malloc allocates a region of memory large enough to hold an object whose size (as measured by the sizeof operator) is size. A pointer to the first element of the region is returned. If it is impossible for some reason to perform the requested allocation, or if size is 0, a null pointer is returned. The region of memory is not specially initialized in any way and the caller must assume that it will contain garbage information. In Draft Proposed ANSI C the parameter to malloc has type size t (section "C-Ref: **NULL, PTRDIFF_T, SIZE_T**") and the pointer returned has type void *.

The function calloc allocates a region of memory large enough to hold an array of elt count elements, each of size elt size (typically given by the sizeof operator). The region of memory is cleared (all bits are set to zero) and a pointer to the first element of the region is returned. If it is impossible for some reason to perform the requested allocation, or if elt count or elt size is zero, a null pointer is returned. In Draft Proposed ANSI C the parameters to calloc both have type size t and the pointer returned has type void *. Programmers should note that arithmetic values consisting of zero bits do not necessarily have the value zero, nor do pointers filled with zero bits necessarily have the value NULL.

The function mlalloc is just like malloc, except that its arguments are of type unsigned long int rather than unsigned int. In some C implementations it is possible to allocate memory regions so large that numbers of type unsigned int are not large enough to specify them. This function is not provided in Draft Proposed ANSI C because the use of type size t for the parameter of malloc avoids the problem.

Similarly, the function clalloc is just like calloc, except that its arguments are of type unsigned long int rather than unsigned int.

The caller of an allocation routine will typically cast the result pointer to an appropriate pointer type:

```
/* Return a pointer to a new object. */
typedef struct { /*...*/ } object;
object *NewObject()
{
  object *objptr = (object *) malloc(sizeof(object));
  if (objptr==NULL)
        printf("NewObject: ran out of memory!\n");
  return objptr;
}
```

18.2. C-Ref: FREE, CFREE

```
#include <stdlib.h>                                    /* ANSI */

void free(ptr)
  char *ptr;
```

```
void cfree(ptr)                                  /* Non-ANSI form */
  char *ptr;
```

The function free deallocates a region of memory previously allocated by malloc, or mlalloc, realloc (section "C-Ref: **REALLOC, RELALLOC**"), or relalloc. The argument to free must be a pointer that is equivalent (except for possible intermediate type casting) to a pointer previously returned by the allocation routine. (If the argument to free is a null pointer then no action should occur, but this is known to cause trouble in some C implementations.) Once a region of memory has been explicitly freed it must not be used for any other purpose.

The function cfree deallocates a region of memory previously allocated by calloc or clalloc. In Draft Proposed ANSI C free is used to deallocate storage returned by calloc, and the argument to free has type void *.

18.3. C-Ref: REALLOC, RELALLOC

```
#include <stdlib.h>                                     /* ANSI */

char *realloc(ptr, size)
  char *ptr;
  unsigned size;

char *relalloc(ptr,size)                         /* Non-ANSI form */
  char *ptr;
  unsigned long size;
```

The function realloc takes a pointer to memory region previously allocated by one of the standard functions and changes its size while preserving its contents. If necessary, the contents are copied to a new memory region. A pointer to the (possibly new) memory region is returned. If the request cannot be satisfied, a null pointer is returned and the old region is not disturbed.

If the first argument to realloc is a null pointer then the function behaves like malloc (section "C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**"). If ptr is not null and size is zero, realloc returns a null pointer and the old region is deallocated. If the new size is smaller than the old size, then some of the old contents at the end of the old region will be discarded. If the new size is larger than the old size, then all of the old contents are preserved and new space is added at the end; the new space is not specially initialized in any way, and the caller must assume that it contains garbage information. Whenever realloc returns a pointer that is different from its first argument the programmer should assume that the old region of memory was deallocated and should not be used.

The function relalloc behaves like relalloc except that the size argument has type unsigned long int instead of int. This function is not present in Draft Proposed ANSI C, in which the size argument to realloc has type size t.

Below is shown a typical use of `realloc` to expand the dynamic array `samples`. The elements of the array are always referenced using array subscripts; any pointers into the array could be invalidated by the call to `realloc`.

```
#define SAMPLE INCREMENT 100
int sample limit = 0;
int sample count = 0;
double *samples = NULL;

int AddSample( new sample )
  double new sample;
{
  if (sample count >= sample limit) {
       sample limit += SAMPLE INCREMENT;
       samples = (double *)
                     realloc((char *) samples,
                             sample limit *
                                  sizeof(double));
  }
  samples[sample count++] = new sample;
  return sample count;
}
```

19. C-Ref: Mathematical Functions

Name	Section
`abs`	"C-Ref: **ABS, FABS, LABS**"
`acos`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`asin`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`atan`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`atan2`	"C-Ref: **ACOS, ASIN, ATAN, ATAN2**"
`ceil`	"C-Ref: **CEIL, FLOOR, FMOD**"
`cos`	"C-Ref: **COS, SIN, TAN**"
`cosh`	"C-Ref: **COSH, SINH, TANH**"
`div`	"C-Ref: **DIV, LDIV**"
`exp`	"C-Ref: **EXP, LOG, LOG10**"
`fabs`	"C-Ref: **ABS, FABS, LABS**"
`floor`	"C-Ref: **CEIL, FLOOR, FMOD**"
`fmod`	"C-Ref: **CEIL, FLOOR, FMOD**"
`frexp`	"C-Ref: **FREXP, LDEXP, MODF**"
`labs`	"C-Ref: **ABS, FABS, LABS**"
`ldexp`	"C-Ref: **FREXP, LDEXP, MODF**"
`ldiv`	"C-Ref: **DIV, LDIV**"
`log`	"C-Ref: **EXP, LOG, LOG10**"
`log10`	"C-Ref: **EXP, LOG, LOG10**"
`modf`	"C-Ref: **FREXP, LDEXP, MODF**"
`pow`	"C-Ref: **POW, SQRT**"
`rand`	"C-Ref: **RAND, SRAND**"
`sin`	"C-Ref: **COS, SIN, TAN**"
`sinh`	"C-Ref: **COSH, SINH, TANH**"
`sqrt`	"C-Ref: **POW, SQRT**"
`srand`	"C-Ref: **RAND, SRAND**"
`tan`	"C-Ref: **COS, SIN, TAN**"
`tanh`	"C-Ref: **COSH, SINH, TANH**"

Most of the facilities described in this section are declared by the library header file `math.h`. All of the operations on floating-point numbers are defined only for ar-

guments of type double, because of the rule that all actual function arguments of type float are converted to type double before the call is performed. In Draft Proposed ANSI C this convention is maintained for compatibility.

Two general kinds of errors are possible with the mathematical functions, although older C implementations may not handle them consistently. When an input argument lies outside the domain over which the function is defined, a *domain error* occurs. The variable errno (section "C-Ref: **ERRNO, STRERROR, PERROR**") is set to the value EDOM and the function returns an implementation-defined value. Zero is the traditional error return value but some implementations may have better choices, such as special "not a number" values. (System V UNIX has a more elaborate mechanism, matherr.)

If the result of a function is too large in magnitude to be represented as a value of the function's return type, then a *range error* occurs. When this happens, errno should be set to the value ERANGE and the function should return the largest representable floating-point value with the same sign as the correct result. In Draft Proposed ANSI C this is the value of the macro HUGE VAL; in System V it is HUGE.

If the result of a function is too small in magnitude to be represented, the function should return zero; whether errno is also set to ERANGE is left to the discretion of the implementation.

19.1. C-Ref: ABS, FABS, LABS

```
#include <math.h>

double fabs(x)
  double x;

#include <stdlib.h>                                         /* ANSI */

int abs(x)
  int x;

long int labs(x)
  long int x;
```

The functions abs, fabs, and labs all return the absolute value of their arguments. More precisely, if the argument is nonnegative, then the value of the argument itself is returned; if the argument is negative, then the result of negating the argument (as if by the unary '-' operator) is returned. The argument and the result are both of type int for abs, of type double for fabs, and of type long int for labs.

In Draft Proposed ANSI C the functions abs and labs are defined in stdlib.h instead of math.h. The absolute value functions are so easy to implement that some compilers may treat them as built-in functions; this is permitted in Draft Proposed ANSI C.

19.2. C-Ref: DIV, LDIV

```
#include <stdlib.h>
typedef ... div t;
typedef ... ldiv t;

div t div(n,d)                                                /* ANSI */
  int n, d;

ldiv t ldiv(n,d)                                              /* ANSI */
  long int n, d;
```

The functions `div` and `ldiv` are found in Draft Proposed ANSI C but are otherwise not common in C implementations. They compute simultaneously the quotient and remainder of the division of `n` by `d`. The type `div t` is a structure containing two components, `quot` and `rem` (in any order), both of type `int`. The type `ldiv t` is also a structure with components `quot` and `rem`, both of type `long int`. The returned quotient has the same sign as `n/d` and its magnitude is the largest integer not greater than `n/d`. The behavior of the functions when `d` is zero, or when `n/d` is not representable, is undefined (not necessarily a domain error) to allow for the most efficient implementation.

19.3. C-Ref: CEIL, FLOOR, FMOD

```
#include <math.h>

double ceil(x)
  double x;

double floor(x)
  double x;

double fmod(x, y)
  double x, y;
```

The function `ceil` rounds its argument up to an integer; the argument and the result are both of type `double`. More precisely, it returns the smallest floating-point number not less than `x` whose value is an exact mathematical integer. If the value of the argument is already a mathematical integer, then the result equals the argument.

Similarly, `floor` rounds its argument down to an integer. It returns the largest floating-point number not greater than `x` whose value is an exact mathematical integer. If the value of the argument is already a mathematical integer, then the result equals the argument.

The function `fmod` returns the floating-point remainder of `x/y`. If `y` is zero, `x` is returned; if the quotient `x/y` cannot be represented, the result is undefined. More

precisely, fmod returns an approximation to the mathematical value f such that f has the same sign as x, the absolute value of f is less than the absolute value of y, and there exists an integer k such that k*y+f equals x. The function fmod should not be confused with modf (section "C-Ref: **FREXP, LDEXP, MODF**"), a function that extracts the fractional and integer parts of a floating-point number.

19.4. C-Ref: EXP, LOG, LOG10

```
#include <math.h>

double exp(x)
  double x;

double log(x)
  double x;

double log10(x)
  double x;
```

The function exp computes a floating-point approximation to the exponential function; that is, *e* raised to the power x, where *e* is the base of the natural logarithms. The argument and the result are both of type double. A range error can occur for large arguments.

The function log computes a floating-point approximation to the natural logarithm function. The argument and the result are both of type double. If the argument is negative, a domain error occurs. Otherwise if the argument is zero or close to zero, a range error occurs (towards minus infinity). Some C implementations may treat zero as a domain error, and some implementations name this function ln.

The function log10 computes a floating-point approximation to the base-10 logarithm function. The conditions causing domain and range errors are the same as for the function log.

19.5. C-Ref: FREXP, LDEXP, MODF

```
#include <math.h>

double frexp(x, nptr)
  double x;
  int *nptr;

double ldexp(x, n)
  double x;
  int n;
```

```
double modf(x, nptr)
  double x;
  int *nptr;
```

The function `frexp` splits a floating-point number into a fraction `f` and an exponent `n`, such that the absolute value of `f` is less than `1.0` but not less than `0.5` and such that `f` times 2 raised to the power `n` is equal to `x`. The fraction `f` is returned and as a side effect the exponent `n` is stored into the place pointed to by `nptr`. If the argument to `frexp` is zero then both returned values will be zero.

The function `ldexp` is the inverse of `frexp`; it computes the value `x` times 2 raised to the power `n`. A range error may occur.

The function `modf` splits a floating-point number into a fractional part `f` and an integer part `n`, such that the absolute value of `f` is less than `1.0` and such that `f` plus `n` is equal to `x`. Both `f` and `n` will have the same sign as the input argument. The fractional part `f` is returned, and as a side effect the integer part `n` is stored into the place pointed to by `nptr`.

The function `modf` should not be confused with `fmod` (section "C-Ref: **CEIL, FLOOR, FMOD**"), a function that computes the remainder from evenly dividing one floating-point number by another. The name `modf` is a misnomer, because the value it computes is properly called a *remainder*.

19.6. C-Ref: POW, SQRT

```
#include <math.h>

double pow(x, y)
  double x, y;

double sqrt(x)
  double x;
```

The function `pow` computes a floating-point approximation to the power function; that is, x raised to the power `y`. When `x` is nonzero and `y` is zero, the result is `1.0`. When `x` is zero and `y` is positive, the result is zero. Domain errors occur if `x` is negative and `y` is not an exact integer, or if `x` is zero and `y` is nonpositive. Range errors may also occur.

The function `sqrt` computes a floating-point approximation to the nonnegative square root of the argument. A domain error occurs if the argument is negative. Implementations that support the concept of a negative floating-point zero may return that number as the square root of itself, while still recording a domain error.

19.7. C-Ref: RAND, SRAND

```
#include <stdlib.h>                                           /* ANSI */

int rand();

void srand (seed)
  /* unsigned */ int seed;
```

Successive calls to rand return values in the range 0 to the largest representable positive value of type int (inclusive) that are the successive results of a pseudo-random-number generator. In Draft Proposal ANSI C the upper bound of the range of rand is given by RAND MAX, which will be at least 32767.

The function srand may be used to initialize the pseudorandom-number generator that is used to generate successive values for calls to rand. After a call to srand, successive calls to rand will produce a certain series of pseudorandom numbers. If srand is called again with the same argument, then after that point successive calls to rand will produce the same series of pseudorandom numbers. Successive calls made to rand before srand is ever called in a user program will produce the same series of pseudorandom numbers that would be produced after srand is called with argument 1.

19.8. C-Ref: COS, SIN, TAN

```
#include <math.h>

double cos(x)
  double x;

double sin(x)
  double x;

double tan(x)
  double x;
```

The function cos computes a floating-point approximation to the trigonometric cosine function of the argument value, where the argument is taken to be in radians. No domain or range errors are possible, but the programmer should be aware that the result may have little or no signficance for arguments whose absolute value is very large.

The functions sin and tan similarly compute floating-point approximations to the trigonometric sine and tangent functions, respectively. A range error may occur in the tan function if the argument is close to an odd multiple of $\pi/2$. The same caution about large-magnitude arguments applies to sin and tan.

19.9. C-Ref: ACOS, ASIN, ATAN, ATAN2

```
#include <math.h>

double acos(x)
  double x;

double asin(x)
  double x;

double atan(x)
  double x;

double atan2(y, x)
  double y, x;
```

The function `acos` computes a floating-point approximation to the principal value of the trigonometric arc cosine function of the argument value. The result is in radians and lies between 0 and π. (The range of these functions is approximate because of the effect of round-off errors.) A domain error occurs if the argument is less than `-1.0` or greater than `1.0`.

The function `asin` computes a floating-point approximation to the principal value of the trigonometric arc sine function of the argument value. The result is in radians and lies (approximately) between $-\pi/2$ and $\pi/2$. A domain error occurs if the argument is less than `-1.0` or greater than `1.0`.

The function `atan` computes a floating-point approximation to the principal value of the trigonometric arc tangent function of the argument value. The result is in radians and lies (approximately) between $-\pi/2$ and $\pi/2$. No range or domain errors are possible. In some implementations of C this function is called `arctan`.

The function `atan2` computes a floating-point approximation to the principal value of the trigonometric arc tangent function of the value `y/x`. The signs of the two arguments are taken into account to determine quadrant information. Viewed in terms of a Cartesian coordinate system, the result is the angle between the positive x-axis and a line drawn from the origin through the point (`x`, `y`). The result is in radians and lies (approximately) between $-\pi$ and π. If `x` is zero the result is $\pi/2$ or $-\pi/2$, depending on whether `y` is positive or negative. A domain error occurs if both `x` and `y` are zero.

19.10. C-Ref: COSH, SINH, TANH

```
#include <math.h>

double cosh(x)
  double x;
```

```
double sinh(x)
  double x;

double tanh(x)
  double x;
```

The function `cosh` computes a floating-point approximation to the hyperbolic cosine function of the argument value. A range error can occur if the absolute value of the argument is large.

The function `sinh` computes a floating-point approximation to the hyperbolic sine function of the argument value. A range error can occur if the absolute value of the argument is large.

The function `tanh` computes a floating-point approximation to the hyperbolic tangent function of the argument value.

20. C-Ref: Time and Date Functions

Name	Section
`asctime`	"C-Ref: **ASCTIME, CTIME**"
`CLK TCK`	"C-Ref: **CLOCK, CLOCK_T, CLK_TCK, TIMES**"
`clock`	"C-Ref: **CLOCK, CLOCK_T, CLK_TCK, TIMES**"
`clock t`	"C-Ref: **CLOCK, CLOCK_T, CLK_TCK, TIMES**"
`ctime`	"C-Ref: **ASCTIME, CTIME**"
`difftime`	"C-Ref: **DIFFTIME**"
`gmtime`	"C-Ref: **GMTIME, LOCALTIME, MKTIME**"
`localtime`	"C-Ref: **GMTIME, LOCALTIME, MKTIME**"
`mktime`	"C-Ref: **GMTIME, LOCALTIME, MKTIME**"
`time t`	"C-Ref: **TIME, TIME_T**"
`times`	"C-Ref: **CLOCK, CLOCK_T, CLK_TCK, TIMES**"
`tm (struct)`	"C-Ref: **GMTIME, LOCALTIME, MKTIME**"

The facilities in this section give the C programmer ways to manipulate the (calendar) date and time and the process time, that is, the amount of time used by the running program.

Calendar time is represented in two forms: a simple arithmetic value returned by the `time` function, and a "broken-down," structured form computed from the arithmetic value by the `gmtime` and `localtime` functions.

Process time is represented by an arithmetic value (usually integral) returned by the `clock` function.

20.1. C-Ref: CLOCK, CLOCK_T, CLK_TCK, TIMES

```
/* unsigned */ long clock()                          /* System V UNIX */

#include<time.h>                                             /* ANSI */
typedef ... clock t;                                        /* ANSI */
#define CLK TCK ...                                         /* ANSI */
clock t clock()                                             /* ANSI */

#include <sys/types.h>                              /* Berkeley UNIX */
#include <sys/times.h>                             /* Berkeley UNIX */
void times(tmsbuf)                                 /* Berkeley UNIX */
struct tms {...} tmsbuf;                           /* Berkeley UNIX */
```

The `clock` function returns the processor time used by the current process. The units in which the time is expressed vary with the implementation; microseconds are customary. Although the return type of `clock` is `long`, the value returned is really of type `unsigned long`; the use of `long` predates the addition of `unsigned long` to the language. Unsigned arithmetic should always be used when computing with process times.

Programmers should be aware of "wrap-around" in the process time. For instance, if type `long` is represented in 32 bits and `clock` returns the time in microseconds, the time returned will "wrap around" to its starting value in about 2,147 seconds or 36 minutes.

The Draft Proposed ANSI C version of `clock` allows the implementor freedom to use any arithmetic type, `clock t`, for the process time. The number of time units ("clock ticks") per second is defined by the macro `CLK TCK`. If the processor time is not available, the value `-1` (cast to be of type `clock t`) will be returned.

Here is a typical example of how the `clock` function can be used to time a program.

```
typedef unsigned long clock t;
extern clock t clock();
#define CLK TCK 1000000
...
clock t start, finish, duration;
start = clock();
process();
finish = clock();
printf("process() took %f seconds to execute\n",
    ((double) (finish - start)) / CLK TCK );
```

Note how the cast to type `double` allows `clock t` and `CLK TCK` to be either floating-point or integral.

The `times` function is found in Berkeley UNIX and in some non-UNIX systems instead of `clock`; it returns a structured value that reports various components of the process time, each typically measured in units of 1/60 seconds. A rough equivalent to the `clock` function can be written using `times`:

```
#include <sys/types.h>
#include <sys/times.h>
#define CLK TCK 60
long clock()
{
  struct tms tmsbuf;
  times(tmsbuf);
  return (tmsbuf.tms utime + tmsbuf.tms stime);
}
```

There is a type, `time t`, used in the above structure; it is a "process time" unit and therefore is *not* the same as the "calendar time" type `time t` defined in conjunction with the `time` function.

20.2. C-Ref: TIME, TIME_T

```
/*unsigned*/ long time(tptr)                      /* Non-ANSI form */
  /*unsigned*/ long *tptr;

#include <time.h>                                          /* ANSI */
typedef ... time t;                                        /* ANSI */
time t time(tptr)                                          /* ANSI */
  time t *tptr;
```

The function `time` returns the current calendar time encoded in an integer of type `long`. If the parameter `tptr` is not `NULL`, the return value is also stored at `*tptr`. If errors are encountered, the value `-1` is returned; in System V UNIX `errno` is also set to `EFAULT`. Although the return type of `time` is `long`, the value returned is usually of type `unsigned long`; the use of `long` predates the addition of `unsigned long` to the language.

The Draft Proposed ANSI C version of `time` gives the implementor more freedom by allowing the return type, `time t`, to be any arithmetic type. If errors are encountered, the value `-1` (cast to be of type `time t`) will be returned.

Typically, the value returned by `time` is passed to the function `asctime` or `ctime` to convert it to a readable form, or is passed to `localtime` or `gmtime` to convert it to a form that is more easily processed. Computing the interval between two calendar times can be done by the Draft Proposed ANSI C function `difftime`; in other implementations the programmer must either work with the broken-down time from `gmtime` or depend on a customary representation of the time as the number of seconds since some arbitrary past date. (January 1, 1970 is popular.)

20.3. C-Ref: ASCTIME, CTIME

```
#include <time.h>
#include <sys/time.h>                             /* Berkeley UNIX */

char *asctime(ts)
  struct tm *ts;

char *ctime(timptr)
  long *timptr;                                   /* Non-ANSI form */
  time t *timptr;                                          /* ANSI */
```

The `asctime` and `ctime` functions both return a pointer to a string that is a printable date and time of the form:

```
Mon Jan 26 12:34:56 1952\n\0
```

The `asctime` function takes as its single argument a pointer to a a structured calendar time; such a structure is produced by `localtime` or `gmtime` from the arithmetic time that is returned by `time`. The `ctime` function takes the value returned

by `time` directly, and therefore `ctime(t)` is equivalent to the expression

```
asctime(localtime(t))
```

In most implementations the functions return a pointer to a static data area, and therefore the returned string should be printed or copied (with `strcpy`) before any subsequent call to either function.

20.4. C-Ref: GMTIME, LOCALTIME, MKTIME

```
#include <time.h>
#include <sys/time.h>                                   /* Berkeley UNIX */

struct tm { ... };

struct tm *gmtime(t)
   long *t;
   time t *t;                                                    /* ANSI */

struct tm *localtime(t)
  long *t;
  time t *t;                                                     /* ANSI */

time t mktime( tmptr )                                           /* ANSI */
  struct tm *tmptr;
```

The functions `gmtime` and `localtime` convert an arithmetic calendar time returned by `time` to a "broken-down" form of type `struct tm`. The `gmtime` function converts to Greenwich Mean Time (GMT) while `localtime` converts to local time, taking into account the time zone and possible daylight savings time. The functions return a null pointer if they encounter errors. These functions are portable across UNIX systems and Draft Proposed ANSI C.

In most implementations the functions return a pointer to a single static data area that is overwritten on every call. Therefore, the returned structure should be used or copied before any subsequent call to either function.

The structure `struct tm` includes these fields:

```
int tm sec;   /* seconds; range 0..59 */
int tm min;   /* minutes; range 0..59 */
int tm hour;  /* hours since midnight; range 0..23 */
int tm mday;  /* day of month; range 1..31 */
int tm mon;   /* month; range 0..11 */
int tm year;  /* year; with 0==1900 */
int tm wday;  /* day of week; range Sun==0..6 */
int tm yday;  /* day of year; range 0..365 */
int tm isdst; /* nonzero implies daylight savings */
```

The function mktime constructs a value of type time t from the broken-down local time specified by the argument tmptr. The values of tmptr->tm wday and tmptr->tm yday are ignored by mktime. If successful, mktime returns the new time value and adjusts the contents of *tmptr, setting the tm wday and tm yday components. If the indicated calendar time cannot be represented as a value of time t, mktime returns the value (time t)-1.

20.5. C-Ref: DIFFTIME

```
#include <time.h>                                     /* ANSI */

double difftime(t1, t0)                               /* ANSI */
  time t t1,t0;
```

The difftime function is found only in Draft Proposed ANSI C; it subtracts calendar time t0 from calendar time t1, returning the difference in seconds as a value of type double. Because the encoding of the calendar time in type time t is not specified, difftime may involve special processing.

21. C-Ref: Control Functions

`abort`	"C-Ref: **EXIT, ABORT**"
`alarm`	"C-Ref: **SLEEP, ALARM**"
`assert`	"C-Ref: **ASSERT, NDEBUG**"
`exec`	"C-Ref: **EXEC, SYSTEM**"
`exit`	"C-Ref: **EXIT, ABORT**"
` exit`	"C-Ref: **EXIT, ABORT**"
`gsignal`	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"
`jmp`	"C-Ref: **SETJMP, LONGJMP, JMP_BUF**"
`kill`	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"
`longjmp`	"C-Ref: **SETJMP, LONGJMP, JMP_BUF**"
`NDEBUG`	"C-Ref: **ASSERT, NDEBUG**"
`onexit`	"C-Ref: **ONEXIT, ONEXIT_T**"
`onexit t`	"C-Ref: **ONEXIT, ONEXIT_T**"
`psignal`	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"
`raise`	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"
`setjmp`	"C-Ref: **SETJMP, LONGJMP, JMP_BUF**"
`signal`	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"
`sleep`	"C-Ref: **SLEEP, ALARM**"
`ssignal`	"C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**"
`system`	"C-Ref: **EXEC, SYSTEM**"

The facilities in this chapter provide extensions to the standard flow of control in C programs. The functions `signal` and `raise` implement a primitive exception handling mechanism; `assert` and `exit` provide program-termination capabilities; and `exec` and `system` allow other programs to be started from within a C program.

21.1. C-Ref: ASSERT, NDEBUG

```
#include <assert.h>
```

```
#ifndef NDEBUG
#define assert(expression) ...
#else
#define assert(expression) ;
#endif
```

The macro assert takes as its single argument a value of any scalar type. If that value is 0 (false) and if additionally the macro NDEBUG is *not* defined, then assert will print a diagnostic message on the standard output stream and halt the program by calling exit. The assert facility is always implemented as a macro and the header file assert.h must be included in the source file to use the facility.

If the macro NDEBUG is defined when the header file assert.h is read, the assert facility is disabled, usually by defining assert to be the empty statement. Not only are no diagnostic messages printed, but the expression that is an argument to assert is not evaluated. The assert facility is typically used during program development to check that certain conditions are true at run time.

21.2. C-Ref: EXEC, SYSTEM

```
#include <stdlib.h>                                            /* ANSI */

int system(command)
 char *command;

execl (name, arg0, arg1, ..., argn, 0 )                   /* UNIX form */
execlp(name, arg0, arg1, ..., argn, 0 )
execle(name, arg0, arg1, ..., argn, 0, envp)
  char *name, *arg0, *arg1, ..., *argn, *envp[];

execv (name, argv )                                       /* UNIX form */
execvp(name, argv )
execve(name, argv, envp)
  char *name, *argv[], *envp[];
```

The function system passes its string argument to the operating system's command processor for execution in some implementation-defined way. In UNIX systems, the command processor is the shell. The value returned by system is implementation-defined but is usually the completion status of the command.

In Draft Proposed ANSI C, system may be called with a null argument in which case 0 is returned if there is no command processor provided in the implementation and a nonzero value is returned if there is a command processor.

The various forms of exec are found mainly in UNIX systems; they are not included in Draft Proposed ANSI C. In all cases, they transform the current process into a new process by executing the file name (typically created by the loader). They differ in how arguments are supplied for the new process:

1. The functions `execl`, `execlp`, and `execle` take a variable number of arguments, all character (string) pointers. The last argument must be a null pointer (0). By convention, `arg0` should point to a string that is the same as `name`, that is, it should be the name of the program being (to be) executed.

2. The functions `execv`, `execvp`, and `execve` supply a pointer to a vector of arguments, such as is provided to function `main`. By convention, `argv[0]` should point to a string that is the same as `name`, that is, it should be the name of the program being (to be) executed. The list specified by `argv` must be terminated by a null pointer.

3. The functions `execle` and `execve` also pass an explicit "environment" to the new process. The parameter `envp` is an array of string pointers terminated by a null pointer. Each string is of the form `"name=value."` In the other versions of `exec`, the environment pointer of the calling process is passed to the new process.

4. The functions `execlp` and `execvp` are the same as `execl` and `execv`, respectively, except that the system looks for the file `name` in the set of directories normally searched for commands (usually the value of the environment variable `path` or `PATH`).

When the new process is started, the arguments supplied to `exec` are made available to the new process' `main` function.

21.3. C-Ref: EXIT, ABORT

```
#include <stdlib.h>                                        /* ANSI */

void exit(status)
  int status;

void  exit(status)
  int status;

void abort()
```

The `exit`, `exit`, and `abort` functions cause the program to terminate. Control does not normally return to the caller of these functions.

The function `exit` causes "normal" termination of a program, performing normal cleanup activities such as flushing buffers and closing any open streams (see `fflush` and `fclose`, section "C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**"). The function `exit` differs in that cleanup activities are not performed. Both functions return an integer value, *status*, to the host environment. By convention, a *status* of 0 signifies successful program termination and nonzero values are used to signify various kinds of abnormal termination. In a C program, returning an integer value from the function `main` with a `return` statement acts like calling `exit` with the same value.

The abort function stops program execution, usually by executing an illegal instruction or causing some other form of hardware fault. Often this fault is translated to a special signal that can be caught or ignored; if ignored, control returns to the caller of abort (returning a value in some implementations). (The signal is SIGABRT in Draft Proposed ANSI C and SIGIOT in System V UNIX.) Whether or not abort causes cleanup actions is implementation-defined. When abort causes the program to terminate, the status value returned is implementation-defined but must be nonzero.

In Draft Proposed ANSI C, exit causes all functions registered with the onexit function to be called in the reverse order of their registration. This happens before the normal cleanup operations. In addition, exit closes files created by tmpfile.

21.4. C-Ref: SETJMP, LONGJMP, JMP_BUF

```
#include <setjmp.h>

typedef ... jmp buf;

int setjmp(env)
  jmp buf env;

void longjmp(env, status)
  jmp buf env;
  int status;
```

The setjmp and longjmp functions implement a primitive form of nonlocal jumps, which may be used to handle abnormal or exceptional situations. This facility was traditionally considered more portable than signal, but the latter has also been incorporated into Draft Proposed ANSI C.

The function setjmp records its caller's environment in the "jump buffer" env, an implementation-defined array, and returns 0 to its caller. (The type jmp buf must be an array type so that a pointer to env is actually passed to setjmp.)

The function longjmp takes as its arguments a jump buffer previously filled by calling setjmp and an integer value, status, that is usually nonzero. The effect of calling longjmp is to cause the program to return from the call to setjmp again, this time returning the value status. Some implementations, including Draft Proposed ANSI C, do not permit longjmp to cause 0 to be returned from setjmp, and will return 1 from setjmp if longjmp is called with a status argument of 0.

If the jump buffer argument to longjmp is not set by setjmp, or if the function containing setjmp returns before the call to longjmp, indeterminate and probably unamusing behavior can result. In some implementations a call to setjmp or longjmp during interrupt processing or signal handling will not operate correctly; Draft Proposed ANSI C specifies that longjmp operate correctly in nonnested signal handlers.

21.5. C-Ref: ONEXIT, ONEXIT_T

```
#include <stdlib.h>                                         /* ANSI */

typedef ... onexit t;

onexit t onexit(func)
  onexit t (*func)();
```

The onexit function is found in Draft Proposed ANSI C. It "registers" a function so that the function will be called when the program terminates when exit is called or when function main returns. The functions are not called when the program terminates abnormally, as with abort or raise. Implementations must allow at least 32 functions to be registered. The onexit function returns a nonzero (true) value of scalar type onexit t if it succeeds, a zero (false) value if it does not.

The registered functions are called in the reverse order of their registration, before any standard cleanup actions are performed by exit. Each function is called with no arguments and should have return type void. A registered function should not attempt to reference any objects with storage class auto or register except those it defines itself. Registering the same function more than once has unpredictable results.

21.6. C-Ref: SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL

```
#include <signal.h>

#define SIG IGN ...
#define SIG DFL ...
#define SIG ERR ...                                         /* ANSI */
#define SIGxxx ...
...

void (*signal(sig,func))()
  int sig;
  void (*func)();

int raise(sig)                                              /* ANSI */
  int sig;

int kill(pid,sig)                                   /* Non-ANSI form */
  int pid, sig;

int (*ssignal(softsig,func))()                      /* System V UNIX */
  int softsig;
  int (*func)();
```

```
int gsignal(softsig)                                    /* System V UNIX */
  int softsig;

void psignal(sig,prefix)                                /* Berkeley UNIX */
  int sig;
  char *prefix;
```

Signals are (potentially) asynchronous events that may require special processing by the user program or by the implementation. Signals are named by integer values and each implementation defines a set of signals in header file `signal.h`. Signals may be triggered or *raised* by the computer's error-detection mechanisms, by the user program itself via `kill` or `raise`, or by actions external to the program. *Software signals* used by the functions `ssignal` and `psignal` are user-defined, with values generally in the range 1 through 15; otherwise they operate like regular signals.

A *signal handler* for signal *sig* is a user function that is invoked when signal *sig* is "raised." The handler function is expected to perform some useful action and then return, generally causing the program to resume at the point it was interrupted. (Handlers may also call `exit` or `longjmp`.) Signal handlers are normal C functions taking one argument, the raised signal:

```
void my handler(sig)
  int sig;  /* the signal */
{
 ...
}
```

Some implementations may pass extra arguments to handlers for certain predefined signals.

The function `signal` is used to associate signal handlers with specific signals. In the normal case `signal` is passed a signal value and a pointer to the signal handler for that signal. If the association is successful, `signal` returns a pointer to the previous signal handler (which may be `SIG DFL` if there was no handler); otherwise it returns the value -1 (`SIG ERR` in Draft Proposed ANSI C) and sets `errno`.

```
void new handler(sig)  int sig; { ... }
void (*old handler)();
...
old handler = signal( sig, &new handler );
if (old handler==SIG ERR)
   printf("?Couldn't establish new handler.\n");
...
if (signal(sig,old handler)==SIG ERR)
   printf("?Couldn't put back old handler.\n")
```

The function argument to `signal` may also have two special values, `SIG IGN` and `SIG DFL`. A call to `signal` of the form

```
signal(sig, SIG IGN)
```

means that signal `sig` is to be ignored. A call to `signal` of the form

```
signal(sig, SIG DFL)
```

means that signal `sig` is to receive its "default" handling, which usually means ignoring some signals and terminating the program on other signals.

The `ssignal` function works exactly like `signal` but is used only in conjunction with `gsignal` for user-defined software signals. Handlers supplied to `ssignal` may return integer values that become the return value of `gsignal`.

The `raise` and `gsignal` functions cause the indicated signal (or software signal) to be raised in the current process. The `kill` function causes the indicated signal to be raised in the specified process; it is less portable.

When a signal is raised for which a handler has been established by `signal` or `gsignal`, the handler is given control. Most implementations reset the associated handler to `SIG DFL` before the handler is given control to prevent unwanted recursion. The handler may return, in which case execution continues at the point of interruption. If the signal was raised by `raise` or `gsignal`, those functions return to their caller.

The `psignal` function prints on the standard error output the string `prefix` (which is customarily the name of the program) and a brief description of signal `sig`. This function may be useful in handlers that are about to call `exit`.

21.7. C-Ref: SLEEP, ALARM

```
void sleep(seconds)
  unsigned seconds;

unsigned alarm(seconds)
  unsigned seconds;
```

The `alarm` function sets an internal system timer to the indicated number of seconds and returns the number of seconds previously on the timer. When the timer expires, the signal `SIGALRM` is raised in the program. If the argument to `alarm` is 0, the effect of the call is to cancel any previous alarm request. The `alarm` function is useful for escaping from various kinds of deadlock situations.

The `sleep` function suspends the program for the indicated number of seconds, at which time the `sleep` function returns and execution continues. Sleep is typically implemented using the same timer as `alarm`, and if the sleep time exceeds the time already on the alarm timer, `sleep` will return immediately after the `SIGALRM` signal is handled. If the sleep time is shorter than the time already on the alarm timer, `sleep` will reset the timer just before it returns so that the `SIGALARM` signal will be received when expected.

Implementation will generally terminate `sleep` when any signal is handled; some, including System V UNIX, supply the number of unslept seconds as the return value of `sleep` (of type `unsigned`).

Some implementations may define these functions as taking arguments of type `unsigned long`. These functions are not part of Draft Proposal ANSI C.

22. C-Ref: Miscellaneous Functions

bsearch	"C-Ref: **BSEARCH**"
ctermid	"C-Ref: **CTERMID, CUSERID**"
cuserid	"C-Ref: **CTERMID, CUSERID**"
getcwd	"C-Ref: **GETCWD, GETWD**"
getenv	"C-Ref: **GETENV, GETLOGIN, GETOPT, PUTENV**"
getlogin	"C-Ref: **GETENV, GETLOGIN, GETOPT, PUTENV**"
getopt	"C-Ref: **GETENV, GETLOGIN, GETOPT, PUTENV**"
getwd	"C-Ref: **GETCWD, GETWD**"
main	"C-Ref: **MAIN**"
putenv	"C-Ref: **GETENV, GETLOGIN, GETOPT, PUTENV**"
qsort	"C-Ref: **QSORT**"

The facilities in this section ways the C programmer can interrogate, and in some cases modify, the environment in which the program is running. They also the function `main`, which programmers must to establish an entry point in their C programs; and the functions `bsearch` and `qsort`, which provide general searching and sorting capabilities.

22.1. C-Ref: MAIN

```
int main()

int main(argc,argv)
  int argc;
  char *argv[];

int main(argc,argv,env)
  int argc;
  char *argv[];
  char *env[];

extern char *environ[];
```

The function `main` is not a library function; it is a function that the programmer defines to designate the entry point of his program and to serve as a vehicle for obtaining information about the program's execution environment. Among all the source files making up a C program, there must be exactly one definition of `main`. Most C implementations, including Draft Proposed ANSI C, permit `main` to be defined with zero or two parameters, customarily called `argc` and `argv`. Some imple-

mentations provide a third parameter, env. The arguments to main are set up by the execution environment and are not directly under control of the C programmer.

The parameter argc is the count of the number of "program arguments" or "options" supplied to the program when it was invoked by a user or another program. Because the first argument is customarily the name of the program itself, argc must be greater than zero.

The parameter argv is a vector of pointers to strings representing the program arguments. argv[0] points to a string which is the name of the program. argv[*i*], where *i*=1,...,argc-1, points the *i*'th program argument. The value of argv[argc] may be a null pointer, but is not so in all implementations.

The parameter env points to a vector of "environment values," each one a pointer to a null-terminated string of the form

"*name*=*value*"

The vector env ends with a null pointer. In some UNIX implementations an external variable, environ, holds the environment pointer instead of it being passed to main as an argument.

For example, the following short program prints out its name and arguments, and its environment:

```
#include <stdio.h>

int main(argc,argv,envp)
  int argc;
  char *argv[],*envp[];
{
  int i;
  printf("Invocation: ");
  for( i=0; i<argc; i++)
    printf("%s ",argv[i]);
  printf("\nEnvironment:\n");
  for( i=0; envp[i] != NULL; i++)
    printf("%s\n",envp[i]);
  return 0;
}
```

22.2. C-Ref: CTERMID, CUSERID

```
#include <stdio.h>                                    /* System V UNIX */

char * ctermid(s)                                    /* System V UNIX */
  char *s;
```

```
char *cuserid(s)                                    /* System V UNIX */
  char *s;

#define L ctermid  n
#define L cuserid  m
```

The function `ctermid` computes a file name that corresponds to the controlling terminal for the current process. This file name can be passed to `fopen` to establish an I/O connection with the terminal. The concept of treating I/O devices like terminals as files is central to UNIX but may be less natural in other systems. However, many C implementations provide a similar facility.

The argument `s` may be null, in which case the terminal file name is stored in an internal buffer whose address is returned as the value of the call. The next call to `ctermid` might overwrite the buffer. If `s` is not null, it is assumed to point to a character array at least `L ctermid` characters long, into which the name is stored. The value of `s` is then returned.

The function `cuserid` retrieves the name of the user who "owns" the current process. The argument `s` may be null, in which case the user name is stored in an internal buffer whose address is returned as the value of the call. The next call to `cuserid` will overwrite the buffer. If `s` is not null, it is assumed to point to a character array at least `L cuserid` characters long, into which the name is stored. The value of `s` is then returned.

22.3. C-Ref: GETCWD, GETWD

```
char *getcwd(buf,size)                             /* System V UNIX */
  char *buf;
  int size;

char *getwd(pathname)                               /* Berkeley UNIX */
  char *pathname;

#include <sys/parm.h>                              /* Berkeley UNIX */
#define MAXPATHLEN ...
```

The functions `getcwd` or `getwd` (depending on the implementation) are used to determine the "current working directory," which is generally the file system directory in which file I/O will take place if a specific directory is not specified in a file name.

The `getcwd` function returns a pointer to the current working directory name. If the argument `buf` is not null, it should point to at least `size` characters of space into which the working directory name will be copied and the address `buf` will be returned by `getcwd`. If the argument `buf` is a null pointer, then `size` bytes of storage are allocated by `malloc`; the working directory name is copied into that storage and its address is returned. In some implementations, `size` must be somewhat

larger than the longest pathname to be returned (e.g., 2 bytes longer). If an error occurs, a null pointer is returned and errno is set.

The getwd function copies into the character array at pathname the working directory name. The array should be at least MAXPATHLEN characters long. If an error occurs, a null pointer is returned and a message is placed in pathname.

22.4. C-Ref: GETENV, GETLOGIN, GETOPT, PUTENV

```
char * getenv( name )
  char * name;

#include <stdlib.h>                                          /* ANSI */
char * getenv( const char *name )                            /* ANSI */
```

The getenv function takes as its single argument a pointer to a string which is interpreted as a "name" understood by the execution environment. The function returns a pointer to another string which is the "value" of the argument name. If the indicated name has no value, a null pointer is returned.

The value string should not be modified by the programmer, and it may be overwritten by a subsequent call to getenv.

The set of (name,value) bindings is also available to the program entry point, the main function.

22.5. C-Ref: BSEARCH

```
#include <stdlib.h>                                          /* ANSI */

char *bsearch( key, base, count, size, compar )
  char *key, *base;
  unsigned count;
  int size;
  int (*compar)();
```

The function bsearch searches an array of count elements whose first element is pointed to by base. The size of each element in characters is specified by size. compar is a function that takes as arguments pointers to two elements and returns -1 if the first element is "less than" the second, 1 if the first element is "greater than" the second, and 0 if the two elements are equal. The array is assumed to be sorted in ascending order (according to compar) at the beginning of the search. bsearch returns a pointer to the element of the array that matches the element pointed to by key, or NULL if no such element is found.

In Draft Proposed ANSI C, key and base have type void *; count and size have type size t; bsearch returns a pointer of type void *; and compar has type

```
int (*compar)(const void *, const void *)
```

22.6. C-Ref: QSORT

```
#include <stdlib.h>                                          /* ANSI */

void qsort( base, count, size, compar )
  char *base;
  unsigned count;
  int size;
  int (*compar)();
```

The function qsort sorts an array of count elements whose first element is pointed to by base. The size of each element in characters is specified by size. compar is a function that takes as arguments pointers to two elements and returns -1 if the first element is "less than" the second, 1 if the first element is "greater than" the second, and 0 if the two elements are equal. The array will be sorted in ascending order (according to compar) at the end of the sort.

In Draft Proposed ANSI C, base has type void *; count and size have type size t; and compar has type

```
int (*compar)(const void *, const void *)
```

PART IV.

C-REF: THE ASCII CHARACTER SET

The ASCII character set is the one most commonly used by C implementations. In the following table, each column gives an ASCII character and its octal, decimal, and hexadecimal value.

```
Oct  Dec   Hex Character
 00    0   0x0  '\0'
 01    1   0x1  '\001'
 02    2   0x2  '\002'
 03    3   0x3  '\003'
 04    4   0x4  '\004'
 05    5   0x5  '\005'
 06    6   0x6  '\006'
 07    7   0x7  '\007'
010    8   0x8  '\b'
011    9   0x9  '\i'
012   10   0xA  '\n'
013   11   0xB  '\v'
014   12   0xC  '\f'
015   13   0xD  '\r'
016   14   0xE  '\016'
017   15   0xF  '\017'
020   16  0x10  '\020'
021   17  0x11  '\021'
022   18  0x12  '\022'
023   19  0x13  '\023'
024   20  0x14  '\024'
025   21  0x15  '\025'
026   22  0x16  '\026'
027   23  0x17  '\027'
030   24  0x18  '\030'
031   25  0x19  '\031'
032   26  0x1A  '\032'
033   27  0x1B  '\033'
034   28  0x1C  '\034'
035   29  0x1D  '\035'
036   30  0x1E  '\036'
037   31  0x1F  '\037'
```

```
Oct  Dec   Hex Character
040   32  0x20  ' ' space
041   33  0x21  '!'
042   34  0x22  '"'
043   35  0x23  '#'
044   36  0x24  '$'
045   37  0x25  '%'
046   38  0x26  '&'
047   39  0x27  '\''
050   40  0x28  '('
051   41  0x29  ')'
052   42  0x2A  '*'
053   43  0x2B  '+'
```

```
054   44  0x2C  ’,’
055   45  0x2D  ’-’
056   46  0x2E  ’.’
057   47  0x2F  ’/’
060   48  0x30  ’0’
061   49  0x31  ’1’
062   50  0x32  ’2’
063   51  0x33  ’3’
064   52  0x34  ’4’
065   53  0x35  ’5’
066   54  0x36  ’6’
067   55  0x37  ’7’
070   56  0x38  ’8’
071   57  0x39  ’9’
072   58  0x3A  ’|’
073   59  0x3B  ’;’
074   60  0x3C  ’<’
075   61  0x3D  ’=’
076   62  0x3E  ’>’
077   63  0x3F  ’?’


 Oct  Dec   Hex Character
0100   64  0x40  ’@’
0101   65  0x41  ’A’
0102   66  0x42  ’B’
0103   67  0x43  ’C’
0104   68  0x44  ’D’
0105   69  0x45  ’E’
0106   70  0x46  ’F’
0107   71  0x47  ’G’
0110   72  0x48  ’H’
0111   73  0x49  ’I’
0112   74  0x4A  ’J’
0113   75  0x4B  ’K’
0114   76  0x4C  ’L’
0115   77  0x4D  ’M’
0116   78  0x4E  ’N’
0117   79  0x4F  ’O’
0120   80  0x50  ’P’
0121   81  0x51  ’Q’
0122   82  0x52  ’R’
0123   83  0x53  ’S’
0124   84  0x54  ’T’
0125   85  0x55  ’U’
0126   86  0x56  ’V’
0127   87  0x57  ’W’
0130   88  0x58  ’X’
```

```
0131   89  0x59  'Y'
0132   90  0x5A  'Z'
0133   91  0x5B  '['
0134   92  0x5C  '\\'
0135   93  0x5D  ']'
0136   94  0x5E  '^'
0137   95  0x5F  ' '
```

Oct	Dec	Hex	Character
0100	64	0x40	’@’
0101	65	0x41	’A’
0102	66	0x42	’B’
0103	67	0x43	’C’
0104	68	0x44	’D’
0105	69	0x45	’E’
0106	70	0x46	’F’
0107	71	0x47	’G’
0110	72	0x48	’H’
0111	73	0x49	’I’
0112	74	0x4A	’J’
0113	75	0x4B	’K’
0114	76	0x4C	’L’
0115	77	0x4D	’M’
0116	78	0x4E	’N’
0117	79	0x4F	’O’
0120	80	0x50	’P’
0121	81	0x51	’Q’
0122	82	0x52	’R’
0123	83	0x53	’S’
0124	84	0x54	’T’
0125	85	0x55	’U’
0126	86	0x56	’V’
0127	87	0x57	’W’
0130	88	0x58	’X’
0131	89	0x59	’Y’
0132	90	0x5A	’Z’
0133	91	0x5B	’[’
0134	92	0x5C	’\\’
0135	93	0x5D	’]’
0136	94	0x5E	’^’
0137	95	0x5F	’ ’

Oct	Dec	Hex	Character
0140	96	0x60	’‘’
0141	97	0x61	’a’
0142	98	0x62	’b’
0143	99	0x63	’c’
0144	100	0x64	’d’
0145	101	0x65	’e’
0146	102	0x66	’f’
0147	103	0x67	’g’
0150	104	0x68	’h’
0151	105	0x69	’i’
0152	106	0x6A	’j’

```
0153  107  0x6B  'k'
0154  108  0x6C  'l'
0155  109  0x6D  'm'
0156  110  0x6E  'n'
0157  111  0x6F  'o'
0160  112  0x70  'p'
0161  113  0x71  'q'
0162  114  0x72  'r'
0163  115  0x73  's'
0164  116  0x74  't'
0165  117  0x75  'u'
0166  118  0x76  'v'
0167  119  0x77  'w'
0170  120  0x78  'x'
0171  121  0x79  'y'
0172  122  0x7A  'z'
0173  123  0x7B  '{'
0174  124  0x7C  '|'
0175  125  0x7D  '}'
0176  126  0x7E  '~'
0177  127  0x7F  '\177'
```

PART V.

C-REF: SYNTAX OF THE C LANGUAGE

This appendix contains a sorted copy of the lexical rules and the syntax for traditional and Draft Proposed ANSI C as it is presented in the text. Lexical rules are identified by the appearance of "(LEXICAL)" in their definition.

abstract-declarator :
    empty-abstract-declarator
    nonempty-abstract-declarator

add-op : one of
    `+  -`

additive-expression :
    multiplicative-expression
    additive-expression add-op multiplicative-expression

address-expression :
    `&` *unary-expression*

array-declarator :
    declarator `[` *constant-expression*$_{opt}$ `]`

assignment-expression :
    conditional-expression
    unary-expression assignment-op assignment-expression

assignment-op : one of
    `=   +=   -=   *=   /=   %=`
    `<<=   >>=   &=   ^=   =`

bit-field :   *declarator*$_{opt}$ `:` *width*

bitwise-and-expression :
    equality-expression
    bitwise-and-expression `&` *equality-expression*

bitwise-negation-expression :
    `~` *unary-expression*

bitwise-or-expression :
    bitwise-xor-expression
    bitwise-or-expression `|` *bitwise-xor-expression*

bitwise-xor-expression :
    bitwise-and-expression
    bitwise-xor-expression `^` *bitwise-and-expression*

break-statement :
    `break ;`

case-label :
    `case` *constant-expression*

cast-expression :
  (*type-name*) *unary-expression*

character : (LEXICAL)
  printing-character
  escape-character

character-constant : (LEXICAL)
  ' *character* '

character-escape-code : *one of* (LEXICAL)
  `n t b r f v \ ' "`

character-escape-code : *one of* (Draft Proposed ANSI C) (LEXICAL)
  `a n t b r f`
  `v \ ' " ?`

character-sequence : (LEXICAL)
  character
  character-sequence character

character-type-specifier :
  `unsigned`$_{opt}$ `char`

character-type-specifier : (Draft Proposed ANSI C)
  `char`
  `signed char`
  `unsigned char`

comma-expression :
  assignment-expression
  comma-expression , *assignment-expression*

component-declaration :
  type-specifier *component-declarator-list* ;

component-declarator :
  simple-component
  bit-field

component-declarator-list :
  component-declarator
  component-declarator-list , *component-declarator*

component-selection-expression :
  direct-component-selection
  indirect-component-selection

compound-statement :
  { *inner-declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

conditional-expression :
  logical-or-expression
  logical-or-expression ? *expression* : *conditional-expression*

conditional-statement :
  if-statement
  if-else-statement

conditional-statement :
  if-statement
  if-else-statement

const-type-specifier : (Draft Proposed ANSI C)
  const

constant : (LEXICAL)
  integer-constant
  floating-point-constant
  character-constant
  string-constant

constant-expression :
  expression

continue-statement :
  `continue ;`

decimal-constant : (LEXICAL)
  nonzero-digit
  decimal-constant digit

declaration :
  declaration-specifiers declarator-list ;

declaration-list :
  declaration
  declaration-list declaration

declaration-specifiers :
  storage-class-specifier
  type-specifier
  declaration-specifiers storage-class-specifier
  declaration-specifiers type-specifier

declarator :
    simple-declarator
    `(` *declarator* `)`
    function-declarator
    array-declarator
    pointer-declarator

default-label :
    `default`

digit : one of (LEXICAL)
    `0 1 2 3 4 5 6 7 8 9`

digit-sequence : (LEXICAL)
    digit
    digit digit-sequence

direct-component-selection :
    postfix-expression `.` *name*

do-statement :
    `do` *statement* `while` `(` *expression* `)` `;`

dotted-digits : (LEXICAL)
    digit-sequence `.`
    digit-sequence `.` *digit-sequence*
    `.` *digit-sequence*

empty-abstract-declarator :

enumeration-constant `=` *expression*

enumeration-constant :
    identifier

enumeration-constant-definition :
    enumeration-constant
    enumeration-constant `=` *expression*

enumeration-definition-list :
    enumeration-constant-definition
    enumeration-definition-list `,` *enumeration-constant-definition*

enumeration-tag :
    identifier

enumeration-type-definition :

enum *enumeration-tag*$_{opt}$
{ *enumeration-definition-list* }

enumeration-type-reference :
enum *enumeration-tag*

enumeration-type-specifier :
enumeration-type-definition
enumeration-type-reference

equality-expression :
relational-expression
equality-expression equality-op relational-expression

equality-op : one of
== !=

escape-character : (LEXICAL)
\ *escape-code*

escape-code : (LEXICAL)
character-escape-code
numeric-escape-code

exponent : (LEXICAL)
e *sign-part*$_{opt}$ *digit-sequence*
E *sign-part*$_{opt}$ *digit-sequence*

expression :
comma-expression

expression-list :
assignment-expression
expression-list , *assignment-expression*

expression-statement :
expression ;

field-list :
component-declaration
field-list component-declaration

float-marker : one of (Draft Proposed ANSI C) (LEXICAL)
f F l L

floating-constant : (LEXICAL)
digit-sequence exponent

dotted-digits exponent$_{opt}$

floating-constant : (Draft Proposed ANSI C) (LEXICAL)
digit-sequence exponent float-marker$_{opt}$
dotted-digits exponent$_{opt}$ *float-marker*$_{opt}$

floating-type-specifier :
`float`
`long float`
`double`

floating-type-specifier : (Draft Proposed ANSI C)
`float`
`double`
`long double`

following-character : (LEXICAL)
letter
underscore
digit

for-expressions :
(*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$)

for-statement :
`for` *for-expressions statement*

function-body :
parameter-declaration-list$_{opt}$ *compound-statement*

function-call :
postfix-expression (*expression-list*$_{opt}$)

function-declarator :
declarator (*parameter-list*$_{opt}$)

function-declarator : (Draft Proposed ANSI C)
declarator (*parameter-type-list* ,...$_{opt}$)
declarator (*parameter-list*$_{opt}$)

function-definition :
declaration-specifiers$_{opt}$ *declarator function-body*

goto-statement :
`goto` *identifier* ;

hex-digit : one of (LEXICAL)

`0 1 2 3 4 5 6 7 8 9`
`A B C D E F a b c d e f`

hex-marker : one of (LEXICAL)
`0x 0X`

hexadecimal-constant : (LEXICAL)
hex-marker
hexadecimal-constant hex-digit

identifier : (LEXICAL)
underscore
letter
identifier following-character

if-else-statement :
`if` (*expression*) *statement* `else` *statement*

if-statement :
`if` (*expression*) *statement*

indirect-component-selection :
postfix-expression `->` *name*

indirection-expression :
`*` *unary-expression*

initialized-declaration :
declaration-specifiers initialized-declarator-list ;

initialized-declarator :
declarator $initializer\text{-}part_{opt}$

initialized-declarator-list :
initialized-declarator
initialized-declarator-list , *initialized-declarator*

initializer :
expression
{ *initializer-list* $,_{opt}$ }

initializer-list :
initializer
initializer-list , *initializer*

initializer-part :
`'='` *initializer*

inner-declaration-list :
    initialized-declaration-list

integer-constant : (LEXICAL)
    decimal-constant *type-marker* $_{opt}$
    octal-constant *type-marker* $_{opt}$
    hexadecimal-constant *type-marker* $_{opt}$

integer-type-specifier :
    signed-type-specifier
    unsigned-type-specifier
    character-type-specifier

iterative-statement :
    while-statement
    do-statement
    for-statement

label :
    named-label
    case-label
    default-label

labeled-statement :
    label `:` *statement*

letter : one of (LEXICAL)

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

logical-and-expression :
    bitwise-or-expression
    logical-and-expression `&&` *bitwise-or-expression*

logical-negation-expression :
    `!` *unary-expression*

logical-or-expression :
    logical-and-expression
    logical-or-expression `||` *logical-and-expression*

long-marker : one of (Draft Proposed ANSI C) (LEXICAL)
    `l L`

mult-op : one of

`*  /  %`

multiplicative-expression :
unary-expression
multiplicative-expression   mult-op   cast-expression

named-label :
identifier

nonempty-abstract-declarator :
`(` *nonempty-abstract-declarator* `)`
abstract-declarator `( )`
abstract-declarator `[` *expression*$_{opt}$ `]`
`*` *abstract-declarator*

nonzero-digit : one of
(LEXICAL)
`1  2  3  4  5  6  7  8  9`

null-statement :
`;`

numeric-escape-code : (LEXICAL)
octal-digit
octal-digit   octal-digit
octal-digit   octal-digit   octal-digit
`x` *hex-digit*   (Draft Proposed ANSI C)
`x` *hex-digit hex-digit*   (Draft Proposed ANSI C)
`x` *hex-digit hex-digit hex-digit*   (Draft Proposed ANSI C)

octal-constant :   (LEXICAL)
`0`
octal-constant   octal-digit

octal-digit : one of (LEXICAL)
`0  1  2  3  4  5  6  7`

parameter-declaration : (Draft Proposed ANSI C)
declaration-specifiers   declarator
declaration-specifiers   abstract-declarator

parameter-declaration-list :
declaration-list

parameter-list :

identifier
parameter-list , *identifier*

parameter-type-list : (Draft Proposed ANSI C)
parameter-declaration
parameter-type-list , *parameter-declaration*

parenthesized-expression :
(*expression*)

pointer-declarator :
* *declarator*

pointer-declarator : (Draft Proposed ANSI C)
* *type-specifier-list*$_{opt}$ *declarator*

postdecrement-expression :
postfix-expression --

postfix-expression :
primary-expression
subscript-expression
component-selection-expression
function-call
postincrement-expression
postdecrement-expression

postincrement-expression :
postfix-expression ++

predecrement-expression :
-- *unary-expression*

preincrement-expression :
++ *unary-expression*

primary-expression :
name
literal
parenthesized-expression

program :
top-level-declaration-list$_{opt}$

relational-expression :
shift-expression

relational-expression relational-op shift-expression

relational-op : one of

`<` `<=` `>` `>=`

return-statement :

`return` *expression*$_{opt}$ `;`

shift-expression :

additive-expression
shift-expression shift-op additive-expression

shift-op : one of

`<<` `>>`

signed-type-specifier :

`short int`$_{opt}$
`int`
`long int`$_{opt}$

signed-type-specifier : (Draft Proposed ANSI C)

`signed`
`signed`$_{opt}$ `int`
`signed`$_{opt}$ `short int`$_{opt}$
`signed`$_{opt}$ `long int`$_{opt}$

simple-component :

declarator

simple-declarator :

identifier

sizeof-expression :

`sizeof ( ` *type-name* ` )`
`sizeof` *unary-expression*

statement :

expression-statement
labeled-statement
compound-statement
conditional-statement
iterative-statement
switch-statement
break-statement
continue-statement
return-statement

goto-statement
null-statement

statement-list :
statement
statement-list statement

storage-class-specifier : one of
`auto extern register static typedef`

string-constant : (LEXICAL)
" *character-sequence*$_{opt}$ "

structure-tag :
identifier

structure-type-definition :
res[struct] structure-tag$_{opt}$ { *field-list* }

structure-type-reference :
`struct` *structure-tag*

structure-type-specifier :
structure-type-definition
structure-type-reference

subscript-expression :
postfix-expression `[` *expression* `]`

switch-statement :
`switch` (*expression*) *statement*

top-level-declaration :
initialized-declaration
function-definition

top-level-declaration-list :
top-level-declaration
top-level-declaration-list top-level-declaration

type-marker : one of (LEXICAL)
`l L`

type-marker : (Draft Proposed ANSI C) (LEXICAL)
long-marker unsigned-marker$_{opt}$
unsigned-marker long-marker$_{opt}$

type-name :
    type-specifier abstract-declarator

type-specifier :
    const-type-specifier (Draft Proposed ANSI C)
    enumeration-type-specifier
    floating-point-type-specifier
    integer-type-specifier
    structure-type-specifier
    typedef-name
    union-type-specifier
    void-type-specifier
    volatile-type-specifier (Draft Proposed ANSI C)

type-specifier-list : (Draft Proposed ANSI C)
    type-specifier-list type-specifier

typedef-name :
    identifier

unary-expression :
    postfix-expression
    cast-expression
    sizeof-expression
    unary-minus-expression
    unary-plus-expression (Draft Proposed ANSI C)
    logical-negation-expression
    bitwise-negation-expression
    address-expression
    indirection-expression
    preincrement-expression
    predecrement-expression

unary-minus-expression :
    - *unary-expression*

unary-plus-expression : (Draft Proposed ANSI C)
    + *unary-expression*

underscore : (LEXICAL)

union-tag :
    identifier

union-type-definition :
`union` *union-tag*$_{opt}$ `{` *field-list* `}`

union-type-reference :
`union` *union-tag*

union-type-specifier :
union-type-definition
union-type-reference

unsigned-marker : one of (Draft Proposed ANSI C) (LEXICAL)
`u U`

unsigned-type-specifier :
`unsigned short int`$_{opt}$
`unsigned int`$_{opt}$
`unsigned long int`$_{opt}$

void-type-specifier :
`void`

volatile-type-specifier : (Draft Proposed ANSI C)
volatile

while-statement :
`while (` *expression* `)` *statement*

width :
expression

Index

! logical negation, 147
!= not equal, 158
" string constants, 236
" string constants, 24
preprocessor command, 29, 48
#define preprocessor command, 31
#elif preprocessor command, 239
#elif preprocessor command, 43
#else preprocessor command, 42
#endif preprocessor command, 42
#error preprocessor command, 239
#ifdef preprocessor command, 45
#ifndef preprocessor command, 45
#if preprocessor command, 42
#include preprocessor command, 41, 238
#line preprocessor command, 48
#pragma preprocessor command, 239
#undef preprocessor command, 37
#undef preprocessor command, 45
$ character, 11, 16
%= assign remainder, 172
% remainder, 151
& , 98
& address operator, 59, 71, 87, 98, 100, 148, 258
& bitwise and, 159
&& logical and, 168
&= assign bitwise AND, 172
* indirection operator, 87, 138
*= assign product, 172
* multiplication, 151
+ addition, 153, 259
++ increment, 143, 149
+= assign sum, 172
- subtraction, 153, 259
- unary minus, 147
- unary plus, 258
. selection operator, 95
/= assign quotient, 172
/ division, 151
0 null pointer, 87

< less than, 157
<< left shift, 155
<<= assign left shift, 172
<= less or equal, 157
-= assign difference, 172
= assignment, 171
== equal, 158
> greater than, 157
-> selection operator'), 98
>= greater or equal, 157
>> right shift, 155
>>= assign right shift, 172
-> structures, 98
? conditional, 169
@ character, 11
@t[*] indirection operator, 149
@t[sizeof] operator applied to arrays, 92
@t[typedef] storage class, 110
abort facility, 357
abs facility, 342
absolute value functions, 342
abstract data type, 95
abstract declarators, 114
acos facility, 347
Ada, 187, 198
addition expressions, 153, 259
addition pointers, 153, 259
additive expressions, 153, 259
address expression, 148, 258
addressing structure, 118
adjustments to type formal parameters, 212
advice on defining external names, 78
aggregate types, 79
agreement of parameters functions, 215
agreement of return values functions, 216
alarm facility, 361
Algol 60, 7, 187
alignment of structures, 102
alignment of unions, 104
alignment restrictions, 119
allocation of storage, 223
American National Standards Institute (ANSI), 8, 233
ANSI C, 233
ANSI C character escapes, 236
ANSI C character set, 233
ANSI C constants, 234

ANSI C floating point constants, 235
ANSI C identifiers, 234
ANSI C libraries, 266
ANSI C line continuation, 233
ANSI C predefined macros, 238
ANSI C type specifiers, 241
`arctan` function, 347
argument conversions, 130, 257
argument conversions functions, 130, 141, 257
argument usual conversions, 130, 141, 257
arithmetic exceptions, 134
arithmetic types, 79
array bounds, 91
array declarators, 65
array initializers, 72
arrays and pointers, 90
arrays subscripting, 90
array to pointer conversions, 71, 90, 126
array types, 90
ASCII character set, 370
`asctime` facility, 351
`asin` facility, 347
asm reserved word, 18
`assert` facility, 355
`assert.h` header files, 355
assignment expressions, 170
assignment operators, 15
assignment usual conversions, 127, 255
associativity expressions, 132
associativity of binary expressions, 151
associativity of expressions, 132
`atan` facility, 347
`atan2` facility, 347
`atof` facility, 295
`atoi` facility, 295
`atol` facility, 295
`auto` storage class, 59
automatic variables, 56
auto storage class specifier, 59
backspace character, 11, 12
`bcmp` facility, 298
BCPL, 7
`bcpy` facility, 298
Bell Laboratories, 7
big endian computers, 118
binary expressions, 151
binary streams, 301

binary usual conversions, 129, 256
bit fields, 100, 248
bit fields alignment, 100
bit fields portability, 100, 102
bit fields size, 100
bit fields structures, 100
bitwise and expressions, 159
bitwise expressions, 147, 159, 160, 161
bitwise expressions portability, 147, 159, 248
bitwise or expressions, 161
bitwise xor expressions, 160
B language, 7
blank character, 11
blocks, 50, 186
block statements, 186
body switch statement, 186
bounds of arrays, 91
break, 203
`break` statements, 202
Brian Kernighan, 8
`bsearch` facility, 366
buffered I/O, 301, 304
`BUFSIZ` value, 304
byte, 117
byte order, 23, 118
byte order portability, 118
`bzero` facility, 299
calendar time conversions, 352
call-by-value, 214
calling functions, 141, 214, 258
`calloc` facility, 337
carriage return character, 11, 12
case, 200
`case` labels, 198
`()` cast, 145
cast expressions, 114
cast expressions), 145
categories of types, 79
`ceil` facility, 343
`cfree` facility, 338
`char` type specifiers, 83, 248
'' character constants, 23
character constants, 23
character sets, 11, 12
character types, 83, 248
`clalloc` facility, 337
C language standardization, 8, 233

`clearerr` facility, 334
`CLK TCK` facility, 349
`clock` facility, 349
`clock t` facility, 349
comma expression, 138, 141, 173
comments, 14
COMMON, 77
comparison pointers, 157, 158, 259
compatibility assignment, 127
compile-time objects, 58
Compiling a C program, 9
component names overloading, 98
components, 95
component selection expressions, 140, 257
component selection structures, 98
components of structures, 95
components of unions, 95
components selection, 140
components structures, 95, 98
components unions, 103
compostion of declarators, 68
compound statements, 186
concatenation of strings, 236
concatenation strings, 236
: conditional, 169
conditional compilation, 42
conditional expressions, 169
conditional statement, 188
conflicting declarations, 55
`const` reserved word, 234
`const` type specifier, 249
constant expressions, 174, 259
constant expressions in initializers, 71
constant expressions initializers, 174
constant expressions in preprocessor commands, 47
constant expressions portability, 174, 259
constant expressions preprocessor, 174
constants strings, 236
continuation of preprocessor commands, 30
continuation of source lines, 12, 24, 233
continuation of string constants, 24, 236
continue, 203
`continue` statements, 202
control expressions, 184
control functions, 355
control library functions, 355

conversions, 122, 254
conversions between pointers, 89, 255
conversions during assignment, 127, 255
conversions during cast expressions, 127
conversions expressions, 128, 129, 255, 256
conversions functions to pointers>), 71
conversions to array, 126
conversions to arrays pointers, 90
conversions to function, 126
conversions to pointer arrays, 90
conversions to pointers arrays, 126
conversions to pointer strings, 71
conversions to void, 127
conversions unsigned, 123
conversion to pointer arrays, 71
conversion to pointer integers, 71
conversion to pointers functions, 126
conversion types, 122, 254
`cos` facility, 346
`cosh` facility, 347
CPL, 7
creating identifiers with preprocessor, 40
C-Ref: **ABS, FABS, LABS**, 342
C-Ref: **ACOS, ASIN, ATAN, ATAN2**, 347
C-Ref: Additive Operators, 153
C-Ref: Addressing a 32-Bit Integer At Address A, 118
C-Ref: Addressing Structure and Byte Ordering, 118
C-Ref: Address Operator, 148
C-Ref: Adjustments to Parameter Types, 212
C-Ref: Advice, 78
C-Ref: Agreement of Actual and Declared Return Type, 216
C-Ref: Agreement of Formal and Actual Parameters, 215
C-Ref: Alignment Restrictions, 119
C-Ref: Allocating and Deallocating Stacks, 223
C-Ref: A Note on Implementation of Typedef Names, 112
C-Ref: An Overview of C Programming, 9
C-Ref: ANSI C #include, 238
C-Ref: ANSI C Addition and Subtraction, 259
C-Ref: ANSI C Address Operator, 258
C-Ref: ANSI C Assignment Conversions, 255
C-Ref: ANSI C Character Escape Codes, 236
C-Ref: ANSI C Character Sets, 233
C-Ref: ANSI C Component Selection, 257

C-Ref: ANSI C const, 249
C-Ref: ANSI C Constant Expressions, 259
C-Ref: ANSI C Conversions and Representations, 254
C-Ref: ANSI C Declarations, 240
C-Ref: ANSI C Declarators, 242
C-Ref: ANSI C Expressions, 257
C-Ref: ANSI C External Names, 247
C-Ref: ANSI C Floating-point Characteristics, 254
C-Ref: ANSI C Floating Point Constants, 235
C-Ref: ANSI C Floating-point Types, 249
C-Ref: ANSI C Forward References to Structures, 241
C-Ref: ANSI C Function Calls, 258
C-Ref: ANSI C Function Prototypes, 243
C-Ref: ANSI C Generic Pointers, 253
C-Ref: ANSI C Identifiers, 234
C-Ref: ANSI C Initializers, 246
C-Ref: ANSI C Integer Constants, 234
C-Ref: ANSI C Integer Types, 248
C-Ref: ANSI C Lexical Elements, 233
C-Ref: ANSI C Lexical Structure, 237
C-Ref: ANSI C Macro Definition and Expansion, 239
C-Ref: ANSI C Minimum Integer Sizes, 254
C-Ref: ANSI C New Commands, 239
C-Ref: ANSI C Number Representation, 254
C-Ref: ANSI C Predefined Macros, 238
C-Ref: ANSI C Preprocessor, 237
C-Ref: ANSI C Relational Expressions, 259
C-Ref: ANSI C Reserved Words, 234
C-Ref: ANSI C Run-time Library, 260
C-Ref: ANSI C Scopes and Name Spaces, 240
C-Ref: ANSI C `sizeof` Operator, 258
C-Ref: ANSI C Statements, 260
C-Ref: ANSI C String Constants, 236
C-Ref: ANSI C Stringization and Merging of Tokens, 237
C-Ref: ANSI C The Function Argument Conversions, 257
C-Ref: ANSI C The Usual Binary Conversions, 256
C-Ref: ANSI C The Usual Unary Conversions, 255
C-Ref: ANSI C Types, 247
C-Ref: ANSI C Type Specifiers, 241
C-Ref: ANSI C Unary Plus Operator, 258
C-Ref: ANSI C Usual Conversions in an Example Signed Implementation, 256
C-Ref: ANSI C volatile, 250

C-Ref: A Package for Manipulating Sets of Integers (1), 162
C-Ref: A Package for Manipulating Sets of Integers (2), 163
C-Ref: A Package for Manipulating Sets of Integers (3), 164
C-Ref: A Package for Manipulating Sets of Integers (4), 165
C-Ref: A Program for Enumerating Subsets of a Given Set, 165
C-Ref: Array Bounds, 91
C-Ref: Array Declarators, 65
C-Ref: Arrays, 72
C-Ref: Arrays and Pointers, 90
C-Ref: Array Types, 90
C-Ref: **ASCTIME, CTIME**, 351
C-Ref: **ASSERT, NDEBUG**, 355
C-Ref: Assignment Expressions, 170
C-Ref: **ATOF, ATOI, ATOL**, 295
C-Ref: Binary Operator Expressions, 151
C-Ref: Bit Fields, 100
C-Ref: Bitwise AND Operator, 159
C-Ref: Bitwise Negation, 147
C-Ref: Bitwise OR Operator, 161
C-Ref: Bitwise XOR Operator, 160
C-Ref: Break and Continue Statements, 202
C-Ref: **BSEARCH**, 366
C-Ref: Casts, 145
C-Ref: **CEIL, FLOOR, FMOD**, 343
C-Ref: Character Constants, 23
C-Ref: Character Encodings, 13
C-Ref: Character Escape Codes, 25
C-Ref: Character Processing, 281
C-Ref: Character Set, 11
C-Ref: Character Type, 83
C-Ref: **CLOCK, CLOCK_T, CLK_TCK, TIMES**, 349
C-Ref: Comments, 14
C-Ref: Compiler Optimization of Memory Accesses, 179
C-Ref: Compile-time Objects, 58
C-Ref: Components, 98
C-Ref: Component Selection, 140
C-Ref: Composition of Declarators, 68
C-Ref: Compound Assignment, 172
C-Ref: Compound Statement, 186
C-Ref: Computing the Greatest Common Divisor, 156

C-Ref: Conditional Compilation, 42
C-Ref: Conditional Expressions, 169
C-Ref: Conditional Statement, 188
C-Ref: Constant Expressions, 174
C-Ref: Constant Expressions in Conditional Commands, 47
C-Ref: Constants, 18
C-Ref: Control Expressions, 184
C-Ref: Control Functions, 355
C-Ref: Conventions for Identifiers, 17
C-Ref: Conversions, 122
C-Ref: Conversions and Representations, 117
C-Ref: Conversions to Array and Function Types, 126
C-Ref: Conversions to Enumeration Types, 126
C-Ref: Conversions to Floating-point Types, 125
C-Ref: Conversions to Integer Types, 123
C-Ref: Conversions to Pointer Types, 126
C-Ref: Conversions to Structure and Union Types, 125
C-Ref: Conversions to the Void Type, 127
C-Ref: Converting Tokens to Strings, 39
C-Ref: C Operators in Order of Precedence, 133
C-Ref: **COSH, SINH, TANH**, 347
C-Ref: **COS, SIN, TAN**, 346
C-Ref: **CTERMID, CUSERID**, 364
C-Ref: Data Structures, 221
C-Ref: Declarations, 49
C-Ref: Declarations Within Compound Statements, 186
C-Ref: Declarators, 64
C-Ref: Default Storage Class Specifiers, 60
C-Ref: Default Type Specifiers, 62
C-Ref: Defining Macros with Parameters, 33
C-Ref: Definition and Replacement, 31
C-Ref: Designing the Stack Module, 220
C-Ref: Difficult Addressing Models, 121
C-Ref: **DIFFTIME**, 353
C-Ref: Discarded Values, 178
C-Ref: **DIV, LDIV**, 343
C-Ref: Do Statement, 192
C-Ref: Draft Proposed ANSI C, 233
C-Ref: Draft Proposed ANSI C Facilities, 266
C-Ref: Draft Proposed ANSI C Libraries (Part 1), 267
C-Ref: Draft Proposed ANSI C Libraries (Part 2), 269
C-Ref: Duplicate Declarations, 55

C-Ref: Duplicate Visibility, 56
C-Ref: Eliding Braces, 75
C-Ref: Enumerations, 73
C-Ref: Enumeration, Structure, and Union Types, 113
C-Ref: Enumeration Types, 92
C-Ref: EOF, 303
C-Ref: Equality Operators, 158
C-Ref: **ERRNO, STRERROR, PERROR**, 274
C-Ref: Escape Characters, 25
C-Ref: Examples of Output Formatting (Part 1), 330
C-Ref: Examples of Output Formatting (Part 2), 331
C-Ref: Examples of Storage Class Specifiers, 60
C-Ref: **EXEC, SYSTEM**, 356
C-Ref: Execution Character Set, 12
C-Ref: **EXIT, ABORT**, 357
C-Ref: Explicit Line Numbering, 48
C-Ref: **EXP, LOG, LOG10**, 344
C-Ref: Expressions, 131
C-Ref: Expressions and Precedence, 132
C-Ref: Expression Statements, 184
C-Ref: Extent, 56
C-Ref: External Names, 58, 76
C-Ref: **FEOF, FERROR, CLEARERR**, 334
C-Ref: **FGETC, GETC, GETCHAR, UNGETC**, 307
C-Ref: **FGETS, GETS**, 308
C-Ref: File Inclusion, 41
C-Ref: Floating-point, 70
C-Ref: Floating-point Constants, 21
C-Ref: Floating-Point Types, 86
C-Ref: **FOPEN, FCLOSE, FFLUSH, FREOPEN**, 303
C-Ref: Formal Parameter Declarations, 211
C-Ref: For Statement, 193
C-Ref: Forward References, 53
C-Ref: **FPRINTF, PRINTF, SPRINTF**, 319
C-Ref: **FPUTC, PUTC, PUTCHAR**, 318
C-Ref: **FPUTS, PUTS**, 318
C-Ref: **FREAD, FWRITE**, 333
C-Ref: **FREE, CFREE**, 338
C-Ref: **FREXP, LDEXP, MODF**, 344
C-Ref: **FSCANF, SCANF, SSCANF**, 309
C-Ref: **FSEEK, FTELL, REWIND**, 306
C-Ref: Function Calls, 141
C-Ref: Function Declarators, 67
C-Ref: Function Definitions, 209

C-Ref: Function Return Types, 216
C-Ref: Functions, 209
C-Ref: Function Types, 107
C-Ref: General Comments, 131
C-Ref: General Syntactic Rules for Statements, 183
C-Ref: **GETCWD, GETWD**, 365
C-Ref: **GETENV, GETLOGIN, GETOPT, PUTENV**, 366
C-Ref: **GMTIME, LOCALTIME, MKTIME**, 352
C-Ref: Goto Statement and Named Labels, 206
C-Ref: Identifiers, 16
C-Ref: Implicit Declarations, 76
C-Ref: Indirection, 149
C-Ref: Initializers, 69
C-Ref: Initial Values, 57
C-Ref: Input/Output Facilities, 301
C-Ref: Integer Constants, 19
C-Ref: Integer Constants Table, 21
C-Ref: Integers, 70
C-Ref: Integer Types, 80
C-Ref: Introduction, 7
C-Ref: Introduction to the Libraries, 265
C-Ref: **ISALNUM, ISALPHA, ISASCII, ISCNTRL**, 282
C-Ref: **ISCSYM, ISCSYMF**, 283
C-Ref: **ISDIGIT, ISODIGIT, ISXDIGIT**, 283
C-Ref: **ISGRAPH, ISPRINT, ISPUNCT**, 284
C-Ref: **ISLOWER, ISUPPER**, 284
C-Ref: **ISSPACE, ISWHITE**, 285
C-Ref: Iterative Statements, 190
C-Ref: Labeled Statements, 185
C-Ref: Lexical Elements, 11
C-Ref: Literals, 137
C-Ref: Logical AND Operator, 168
C-Ref: Logical Negation, 147
C-Ref: Logical Operator Expressions, 167
C-Ref: Logical OR Operator, 168
C-Ref: **MAIN**, 363
C-Ref: Main Programs, 217
C-Ref: **MALLOC, CALLOC, MLALLOC, CLALLOC**, 337
C-Ref: Mathematical Functions, 341
C-Ref: **MEMCHR**, 297
C-Ref: **MEMCMP, BCMP**, 298
C-Ref: **MEMCPY, MEMCCPY, MEMMOVE, BCPY**, 298
C-Ref: Memory Functions, 297

C-Ref: **MEMSET, BZERO**, 299
C-Ref: Miscellaneous Functions, 363
C-Ref: Missing Declarators, 63
C-Ref: Mixed Common Model, 77
C-Ref: Modularization, 219
C-Ref: More About Array Types, 113
C-Ref: More About Typedef Names, 113
C-Ref: Multidimensional Arrays, 91
C-Ref: Multiple Control Variables, 197
C-Ref: Multiplicative Operators, 151
C-Ref: Multiway Conditional Statements, 188
C-Ref: Names, 135
C-Ref: **NULL, PTRDIFF_T, SIZE_T**, 273
C-Ref: Null Statement, 207
C-Ref: Numeric Escape Codes, 26
C-Ref: Objects and LValues, 131
C-Ref: **ONEXIT, ONEXIT_T**, 359
C-Ref: Operations, 92
C-Ref: Operations on Stacks, 225
C-Ref: Operations on Structures, 98
C-Ref: Operators and Separators, 15
C-Ref: Order of Evaluation, 176
C-Ref: Organization of Declarations, 50
C-Ref: Other Function Conversions, 130
C-Ref: Other Problems, 41
C-Ref: Overflow and Other Arithmetic Exceptions, 134
C-Ref: Overloading of Names, 53
C-Ref: Packaging the Module, 228
C-Ref: Parameter-Passing Conventions, 214
C-Ref: Paranthesized Expressions, 137
C-Ref: Pointer Arithmetic, 88
C-Ref: Pointer Declarators, 65
C-Ref: Pointers, 71
C-Ref: Pointer Sizes, 121
C-Ref: Pointer Types, 87
C-Ref: Portability Problems, 102
C-Ref: Postdecrement Operator, 144
C-Ref: Postfix Expressions, 138
C-Ref: Postincrement Operator, 143
C-Ref: **POW, SQRT**, 345
C-Ref: Precedence and Associativity of Operators, 132
C-Ref: Precedence Errors in Macro Expansions, 38
C-Ref: Predecrement Operator, 150
C-Ref: Predefined Macros, 36
C-Ref: Preface, 2
C-Ref: Preincrement Operator, 149

C-Ref: Preprocessor Commands, 29
C-Ref: Preprocessor Lexical Conventions, 30
C-Ref: Primary Expressions, 135
C-Ref: Printargs Function in Draft Proposed ANSI C, 279
C-Ref: Printargs Function in Traditional C, 278
C-Ref: Program Structure, 219
C-Ref: **QSORT**, 367
C-Ref: **RAND, SRAND**, 346
C-Ref: **REALLOC, RELALLOC**, 339
C-Ref: Redefining Typedef Names, 111
C-Ref: Relational Operators, 157
C-Ref: **REMOVE, RENAME**, 335
C-Ref: Representational Issues, 117
C-Ref: Representation Changes, 122
C-Ref: Rescanning of Macro Expressions, 35
C-Ref: Reserved Words, 18
C-Ref: Return Statement, 205
C-Ref: Robustness, 221
C-Ref: Sample Output From Enumerating Subsets, 166
C-Ref: Scope, 51
C-Ref: Semicolons, 183
C-Ref: Sequential Expressions, 173
C-Ref: **SETBUF, SETVBUF**, 304
C-Ref: **SETJMP, LONGJMP, JMP_BUF**, 358
C-Ref: Shift Operators, 155
C-Ref: Side Effects in Macro Arguments, 39
C-Ref: **SIGNAL, RAISE, GSIGNAL, SSIGNAL, PSIGNAL**, 359
C-Ref: Signed Integer Types, 80
C-Ref: Simple Assignment, 171
C-Ref: Simple Declarators, 64
C-Ref: Simple Macro Definitions, 31
C-Ref: Size of Operator, 145
C-Ref: Sizes of Structures, 102
C-Ref: Sizes of Unions, 104
C-Ref: **SLEEP, ALARM**, 361
C-Ref: Some Problems with Pointers, 89
C-Ref: Stack Example: Allocation of Stacks, 224
C-Ref: Stack Example: Conditionally Compiled Debugging Code, 222
C-Ref: Stack Example: Deallocation of Stacks, 225
C-Ref: Stack Example: Determining Stack Sizes, 228
C-Ref: Stack Example: Header File (Part 1, Types), 229

C-Ref: Stack Example: Header File (Part 2, Operations), 229
C-Ref: Stack Example: Peek Operation, 227
C-Ref: Stack Example: Push and Pop Operations, 226
C-Ref: Standard Language Additions, 273
C-Ref: Statements, 183
C-Ref: **STDIN, STDOUT, STDERR**, 306
C-Ref: Storage Allocation, 337
C-Ref: Storage Class Specifiers, 59
C-Ref: Storage Units and Data Sizes, 117
C-Ref: **STRCAT, STRNCAT**, 288
C-Ref: **STRCHR, STRPOS, STRRCHR, STRRPOS**, 290
C-Ref: **STRCMP, STRNCMP**, 289
C-Ref: **STRCPY, STRNCPY**, 289
C-Ref: String Constants, 24
C-Ref: String Processing, 287
C-Ref: **STRLEN**, 290
C-Ref: **STRSPN, STRCSPN, STRPBRK, STRRPBRK**, 291
C-Ref: **STRSTR, STRTOK**, 292
C-Ref: **STRTOD, STRTOL, STRTOUL**, 293
C-Ref: Structure Component Layout, 99
C-Ref: Structures, 74
C-Ref: Structure Type References, 97
C-Ref: Structure Types, 95
C-Ref: Subscripting Expressions, 138
C-Ref: Switch Statement; Case and Default Labels, 198
C-Ref: Syntax Notation, 10
C-Ref: Syntax of the C Language, 378
C-Ref: Terminology, 51
C-Ref: The #elif Commands, 43
C-Ref: The #if, #else, and #endif Commands, 42
C-Ref: The #ifdef and #ifndef Commands, 45
C-Ref: The ASCII Character Set, 370
C-Ref: The Assignment Conversions, 127
C-Ref: The Casting Conversions, 127
C-Ref: The C Language, 6
C-Ref: The C Libraries, 264
C-Ref: The Common Model, 77
C-Ref: The C Preprocessor, 29
C-Ref: The Dangling Else Problem, 189
C-Ref: The `defined` Operator, 47
C-Ref: The Function Argument Conversions, 130
C-Ref: The Initializer Model, 77
C-Ref: The Omitted Storage Class Model, 77

C-Ref: The Usual Binary Conversions, 129
C-Ref: The Usual Conversions, 127
C-Ref: The Usual Unary Conversions, 128
C-Ref: Time and Date Functions, 349
C-Ref: **TIME, TIME_T**, 351
C-Ref: **TMPFILE, TMPNAM, MKTEMP**, 336
C-Ref: **TOASCII**, 285
C-Ref: **TOINT**, 285
C-Ref: Token Merging in Macro Expansions, 40
C-Ref: Tokens, 15
C-Ref: **TOLOWER, TOUPPER**, 285
C-Ref: Trivial Conversions, 123
C-Ref: Type Categories, 79
C-Ref: Typedef Names, 110
C-Ref: Typedef Names for Function Types, 111
C-Ref: Type Equivalence, 112
C-Ref: Type Names and Abstract Declarators, 114
C-Ref: Types, 79
C-Ref: Types of Functions, 210
C-Ref: Type Specifiers, 62
C-Ref: Unary Expressions, 144
C-Ref: Unary Minus, 147
C-Ref: Undefining and Redefining Macros, 37
C-Ref: Union Component Layout, 104
C-Ref: Unions, 75
C-Ref: Union Types, 103
C-Ref: Unreferenced External Declarations, 78
C-Ref: Unsigned Integer Types, 82
C-Ref: Use of Compound Statements, 187
C-Ref: Use of Switch Statements, 200
C-Ref: Using `break` and `continue`, 203
C-Ref: Using the For Statement, 194
C-Ref: Using the `goto` statement, 206
C-Ref: Using Union Types, 105
C-Ref: **VARARG, STDARG**, 276
C-Ref: **VFPRINTF, VPRINTF, VSPRINTF**, 332
C-Ref: Visibility, 52
C-Ref: Void, 109
C-Ref: While Statement, 191
C-Ref: Whitespace and Line Termination, 12
C-Ref: Who Defines C?, 8
C-Ref: `DATE` , `FILE` , `LINE` , `TIME` , `STDC` , 275
`ctermid` facility, 364
`ctime` facility, 351
`ctype.h` header files, 281
`cuserid` facility, 364

dangling else, 189
dangling else conditional statement, 189
data objects size, 117
data representation, 117
data tags, 105
data tags unions, 105
date, 238
date facilities, 349
decimal point, 21
declaration of functions, 107
declaration of structures, 95
declaration of unions, 103
declaration point, 51, 53, 56
declarations, 49
declarations at head of blocks, 60
Declarations extent, 56
declarations in compound statements, 186
declarations scope, 51, 240
declaration syntax enumerations, 92
declarators, 49, 64, 242
declarators for arrays, 65
declarators for functions, 67, 242
declarators for pointers, 65, 242
declarators for Variables, 64
`--` decrement, 144, 150
decrement expression, 144
decrement expressions, 150
default, 200
`default` labels, 198
default declarations, 76
default storage classes declarations, 60
default storage class specifier, 60, 62
default type specifiers, 62
`defined` preprocessor command, 47, 239
defining declarations, 76
defining integer values of enumerations, 92
defining macros, 31, 33, 49
defining your own `void` type specifier, 18
definition functions, 209
definition of functions, 107
Dennis Ritchie, 7, 8
dereferencing null, 149
`difftime` facility, 353
discarded expressions, 173, 178, 184, 193
`div` facility, 343
divide by 0, 134, 151
division expressions, 151

`do` statement, 192
domain error, 341
`double` type specifier, 86
duplicate declarations, 55
duplicate visibility, 56, 97
duplicate visibility declarations, 56
`EDOM` macro, 341
`else` conditional statement, 188
enclosed by declarators identifiers, 64
encodings of characters, 13
end-of-file, 301
end of line, in source program, 11, 24
entry point of programs, 217
enum, 92
enumeration constants, 92
enumeration constants identifiers, 92
enumeration constants in expressions, 135
enumeration constants scope, 58, 92
enumeration initializers, 73
enumerations constants, 92
enumerations size, 92
enumeration tags, 92
enumeration tags scope, 58, 240
enumeration types, 92
enumeration type specifiers, 92
environmental functions, 363
environmental library functions, 363
`EOF` facility, 281, 303
equality expressions, 158, 259
equivalence of typedef names, 113
equivalence of types, 112
`ERANGE` macro, 341
`errno` facility, 274, 341
`errno` variable, 225
error indication in files, 301
escape characters, 12, 25, 236
Euclid's GCD algorithm, 151
evaluation order, 176
`exec` facility, 356
executable program, 9
`exit` facility, 357
`exp` facility, 344
expansion macros, 239
expansion of macros, 33
exported identifiers, 58
expressions, 131, 257
expressions assignment, 170

expressions as statements, 184
expressions operators, 131
expression statements, 184
extent of declarations, 56
`extern` storage class, 59
external identifiers, 17, 51, 58, 240, 247
external names, 51, 58, 76, 234, 240, 247
external names portability, 58
extern identifiers, 234
extern storage class specifier, 59
`fabs` facility, 342
`fclose` facility, 303
`feof` facility, 303, 334
`ferror` facility, 334
`fflush` facility, 303
`fgetc` facility, 307
`fgets` facility, 308
field pointers, 89
fields of structures, 95
fields of unions, 95
file, 238
`FILE` type, 301
file inclusion, 41, 238
file names in `#include`, 41, 238
file pointer, 301
file position, 301, 306
`float` type specifier, 86
`float.h`header files, 254
floating-point constants, 21, 235
floating-point initializers, 70
floating-point objects size, 86
floating-point to floating-point conversions, 125
floating-point to integer conversions, 123
floating-point types, 86, 235
`floor` facility, 343
`fmod` facility, 343
`fopen` facility, 303
`for` statement, 193
formal parameters, 33, 211, 214
formal parameters passing conventions, 214
formatting characters, 11, 23
form feed character, 11, 12
FORTRAN, 77, 198
fortran reserved word, 18
forward references, 53, 241
`fprintf` facility, 215, 319
`fputc` facility, 318

fputs facility, 318
fread facility, 333
free facility, 223, 338
freopen facility, 303
frexp facility, 344
fscanf facility, 309
fseek facility, 306
ftell facility, 306
() function call, 141, 258
function call, 141
function call expressions, 141, 258
function declarations, 107
function declarators, 67, 242
function prototypes, 243
function return type, 216
functions conversion to pointers>), 71
functions definition, 50
functions parameters, 211
functions pointers, 107
functions returning structures, 140
functions types, 107, 210
function to pointer conversions, 126
fwrite facility, 333
generic pointers, 253
generic pointers portability, 121
getc facility, 307
getchar facility, 307
getchar function, 25, 83
getchar., 83
getcwd facility, 365
getenv facility, 366
getlogin facility, 366
getopt facility, 366
gets facility, 308
getwd facility, 365
gmtime facility, 352
goto statement, 206
goto statement , 53
goto statement effect on initialization, 57
graphic characters, 11
Greatest Common Divisor, 151
gsignal facility, 359
header files, 9, 228, 260
heap sort, 60
hexadecimal constants, 19
hexadecimal escape characters, 26
hidden declarations, 52

holes in structures, 98
horizontal tab character, 11, 12
host computer, 12
HUGE macro, 341
HUGE VAL macro, 341
hyperbolic functions, 347
identifier names, 16
identifier naming conventions, 17
identifiers declarations, 49
identifiers in expressions, 135
identifiers scope, 51
identifiers spelling rules, 16
if statement, 188
if conditional statement, 188
illegal declarators, 68
implicit declarations, 76
increment expression, 143, 149
index facility, 290
indirection expression, 149
initializer arrays, 72
initializer enumerations, 73
initializers, 246
Initializers, 57, 69
initializers for automatic variables, 69
initializers for automatic variables>), 246
initializers for integers, 70
initializers for static variables, 69, 246
initializers in compound statements, 186
initializers pointers, 71
initializer structures, 74
initializer unions, 75
inner declarations, 50
input/output portability, 301
insertion sort, 194
instr facility, 291
int type specifier, 82, 248
int type specifiers, 80
integer arithmetic portability, 134
integer constants, 19, 234
integer conversions overflow, 123
integer conversions pointers, 126
integer initializers, 70
integer sizes representation, 254
integer to floating-point conversion, 125
integer to integer conversions, 123
integer to pointer conversions, 71, 126
integer types, 80

integer type specifiers, 80
integer types portability, 80
integral types, 80
`isalnum` facility, 282
`isalpha` facility, 282
`isascii` facility, 282
`iscntrl` facility, 282
`iscsym` facility, 283
`iscsymf` facility, 283
`isdigit` facility, 283
`isgraph` facility, 284
`islower` facility, 284
`isodigit` facility, 283
`isprint` facility, 284
`ispunct` facility, 284
`isspace` facility, 285
`isupper` facility, 284
`iswhite` facility, 285
`isxdigit` facility, 283
iterative loops, 190
iterative statements), 190
`jmp_buf` facility, 358
Ken Thompson, 7
keywords, 18, 234
labeled statements, 185, 206
`labs` facility, 342
LALR(1) grammar, 62, 112
`ldexp` facility, 344
`ldiv` facility, 343
length of source lines, 12
`lenstr` facility, 290
lexical constants, 18
lexical converting to strings, 237
lexical elements, 11, 233
lexical merging by preprocessor, 237
lexical tokens, 15
library functions, 9, 260
library functions character processing, 281
library functions characters, 281
library functions string processing, 287
library functions strings, 287, 297
`limits.h` header files, 254
line, 238
line break characters, 12
line termination characters, 12
linker, 9
list expressions, 173

literal constants, 18
literals, 18
little endian computers, 118
`ln` function, 344
`localtime` facility, 352
`log` facility, 344
`log10` facility, 344
logical and expressions, 168
logical expressions, 167
logical negation, 147
logical or expression, 168
`long double` type specifier, 249
`long float` type specifier, 86, 249
`long` type specifier, 82, 86, 248
`long` type specifiers, 80
long constants, 19
`longjmp` facility, 358
loops, 190
lvalues, 131
lvalues expressions, 131
`L tmpnam` macro, 336
macro arguments, 33
macro body, 31
() macro call, 33
macro call, 33
macro calling, 33
macro expansion, 33
macro parameters, 33
macro parameters in characters, 39
macro parameters in strings, 39, 237
macro pitfalls, 31, 38
macro replacement, 35
macro replacement in file names, 41
macros precedence, 38
macros replacement, 48
macros scope, 51
`main` facility, 363
`main` functions, 217, 363
main program, 9, 217, 363
`malloc` facility, 89, 119, 223, 337
Martin Richards, 7
`math.h` header files, 341
mathematical library functions, 341
`matherr` facility, 341
members of structures, 95
members of unions, 95
`memccpy` facility, 298

memchr facility, 297
memcmp facility, 298
memcpy facility, 298
memmove facility, 298
memory.h header files, 297
memory accesses, 179
memory alignment, 119
memory functions, 297
memory library functions, 297
memset facility, 299
merging by preprocessor tokens (lexical), 40
merging of tokens, 40, 237
minimum integer sizes, 254
minimum sizes types, 254
minus expressions, 147
minus operator, 147
missing declarators, 63
mixed common model, 247
mktemp facility, 336
mktime facility, 352
mlalloc facility, 337
modf facility, 344
Modula-2, 187
modularization, 219
modules, 219
multicharacter constants, 23
multidimensional arrays, 65, 91, 138
multiplicative expressions, 151
names, 135
NDEBUG facility, 355
negation arithmetic expressions, 147
negation (bitwise) expressions, 147
negation logical expressions, 147
nested comments, 14
newline character, 12
notation representation, 123
notstr facility, 291
NULL facility, 273
null character, 12
null pointer, 87, 123, 273
null pointers, 123
null statement, 207
numeric escape characters, 25, 26
numeric escapes characters, 236
object code, 9
object module, 9
objects, 131

objects expressions, 131
octal constants, 19
of arrays length, 65
omitted storage class model, 247
`onexit` facility, 359
`onexit_t` facility, 359
operations on arrays, 92
operations on functions, 107
operations on pointers, 88
operations on structures, 98
operator characters, 15
optimization of memory accesses, 179, 250
order of evaluation, 176
order of evaluation expressions, 176, 179
organization of declarations, 50
overflow, 134
overflow floating-point conversion, 125
overloading, 52, 53, 135
overloading class components, 53, 98, 103, 240
overloading class enumeration constants, 53, 92
overloading class labels, 53, 240
overloading class macros, 53
overloading class name space, 53
overloading class statement labels, 53, 240
overloading class tags, 240
overloading class tags , 53
overloading class typedef names, 53
overloading identifiers, 53
overloading of identifiers, 53
packing of components structures, 98
packing of components unions , 104
parameter conversions functions, 130
parameter declarations, 60, 211
parameters functions, 212, 214
`()` parenthesized expression, 137
parenthesized expressions, 137
Pascal, 187, 191, 192, 198
`perror` facility, 274
PL/I, 187
plus operator, 258
plus (unary) expressions, 258
pointer arguments functions, 141
pointer arithmetic portability, 153, 157
pointer conversions integers, 126
pointer declarators, 242
pointer Declarators, 65
pointer function arguments, 141
pointer Initializers, 71

pointers and arrays, 90
pointers and integers portability, 71, 126
pointers cast, 119
pointers size, 126
pointers subscripting, 90
pointers to functions, 107, 255
pointer to array conversions, 90
pointer to integer conversions, 123
pointer to pointer conversions, 89, 126
pointer types, 87, 253
portability, 117, 233
portability character sets, 16, 26
portability comments, 14
portability constant expressions, 23
portability of external names, 17
portability of floating-point types, 86
portability of macros, 39
portability of unions, 105
Portable C Compiler (PCC), 8
position in file, 301, 306
postfix expressions, 138
`pow` facility, 345
precedence expressions, 132
precedence of binary expressions, 151
precedence of declarators, 68
precedence of expressions, 132
precedence of unary operators, 144
predefined macros, 36, 45, 238
preprocessor, 29, 237
preprocessor commands, 29, 239
preprocessor comments, 14, 40
preprocessor lexical conventions, 30, 237
preprocessor pitfalls, 31, 38
preprocessor stringization, 237
preprocessor token merging, 40, 237
primary expressions, 135
`printf` facility, 319
printing characters, 23, 24
process time, 349
program, 9, 50
prototypes, 240, 243
prototypes functions, 240
pseudo-character types, 83
pseudo-unsigned characters, 83
`psignal` facility, 359
`ptrdiff_t` facility, 273
`putc` facility, 318
`putchar` facility, 318

`putenv` facility, 366
`puts` facility, 318
qsort, 367
`qsort` facility, 367
`raise` facility, 359
`rand` facility, 346
range error, 341
`realloc` facility, 339
redefining macros, 37, 239
redefining typedef names, 111
referencing declarations, 76
`register` storage class, 59, 148
register declarations portability, 148
`relalloc` facility, 339
relational expressions, 157, 259
relaxed ref/def model, 247
remainder, 343
remainder), 151
remainder expressions, 151
`remove` facility, 335
`rename` facility, 335
replacement macros, 41, 43
representation of boolean values, 80
representation of characters, 80
representation of data, 117, 254
representation of pointers, 89
representation of types, 122
reserved word entry, 18
reserved words, 18, 234
respresentation changes conversions, 122
`return` statement, 205, 216
return character, 11
returning from functions structures, 140
returning void functions, 141
return statement functions, 205
return types functions, 216
`rewind` facility, 306
`rindex` facility, 290
rvalue, 131
scalar types, 79
`scanf` facility, 309
`scnstr` facility, 290
scope declarations, 51, 240
scope identifiers, 51
scope of component names structures, 58, 240
scope of component names unions, 58, 240
scope of constants enumerations, 58, 92

scope of enumeration tags, 92
scope of tag enumerations, 92
scope of tags enumerations, 58
scope of tags structures, 58, 240
scope of tags unions, 58, 240
scope statement labels, 51
SEEK CUR macro, 306
SEEK END macro, 306
SEEK SET macro, 306
selection components, 257
selection of components, 140, 257
selection of components structures, 95
self-referential structures, 97, 241
separator characters, 15
sequence point, 250
, sequential expression, 173
sequential expressions, 173
setbuf facility, 304
setjmp facility, 358
setjmp.h header files, 355
set package (example), 161
setvbuf facility, 304
shell sort, 194
shift expressions, 155
shift expressions portability, 155
shift left expressions, 155
shift right expressions, 155
short type specifier, 82, 248
short type specifiers, 80
side effects of macros, 39
signal facility, 359
signal.h header files, 355
signed reserved word, 234
signed type specifier, 248
signed and unsigned characters, 83
signed and unsigned conversions, 82
signed types, 80, 248
signed type specifiers, 80, 248
signed versus unsigned characters, 23, 281
simple declarators, 64
simple macros, 31
sin facility, 346
sinh facility, 347
size arrays, 90, 92
size characters, 117
sizeof expressions, 145, 258
sizeof operator, 24, 126, 145, 258

sizeof operator , 117
sizeof operator applied to arrays, 90
sizeof operator applied to functions, 107
sizeof operator type name arguments, 114
size of arrays, 65, 90, 92
size of characters, 117
size of enumerations, 92
size of floating-point, 86
size of integers, 80, 82, 254
size of pointers, 121, 126
size of structures, 102
size of unions, 104
size types, 117
size t facility, 273
size t type, 258
sleep facility, 361
sorting library facilities, 367
source computer, 12
source files, 9
space character, 11
sprintf facility, 319
sqrt facility, 345
srand facility, 346
sscanf facility, 309
ssignal facility, 359
stack module example, 219
standard characters, 11
standard I/O functions, 301
standard I/O library functions, 301
: statement label, 185
statement labels, 53, 185, 206, 240
statement labels scope, 240
statements syntax, 183
static storage class, 59
static storage class specifier, 59
stdarg facility, 276
stdarg.h header files, 273
STDC, 238
stddef.h header files, 273
stderr facility, 306
stdin facility, 306
stdio.h file, 83
stdio.h header files, 301
stdio.h., 83
stdlib.h header files, 337, 341, 355
stdout facility, 306
storage allocation, 223, 337

storage allocation library functions, 337
storage classes functions, 60, 209
storage class specifier, 59
storage units, 117
storage units size, 117
`strcat` facility, 288
`strchr` facility, 290
`strcmp` facility, 289
`strcpy` facility, 289
`strcspn` facility, 291
streams, 301
`strerror` facility, 274
strict ref/def model, 247
`string.h` header files, 287
string concatenation, 288
string constants, 236
stringization of tokens, 39, 237
strings used to initialize array of char, 72
string to pointer conversions, 71
`strlen` facility, 290
`strncat` facility, 288
`strncmp` facility, 289
`strncpy` facility, 289
`strpbrk` facility, 291
`strpos` facility, 290
`strrchr` facility, 290
`strrpbrk` facility, 291
`strrpos` facility, 290
`strspn` facility, 291
`strstr` facility, 292
`strtod` facility, 293
`strtok` facility, 292
`strtol` facility, 293
`strtoul` facility, 293
struct, 95
structure component names scope, 58, 240
structure components, 95
structure components alignment, 98
structure conversions, 125
structure declarations, 95
structure initializers, 74
structures alignment , 102
structures packing of components, 99
structures portability problems, 102
structures size, 102
structure tags, 95
structure tags scope, 58, 240

structure type, 95
structure types, 95
subscript expressions, 138
subscripting, 90, 138
subscripting arrays, 138
[] subscripts, 138
subtraction expressions, 153, 259
subtraction pointers, 153, 259
switch, 200
switch statement, 198
switch statement body, 186
switch statement effect on initialization, 57
switch statement use, 200
syntax listing, 378
syntax notation, 10
sys/times.h header files, 349
sys/types.h header files, 349
system facility, 356
SYS OPEN macro, 303
tag enumerations, 92
tags structures, 95
tan facility, 346
tanh facility, 347
target computer, 12
text streams, 301
time, 238
time facility, 351
time.h header files, 349
time and date library functions, 349
time-of-day facilities, 349
times facility, 349
time t facility, 351
tm library structure, 352
tmpfile facility, 336
tmpnam facility, 336
TMP MAX macro, 336
toascii facility, 285
toint facility, 285
tolower facility, 285
top-level declarations, 50, 60
toupper facility, 285
trigonometric functions, 346, 347
trigraph characters, 233
trigraphs, 233, 238
trivial conversions, 123
two's complement representation, 123
type characters, 83, 248

type checking formal parameters, 215
type checking of function parameters, 215
type checking of function return values, 216
`typedef` storage class, 59
`typedef` storage class specifier, 59
typedef names, 110
typedef names effect on LALR(1) grammar, 112
typedef names for functions, 111
typedef names scope, 58
type equivalence, 112
type equivalence enumerations, 113
type equivalence functions, 112
type equivalence pointers, 112
type equivalence structures, 113
type equivalence unions, 113
type managers, 219
type names, 114
type names in `sizeof` expression, 145
type of arrays, 90
type of enumerations, 92
type of functions, 79, 107, 210
type of structures, 95
type of unions, 105
types, 247
types floating-point, 86
type specifiers, 49, 62, 110, 241
type specifiers without declarators, 63
types pointers, 87
type strings, 83
unary expressions, 144
unary minus expression, 147
unary plus expression, 258
unary usual conversions, 128, 255
undefining macros, 37
underflow, 134
underflow floating-point conversion, 125
underscore character in external names, 17
`ungetc` facility, 306, 307
`union` type specifier, 103
union component names scope, 58, 240
union components overloading, 103
union conversions, 125
union initializers, 75, 246
unions alignment, 104
unions components, 103
unions declarations, 103
unions size, 104
unions types, 105

union tags, 103
union tags scope, 58, 240
union type, 103
union types portability, 105
UNIX, 7, 24, 78, 112
`unix` macro, 36
`unsigned` type specifier, 82, 248
`unsigned` type specifiers, 83
unsigned integers, 82
unsigned integers arithmetic rules, 134
unsigned integers conversions, 123
unsigned operators, 82
unsigned types, 82
unsigned type specifiers, 248
unspecified bounds arrays, 65, 72
use of semicolons in statements, 183
use of void cast expressions , 109
user-defined types, 110
usual argument conversions, 130, 141, 257
usual assignment conversions, 127, 255
usual binary conversions, 129, 256
usual casting conversions, 127
usual conversions casts, 127
usual unary conversions, 255
value of constants, 137
value of enumeration constant, 92
value of name arrays, 135
value of name functions, 135
`vararg` facility, 276
`varargs.h` header files, 273
variable argument lists portability, 215, 243
variables in expressions, 135
`vax` macro, 36
VAX-11, 9, 36
vertical tab character, 11, 12
`vfprintf` facility, 332
visibility declarations, 52
visibility identifiers, 52
`void` types, 109
`void` type specifier, 62, 79, 109, 178, 216
`void` type specifier function result, 141
`void` type specifier in casts, 109
`void` type specifiers, 253
void discarded expressions, 178
void function result, 141
void in casts, 109
void pointers, 253

void return type, 216
volatile reserved word, 234
volatile type specifier, 250
vprintf facility, 332
vsprintf facility, 332
what type integers to use, 80
while statement, 191, 192
whitespace characters, 12
writing into strings, 24
X3J11, 8, 233
YACC, 112
\ line continuation, 233
\ line continuation in strings, 236
\? trigraph escape, 233
\a alert, 236
\' apostrophe, 25
\b backspace, 25
\ character escape, 25
\" double quote>}, 25
\f form feed, 25
\ line continuation in macro calls, 33
\ line continuation in preprocessor commands, 30
\line continuation, in strings, 24
\n newline, 25
\r carriage return, 25
\t tabulate, 25
\v vertical tabulate, 25
\x hexadecimal escape, 236
\\ back slash, 25
^ bitwise xor, 160
^= assign bitwise XOR, 172
_ character in identifiers, 16
_external, 17
FILE facility, 36
_ in external names, 17
IOFBF value, 304
IOLBF value, 304
IONBF value, 304
LINE facility, 36
tolower facility, 285
toupper facility, 285
DATE facility, 238, 275
FILE facility, 238, 275
LINE facility, 238, 275
STDC facility, 238, 275
TIME facility, 238, 275
‘ character, 11

{ } compound statements, 186
{ } enumeration definition, 92
{ } initializers, 69
{ } structure definition, 95
{ } union definition, 103
| bitwise or, 161
|= assign bitwise OR, 172
|| logical or, 168
~ bitwise negation, 147