C Programming with GNU/Linux

Shiv Dayal

May 13, 2015

C Programming with GNU/Linux
by Shiv Dayal

Contents

Preface xiii

1 Introduction 1
- 1.1 Why C? . . . 2
- 1.2 History . . . 2
- 1.3 Comparison with other Languages . . . 3
- 1.4 How to Learn Programming? . . . 3
- 1.5 What is a Computer Program? . . . 3
- 1.6 Attributes of a Program . . . 5
- 1.7 Tools of Trade . . . 6
- 1.8 Bits and Bytes . . . 6
- 1.9 Notes on Number System . . . 7
- 1.10 Binary Number System . . . 8
  - 1.10.1 Conversion of Unsigned Decimals and Binaries . . . 8
  - 1.10.2 2's Complement and 1's Complement . . . 9
- 1.11 Compiling and Executing . . . 11

2 Basics of C 13
- 2.1 The C Character Set . . . 13
- 2.2 Keywords . . . 14
- 2.3 Identifers . . . 14
- 2.4 Programming . . . 15
- 2.5 Data Types . . . 18
  - 2.5.1 Integers . . . 20
  - 2.5.2 Characters . . . 21
  - 2.5.3 Floating Types . . . 22
    - 2.5.3.1 Representation of Floating-Point Numbers . . . 23
- 2.6 New Data Type of C99 . . . 24
  - 2.6.1 Boolean Type . . . 24
  - 2.6.2 Complex Types . . . 25
- 2.7 `void` and `enum` Types . . . 30
- 2.8 Constants . . . 31
- 2.9 Escape Sequences . . . 33

3 Console I/O 37
- 3.1 C's Memory Model . . . 37
- 3.2 printf . . . 39
- 3.3 scanf . . . 42
- 3.4 Character String I/O Functions . . . 46

3.5 Single Character I/O . . . 47

4 Operators and Expressions 49
4.1 Scope of an Identifier . . . 49
4.2 Linkages of an Identifier . . . 50
4.3 Storage Duration of Objects . . . 50
4.4 Usual Arithmetic Conversions . . . 50
4.5 Arithmetic Operators . . . 51
4.6 Relational Operators . . . 53
4.7 Equality Operators . . . 54
4.8 Increment and Decrement Operators . . . 54
4.9 Logical Operators . . . 55
4.10 Bitwise Operators . . . 56
4.11 Bitwise Shift Operators . . . 57
4.12 Assignment Operators . . . 57
4.13 Conditional Operators . . . 58
4.14 Comma Operator . . . 59
4.15 sizeof Operator . . . 59
4.16 Unary Arithmetic Operators . . . 59
4.17 Grouping parentheses . . . 59

5 Control Flow 61
5.1 if else Statement . . . 61
5.1.1 Dangling else Problem . . . 64
5.2 switch Statement . . . 64
5.3 while Loop . . . 66
5.4 do-while Loop . . . 66
5.5 for Loop . . . 68
5.6 break and continue Statements . . . 69
5.7 typedef and return Statements . . . 70

6 Arrays and Pointers 71
6.1 Single-Dimensional Array . . . 71
6.2 Multi-Dimensional Array . . . 74
6.3 Pointers . . . 75
6.4 Address and Indirection Operators . . . 79
6.5 Arrays of Pointers . . . 80
6.6 Pointers of Pointers . . . 82
6.7 realloc() Function . . . 83
6.8 free() Function . . . 84
6.9 Constness . . . 85

7 Functions **87**
7.1 Pass by Value . . . 87
7.2 Pass by Address . . . 90
7.3 Recursion . . . 90
7.4 Function like Macros . . . 95
7.5 Inline Functions . . . 96
7.6 Function Pointers . . . 96
7.7 Passing and Receiving Function Pointers . . . 99
7.8 Type Generic Functions . . . 101

8 Structures and Unions **103**
8.1 Pointer members of a Structure . . . 106
8.2 Usage of Structures and Unions . . . 107
8.3 Structures and Arrays . . . 108

9 Preprocessing Directives **111**
9.1 Conditional Inclusion . . . 112
9.2 Source File Inclusion . . . 113
9.3 Macro Replacement . . . 114
9.3.1 Argument Substitution . . . 116
9.3.2 The # Operator . . . 116
9.3.3 The ## Operator . . . 116
9.3.4 Rescanning and Further Replacement . . . 117
9.3.5 Scope of Macro Definitions . . . 117
9.4 Line Control . . . 117
9.5 Error Directive . . . 118
9.6 Pragma Directive . . . 118
9.7 Null Directive . . . 119
9.8 Predefined Macro Names . . . 119
9.9 Pragma Operator . . . 120
9.10 Usage . . . 120
9.10.1 #include . . . 120
9.10.2 Why we need headers? . . . 122
9.10.3 #define . . . 122
9.10.4 #undef . . . 123
9.10.5 # and ## . . . 124
9.10.6 #error . . . 124
9.10.7 #pragma . . . 125
9.10.8 Miscellaneous . . . 125

10 The C Standard Library **127**
10.1 Introduction . . . 127
10.1.1 Definition of Terms . . . 127
10.1.2 Standard Headers . . . 128
10.1.3 Reserved Identifiers . . . 128
10.1.4 Use of Library Functions . . . 129

11 Diagnostics <assert.h> **131**
11.1 Program Diagnostics . . . 131
11.1.1 The assert Macro . . . 131

12 Complex arithmetic <complex.h> **133**
12.1 Introduction . . . 133
12.2 Conventions . . . 133
12.3 Branch Cuts . . . 133
12.4 The CX_LIMITED_RANGE Pragma . . . 134
12.5 Trigonometric functions . . . 134
12.5.1 The cscos functions . . . 134
12.5.2 The casin functions . . . 135
12.5.3 The cstan functions . . . 136
12.5.4 The ccos functions . . . 137
12.5.5 The csin functions . . . 138
12.5.6 The ctan functions . . . 139
12.6 Hyperbolic functions . . . 140
12.6.1 The cscosh functions . . . 140
12.6.2 The casinh function . . . 141
12.6.3 The catanh functions . . . 142
12.6.4 The ccosh functions . . . 143
12.6.5 The csinh functions . . . 144
12.6.6 The ctanh functions . . . 145
12.7 Exponential and logrithmic functions . . . 146
12.7.1 The cexp functions . . . 146
12.7.2 The clog functions . . . 147
12.8 Power and absolute-value functions . . . 148
12.8.1 The cabs functions . . . 148
12.8.2 The cpow functions . . . 149
12.8.3 The csqrt functions . . . 150
12.9 Manipulation functions . . . 151
12.9.1 The carg functions . . . 151
12.9.2 The cimag functions . . . 152
12.9.3 The conj functions . . . 153
12.9.4 The cproj functions . . . 154
12.9.5 The creal function . . . 155

13 Character Handling <ctype.h> **157**
13.1 Character classification functions . . . 157
13.1.1 The isalnum functions . . . 157
13.1.2 The isalpha function . . . 158
13.1.3 The isblank function . . . 159
13.1.4 The iscntrl function . . . 160
13.1.5 The isdigit function . . . 161
13.1.6 The isgraph function . . . 162
13.1.7 The islower function . . . 162
13.1.8 The isprint function . . . 163
13.1.9 The ispunct function . . . 164
13.1.10 The isspace function . . . 165
13.1.11 The isupper function . . . 166
13.1.12 The isxdigit function . . . 167

14 Errors <errno.h> **169**

15 Floating-point environment <fenv.h> **171**
15.1 The FENV_ACCESS pragma . . . 172
15.2 Floating-point exceptions . . . 173
15.2.1 The feclearexcept function . . . 174
15.2.2 The fegetexceptflag function . . . 176
15.2.3 The feraiseexcept function . . . 176
15.2.4 The fesetexceptflag function . . . 177
15.2.5 The fetestexcept function . . . 177
15.3 Rounding . . . 178
15.3.1 The fegetround function . . . 178
15.3.2 The fesetround function . . . 178
15.4 Environment . . . 179
15.4.1 The fegetenv function . . . 179
15.4.2 The feholdexcept function . . . 179
15.4.3 The fesetenv function . . . 180
15.4.4 The feupdateenv function . . . 180

16 Characterictics of floating types <float.h> **183**

17 Format conversion of integer types <inttypes.h> **185**
17.1 Macros for format specifiers . . . 185
17.2 Functions for greatest-width integer types . . . 186
17.2.1 The imaxabs function . . . 186
17.2.2 The imaxdiv function . . . 187
17.2.3 The strtoimax and strtoumax functions . . . 187
17.2.4 The wcstoimax and wcstoumax functions . . . 188

18 Alternative spellings <iso646.h> **189**

19 Sizes of integer types <limits.h> **191**

20 Localization <locale.h> **193**
20.1 Locale Control . . . 194
20.1.1 The setlocale function . . . 194
20.2 Numeric formatting convention inquiry . . . 195
20.2.1 The localeconv function . . . 195

21 Mathematics <math.h> **199**
21.1 Treatment of error conditions . . . 201
21.2 The FP_CONTRACT pragma . . . 202
21.3 Classification macros . . . 202
21.3.1 The fpclassify macro . . . 202
21.3.2 The isfinite macro . . . 203
21.3.3 The isinf macro . . . 203
21.3.4 The isnan macro . . . 204
21.3.5 The isnormal macro . . . 204
21.3.6 The signbit macro . . . 204
21.4 Trigonometric functions . . . 205
21.4.1 The acos functions . . . 205
21.4.2 The asin functions . . . 205
21.4.3 The atan functions . . . 205
21.4.4 The atan2 functions . . . 206
21.4.5 The cos functions . . . 206
21.4.6 The sin functions . . . 206
21.4.7 The tan functions . . . 207
21.5 Hyperbolic functions . . . 207
21.5.1 The acosh functions . . . 207
21.5.2 The asinh functions . . . 207
21.5.3 The atanh functions . . . 208
21.5.4 The cosh functions . . . 208
21.5.5 The sinh functions . . . 208
21.5.6 The tanh functions . . . 209
21.6 Exponential and logrithmic functions . . . 209
21.6.1 The exp functions . . . 209
21.6.2 The exp2 functions . . . 209
21.6.3 The expml functions . . . 210
21.6.4 The frexp functions . . . 210
21.6.5 The ilogb functions . . . 210
21.6.6 The ldexp functions . . . 211
21.6.7 The log functions . . . 211
21.6.8 The log10 functions . . . 211

21.6.9 The log1p functions . . . 212
21.6.10 The log2 functions . . . 212
21.6.11 The logb functions . . . 212
21.6.12 The modf functions . . . 213
21.6.13 The scalbn and scalbln functions . . . 213
21.7 Power and absolute-value functions . . . 214
21.7.1 The cbrt functions . . . 214
21.7.2 The fabs function . . . 214
21.7.3 The hypot functions . . . 214
21.7.4 The pow functions . . . 215
21.7.5 The sqrt functions . . . 215
21.8 Error and gamma functions . . . 215
21.8.1 The erf functions . . . 215
21.8.2 The erfc functions . . . 216
21.8.3 The lgamma functions . . . 216
21.8.4 The tgamma functions . . . 216
21.9 Nearest integer functions . . . 217
21.9.1 The ceil functions . . . 217
21.9.2 The floor functions . . . 217
21.9.3 The nearbyint functions . . . 217
21.9.4 The rint functions . . . 218
21.9.5 The lrint and llrint functions . . . 218
21.9.6 The round functions . . . 218
21.9.7 The lround and llround functions . . . 219
21.9.8 The trunc functions . . . 219
21.10 Remainder functions . . . 219
21.10.1 The fmod functions . . . 219
21.10.2 The remainder functions . . . 220
21.10.3 The remquo functions . . . 220
21.11 Manipulation functions . . . 221
21.11.1 The copysign function . . . 221
21.11.2 The nan functions . . . 221
21.11.3 The nextafter functions . . . 222
21.11.4 The nexttoward functions . . . 222
21.12 Maximum, minimum and positive difference functions . . . 222
21.12.1 The fdim funcitons . . . 222
21.12.2 The fmax function . . . 223
21.12.3 The fmin functions . . . 223
21.13 Floating multiply-add . . . 223
21.13.1 The fma functions . . . 223
21.14 Comparison macros . . . 224
21.14.1 The isgreater macro . . . 224
21.14.2 The isgreaterequal macro . . . 224

21.14.3 The isless macro . . . 225
21.14.4 The islessequal macro . . . 225

22 Nonlocal jumps <setjmp.h> 227
22.1 Save calling environment . . . 227
22.1.1 The setjmp macro . . . 227
22.2 Restore calling environment . . . 228
22.2.1 The longjmp function . . . 228

23 Signal handling <signal.h> 231
23.1 Specify signal handling . . . 232
23.1.1 The signal function . . . 232
23.2 Send signal . . . 233
23.2.1 The raise function . . . 233

24 Variable arguments <stdarg.h> 235
24.1 Variable argument list access macros . . . 235
24.1.1 The va_arg macro . . . 236
24.1.2 The va_copy macro . . . 236
24.1.3 The va_end macro . . . 236
24.1.4 The va_start macro . . . 237

25 Boolean types and values <stdbool.h> 241

26 Common definitions <stddef.h> 243

27 Integer types <stdint.h> 245
27.1 Integer types . . . 245
27.1.1 Exact-width integer types . . . 246
27.1.2 Minimum-width integer types . . . 246
27.1.3 Fastest minimum-width integer types . . . 246
27.1.4 Integer types capable of holding object pointers . . . 247
27.1.5 Greatest-width integer types . . . 247
27.2 Limits of specific-width integer types . . . 247
27.2.1 Limits of exact-width integer types . . . 248
27.2.2 Limits of minimum-width integer types . . . 248
27.2.3 Limits of fastest minimum-width integer types . . . 248
27.2.4 Limits of integer types capable of holding object pointers . . . 248
27.2.5 Limits of greatest-width integer types . . . 249
27.3 Limits of other integer types . . . 249
27.4 Macros for integer constants . . . 250
27.4.1 Macros for minimum-width integer constants . . . 250
27.4.2 Macros for greatest-width integer constants . . . 250

28 Input/Output <stdio.h> 253
28.1 Introduction . . . 253
28.2 Streams . . . 255
28.3 Files . . . 256
28.4 Operations on files . . . 258
28.4.1 The remove function . . . 258
28.4.2 The rename function . . . 258
28.4.3 The tmpfile function . . . 259
28.4.4 The tmpnam function . . . 259
28.5 File acces functions . . . 260
28.5.1 The fclose functions . . . 260
28.5.2 The fflush function . . . 260
28.5.3 The fopen function . . . 260
28.5.4 The freopen function . . . 262
28.5.5 The setbuf function . . . 262
28.5.6 The setvbuf function . . . 262
28.6 Formatted input/output function . . . 263
28.6.1 The fprintf function . . . 263
28.6.2 The fscanf function . . . 268
28.6.3 The printf function . . . 274
28.6.4 The scanf function . . . 274

Index 275

Preface

Welcome to this book on C programming using a particular specification version C99. As of now C99 is the latest specification fully implemented by C compilers. C11 is latest but it is yet to be implemented by compiler authors. I have been writing this book for a long time but it has yet to reach completion. Very soon a C11 version will be written as compilers start supporting C11 specification. These days I have noticed a tendency that people think Java, PHP or Python are more used and they have a tendency to overlook the merits of C as a programming language. The benefits of learning C is described further in Chapter 1 "Introduction".

C is one of the language which is closest to system and very useful for system programming though not limited to it. It is also very fundamental to many languages as you will discover along your programming endeavours. I highly recommend that all programmers must learn C programmer in order to become a better programmer. You do not have many choices in the category of compiled languages anyway.

Acknowledgements

I do not have much to write in preface as I have covered most of it in Chapter 1. Therefore, I will skip directly to acknowledgements section. First, I would like to thank my parents, wife and son for taking out their fair share of time and the support which they have extended to me during my bad times. After that I would like to pay my most sincere gratitude to my teachers particularly H. N. Singh, Yogendra Yadav, Satyanand Satyarthi, Kumar Shailesh and Prof. T. K. Basu. Now is the turn of people from software community. I must thank the entire free software community for all the resources they have developed to make computing better. However, few names I know and here they go. Richard Stallman is the first, Donald Knuth, Dennis Ritchie, Ken Thompson, Bjarne Stroustrup after that. I possibly cannot write all the names because then possibly an entire chapter will be full of those names. :-) I am using Docbook to write this book therefore obviously Normal Walsh deserves praise for his work on it.

I am not a native English speaker and this book has just gone through one pair of eyes therefore chances are high that it will have lots of errors. At the same time it may contain lots of technical errors. Please feel free to drop me an email at shivshankar.dayal@gmail.com. I will try to fix all errors and incorporate your valuable suggestions.

Chapter 1

Introduction

C programming language is a relatively low-level programming language with an inclination towards system programming. This book will try to serve both as a tutorial and reference of C programming language. If you have not programmed earlier then you should read in linear manner. Even if you have programmed it would be better if you read the book in linear fashion because there are certain points which you may miss if you skip chapters. Of all popular mainstream languages C, and Lisp are two oldest but we cannot really say that Lisp is really popular. It has a niche area(artificial intelligence) and it is there for that. So let me redefine my statement. Of all general-purpose and popular programming language C is the oldest. FORTRAN is another very old language but it is not general purpose programming language. So what makes C so special that it is still out there. Well, C and Unix were born almost together in early 1970s. Then Unix was ported in C and the notion that operating systems can be only written in assembly language, because it has to do time critical things, was destroyed. After that Unix became very popular. Then when C++ was not yet there Windows was written in C and more and more programs were written in C. It might have been the case that Microsoft and Apple would have written their OS in C++ had it been there. So, essentially what happened that there is a lot of code base which is there in C. Also, C++'s backward compatibility is one of the reasons why C++ is so popular. When C was invented there was no structured programming language and code was mostly written in assembly. With C it gave the power of assembly and benefits of structured language like code reuse, modularity, and portability among others. Because of these reasons C became immensely popular and is still popular.

C is simple, small, succinct. It may be dirty but is quick. It may have its quirks but it is a success. C is really so simple yet so deceptive. It will take one years of programming to really thoroughly understand it.

C is currently described by ISO/IEC 9899:2011 which you can buy on web or you can get the draft revision n1570.pdf. It is not necessary to buy the specification as draft is very much similar. I will refer to specification in the form of (§ iso.x.x.x.x) to point to sections where exact information is discussed. Note that specification is written for accuracy and the language is not trivial to understand. At the same time, it does not directly address programmers but compiler writers because there are certain decisions which specification does not make but rather leaves for compiler authors' decision.

1.1 Why C?

Because it is the most common denominator. Any language be it C++, Java, Perl, Python etc have got bindings in C. Whenever you are willing to extend these languages you need to know C. Also, if by any chance you are going towards system programming you need to C. C is everywhere. There is no escape from learning it; it does not matter whether you like it or not.

There is one more important point worth noting here is that C++ is far more complex compared to C. It is generally accepted that one should not try to learn C++ as one's first programming language. If you search web then you will find that many great computer scientists like Ken Thompson, Richard Stallman, Donald Knuth etc. clearly say that they do not like C++ because it is overly complex. Also, the runtime(virtual functions) calculations of C++ make it slightly slower than C. However, C++ has its own strengths and it excels as well if used properly. Wherever there is a memory constraint or extreme high performance is needed C is preferred. The simple syntax of C means its code is very verbose for programmer in the sense that if you read code then you can very easily see what instructions the code is going to translate into.

It is very easy to write interfaces to other languages because other languages expose there objects in terms of C structures not the other way around. The reason for this is huge popularity of C, maturity and large code base.

One more important feature is portability. Note that if you want your program to have high degree of portability then you should not use C99 features but rather ANSI C because ANSI C compilers are available on most platforms. Even though Java claims to be portable or other interpreted languages they are limited by the fact that the interpreters or VMs(JVM in case of Java) is not available on all the platforms. Therefore, C is the MOST portable language. :-)

One of the advantages with C is that since it is very old a lot of software has been written using it. GNU/Linux for example uses C for its kernel. GNOME which is one of the desktop environments of GNU/Linux is also written in C. The fact is that since a lot of libraries are available in C thus it makes a lot of work very easy to do in C since the infrastructure or ecosystem is developed and mature. At the same time, since C is taught in many colleges and universities as first course in programming a lot of people know C thus if you are writing free(as in freedom) software then you have a much higher chance of getting support if you write your code in C.

1.2 History

C was formally delivered to this world in 1972 and started by Dennis MacAlistair Ritchie in 1968. In 1972 C was formally announced. C took its features from BCPL a language by Martin Richard and B by Ken Thompson. AT&T Bells labs gave Unix and a C compiler to many universities at a nominal distribution fees because there was a contract between government of USA AT&T which forbid them from selling software. It grew with leaps and bounds from there and became a ubiquitous language along with Unix, which became an immensely popular operating system. For many years "The C Programming Language"(a book written by Dennis Ritchie and Brian Kernighan published in 1978) served as a reference of C. Later it was standardized by ANSI and then by ISO standards.

1.3 Comparison with other Languages

C is a structured, statically typed, somewhat low-level, high-performance compiled language. It does not support object-oriented programming like most modern programming including C++, Java, Perl, Python, Ruby etc. However, that does not mean you cannot do object-oriented programming in C. In fact, CFront, written by Bjarne Stroustrup which was first C++ compiler was actually a translator from C++ to C. It is just that C does not have support at the language level and it is painful to do so. However, GNOME project has implemented many features of object-oriented paradigm in GObject library. C is low level because it allows you to handle memory contents directly. You have something called `void` which is raw representation of memory content. C also does not support functional or generic programming but again it is possible to do so in a limited manner with painful hacks. One of the coveted features is C programs deliver very high performance if written correctly as it does not have reunite penalties of virtual functions of OOP (object-oriented programming) languages.

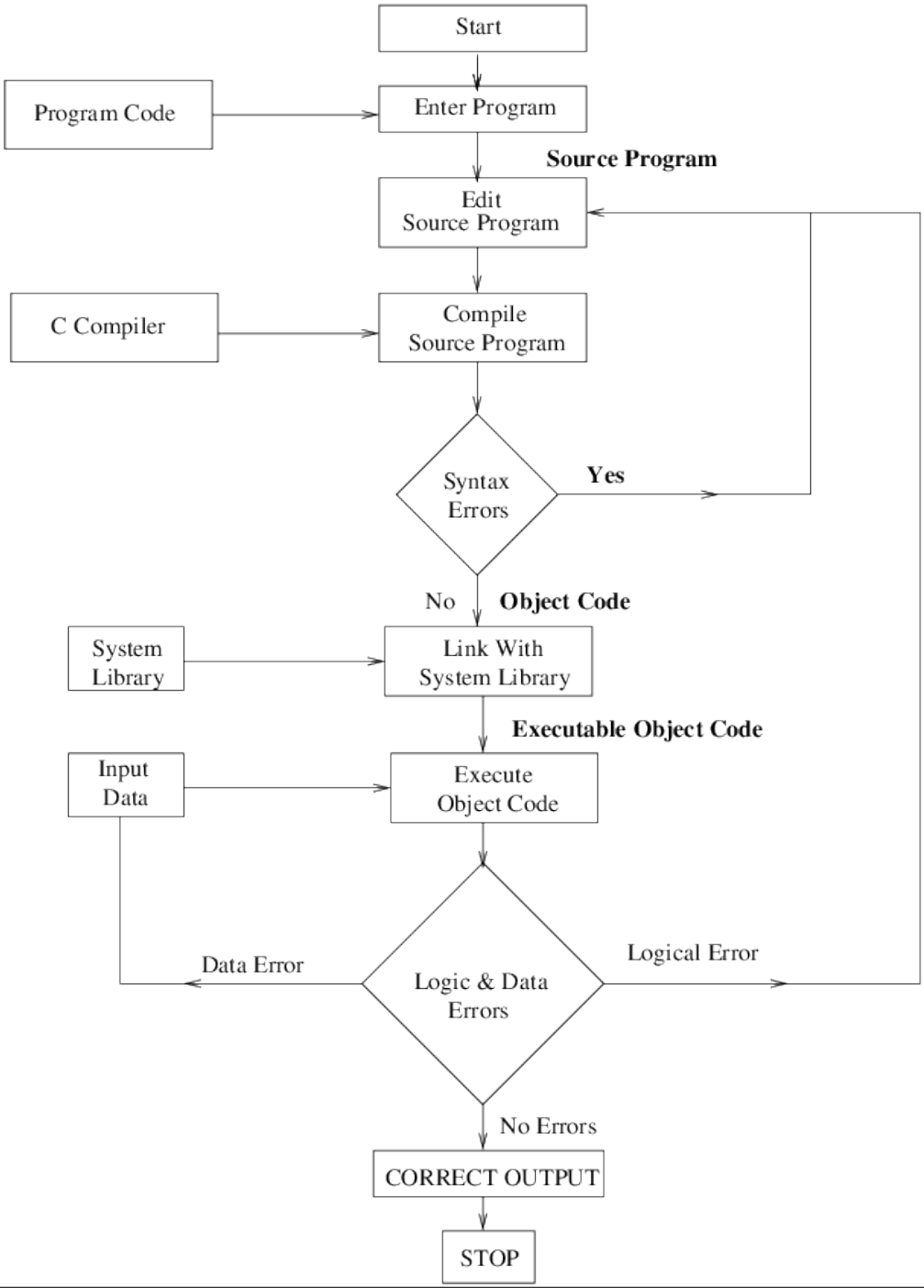
1.4 How to Learn Programming?

Programming is exactly like Mathematics. As in Mathematics you need to read theory, understand solved problems and then solve more and more problems by yourself. If you cannot solve ask your teacher. Similarly, in programming you need to read about language, try examples given, read code written by others and then develop your own code. If you get stuck there are umpteen number of tutorials, mailing lists and groups to help you. I recommend comp.lang.c user group for C programming. Its interface is at http://groups.google.com/group/comp.lang.c/. You should join it and participate there. http://www.stackoverflow.com/ is also a very good forum to ask questions about programming in general.

1.5 What is a Computer Program?

As you may know a computer consists of many components and one of the most or rather most important part is processor often named as CPU (central processing unit). The logic gates in CPUs are formed and instructions like ADD (addition), SUB (subtraction), MUL(multiplication), DIV(division) etc are implemented in hardware of CPU. When we write a program say C program the instructions given in our program is translated to a format which operating system can understand. In our case that is GNU/Linux this executable format is known as ELF (executable and linkable format). For the curious you can read http://en.wikipedia.org/wiki/Executable_and_Linkable_Format and there are lots of specification for different CPUs. Then operating system interprets these files and ask CPU to perform action. So a C program does not directly talk to processor but it rather talks to operating system or rather kernel of the operating system and in turn the operating system or kernel provides services to your program. There is a typical life cycle in development of a program. It has been given as a flowchart below.

Figure 1.1 Development flowchart of a computer program.



1.6 Attributes of a Program

You may be wondering so that is very easy. You just learn programming in C and start hacking on keyboard to produce software. Well, that is partially true but a program has several desired attributes which you must consider. Any program cannot be considered a good program unless it satisfies following requirements or possess following attributes (Note: These are generic attributes and not specific to C programming language):

1. **Correctness:** Correctness means that a program satisfies its requirement specification. It means that for a specified input the specified output should be produced. This particular attribute is of most significance. It does not matter whether other attributes are present or not but this one is a must. If a program behavior is not correct then it is of no use.

2. **Efficiency:** Efficiency is second to correctness only. Say you are developing a text editor and you take 5 seconds to load a 10KB text file then by no means you can persuade a user to use you text editor. A programmable must be as efficient as possible. Sometimes it clashes with other attributes and also depend on the problem domain that how strict are the requirements.

3. **Security:** A very highly desirable feature in programs which deal with more than one computer and also for desktop applications. It is very bad if someone can take advantage of buffer overflow, stack overflow, integer overflow etc. in your program and you must guard against these at all times. Note that to provide security you must put extra checks which will go against efficiency.

4. **Robustness:** Sometimes users will not give correct inputs. For example they may enter a character when an integer is asked for or they can give input beyond range. In such cases you must handle the erroneous input. This is just one example. Sometimes your memory allocation may fail. The rule is program defensively. All such input validations and checks on memory do take a toll on our second attribute but that does not mean that we can neglect it.

5. **Maintainability:** Even a one line program has to be maintained if it is worth it! Typically the life of a program far exceeds the development time. In almost all the cases the original programmer is not maintainer. Because of these reasons you must strive for maintainability. You should follow some coding standards like I highly recommend GNU Coding Stadards. Clear documentation is one of the prerequisites of maintainability.

6. **Extensibility:** Let us take our example of text editor and say our editor is complete. Now someone else would like to provide a plugin which will enable syntax highlighting and project management for this editor. So, in order to do so you can choose a plugin-based extensible architecture or you can allow them to extend the editor using scripting languages like Guile, Python, Lua etc.This features allows user to collaborate and make your program better. Remember the rule is the more the merrier here.

7. **Portability:** It is an elusive and painful goal. Let us say we write our text editor GUI using something like Xlib directly then we will have to port the entire GUI for other non X-based OSes. So we can choose some cross-platform GUI libraries like GTK+, Qt, WxWidgets etc. Even then

when system calls come in your software you can do not much but either write wrappers and do conditional compilation.

1.7 Tools of Trade

At the very least you need a compiler, an editor and a linker. Almost all GNU/Linux systems install GCC by default which is the compiler we are going to use and it includes linker ld. There are several editors you can use from but I am going to use Emacs along with Sr-speedbar and Flymake plugins. Other options include VI, Kate, Gedit, Kwrite etc. A debugger is optional but if you want to go far with C programming then you must learn to use a debugger. GDB is a very nice editor and we are going to use it for debugging in Emacs itself. Emacs has native support for debugging with GDB. For dynamic memory checking, heap corruption, cache corruption etc I am going to show you how to use valgrind. Again, valgrind is optional but it becomes mandatory if you want to work on large projects. For profiling gprof and for code coverage gcov. Note that you can use gcc for compiling programs. Most of the GNU/Linux systems come with gcc. For compiling programs I will use GNU Make though in the beginning I will show you how to compile on command line. Again, profiling, code coverage and make are optional to learn C but practically they are necessary to develop any software worth its value.

You may choose another editor or IDE but I will recommend against IDEs for beginners as they hide much of compilation process from the users. The reason of choosing Emacs as an editor is its power. Emacs is hard to learn but it is very powerful and I implore you to spend some time and learn it. Learning Emacs will pay rich dividends in future to you. Emacs comes with its manual which you can read in menu for `Help`. For using more tools like `Sr-speedbar` and `Flymake` mode you can read more on Emacs Wiki. A lot of extensions are available at Marmalade Repo. In fact it is wrong to say that Emacs is an editor. You can read your email, play games, have a shell, read news, do remote editing, browse web and many other things. It is so powerful that some people set it to run when they login and they never get out of it.

1.8 Bits and Bytes

The smallest unit a computer can understand is called a bit. The values for a bit is either 0 or 1. Consider a voltage. It can be 0V or 1.5V or whatever the core CPU voltage is. CPU does not understand numbers but voltages :-). You cannot expect an electronics hardware to understand the same semantics of 0 and 1 which we know. 0 and 1 are abstraction of CPUs voltages in programming. Four bits form a nibble and eight form a byte. A byte is the area of memory which can be addressed by CPU and its content manipulated. To address a memory a CPU has say 4 or 8 or up to 256 pins. For example, in a common 32-bit CPU there are 32 pins whose voltages may represent 0 or 1. Consider all pins are low i.e. 0 then the memory location pointed to is 00000000000000000000000000000000 i.e. a 8 bit memory at location 0 can be accessed. This memory is also called primary memory or RAM (Random Access Memory). So computing this way we can see that a 32-bit processor can access 2^{32} bytes or 4,294,967,296 bytes. You can arrive at this number by 4*1024*1024*1024. This is equivalent to 4GB of RAM. However, modern Intel processors have 36 physical pins to address up

to 64GB of memory. That does not mean that all 64-bit CPUs have 64 pins for addressing memory as 16 Exabytes(approximately $16 * 10^{18}$) is really, really huge amount of memory which is not needed by any single monolithic system practically and will be very expensive, thus it is not practical Another point is that this much memory will require huge area because RAM is not as compact as hard disk. There are more important practical aspects that if such a computer fails then it will cause massive loss in productivity of the system which employs such a computer. Thus computers are kept much smaller than this size and tasks are divided on those computers and in case one of them fails then it does not affect the entire system. But all that is an architectural concern the point I am saying is it is impractical as of now to have so much RAM in a system but again no one knows future.

Since a byte has 8 bits, its value may range from 0 to 255 as 2^8 is 256. For unsigned data type this will be the range. When all bits are 0 value is zero and when all are high it is 255. Computers use two's complement form to represent binary number. So if these 8-bits represent signed number the range will be from -2^8 to $-2^8 - 1$ that is -128 to 127. As you will see later at lowest levels C allows you to access even one bit using something called bit-fields. If you read specification it will signify the range of one 8-bit byte as -127 to 127 because it also takes in to consideration of 1's complement computers in which positive and negative zeroes are different.

1.9 Notes on Number System

A number system is a system which determines the rules and symbols for numbers on how we are going to use them. A number system consists of symbols for representing numbers and a dot for representing fractional numbers. Minus sign is used to represent negative numbers. A number system ranges from$-\infty$ to $-\infty$. It is best represented by a straight line given below:

Figure 1.2 Number axis.

Each point on this axis represents a number. It may be integer or fractional number. An integer is a whole number like -1, -2, 0, 5, 7 etc. Floating-point numbers have fractional parts like 1.234. The important fact to note is that between any two points there exists infinite numbers. In other words between any two numbers there exists infinite numbers. For example, between 1.2 and 1.3 there are 1.21, 1.22, 1.23..., 1.29. Moreover between 1.21 and 1.22 there are 1.211, 1.212, 1.213 and so on. It enables us to represent a point on this axis. The numbers I have written are supposedly in decimal number system. Base of decimal number system is 10. Why because it consists of 10 distinct symbols 0 through 9 or vice-versa. Similarly we can have any other number system. Popular number systems in computers are octal and hexadecimal not to mention decimal of course. Binary is already mentioned as the native number system for computers.

A number in a generic number system is given below:

$$(..c_m b^{m-1} + c_{m-1}b^{m-2} + ... + c_2 b^1 + c1_b^0 + c_{-1}b^{-1} + ... + c_{-m}b^{-m}) \\ = (...c_m c_{m-1}...c_2 c_1 . c_{-1}...c_{-m})_b$$

All the terms with c are called digits. The leftmost or leading digit is called *most significant digit* and the rightmost or trailing digit is called *least significant digit*. The . is called a point which separates the integral part which is towards its left from the fractional part which is towards its right. b is known as radix or base of the number system. Note that all digits will be between 0 to $b-1$. So in our decimal system b is 10 therefore we have digits from 0 to 9. In binary number system it is 2 therefore digits permitted are 0 and 1.

1.10 Binary Number System

As the name suggests binary number system has base of 2. Therefore it has only two symbols. 0 and 1. This is the most popular system for computers because TTL NAND and NOR gates which are the most basic logic gates using which other gates are implemented in processor has only two voltage output levels because of their operation in cut-off and saturation zones. These terms are better understood with the help of a book on electronics which is out of scope of this book. All binary numbers consist of 0 and 1. So the count is like 0, 1, 10, 11, 100, 101, 110, 111, 1000 and so on.

1.10.1 Conversion of Unsigned Decimals and Binaries

Consider a decimal number. Let us say 53 then how would be convert it to binary. The technique is that of division. Please examine following carefully:

```
2 | 53 | 1
----------
2 | 26 | 0
----------
2 | 13 | 1
----------
2 | 6  | 0
----------
2 | 3  | 1
----------
  | 1  |
```

So the binary is 110101_2. First we divide 53 by 2 and write the remainder. Then quotient is 26. We repeat the process for 26 therefore remainder is 0 and quotient is 13. This we go on repeating till we have 1 as quotient. Note that all the remainders will be 0 or 1 because divisor is 2. Similarly, final quotient is always 1. Now we take final quotient and start writing remainders from top to bottom.

To convert binary to decimal let us examine following:

$$1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 53_{10}$$

The power is to 2 because 2 is the base of source. It starts from 0 for unit's position and increases to 1 and 2 for ten's and hundred's position and so on. 1's and 0's are the values of that place. If you note carefully powers of 2 grow like 1, 2, 4, 8, 16, 32, 64, 128 and so on. Any number can be written by using these powers at most one time. For example consider 100. I know it is less than 128 so I will use 64. Then 36 remains. So I will use 32 and then 4. This means 100=64+32+4 which means power 6, 5 and 2 have been used. Therefore, I can quickly write down number as 1100100_2.

Fractional numbers are slightly more complicated. Let us consider 1.1_2. In decimal it will be $1+\frac{1}{2}$. This is 1.5 in decimal. Note that when you convert a fractional part of binary to decimal denominator will always be power of 2. For that matter when you convert from any base to decimal denominator will be powers of that base. **Important:** Therefore, when you convert from decimal to some base *n* then denominator of that decimal number can have only those prime factors which are available in the set of prime factors of *n*.

Let us say we have a fractional number in decimal .59 then to convert it to decimal we multiply it with 2 which yields 1.018 which is greater than 1 so our equivalent binary number is .1. Now we subtract 1 from 1.18 to get .18 which is less than 1 so we multiply it with 2 again to get .36. Now since this is less than 1 our equivalent binary number is .10. Repeating the process we get .72 and .100 then 1.44 and .1001. We put 1 in binary part because decimal part has become greater than 1. Now again we subtract 1 from decimal part to get .44 and repeat the process. Operations such as addition, subtraction, multiplication and division are similar in all number systems.

1.10.2 2's Complement and 1's Complement

2's complement and 1's complement are used to convert binary numbers to decimal values. In 1's complement the number is obtained by inverting bits i.e. making 0 bit to 1 bit and 1 bit to 0 bit of the binary number in question.

The 2's complement of an N-bit number is defined as the complement with respect to 2^N; i.e. it is the result of subtracting the number from 2^N, which in binary is one followed by N zeroes. This is also equivalent to taking the 1's complement and then adding one, since the sum of a number and its 1's complement is all 1 bits.

Consider the following table which contains some numbers for 1's complement of some 8-bit numbers.

For signed numbers MSB(most significant bit) decides sign in both 1's complement as well as 2's complement. 1's complement has two zeroes. Positive and negative. As you see in table that 1111 1111 is -0 because MSB is 1 so it is a negative number and then if you invert all remaining bits then it turns out to be 0. In a 1's complement system negative numbers are represented by the arithmetic negative of the value. An N-bit 1's complement number system can represent integers in the range $-2^{N-1}-1$ to $-2^{N-1}-1$.

Now it is easy to do addition, subtraction, multiplication, division and other arithmetic operations. Subtraction for 1's complement is a bit different. Consider the following:

```
+ 0000 0110       6
- 0001 0011      19
============    ====
1 1111 0011     -12    -An end-around borrow is produced, and the sign bit
```

Table 1.1 Example of 1's complement

Bits	Unsigned Value	1's Complement
0111 1111	127	127
0111 1110	120	126
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	255	-0
1111 1110	254	-1
1000 0010	130	-125
1000 0001	129	-126
1000 0000	128	-127

```
                          of the intermediate result is 1.
- 0000 0001       1     -Subtract the end-around borrow from the result.
============    ====
  1111 0010      -13     -The correct result (6 - 19 = -13)
```

Borrows are propagated to the left. If the borrow extends past the end then it is said to have "wrapped around", a condition called an "end-around borrow". When this occurs, the bit must be subtracted from the right-most bit or least significant bit(LSB). This does not occur in 2's complement arithmetic.

As you see in table and also you can verify the value becomes negative if its 1's complement is computed. However, 2's complement is used on most of computers because of two zeroes in 1's complement, borrowing being complicated etc.

Table 1.2 Example of 2's complement

Bits	Unsigned Value	2's Complement
0111 1111	127	127
0111 1110	120	126
0000 0010	2	2
0000 0001	1	1
0000 0000	0	0
1111 1111	255	-1
1111 1110	254	-2
1000 0010	130	-126
1000 0001	129	-127
1000 0000	128	-128

Clearly, since N-bit 1's complement can represent numbers in range $-2^{N-1}-1$ to $2^{N-1}-1$ 2's complement of Nbit can represent -2^{N-1} to $2^{N-1}-1$s it does not have negative 0 i.e. its range is more by 1 number.

The 2's complement system has the advantage that operations of addition, subtraction, and multiplication are same as unsigned binary numbers (as long as the inputs are represented in the same number of bits and any overflow beyond those bits is discarded from the result). This property makes the system both simpler to implement and capable of easily handling higher precision arithmetic. Also, as mentioned above zero has only a single representation, avoiding the subtleties associated with negative zero, which exists in 1's complement systems.

The value v of an N-bit integer $b_{N-1}b_{N-2}\dots b_0$ is given by the following formula:

$$v = -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i$$

I will leave it up to you, the reader, to perform basic operations like addition, subtraction, multiplication, division etc.

1.11 Compiling and Executing

To compile and execute a program create a new file, edit it and save it. The extension of file should be `*.c`. For example, `myprogram.c`. After that you can give this command at terminal. Here is the first program for you.

```
#include <stdio.h>

int main()
{
  return 0;
}
```

Execute the following command on your command prompt:

$**gcc nothing.c -o nothing** where `nothing.c` is the name of the program.

Then you will see a file named `nothing` is created by compiler if no errors were there in your program. In case of errors, like we had in one shown to you they have to be resolved first. Suppose nothing is produced then you can execute it like

$**./nothing**

Note that in both the commands $ is not part of command but it is prompt. For you it may be `%` or # or something fancier (depends on the imagination of your system administrator). To execute this command your working directory must be same as the directory your program is in. Also, note that on some systems TAB auto completes filename so do not do the following by accident:

DO NOT DO THIS.

$**gcc nothing.c -o nothing.c**

This will overwrite your nothing.c by nothing. Let us see how to compile this program using a Makefile. Edit your Makefile like this:

```
#sample Makefile
check-syntax:
    gcc -o nul -Wall -S $(CHK_SOURCES)

nothing:nothing.c
    gcc nothing.c -o nothing
```

2nd and 3rd lines in the above Makefile is for Flymake to work in Emacs. Now from do this from menu of Emacs. From Tools select compile. As the command issue **make -k nothing**. Your code will be compiled. Makefiles are better than executing commands however you must know underlying commands. Alternatively, you can execute `make` directly from your shell or type 5th line yourself on prompt to compiler the program. You can also use something like CMake or Scons but I think that should be part of a book covering build systems.

Chapter 2

Basics of C

Now is the time for learning basics. There are certain rules in every language, certain grammar which dictates the way language will be spoken and written. It has a script to write using. Similarly, programming languages have BNF (Backus-Naur Form) context-free grammar. There are valid characters in a programming language and a set of keywords. However, programming language ruleset is very small compared to a natural programming language. Also, when using natural programming language like talking to someone or writing something the other person can understand your intent but in programming you cannot violate rules. The grammar is context-free. Compilers or interpreters cannot deduce your intent by reading code. They are not intelligent. You make a mistake and it will refuse to listen to you no matter what you do. Therefore, it is very essential to understand these rules very clearly and correctly.

2.1 The C Character Set

The following form the C character set you are allowed to use in it:

```
[a-z] [A-Z] [0-9] ~ ! # % ^ & * ( ) - = [ ] \ ; ' , . / _ + { } | : " < >  ↩
    ?
```

This means along with other symbols you can use all English alphabets (both uppercase and lowercase) and Arabic numerals. However, English is not the only spoken language in the world. Therefore in other non-English speaking counties there are keyboard where certain characters present in above set are not present. The inventors of C were wise enough to envision this and provide the facility in form of trigraph sequences. Given below is the table containing all trigraph sequences. Note that GNU coding standards advice against using trigraph sequences for portability reasons.

Table 2.1 Trigraph Sequence

Trigraph	Equivalent	Trigraph	Equivalent	Trigraph	Equivalent
??=	#	??'	^	??!	|
??(	[	??)	]	??<	{
??>	}	??/	\	??-	~

2.2 Keywords

The following are reserved keywords for C programming language by C99 specification which you are not allows to use other than what they are meant for:

Table 2.2 Keywords of C99

auto	enum	restrict	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool
continue	if	static	_Complex
default	inline	struct	_Imaginary
do	int	switch	
double	long	typedef	
else	register	union	

These keywords serve specific purpose. You will come to know about all of them as you progress through the book. Next we look at identifiers. Following keywords were added in C11 specification:

Table 2.3 Keywords added in C11 specification

_Alignof	_Atomic	_Generic	_Noreturn	_Static_assert	_Thread_local

2.3 Identifers

The names which we give to our variables are known as identifiers. Something with which we identify. As you have already seen what is allowed in C's character set but not all are allowed in an identifiers name. Only alphabets from English language both lowercase and uppercase, Arabic digits from zero to nine and underscore (_) are allowed in an identifiers name. The rule for constructing names is that among the allowed characters it can only begin with only English alphabets and underscore. Numbers must not be first character. For example, `x, _myVar, varX, yourId78` are all valid names. However, take care with names starting from underscore as they are mostly used by different library authors. Invalid identifier examples are `9x, my$, your age`. Please read this section carefully

and make sure understand the rules for naming identifiers. Later at the end of chapter there are some simple problems to work on.

2.4 Programming

Now is time for some programming. Let us revisit our first program and try to understand what it does. Here I am giving code once again for quick reference:

```
//My first program
/* Author: Shiv Shankar Dayal
   Description: This program does nothing.*/

#include <stdio.h>

int main(int argc, char* argv[])
{
  return 0;
}
```

You can now issue a command as **$gcc nothing.c** where `nothing.c` is the filename by which you saved the source code. Note that `$` is the prompt not part of command itself. Then you can do an **ls** and you will find that `a.out` is a file which has been produced by gcc. Now you can run this program by saying **./a.out** and nothing will happen. But if you type **echo $?** then you will find that 0 is printed on screen which is nothing but 0 after return of our program.

As you can see this program does almost nothing but it is fairly complete program and we can learn a lot from it about C. The first line is a comment. Whenever C compiler parses C programs and it encounters `//` it ignores rest of line as code i.e. it does not compile them. This type of single line comment were introduced in C99 standard and if your compiler is really old the compiler may give you error message about it. The second and third lines are also comments. Anything between `/*` and `*/` is ignored like `//`. However, be careful of something like `/* some comment */more comment */`. Such comments will produce error messages and your program will fail to compile.

Comments are very integral part of programming. They are used to describe various things. You can write whatever you want. They may also be used to generate documentation with tools like doxygen. Typically comments tell what the program is doing. Sometimes how, when the logic is really complex. One should be generous while commenting the code.

`#include` is a pre-processor directive. It will look for whatever is contained in angular brackets in the `INCLUDEPATH` of compiler. For now you can assume that `/usr/include` is in include path of compiler. Basically what it does is that it looks for a file names `stdio.h` in the `INCLUDEPATH`. If that is found the content of that file is pasted here in our program.If you really want to see what happens then you can type **$gcc -E nothing.c**. You will see lots of text scrolling on your screen. The `-E` switch tells gcc that just preprocess the file, do not compile it, and send the resulting output to standard output (we will know about this more later), which happens to be your monitor in this case.

Next line is `int main(int argc, char* argv[])`. Now this is a special function. Every complete executable(shared objects or dlls do not have main even though they are C programs) C

program will have one main function unless you do assembly hacking. This function is where the programs start. The first word `int` is a keyword which stands for integer. This signifies the return type of function. `main` is the name of the function. Inside parenthesis you see `int argc` which tells how many arguments were passed to program. While `char* argv[]` is a pointer to array which we will see later. For now it holds all the arguments to the program.

Next is a brace. The scope in C is determined by braces. Something outside any brace has global scope (we will see these later), something inside first level of brace has function or local scope. Something inside second or more level of braces have got that particular block scope. Scope here means that when there will be a closing brace that particular variable which is valid in that scope will cease to exist. However, we do not have to worry about that yet as we do not have any variable. Just note that a corresponding closing brace will be the end of main function.

Next line is `return 0;` This means whoever has called `main()` will get a 0 as return is returning 0. In this case, receiver is the shell or operating system which has invoked the very program. The semicolon is called the terminator and used also on Java or C++ for example. The very requirement of semicolon is to terminate the statement and move on to next statement.

However, the program shown does not do much. Let us write a program which has some more functionality and we can explore more of C. So here is a program which takes two integers as input from users and presents their sum as output. Here is the program:

```
// My second program
// Author: Shiv S. Dayal
// Description: It adds two numbers

#include <stdio.h>

int main()
{
  int x=0, y=0, sum=0;

  printf("Please enter an integer:\n");
  scanf("%d", &x);

  printf("Please enter another integer:\n");
  scanf("%d", &y);

  sum = x + y;

  printf("%d + %d = %d\n", x, y, sum);

  return 0;
}
```

and the output is:

```
shiv@shiv:~/book/code$ ./addition
Please enter an integer:
7
```

```
Please enter another integer:
8
7 + 8 = 15
shiv@shiv:~/book/code$
```

Note that `shiv@shiv:~/book/code$` is the prompt. The Makefile is also updated:

```
check-syntax:
    gcc -o nul -Wall -S $ (CHK_SOURCES)

nothing:nothing.c
    gcc nothing.c -o nothing

addition:addition.c
    gcc addition.c -o addition
```

You can choose `Tools->Compile` then enter `make -k addition` as make commands in the Emacs's minibuffer and execute like `$./addition`.

Let us discuss new lines one by one. The line `int x=0, y=0, z=0;` is declaration and definition or initialization of three ints. `int` keyword in C is used to represent integers. Now we have three integers with there values set to 0. Note that how the variables are separated by commas and terminated by semicolon(as we saw in last program also). We could have also written it like this:

```
int x;
int y;
int z;

x = 0;
y = 0;
z = 0;
```

or:

```
int x, y, z;

x = y = z = 0;
```

However, the first method is best and most preferred as it prevents use before definition. int is a data-type in C. `x, y,` and `z` are variables of type `int`. This means that the size of these variables will be same as `int`. Note that C is a statically typed language and all types have predefined memory requirements. In cour case, int requires 4 bytes on 32-bit systems.

Now I will talk about `printf()` function. This function is declared in `stdio.h`. The prototype of `printf()` is

```
int printf(const char *restrict format, ...);
```

The first argument format is what we have in first two function calls. The second is a `...` which means it can take variable number of arguments known as variable-list. We have seen this in the third call.This means it will take a string with optional variable no. of arguments. The string is called

the format-string and determines what can be printed with supplied arguments. These `...` are used to supply variable no. of arguments. In the first two `printf()` statements we just print the format-string so that is simple. However, in the last one, we have format as `%d` which signifies a decimal integer. The integers printed are in the same order in which they were supplied.

Time for some input. `scanf()` is scan function which scans for keyboard input. As by now you know that `%d` is for decimal integer but I have not said `x` or `y`. The reason is `x` and `y` are values while `&x` and `&y` are the addresses of `x` and `y` in memory. `scanf()` needs the memory address to which it can write the contents to. You will see `&` operator in action later when we deal with pointers. Just remember for now that to use a simple variable with `scanf()` requires `&` before its name.

Now I am going to take you on a tour of data types. Till now we have just seen only `int`. So onward to data types.

2.5 Data Types

C is a statically typed language that is every variable has a type associated with it. Types are discussed in specification in great length in Types(§ iso.6.2.5) to (§ iso.6.2.8). Since C is statically typed the sizes of types have to be known at compile time. There are four types of data types. Integral, floating-point, arrays and pointers. There are two user-defined data types; structures and unions. Here, I will discuss the integral and floating-point types and leave the rest for later. The integral types are `char, short int, int, long` and `long long` and floating-point types are `float, double` and `long double`.`signed` and `unsigned` are sign modifiers which also modified the range of data types but do not affect their memory requirements. By default all basic data types are signed in nature and you must qualify you variables with `unsigned` if you want that behavior. `short` and `long` are modifiers for size which the data type occupies. The ranges of integral data types directly reflect their memory requirements and if you know how much memory they are going to occupy you can easily compute their ranges. The range of floating-point comes from IEEE specification IEEE-754.

The range of data types is given in Numerical limits(§ iso.5.2.4.2). For example, in the range program given below size of *int* is 4 bytes which is double than what is specified by specification i.e. 2 bytes. Given below is the table for numerical limits for reference from specification. Note that these are in 1's complement form thus you have to adjust for 2's complement. Note that these limits are minimum limits imposed by specification and actual limits of data types may be different on your particular platform. The actual values of these limits can be found in headers `<limits.h>, <float.h>` and `<stdint.h>`. The values given below are replaced by constant expressions suitable for use in `#if` preprocessing directives. Moreover, except for `CHAR_BIT` and `MB_LEN_MAX`, the following are replaced by expressions that have the same type as an expression that is an object of the corresponding type converted according to the integer promotions. Their implementation-defined values are equal or greater in magnitude (absolute value) to those shown, with the same sign.

- number of bits for smallest object that is not a bit-field (byte)

  `CHAR_BIT` 8

- minimum value for an object of type `signed char`

`SCHAR_MIN` -127 $-2^7 - 1$

- maximum value for an object of type `signed char`
  `SCHAR_MAX` 127 $2^7 - 1$
- maximum value for an object of type `unsigned char`
  `UCHAR_MAX` 255 $2^8 - 1$
- minimum value for an object of type `char`
  `CHAR_MIN` *see below*
- maximum value for an object of type `char`
  `CHAR_MAX` *see below*
- maximum number of bytes in a multibyte character, for any supported locale
  `MB_LEN_MAX` 1
- minimum value for an object of type `short int`
  `SHRT_MIN` -32767 $-2^{15} - 1$
- maximum value for an object of type `short int`
  `SHRT_MAX` +32767 $2^{15} - 1$
- maximum value for an object of type `unsigned short int`
  `USHRT_MAX` 65535 $2^{16} - 1$
- minimum value for an object of type `int`
  `INT_MIN` -32767 $-2^{15} - 1$
- maximum value for an object of type `int`
  `INT_MAX` +32767 $2^{15} - 1$
- maximum value for an object of type `unsigned int`
  `UINT_MAX` 65535 $2^{16} - 1$
- minimum value for an object of type `long int`
  `LONG_MIN` -2147483647 $-2^{31} - 1$
- maximum value for an object of type `long int`
  `LONG_MAX` +2147483647 $2^{31} - 1$
- maximum value for an object of type `unsigned long int`
  `ULONG_MAX` 4294967295 $2^{32} - 1$

- minimum value for an object of type `long long int`
  `LLONG_MIN` -9223372036854775807 $-2^{63} - 1$
- maximum value for an object of type `long long int`
  `LLONG_MAX` +9223372036854775807 $2^{63} - 1$
- maximum value for an object of type `unsigned long long int`
  `ULLONG_MAX` 18446744073709551615 $2^{64} - 1$

If the value of an object of type char is treated as a signed integer when used in an expression, the value of `CHAR_MIN` is the same as that of `SCHAR_MIN` and the value of `CHAR_MAX` is the same as that of `SCHAR_MAX`. Otherwise, the value of `CHAR_MIN` is 0 and the value of `CHAR_MAX` is the same as that of `UCHAR_MAX`. The value `UCHAR_MAX` equals $2^{CHAR\_BIT} - 1$.

Let us write a program to find out ranges for integral data types:

```
// My range program
// Author: Shiv S. Dayal
// Description: It gives ranges of integral data types

#include <stdio.h>
#include <limits.h>

int main()
{
  printf("Size of char is.........%d\n", sizeof(char));
  printf("Size of short int is.....%d\n", sizeof(short int));
  printf("Size of int is...........%d\n", sizeof(int));
  printf("Size of long is.........%d\n", sizeof(long));
  printf("Size of long long is.....%d\n", sizeof(long long));

  return 0;
}
```

and the output will be:

```
Size of char is.........1
Size of short int is.....2
Size of int is...........4
Size of long is.........4
Size of long long is.....8
```

Based on this it is left as an exercise to reader to compute the ranges of these data types.

2.5.1 Integers

Integers are probably simplest to understand of all data types in C so I am discussing them before any other type. As you have seen the keyword for declaring integer type is `int`. An integer can be

2 bytes or 4 bytes. A 16-bit compiler will have integer of 2 bytes while a 32-bit or 64-bit compiler will have a 4 byte integer. The specified minimum size of an integer is 2 bytes. Since most modern computers are either 32-bit with 64-bit becoming more dominant we will assume in this book that integer's size is 4 bytes or 32-bit implicitly because 32-bit `gcc` gives a 32-bit integer. As said earlier keyword `signed` which when applied to a data type splits the range into two parts. Since integer is 32 bit so it will be split in the range from -2^{31} to $2^{31}-1$. By default integers, characters and long are `signed`. Floats and doubles are always `signed` and have no unsigned counterpart. When the integer will be `unsigned` then the positive range doubles and it becomes 0 to $2^{32}-1$. When the value of intger is more than its range then the values rotate in the using modulus with the largest value of the range which is also known as `INT_MAX` or `INT_MIN`. For `unsigned` types it is `UINT_MAX`. These are macros and are defined in `limits.h` which you can find in `/usr/include` or `/usr/local/include` by default.

There are four different types of integers based on their storage requirement. `short int`, `int`, `long` and `long long`. `long` is short for `long int`. Short integers are always two bytes. Signed short integer has a range of -32768 to 32767 while unsigned of that has a range of 0 to 65535. Plain integers i.e. `int` have already been discussed. `long` are having a minimum storage requirement of 4 bytes. Usually it is large enough to represent all memory addresses of the system because `size_t` is `unsigned long` and `size_t` is th type of argument accepted by memory allocation functions.

2.5.2 Characters

A `char` is 1 byte i.e. 8 bits or `CHAR_BIT` bits. So its signed version i.e. 2's complement where half the range is negative and half is positive will have value from -128 to 127. Well that is not exactly opposite because we have only one zero for positive and negative numbers. If it would have been 1's complement then range would have been from -127 to 127 which the specification clearly mentions but since computers follow 2's complement that range should be from -2^7 to 2^7-1. Note that chars are fundamentally integral types and ASCII symbols are first 128 numbers or in other words they are 7-bit numbers.

So a character '0' is internally 48 in decimal which is its integral or internally it is handled as a sequence of binary numbers representing `0x30` in hexadecimal. These integral values for characters are known as ASCII value. A full table of ASCII values is given in the appendix A.

A simple program which takes input for few characters and then prints them on console along with their ascii values is given below:

```
#include <stdio.h>

int main()
{
  char c = 0;
  char c1 = 0, c2 = 0;

  printf("Enter a character on your keyboard and then press ENTER:\n");
  scanf("%c", &c);
```

```
  printf("The character entered is %c and its ASCII value is %d.\n", c, c) ↩
      ;
  // Their remains '\n' in the stdin stream which needs to be cleared.
  getchar();
  printf("Enter a pair of characters on your keyboard and then press \
          ENTER:\n");
  scanf("%c%c", &c1, &c2);
  printf("The characters entered are %c and %c and their ASCII \
          values are %d and %d respectively.\n", c1, c2, c1, c2);

  short int si = 0;

  si = c1 + c2;

  printf("The sum of c1 and c2 as integers is %hd.\n", si);

  return 0;
}
```

A sample run may have following output:

```
Enter a character on your keyboard and then press ENTER:
1
The character entered is 1 and its ASCII value is 49.
Enter a pair of characters on your keyboard and then press ENTER:
12
The characters entered are 1 and 2 and their ASCII values are 49 and 50
respectively.
The sum of c1 and c2 as integers is 99.
```

As you can see from the program that characters are internally stored as integers and we can even perform integers which we normally perform on numbers like addition as shown. We can perform other operation as subtraction, multiplication and division, however, most of the time addition or subtraction only makes sense to advance the characters in their class. Multiplication and division of characters with other characters or integers does not make sense.

One problem of concern is the extra `\n` in the input stream. It does not cause trouble with integers but when you want to read characters then the `Enter` or `Return` keys which may be left over from the last input will cause trouble. `\n` is recognized as a character and will be assigned to next variable if it is in `stdin`, which is standard input stream. One of the ways to remove it is to make a call to `getchar` which reads one character from the `stdin` stream and this `\n` will be removed.

2.5.3 Floating Types

Floating point representation is a lot more complicated in computers than it is for us human beings. C specification takes floating points description and specification from **IEC 60559:1989** which is a standard for floating point arithmetic which is same as IEEE 754. In C there are three types of

floating point numbers `float, double` and `long double`. These are described in specification in §(iso.5.2.4.2.2).

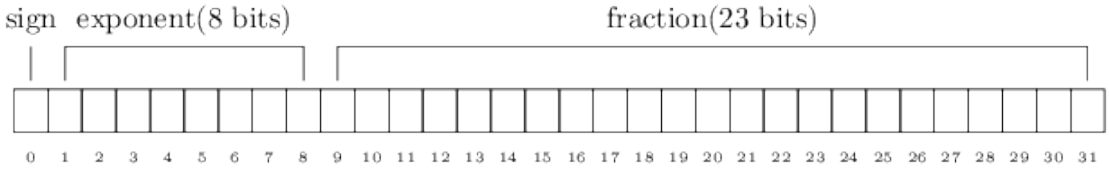
A floating-point number is used to represent real-world fractional value which is a trade-off between range and accuracy because as I said in fractional binary numbers, a decimal fraction cannot represented in binary unless the denominator of that number is an integral power of 2. A number is, in general, represented approximately to a fixed number of significant digits (the *significand*) and scaled using an exponent; numbers are usually binary, octal, decimal or hexadecimal. A number that can be represented exactly is of the following form: significand $\times$ base$^{\text{exponent}}$ where significand $\in \mathbb{Z}$, base $\in \mathbb{N}$, exponent $\in \mathbb{Z}$.

The term floating point refers to the fact that a number's radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number.

2.5.3.1 Representation of Floating-Point Numbers

Given below are pictorial representations of 32-bit floating point numbers:

Figure 2.1 32-bit floating-point numbers.



Similarly in 64-bit floating point numbers we have 1 bit for sign, 11 bits for exponent and 52 bits for fractional part. Clearly zero will be represented by all sign and exponent bits having value 0 for them.

C also has concept of positive and negative infinities. Sign bit is 0 for positive infinity and 1 for negative infinity. Fractional bits are 1 while exponent bits are all 1.

Certain operations cause floating point exceptions like division from zero or square rooting a negative number. Such exceptions are represented by NANs which stands for "not a number". Sign for NaNs is similar i.e. 0 for positive and 1 for negative. Exponent bits are 1 and fractional part is anything but all 0s because that represents positive infinity.

Now let us see a program to see how we can take input and print the floating point numbers.

```
#include <stdio.h>

int main()
{
  float f = 0.0;
  double d = 0.0;
  long double ld = 0.0;
```

```
  printf("Enter a float, double and long double separated by space:\n");
  scanf("%f %lf %Lf", &f, &d, &ld);

  printf("You entered %f %lf %Lf\n", f, d, ld);

  return 0;
}
```

If you run this you might have following output:

```
Enter a float, double and long double separated by space:
3.4 5.6 7.8
You entered 3.400000 5.600000 7.800000
```

By default these print upto six significant digits but doubles have double precision as we have studied.

2.6 New Data Type of C99

There are some new data types introduced in C99. They are `_Bool`, `_Complex` and `_Imaginary`.

2.6.1 Boolean Type

Booleans were not there before C99 specification. It was introduced in ISO C99 specification. The keyword is `_Bool` to declare a boolean variable. Booleans have two possible values as you may know from algebra. The two possible values are `true` and `false` or 1 and 0 respectively. Any non-zero value is treated as `true` including negative value while zero is treated as `false`. `bool`, `true` and `false` are macros which expand to `_Bool`, 1 and 0 respectively. `bool` is used to maintain compatibility with C++. Since `bool` keyword was reserved in C++ so C used `_Bool` but the macro `bool` is used to maintain compatibility between C and C++ code.

Let us see a small program to see demo of booleans.

```
// Boolean demo Program
// Author: Shiv S. Dayal
// Description: Demo of boolean data type

#include <stdio.h>
#include <stdbool.h>

int main()
{
  bool bcpp      = 4;
  _Bool bc       = 5;
  bool True      = true;
  _Bool False    = false;
```

```
  bool bFalseCPP = -4;
  _Bool bFalseC  = -7;

  printf("%d %d %d %d %d %d\n", bcpp, bc, True, False, bFalseCPP, bFalseC) ↩
      ;

  getchar();

  return 0;
}
```

and the output is:

```
1 1 1 0 1 1
```

Note that `true` and `false` are keywords while `True` and `False` are identifiers.

2.6.2 Complex Types

For complex types, there is a system header `complex.h` which internally includes various other headers. However I am giving you the summary here. There are following `#define` macros:

`complex`: Expands to `_Complex` `_Complex_I`: Expands to a constant expression of type `const float _Complex` with the value of the imaginary.

`imaginary`: Expands to `_Imaginary`. `_Imaginary_I`: Expands to a constant expression of type `const float _Imaginary` with the value of the imaginary value.

`I`: Expands to either `_Imaginary_I` or `_Complex_I`. If `_Imaginary_I` is not defined, `I` expands to `_Complex_I`.

Complex types are declared as given below:

1. `float complex fCompZ;`
2. `double complex dCompZ;`
3. `long double ldCompZ;`

Now I will present a summary of library functions provided by `complex.h`

```
//cabs, cabsf, cabsl - these compute and return absolute value
//of a complex number z

double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);

//carg, cargf, cargl - these compute and return argument of a complex
//number z. The range of return value's range from one +ve pi radian
//to one -ve pi radian.
```

```
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);

//cimag, cimagf, cimagl - these compute imaginary part of a complex
//number z and return that as a real number.

double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);

//creal, crealf, creall - these compute real part of a complex
//number z and return the computed value.

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);

//conj, conjf, conjl - these functions compute the complex conjugate
//of z, by reversing the sign of its imaginary part and return the
//computed value.

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);

//cproj, cprojf, cprojl - these functions compute a projection of z
// onto the Riemann sphere: z projects to z, except that all complex
//infinities (even those with one infinite part and one NaN (not a
//number) part) project to positive infinity on the real axis. If z
//has an infinite part, then cproj( z) shall be equivalent to:
//INFINITY + I * copysign(0.0, cimag(z))
//These functions return the computed value.

double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);

//cexp, cexpf, cexpl - these functions shall compute the complex
//exponent of z, defined as e^z and return the computed value

double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);

//clog, clogf, clogl - these functions compute the complex
//natural (base e) logarithm of z, with a branch cut along
```

```
//the negative real axis and return complex natural logarithm
//value, in a range of a strip mathematically unbounded along
//real axis and in the interval -ipi to +ipi along the
//imaginary axis.

double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);

//csqrt, csqrtf, csqrtl - these functions compute the complex
//square root of z, with a branch cut along the negative real
//axis and return the computed value in the range of the right
//half-plane (including the imaginary axis)

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);

//cpow, cpowf, cpowl - these functions compute the complex
//power function x^y, with a branch cut for the first
//parameter along the negative real axis and return the
//computed value.

double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
long double complex y);

//csin, csinf, csinl - these functions compute the complex
//sine of z and return the computed value.

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

//ccos, ccosf, ccosl - these functions compute the complex
//cosine of z and return the computed value.

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);

//ctan, ctanf, ctanl - these functions compute the complex
//tangent of z and return the computed value.

double complex ctan(double complex z);
float complex ctanf(float complex z);
```

```
long double complex ctanl(long double complex z);

//casin, casinf, casinl - these functions compute the complex
//arc sine of z, with branch cuts outside the interval
//[-1, +1] along the real axis and return the computed value
//in the range of a strip mathematically unbounded along the
//imaginary axis and in the interval -0.5pi to +0.5pi radian
//inclusive along the real axis.

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

//cacos, cacosf, cacosl - these functions compute the complex
//arc cosine of z, with branch cuts outside the interval
//[-1, +1] along the real axis and return the computed value
//in the range of a strip mathematically unbounded along the
//imaginary axis and in the interval -0 to +pi radian
//inclusive along the real axis.

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

//catan, catanf, catanl - these functions compute the complex
//arc tangent of z, with branch cuts outside the interval
//[-i, +i] along the real axis and return the computed value
//in the range of a strip mathematically unbounded along the
//imaginary axis and in the interval -0.5pi to +0.5pi radian
//inclusive along the real axis.

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

//csinh, csinhf, csinhl - these functions compute the complex
//hyperbolic sine of z and return the comupted value.

double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

//ccosh, ccoshf, ccoshl - these functions shall compute the
//complex hyperbolic cosine of z and return the computed
//value

double complex ccosh(double complex z);
```

```
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);

//ctanh, ctanhf, ctanhl - these functions compute the
//complex hyperbolic tangent of z and return the computed
//value.

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);

//casinh, casinhf, casinhl - these functions compute the
//complex arc hyperbolic sine of z, with branch cuts
//outside the interval [-i, +i] along the imaginary axis and
//return the complex arc hyperbolic sine value, in the range
//of a strip mathematically unbounded along the real axis
//and in the interval [-i0.5pi, +i0.5pi] along the imaginary
//axis.

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
cacosh, cacoshf, cacoshl - theese functions compute the

//complex arc hyperbolic cosine of z, with a branch cut at
//values less than 1 along the real axis and return the complex
//arc hyperbolic cosine value, in the range of a half-strip
//of non-negative values along the real axis and in the
//interval [-ipi, +ipi] along the imaginary axis.

double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);

//catanh, catanhf, catanhl - these functions shall compute the
//complex arc hyperbolic tangent of z, with branch cuts outside
//the interval [-1, +1] along the real axis and return the
//complex arc hyperbolic tangent value, in the range of a strip
//mathematically unbounded along the real axis and in the
//interval [-i0.5pi, +i0.5pi] along the imaginary axis.

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Hers is a small demo program which explains three functions:

```
// Complex Number Program
```

```
// Author: Shiv S. Dayal
// Description: Demo of complex data type

#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 4.0 + 3.0i;

  printf("Absolute value of z is %lf\n", cabs(z));

  double complex zConj = conj(z);
  printf("Imaghinary part of conjugate is now %lf\n", cimag(zConj));

  return 0;
}
```

and the output is:

```
Absolute value of z is 5.000000
Imaghinary part of conjugate is now -3.000000
```

You must note that in Makefile you must compile it like **$gcc complex.c -o complex -lm**. Note the `-lm` part. It tells to look for definition of these functions in Math library of C. Without it the program won't compile. At this point I encourage you to further explore different functions presented in the summary. There are even more data types for integral type. I am sorry but I am unwrapping the layers one by one. These types are defined in `inttypes.h` and `stdint.h`. The types are `int8_t`, `int16_t`, `int32_t`, `uint8_t`, `uint16_t` and `uint32_t`. The numbers tell you how many bits each data type will occupy. The types without leading `u` are of signed type and the ones with it are of unsigned type. You can use the good old `%d` or `%i` for decimal integers and `%o` and `%x` for octals and hexes. Have a look at headers and try to decipher them.

2.7 `void` and `enum` Types

There are these two types remianing. void type comprises an empty set of values; it is an incomplete type that cannot be completed. You cannot declare an array of `void`s. It is a generic type in the sense that any other pointer to any type can be converted to pointer type of void and vice-versa. It is a low level type and should be only used to convert data types from one type to another and sparingly. A type occupies one byte. Typically you never declare a variable of void type. It is used mostly for casting.

`enum` comprises a set of named integer constant values. Each distinct enumeration constitutes a different enumerated type. In C enums are very much equivalent to integers. You can do all operations of an enum on an enumeration member. An enumeration is is a set of values. It starts from zero by default and increments by one unless specifically specified. Consider the following example:

```
// Description: Demo of enum

#include <stdio.h>

int main()
{
  typedef enum {zero, one, two} enum1;
  typedef enum {alpha=-5, beta, gamma, theta=4, delta, omega} enum2;

  printf("zero = %d, one = %d, two=%d\n", zero, one, two);
  printf("alpha = %d, beta = %d, gamma=%d, theta=%d, delta=%d, omega=%d\n ↩
      ", \
          alpha, beta, gamma, theta, delta, omega);

  return 0;
}
```

and the output is:

```
zero = 0, one = 1, two=2
alpha = -5, beta = -4, gamma=-3, tehta=4, delta=5, omega=6
```

2.8 Constants

We have seen some variables now let us see some constants. There are five categories of constants: character, integer, floating-point, string, and enumeration constant. We will see enumeration constants later first we see remaining four types of constants. There are certain rules about constants. Commas and spaces are not allowed except for character and string constants. Their range cannot outgrow the range of there data type. For numeric type of stants they can have a leading (-)minus sign.

Given below is an example:

```
// Integer constants
// Description: Demo of integer constants

#include <stdio.h>

int main()
{
  int decimal = 7;
  int octal = 06;
  int hex = 0xb;

  printf("%d %o %x\n", decimal, octal, hex);
```

```
  return 0;
}
```

and the output is:

```
7 6 b
```

As you can see there are three different categories for integer constants: decimal constants (base 10), octal constants (base 8) and hexadecimal constants (base 16). Also, you must have noticed how a zero is prefixed before octal type and a zero and x for hexadecimal type. The `%d` format specifier is already known to you for signed decimals. However, now you know two more `%o` and `%x` for unsigned octal and unsigned hexadecimal respectively. For unsigned integer it is `%u`. There is one more format specifier which you may encounter for signed decimal and that is `%i`.

Note that there is nothing for binary constants. I leave this as an exercise to you to convert a number in any base shown above to binary and print it. Also vice-versa that is take a input in binary and convert to these three. Later I will show you this program.

Now let us move to floating-point constants. Again, I will explain using an example:

```
// Floating-point constants
// Description: Demo of floating-point constants

#include <stdio.h>

int main()
{
  float f = 7.5384589234;
  double d = 13.894578834538578234784;
  long double ld = 759.826347823472940235402835820835823082930 4;

  printf("%f %lf, %Lf\n", f, d, ld);

  return 0;
}
```

and the output is:

```
7.538459 13.894579, 759.826348
```

We will learn to change precision later when we deal with format specifiers along with printf and all input/output family. Here also, you learn three format specifiers. Other are `%e` or `%E` for scientific notation of float family. Then there is `%g` or `%G` which uses shorter of `%e` and `%f` types.

Now we move on to character and string type constants and as usual with a small program.

```
// Character constants
// Description: Demo of character constants

#include <stdio.h>
```

```
int main()
{
  char c = 'S';
  char* str ="Shiv S. Dayal";

  printf("%c %s\n", c, str);

  return 0;
}
```

and the output is:

```
S Shiv S. Dayal
```

As I had said that commas and blanks are not allowed in numeric types but you can see both are allowed on character and string types. Also, the string is a character pointer that is it can point to memory location where a character is stored. In this case the string is stored in an area of memory called stack. When memory is allocated the compiler knows how much has been allocated. For string there is something called null character represented by '`\0`' which is used to terminate string. By using this mechanism the program knows where the string is terminating. It is treated in next section as well.A very interesting thing to be noted is char is considered to be an integral type. It is allowed to perform addition etc on char type. Till now you have learnt many format specifiers and have seen they all start with `%`. Think how will you print `%` on stdout. It is printed like `%%`. It was simple,wasn't it? C program have got something called ASCII table which is a 7-bit character table values ranging from0 to 127. There is also something called escape sequences and it is worth to have a look at them.

2.9 Escape Sequences

All escape sequences start with a leading `\` . Following table shows them:

Note that there is no space between two backslashes. Sphinx does not allow me to write four continuous backslashes. Now we will talk about all these one by one. `\0` which is also known as `NULL` is the string terminating character, as said previously, and must be present in string for it to terminate. For example, in our character constant program the str string is "`Shiv S.Dayal`". So how many characters are there 13? Wrong 14! The `NULL` character is hidden. Even if we say `str= "";` then it will contain one character and that is this `NULL`. Many standard C functions rely on this presence of `NULL` and causes a lot of mess because of this. The bell escape sequence if for a bell from CPU. Let us write a program and see it in effect.

```
// Bell Program
// Description: Demo of bell escape sequence

#include <stdio.h>

int main()
```

Table 2.4 Escape Sequences

Character	Escape Sequences	ASCII Value
null	\0	000
bell (alert)	\a	007
backspace	\b	008
horizontal tab	\t	009
newline(line feed)	\n	010
vertical tab	\v	011
form feed	\f	012
carriage return	\r	013
quotation mark (")	\"	034
apostophe (')	\'	039
question mark	\?	063
backslash	\\	092

```
{
  printf("hello\a");

  getchar();

  return 0;
}
```

The output of this program will be `hello` on stdout and an audible or visible bell as per settings of your shell. Notice the `getchar()` function which waits for input and reads a character from stdin. Next is backspace escape sequence. Let us see a program for its demo as well:

```
// Backspace Program
// Description: Demo of backspace escape sequence

#include <stdio.h>

int main()
{
  printf("h\b*e\b*l\b*l\b*o\b*\n");
  printf("\b");

  getchar();

  return 0;
}
```

and the output is:

```
*****
```

It is hello replaced by *. A minor modification in this program to replace the character as soon as key is pressed by some other character will turn it into a password program. Backspace escape sequence means when it is encountered the cursor moves to the previous position on the line in context. If active position of cursor is initial position then C99 standard does not specify the behavior of display device. However, the behavior on my system is that cursor remains at initial position. Check out on yours. The second `printf` function determines this behavior.

Next we are going to deal with newline and horizontal tab escape sequences together as combined together they are used to format output in a beautiful fashion. The program is listed below:

```
// Newline and Horizontal tab program Program
// Description: Demo of newline and horizontal tab escape sequence

#include <stdio.h>

int main()
{
  printf("Before tab\tAftertab\n");
  printf("\nAfter newline\n");

  getchar();

  return 0;
}
```

and the output is:

```
Before tab      Aftertab

After newline
```

Here I leave you to experiment with other escape sequences. Feel free to explore them. Try various combinations; let your creative juices flow.

Chapter 3

Console I/O

IO stands for input/output. C provides only mechanism to interact through console using its standard library. C does not provide ways to have GUI although that is possible with various GUI libraries most notable being GTK and KDE. However, discussing about GTK or KDE is out of scope of this book. In this chapter we will focus on console output facilities of C because any program we write at this stage will be meaningless if it has no input/output. Typically when we interact with a C program we give input using keyboard which is also referred as `stdin` stream. The output is monitor or `stdout` stream. There is one more stream `stderr` which is generally redirected to monitor or a log file. For historical reasons these are known as `FILE` stream which represents the datatype of these streams. `FILE` is capable of representing other streams which are disk based for example a file on your hard drive. There are more type of input devices on a modern computer. For example, network i/o is there. Whenever you browse web or download a file through your Internet connection network i/o comes into play. There is an opengroup which specifies functions for network related functions. Operating systems like GNU/Linux are POSIX compatible which defines how network i/o will be used. Even a printer is a special output device, a camera input, speakers output, microphone input and so on. In this books we are concerned with keyboard input, output on monitor and i/o using files. Other types of i/os are out of scope of this book.

However, before we go on with i/o I would like to present C's memory model which will be needed by our discussion of i/o related functions. However, if things do not make sense even then please go through it and come later to understand more.

3.1 C's Memory Model

As you may be knowing RAM(random access memory) is the area which is used as primary memory. Whenever we execute a program the first thing which happens is that it gets loaded into memory. Now a binary program becomes a process when it is running i.e. a running program is referred as process. All processes have memory area divided into different portions. These portions are known as data segment, stack and code or text segment. Data segment is further split in three parts; initialized data segment, uninitialized data segment or BSS which is name after an ancient assembler Block

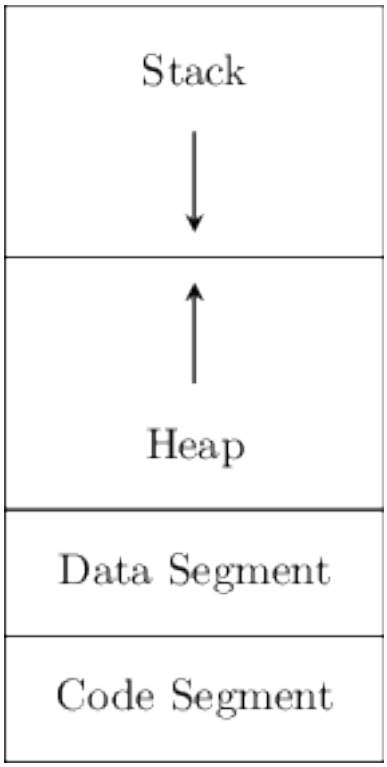
Started by Symbol and heap. Initialized data segment contains initialized global variables and static variables. For uninitialized data segment it is same as above just that the variables are not initialized explicitly but implicitly to zero.

Heap is the largest area of memory used for dynamic memory allocation. As you will see later that you can manage heap using `malloc(), calloc(), realloc(),` and `free()`. Note that compiler does not manage memory allocated for you. You, the programmer, are responsible for allocating and freeing up memory in area. If heap gets full os will use virtual memory or swap space on hard disk. Objects allocated on heap persist across function calls. However, there are some very nasty problems, which, come in picture when you use heap. There are several of them. You may forget to allocate memory and want to dereference unallocated pointer. You may have initialized it to `NULL` and try to dereference that. You may allocate and free twice. You forgot to set pointer to `NULL` after freeing it. And last but not the least you loose all pointers to the memory area before you can free. The nature with this particular problem is that if your program is going to run for long time then it is going to consume more and more memory. Because of its nature it is known as memory leak. It is very difficult to detect such problems in code which does not run for long periods of time. Our friend valgrind will come to help up with this problem. When a memory leak happens it eats up RAM slowly and then operating system has to use virtual memory as explained above. In a nutshell, I will say that heap means you have to handle it.

Stack is relatively simple. All non-static and non-register variables go on stack if not allocated dynamically. Stack variables do not retain there value across function calls unless they are passed as pointers. Also, when they go out of scope, that is the scope in which they were declared ends, they will be kind of lost. The way in which stack frame moves the same area will be used for new variables. However, stack is very limited (compared to heap) and in deeply nested function calls or recursion (you will see these in Functions chapter) stack may get full and program may crash. The reason for crashing is that operating system will not use virtual memory but will do a segmentation fault in its place. GNU/Linux allow its users to modify the stack size by `ulimit` command. Note that stack and heap are adjacent in memory and grow in opposite direction.

Code segment or text segment is an area where the executable instructions of program reside. It is typically constant and read-only area unless your system allows self-modifying code. Following diagram shows the memory layout.

Figure 3.1 C's Memory Model



In this chapter we will look at only those functions, which, allow us to do console i/o. We will begin with our familiar friends `printf` and `scanf`.

3.2 printf

The prototype of printf is given by:

```
int printf(const char* fmt, ...);
```

Let us take a minute to understand this as we have not yet covered functions. The first word is `int` which denotes the return type of the `printf` function. This is no. of characters printed. Then we have name of the function. `fmt` is the format string of type `const char`. In C, strings are either character arrays or character pointers. Here, `const` means `printf` will not modify the format string. The `...` means variable no. of arguments, which can be 0 also, to be supplied to `printf`.

`printf` is a string based output function that is It writes character strings to `stdout`. The data which has to be written is formatted by format string as shown previously. After the format specifier it expects as many arguments as specified in format string. The characters which are not like, say

`%d` for example, arecalled ordinary characters. These are simply copied to output stream, which, is `stdout` for `printf`. The `%d` like conversion charcaters are known as conversion specification or format specifiers. Each conversion specification should be augmented with one one argument. The results are undefined if there are insufficient arguments for the format. If extra arguments are given the excess arguments will be evaluated but are otherwise ignored. However, there is a big problem here! There is no type-safety. In general compiler will warn you about it and you, the programmer, are responsible for giving correct format string, correct no. of correct type of arguments. Consider the following program for example:

```
// printf demo
// Description: printf demo

#include <stdio.h>

int main()
{
  printf("%d %d\n", 3, 8);

  //do not mess it. undefined behavior
  printf("%d %d\n", 5);

  //extra arguments ignored
  printf("%d %d\n", 3, 5, "hello");

  //legal because char is integer type
  printf("%d\n", 's');

  //wrap around of integer as char
  printf("%c\n", 836);

  //do not mess with type-safety
  int i = printf("%d\n", "hello");
  prinf("%d\n", i);

  return 0;
}
```

now that if you give the command like `clang printf.c` then you will be shown following warnings:

```
printf.c:12:14: warning: more '%' conversions than data arguments [- ↩
    Wformat]
printf("%d %d\n", 5);
               ~^
printf.c:15:26: warning: data argument not used by format string [-Wformat ↩
    -extra-args]
        printf("%d %d\n", 3, 5, "hello");
             ~~~~~~~~~        ^
```

```
printf.c:24:19: warning: conversion specifies type 'int' but the argument  ↩
    has type
'char *' [-Wformat]
      int i = printf("%d\n", "hello");
                      ~^     ~~~~~~~
                      %s
3 warnings generated.
```

Clearly this is not a good sign for any program. A program should compile cleanly. In our case compiler is generating binary even though there are warnings. You can make compiler generate more warnings by issuing a `-Wall` flag. You can also treat all warnings as errors by passing `-Werror` to compiler. These two options will ensure that your code has no warnings. Now let us move to output and try to understand it. The output on my system is as given below. It may differ on your system:

```
3 8
5 8
3 5
115
D
134514119
10
```

First `printf` is correct as expected. The second line causes undefined behavior. You may think it is the previous 8 but rest assured it is not guaranteed that it will always the case. Ii is **UNDEFINED**. Third printf is also fine in the sense that extra argument is ignored. Fourth and fifth are normal. Sixth is again a big problem. You are trying to print a decimal integer while argument is a character string. There is no way for compiler to determine that what should be printed which will fit on standards.

A full detail of all conversion specification is given in specification at Section28.6.1, "The fprintf function" (§iso.7.21.6.1), which lists `fprintf` function but conversion specifiers are same as `printf`.

In real-world most of the time the conversion specifiers are kept simple. Given below is a sample program showing some of the things given above:

```
// Format Specifiers
//Desciption: It is a demo of several format specifiers

#include<stdio.h>

int main()
{
  int i   = 343456;
  float f = 123;
  long double ld = 78939.9347;

  printf("% d\n", i);
  printf("%+d\n", i);
```

```
  printf("%#o\n", i);
  printf("%#f\n", f);
  printf("%-08i\n", i);
  printf("%08i\n", i);
  printf("%8i\n", i);
  printf("%hhi\n", i);
  printf("%hi\n", i);
  printf("%li\n", i);
  printf("%lli\n", i);
  printf("%ji\n", i);
  printf("%zi\n", i);
  printf("%ti\n", i);
  printf("%8.8f\n", f);
  printf("%8.8Lf\n", ld);

  return 0;
}
```

and the output is:

```
 343456
+343456
01236640
123.000000
343456
00343456
  343456
-96
15776
343456
4638355772471066016
4638355772471066016
343456
343456
123.00000000
78939.93470000
```

I suggest you to read the desciption of conversion specifiers and experiment with various parameters to get different kind of output.

3.3 scanf

The prototype of `scanf` is given below which is very similar to `printf`.

```
int scanf(const char * restrict format, ...);
```

`scanf()` is sister of `printf()`. They work in tandem. As its name says scan function it scans `stdin` or keyboard for input. Its signature is same as that of `printf()`. It raeds bytes from keyboard input, interprets them according to format string. It also expects a set of pointer arguments as opposed to values for `printf()`. The pointers indicate where the interpreted data from the input will be stored. The result is UNDEFINED if there are less number of pointer arguments than the number of conversion specifers in format string. Excess arguments will be evaluated but ignored. The format string can have only white-space characters or an ordinary character (neither '%'; nor a white-space character) or a conversion specification.

The full detail of conversion specification can be found at Section28.6.2, "The fscanf function" (§iso.7.12.6.2) which lists `fscanf` function but conversion specifiers are same for both.

Time for some code. You have already seen many examples of `scanf` so I will just explain some concepts here. Consider the following program:

```
// Author: Shiv S. Dayal
// Description: Demo of string input

#include <stdio.h>

int main()
{
  char str[128] = {0};

  scanf("%s", str);
  printf("You entered:\n%s\n", str);

  return 0;
}
```

and the output is:

```
Hi! My name is Shiv.
You entered:
Hi!
```

It is certainly not the corect output. We had expected to see like: "Hi! My name is Shiv.". What happend to input string after "Hi!". Well, in a form given above for `scanf()` it will stop taking input after white-space for character strings. For numerics it does not matter as it does not match the format. For characters it is character-by-character so no confusion either. So what if you want to have the entire string including white-spaces. Use `[^n]` as given below:

```
// Author: Shiv S. Dayal
// Description: Corrected demo of string input

#include <stdio.h>

int main()
{
  char str[128] = {0};
```

```
  scanf("%[^\n]s", str);
  printf("You entered:\n%s\n", str);

  return 0;
}
```

and the output is:

```
Hi! My name is Shiv.
You entered:
Hi! My name is Shiv.
```

What if you want to filter a string based on certain patterns. For example, a charcater string does not contain more that a single space, English alphabets, period and digits. To scan such a string you can define a pttern as program given below shows:

```
// Author: Shiv. S Dayal
// Description: Demo of []

#include <stdio.h>

int main()
{
  char c[100]={0};

  scanf("%[ A-Za-z0-9!.]", c);
  printf("%s\n", c);

  return 0;
}
```

and the output is:

```
Hi! My name is Shiv! My phone no. is 1234. %^$&*
Hi! My name is Shiv! My phone no. is 1234.
```

There is also a major problem associated with input and that comes when you have characters involved. Consider the following program:

```
// Author: Shiv S. Dayal
// Description: Demo of scanf() function

#include <stdio.h>

int main()
{
  int   i = 0;
  float f = 0.0;
```

```
  char  c1 = '\0';
  char  c2 = '\0';
  char  c3 = '\0';

  printf("Enter an integer, a float and three character one by one:\n");

  scanf("%d", &i);
  scanf("%f", &f);
  scanf("%c", &c1);
  scanf("%c", &c2);
  scanf("%c", &c3);

  printf("You entered\n");
  printf("%d\n", i);
  printf("%f\n", f);
  printf("%c\n", c1);
  printf("%c\n", c2);
  printf("%c\n", c3);

  return 0;
}
```

and the output is:

```
2
3.4
s
You entered
2
3.400000


s
```

What is happening here is that newline entered by our `RET` key is getting assigned to `c1` and `c3`. That is why the program accepted only second character. The enter after `float f;` was assigned to `c1` and the character entered to `c2` and then the RET newline to `c3`. There is a very simple way to recover from this:

```
// Author: Shiv S. Dayal
// Description: Demo of scanf() function

#include <stdio.h>

int main()
{
  int   i = 0;
  float f = 0.0;
```

```
  char  c1 = '\0';
  char  c2 = '\0';
  char  c3 = '\0';

  printf("Enter an integer, a float and three character one by one:\n");
  scanf("%d", &i);
  scanf("%f", &f);
  scanf(" %c", &c1);
  scanf(" %c", &c2);
  scanf(" %c", &c3);

  printf("%d\n", i);
  printf("%f\n", f);
  printf("%c\n", c1);
  printf("%c\n", c2);
  printf("%c\n", c3);

  return 0;
}
```

The whitespace character shown will eat up all the white-space given after the previous input. This concludes our discussion on `printf()` and `scanf()`. Now we will move to another set of i/o functions which take character string without filtering and print it to screen without filtering. What I am going to discuss are `gets(), fgets(), puts()` and `fputs()`.

3.4 Character String I/O Functions

These functions are very simple compared to `printf()` and `scanf()`. They take a pointer to a character array or a character pointer and fill it with input or print it to monitor. Note that `gets()` and `puts()` work only with stdin and stdout respectively while `fgets()` and `fputs()` work with `FILE` streams. They can read and write to file streams that is. Here is a sample program:

```
// Author: Shiv S. Dayal
// Description : Demo of string i/o
#include <stdio.h>
#include <stdlib.h>

int main()
{
  char cStack[1024] = "";
  char *cHeap = (char*)malloc(sizeof(1024));

  gets(cStack);
  puts(cStack);

  cHeap = fgets(cHeap, 1024, stdin);
```

```
  fputs(cHeap, stdout);

  return 0;
}
```

and the output is:

```
Hi!
Hi!
Hello!
Hello!
```

First "Hi!" and "Hello!" are keyboard inputs. Do not worry about array and pointer syntax at the moment. Just see the difference between function calls. Their is a problem with `gets()` that it can cause buffer overflow. If input is bigger than 1024 bytes including the null terminator then buffer overflow will happen. Note how you can prevent it with fgets() by specifying the number of characters you want to read. Rest of input will be ignored by `fgets()`. This is a security hole and therefore you should never ever use `gets()`.

Time for single character input/output.

3.5 Single Character I/O

There are several functions for single character i/o. They are `getc(), putc(), getchar(), putchar(), fgetc()` and `fputc()`. Apart from `getchar()` and `putchar()` rest can do any FILE stream-based i/o. Let us see them as they are mostly trivial.

```
// Author: Shiv S. Dayal
// Description: Single character funciton demo
#include <stdio.h>

int main()
{
  char c ='';

  c = getchar();
  putchar(c);

  c = getchar();
  putchar(c);

  c = fgetc(stdin);
  fputc(c, stdout);

  c = getchar();
  putchar(c);
```

```
  c = getc(stdin);
  putc(c, stdout);

  return 0;
}
```

and the output is:

```
4
4
5
5
6
6
```

The first 4, 5 and 6 were keyboard inputs. Note the use of extra `getchar()` and `putchar()` to handle the situation we faced during `scanf()`.

So we have seen many functions and programs for console i/o. File i/o is still there and will be covered later. This chapter ends here. In the next chapter we will have operators and expressions.

Chapter 4

Operators and Expressions

Operators and expressions are in the core of every programming language. They form the major part of BNF grammar. They also decide how the syntax will look like. You as a programmer will spend considerable time using C operators. C has sevral type of operators like arithmetic operators, relational operators, bitwise operators, unary operators, logical operators to name some of them. Since C was first of very poopular structured general-pupose lnguages therefore many modern language use almost all the operators and supplement with their own. It is needless to say that to become a good programmar you must know all the operators of C and know where to use which one as it may decide performance, readability, simplicity of your code. Whenever you see array and pointer in following sections just plow through them. All will be clear soon.

Before we can proceed to discuss operators and expressions I will explain scope, linkage and storage durations which can be applied to variables. These are given in specification starting at §(iso.6.2.1) and ending at §(iso.6.2.4).

4.1 Scope of an Identifier

Till now we have seen plain variables and their identifiers. However, there are other identifiers as well which will be discussed later. For now we will consider scope of plain variables. In general there are three kinds of scope. Global scope, function scope and block scope. Variables declared outside any function have global scope and they persist throughout the lifetime of the program. Variables declared inside functions at outermost level have function scope and they live as long as function remains active. A block in C is marked by braces { and }. Function bodies are also marked by this. Here I mean blocks inside a function. Starting from C99 you can declare variables anywhere inside a function and this block variables which have less lifetime than functions are possible. We will see more of these when we see more code. Note that identifiers can be reused in different scopes. For example, a loop index integer identifier is repeated many times but every time it is a new variable(We will see loops soon). Two identifiers have same scope if and only if their scope terminates at the same point.

4.2 Linkages of an Identifier

There are three different kinds of linkages. External, internal and none. Global variables and functions have external linkage as long as they are not static. If they are static then they have internal linkage. By external linkage we mean that for a program which consists of multiple source code files these functions and variable identifiers can be referred in files other than in which they are declared. When functions and global variables are static i.e. they have internal linkage they cannot be accessed in other source code files.

The following identifiers have no linkage: an identifier declared to be anything other than an variable or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier `extern`.

4.3 Storage Duration of Objects

There are four storage durations. Static, thread, automatic and allocated. Here, we will not discuss thread which we will talk about later. A static variable which is local to a function of global variable has static duration and it lives in data segment in memory and has static storage duration. A variable local to a function or block which is not dynamically allocated on heap by using either of `malloc, calloc` or `realloc` has automatic storage and has function or block has automatic storage and is cleaned up automatically and it lives on stack. Allocated storage duration variables can persist as long as they want after allocation on heap by using one of `malloc, calloc` and `realloc` as long as the name is kept in scope and a corresponding `free` is not called on that name of the variable. Now let us discuss operators and expressions.

Whenever operators and expressions come in picture you may have a set of mixed data then to perform oration data is converted from one type to another. This has an entire section devoted to it in specification at §(iso.6.3). There are two types of conversions. Many operators convert their operands silently which is called “implicit conversion” and then we have cast operators which we can use to explicitly convert values from one type to another which is called “explicit conversion”. We will first see implicit conversion.

4.4 Usual Arithmetic Conversions

Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is alo the corresponding real type of the result, whose type domain is the type domian of the operands if they are the same, and complex otherwise. This pattern is called the usual *arithmetic conversions*:

- First, if the corresponding real type of either operand is `long double`, the other operand is converted, without change of type domain, to a type whose corresponding real type is `long`

`double`.

- Otherwise, if the corresponding real type of either operand is `double`, the other operand is converted, without change of type domain, to a type whose coreesponding real type is `double`.
- Otherwise, if the corresponding real type of either operand is `float`, the other operand is converted, without change of type domain, to a type whose coreesponding real type is `float`.
- Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
  - If both operands have the same type, then no further conversion is needed.
  - Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
  - Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
  - Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
  - Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.
- The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.

4.5 Arithmetic Operators

There are five here. +, -, *, / and %. Let us see a simple example:

```
// Arithmetic operators
// Description: Demo of arithmetic operators

#include <stdio.h>

int main()
{
  int i = 10;
  float f= 6.45;
  char c = 'A';
  int iResult = 0;
  float fResult = 0.0;
  char cResult = '\0';
```

```
  cResult = c + i;
  printf("cResult = %c\n", cResult);
  cResult = cResult - 5;
  printf("cResult = %c\n", cResult);

  iResult = i - 10;
  printf("iResult = %d\n", iResult);
  iResult = i * c;
  printf("iResult = %d\n", iResult);
  iResult = (i + c)/3;
  printf("Result = %d\n", iResult);
  iResult = (i + c)%2;
  printf("iesult = %d\n", iResult);

  fResult = f * 2.12;
  printf("fesult = %f\n", fResult);
  fResult = f - i;
  printf("fesult = %f\n", fResult);
  fResult = f / 1.12;
  printf("fesult = %f\n", fResult);
  fResult = 1 % 3;
  printf("fesult = %f\n", fResult);

  return 0;
}
```

and the output is:

```
cResult = K
cResult = F
iResult = 0
iResult = 650
Result = 25
iesult = 1
fesult = 13.674000
fesult = -3.550000
fesult = 5.758928
fesult = 1.000000
```

First `cResult` is sum of `'A' + i` which is `'K'` as `'K'` comes ten positions after A in ASCII table. Then we subtract five and go back to F.

First `iReasult` is 10 - `i` where value of `i` is 10 hence result is 0. Next we multiply it with `i` which contains `'A'` who has got ASCII value of 65 and result becomes 650. Then We take sum of `'A'` and `i` and divide by 3 so the result is 25 as it is a division of 75 by 3. Next we use modulus operator and remainder is 1. Note that in case of / and % if denominator is zero the behavior is undefined.

Same way you can udnerstand floating-point operations. Note that you cannot use modulus operator if either of the operands are floating-point numbers as it will make no sense because of

data type promotion rules. Here data type promotion rule says smaller data types will be converted to bigger data types. Also, if there is a data type on left side of assignment the result of applying the operator to operands will be converted to the type of that. chars are promoted to ints, ints are promoted to floats anf floats to double. The point is that conversion will try to keep as much data as possible.

4.6 Relational Operators

There are four relational operators: <, >, <= and >=. Once again these are described in chapter 4. Let us see an example:

```
// Author : Shiv S. Dayal
// Description : Demo of relational operator

#include <stdio.h>
#include <stdbool.h>

int main()
{
  int i = 4, j = 5;
  _Bool result = 0;

  result = i < j;
  printf("%d\n", result);

  result = i > j;
  printf("%d\n", result);

  result = i <= j;
  printf("%d\n", result);

  result = i >= j;
  printf("%d\n", result);

  return 0;
}
```

and the output is:

```
1
0
1
0
```

Note that you should not apply these to floating-point data types as they may not be represented correctly and two different entities have the same internal representation.

4.7 Equality Operators

There are two equlity operators == and !=.

```
// Author : Shiv S. Dayal
// Description : Demo of equality operator

#include <stdio.h>
#include <stdbool.h>
int main()
{
  int i = 4, j = 5;
  _Bool result = 0;

  result = i == j;
  printf("%d\n", result);

  result = i != j;
  printf("%d\n", result);

  return 0;
}
```

and the output is:

```
0
1
```

4.8 Increment and Decrement Operators

There is one increment and one decrement operator. ++ and --. Both come in two forms prefix and postfix. First we will see prefix versions then postfix ones. There is only one constraint on prefix operators of these and that is the operand of the prefix increment or decrement operator will have qualified or unqualified real or pointer type and will be a modifiable lvalue.

```
// Author : Shiv S. Dayal
// Description : Demo of increment decrement operators

#include <stdio.h>

int main()
{
  float f = 7.123;

  printf("%f\n", ++f);
  printf("%f\n", --f);
```

```
  printf("%f\n", f++);
  printf("%f\n", f--);
  printf("%f\n", f);

  return 0;
}
```

and the output is:

```
8.123000
7.123000
7.123000
8.123000
7.123000
```

4.9 Logical Operators

There are two such operators. && logical AND and II locical OR. Both the operators have the same constraints and it is that both the operands will have scalar type.

The && operator gives 1 if both the operands are non-zero else 0. The result type is int. It is different from bitwise & operator in the sense that it guarantess left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand is 0 then the second operand is not evaluated. This is known as "short-circuit evaluation".

The II operator gives 1 if any of operands are non-zero else it gives 0. Same as logical AND operator and unlike bitwise I operator it guarantees left-to-right evaluation and same goes for sequence points. If first operand is non-zero, the second is not evaluated.

```
// Author : Shiv S. Dayal
// Description : Demo of logical AND & OR operators

#include <stdio.h>
#include <stdbool.h>

int main()
{
  int i = 4, j = 5, k = 0;
  bool result;

  result = i&&j;
  printf("%d\n", result);

  result = i||j;
  printf("%d\n", result);
```

```
  result = k&&j;
  printf("%d\n", result);

  result = k||j;
  printf("%d\n", result);

  return 0;
}
```

and the output is:

```
1
1
0
1
```

note the use of `bool` here instead of `_Bool`.

4.10 Bitwise Operators

There are three bitwise operators. &, I, and ˆ. AND, OR and EX-OR respectively. OR is also called inclusive OR. These have the same contsraints and it is that operands should be integer types. The usual arithmetic conversions are performed on the operands.

```
// Author : Shiv S. Dayal
// Description : Demo of bitwise operators

#include &stdio.h>
#include &stdbool.h>

int main()
{
  int i = 4, j = 5;
  int result;

  result = i&j;
  printf("%d\n", result);

  result = i|j;
  printf("%d\n", result);

  result = i^j;
  printf("%d\n", result);

  return 0;
}
```

and the output is:

```
4
5
1
```

4.11 Bitwise Shift Operators

The constraint is same as other bitwise operators that operands should be integers. The integer promotions are performed on each of the operands. The type of the result if that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

```
// Author : Shiv S. Dayal
// Description : Demo of shift operators

#include <stdio.h>

int main()
{
  int i  = 4;
  char c ='A';
  int result;

  result = c<<i;
  printf("%d\n", result);

  result = c>>i;
  printf("%d\n", result);

  return 0;
}
```

4.12 Assignment Operators

These are = *= /= %= += -= <<= >>= &= ˆ= and |= The only constraint is that left operand should be modifiable lvalue. An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand is sequenced after the value computations of the left and right operands. The evaluations of the operands are unsequenced.

```
// Author: Shiv S. Dayal
// Description: Demo of compound assignments.

#include <stdio.h>

int main()
{
  int i   = 3;
  int j   = 3;
  float f = 4.7;
  float result=0.0;

  result += i+f;
  printf("%f\n", result);

  result -= f;
  printf("%f\n", result);

  j <<= i;
  printf("%d\n", j);

  return 0;
}
```

and the output is:

```
7.700000
3.000000
24
```

4.13 Conditional Operators

```
// Author : Shiv S. Dayal
// Description : Demo of conditional operator

#include <stdio.h>

int main()
{
  int i = (4 < 5)? 7:10;

  printf("%d\n", i);

  return 0;
}
```

output is 7 as 4 is less than 5 which is true.

4.14 Comma Operator

It is a very simple operator. The left operand of a comma operator is evaluated as a void expression; there is a sequence point between its evaluation and that of the right operand. Then the right operand is evaluated; the result has its type and value. A comma operator does not give an lvalue.

4.15 sizeof Operator

You have already see sizeof operator in second chapter when we saw sizes of data types. However here is the constraint: the sizeof operator will not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member.

The sizeof operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

When applied to an operand that has type char, unsigned char, or signed char, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

4.16 Unary Arithmetic Operators

We will see casting, array subscripting, function parenthes, address and indirection operators later at appropriate time. For now I am going to tell you about operator precedence and associativity and then about grouping parenthes. Given below is the table for operator precedence and associativity, however, you may not be familiar with few of them but later you will be:

4.17 Grouping parentheses

Grouping parentheses are used to override operator precedence and group expressions. NEVER EVER try to memorize and rely on precedence of operators. Always use grouping parentheses. Till now I have shown very simple examples of operators; here are some complex ones:

```
// Author: Shiv Shankar Dayal
// Description: Demo of grouping parentheses

#include <stdio.h>
```

Table 4.1 Priority and assiciativity table

Operators	Associativity
() [] . -> ++ – (postfix)	left-to-right
++ – + - (unary) ! ~ (types) * & sizeof	right-to-left
* / %	left-to-right
+ - (Addition/Subtraction)	left-to-right
<< >>	left-to-right
< > <= >=	left-to-right
== !=	left-to-right
&	left-to-right
^	left-to-right
|	left-to-right
&&	left-to-right
||	left-to-right
Assignemnt operators	right-to-left
,	left-to-right

```
int main()
{
  printf("%f\n", 5.2*(3.7+2.3));
  printf("%d\n", ((4<5)||(7^5)));

  return 0;
}
```

This small program shows you what can go wrong if you rely on memory. It allows you do addition first and then multiplcation. Inner parentheses are evaluated first then outer ones. This concludes our chapter on operators and expressions. Next we focus on control statements and flow statements.

Chapter 5

Control Flow

There are three things you will learn in this chapter. Switching the path of execution in program depending upon program variables or states using control statements. Repeating a set of instructions using loops. Bypassing certain set of instructions in a loop and jump around. Collectively, these elements of C allow or enable you to take driver's seat over the control over a C program. You will spend much of your programming time even in future using these basic elements. Let us begin with if-else without spending much time over boring stuff. Before we proceed I would like to tell you about storage classes of array and their scope. I could have covered it in second chapter but I did not want to scare you with too many things in itself.

5.1 if else Statement

An if-else statement may consist of only `if` or both `if` and `else` or `if` and `else if` or `if, else if` and `else`. An if-else statement must have `if` at the beginning, zeor or more `else if` may come after `if` or before `else` and `else` must come at end. `else if` and `else` are optional and may not come. Consider the following program:

```
//Description : Demo of if-else statements.

#include <stdio.h>

int main()
{
  int i = 0, j= 0;

  printf("Please enter two integers i and j:\n");
  scanf("%d%d", &i , &j);

  if(i==4)
    printf("you entered 4 for i.\n");
```

```
  if(i==7)
  {
    printf("you entered 7 for i.\n");
    printf("I am happy for you.\n");
  }
  else
  {
    printf("You did not enter 7 for i.\n");
  }
  if(i==7)
  {
    printf("you entered 7 for i.\n");
    printf("I am happy for you.\n");
  }
  else if(j==8)
    printf("You entered 8 for i.\n");

  if(i==7)
    printf("you entered my lucky number.\n");
  else if((i==7) &&(j==8))
    printf("May god bless you!\n");
  else
    printf("You entered bad number.\n");

  return 0;
}
```

and the output is:

```
Please enter two integers i and j:
4
6
you entered 4 for i.
You did not enter 7 for i.
You entered bad number.
```

As you can see from first if sttatement that if you enter the value of i as 4 then the printf will be executed and you will be able to see it. Note that if there are multiple lines below if which you want to execute then you must put them in a block using curly braces. If you just want to execute one line then these curly braces are optional. Note that how you must use curly braces if you have more than one line and you want to execute them. Also, see the syntax for missing else and missing else if. One if-else can be nested inside another for example see the following code:

```
//Author: Shiv S. Dayal
//Description : Demo of if-else statements.

#include <stdio.h>
#include <string.h>
```

```
int main()
{
  char fName[128]={0}, lName[128]={0};

  printf("Enter your first name and last name in that order:\n");
  gets(fName);
  gets(lName);

  if(strcmp(fName, "Shiv") == 0)
  {
    if(strcmp(lName, "Dayal") == 0)
      printf("Your name is Shiv Dayal.\n");
  }
  else
  {
    printf("Your name is %s %s.\n", fName, lName);
  }

  return 0;
}
```

and the output is:

```
Enter your first name and last name in that order:
Shiv
Dayal
Your name is Shiv Dayal.
```

another run:

```
Enter your first name and last name in that order:
Richard
Stallman
Your name is Richard Stallman.
```

when first if matches but else does not:

```
Enter your first name and last name in that order:
Shiv
Stallman
```

Note the usage of nested if-else. Also, note how `strcmp` has been used to compare two strings and `gets` to read the input. `gets` is dangerous but it is simple that is why has been used here. You can read about it at the link of opengroup. We will see this in more detail towards the end when we deal with chapter named C Standard Library.

Assignment in if/else-if

Always remember the expression inside if evaluates to a boolean so you should never do an ASSIGNMENT inside if and else if as it will always evaluate to what is assigned. It can render all your logic meaningless. C is not Python, where assignment inside if is not allowed. However, if you assign 0 to some variable it will evaluate to `false`.

5.1.1 Dangling else Problem

The `else` part has a property that it will cling to closest if. So the following piece of code may give you surprise:

```
if(x==1)
  if(y>2)
    printf("foo\n");
else
  printf("bar\n");
```

Now consider `x!=1` then you may think that bar will be printed. However, that will not be the case. The else part clings to inner if. This can be fixed by using curly braces.

5.2 switch Statement

`switch` statement is kind of if-else replacement to simplify it. Usage of switch statement is to compare one expression with others, and then execute a series of sub-statements inside case and default based on the result of the comparisons. Note that switch statement takes only integers or integreal type as its argument and same is valid for its cases. Consider the following example:

```
//Author: Shiv S. Dayal
//Description : Demo of if-else statements.

#include <stdio.h>

int main()
{
  int i  = 65;

  switch(i)
  {
    case 'A':
      printf("Value of i is 'A'.\n");
      break;
```

```
    case 'B':
      printf("Value of i is 'B'.\n");
      break;
    default:
      break;
  }

  return 0;
}
```

and the output is:

```
Value of i is 'A'.
```

Notice the usage of `break`. It is used to terminate execution once a match has been found for a particular case else what will happen is shown below:

```
//Author: Shiv S. Dayal
//Description : Demo of switch statement.

#include <stdio.h>

int main()
{
  int i  = 65;

  switch(i)
  {
    case 'A':
      printf("Value of i is 'A'.\n");
    case 'B':
      printf("Value of i is 'B'.\n");
    default:
      printf("Value of i is %c.\n", i);
      break;
    }

  return 0;
}
```

and the output is:

```
Value of i is 'A'.
Value of i is 'B'.
Value of i is A.
```

This is also known as fall through of a `switch` statement. Notice, the use of default that how it is analogous to `else` statement. `switch` statements can also be nested inside each other. However, node that lots of nesting is not good. At most 2-3 levels are more than enough else you should look at alternative ways of writing code.

5.3 while Loop

Of three loops I am first going to cover `while` loop. It is simplest of three. I will just give an example for you to understand.

```
//Author: Shiv S. Dayal
//Description : Demo of while statement.

#include <stdio.h>

int main()
{
  int i = 0;

  while(i <= 10)
  {
    printf("%d * %2d = %4d\n", 2, i, 2*i);
    i++;
  }

  return 0;
}
```

and the output is:

```
2 *  0 =    0
2 *  1 =    2
2 *  2 =    4
2 *  3 =    6
2 *  4 =    8
2 *  5 =   10
2 *  6 =   12
2 *  7 =   14
2 *  8 =   16
2 *  9 =   18
2 * 10 =   20
```

`while` loop just has one expression which is its terminating condition. We have written `i<=10` which is terminating condition for our loop. The moment i will become greater than that the loop will terminated. We are initializing our loop index to 0 and incrementing within while loop. Note that you must use curly braces for body of block of loop. If you have only one statement as body of loop then braces are optional.

5.4 do-while Loop

It is very much similar to `while` loop but with a very subtle difference. Consider the following code:

```
//Author: Shiv S. Dayal
//Description : Demo of do while statement.

#include <stdio.h>

int main()
{
  int i = 0;

  do {
    printf("I am Shiv.\n");
    i++;
  }while(i<5);

  return 0;
}
```

and the output is:

```
I am Shiv.
I am Shiv.
I am Shiv.
I am Shiv.
I am Shiv.
```

Notice the semicolon at the end of while. Now time for that subtle difference:

```
//Author: Shiv S. Dayal
//Description : Demo of do while statement.

#include <stdio.h>

int main()
{
  int i = 10;

  do {
    printf("2 * 10 = 20\n");
    i++;
  }while(i<5);

  return 0;
}
```

and the output is:

```
2 * 10 = 20
```

Notice how `do while` loop executes once even if the loop index is more than the terminating condition in the while part.

5.5 for Loop

`for` loop is the last of loops and most versatile. It has three parts: initialization of loop counters, terminating condition, and loop index modification. If you declare a variable in the initialization part then that variable has just loop scope while for `while` and `do while` loop indices have at least outer block scope. This makes `for` loop better. Consider the following example for computing squares of numbers:

```
//Author: Shiv S. Dayal
//Description : Demo of for statement.

#include <stdio.h>

int main()
{
  for(int i=1, j=1; (i<=10)||(j<=10); i++, j++)
    printf("%2d * %2d = %4d\n", i, j, i*j);

  return 0;
}
```

and the output is:

```
1 *  1 =    1
2 *  2 =    4
3 *  3 =    9
4 *  4 =   16
5 *  5 =   25
6 *  6 =   36
7 *  7 =   49
8 *  8 =   64
9 *  9 =   81
10 * 10 =  100
```

Notice how various things are coming in picture here: initialization, terminating conditions loop counter incrementation and output formatting. Here is how you can write an infinite for loop `for(;;)`. You can write an infinite loop anywhere if your loop index counters are not getting incremented/decremented properly or your termination condition is incorrect. Also, always make sure that loop indices are initialized. As an exercise you can try to implement this program using `while` and `do while` loop. Last line of the above output is not having first space properly.

5.6 break and continue Statements

`break` statement breaks out of innermost `for, do, while` and `switch` statements. It terminates that loop. Consider for example:

```
//Author: Shiv S. Dayal
//Description : Demo of break statement.

#include <stdio.h>

int main()
{

  for(int i = 0;;i +=10)
  {
    if(i>100)
      break;
    printf("%d\n", i);
  }

  return 0;
}
```

and the output is:

```
0
10
20
30
40
50
60
70
80
90
100
```

Notice how the `for` loop is terminated once `i` goes beyond 100 even though there is no terminating condition. Try the same in `while` and `do-while` loop and produce the same result.

`continue` statement is slightly different than `break` in the sense that it does not stop the execution of that loop but simply does not execute remaining instructions of that block. Consider for example:

```
//Author: Shiv S. Dayal
//Description : Demo of continue statement.

#include <stdio.h>
```

```
int main()
{

  for(int i = 0;i<=100;i +=10)
  {
    if(i==50)
      continue;
    printf("%d\n", i);
  }

  return 0;
}
```

and the output is:

```
0
10
20
30
40
60
70
80
90
100
```

Notice how 50 is missing from output.

5.7 typedef and return Statements

`typedef` statement is used to define new types from existing types. For example:

```
typedef char s8;
typedef unsigned char s8;
typedef short int s16;
typedef unsigned short int u16;
```

You will be seeing its usage in function pointers, structures and unions heavily.
`return` statement is used to return from function. Optionally you can return a value.

Chapter 6

Arrays and Pointers

In this chapter I am going to tell you about two very powerful constructs of C programming; arrays and pointers. Arrays are what they are; array of some data type. There can be an array of any complete type. You cannot create an array of any incomplete type, therefore, an array of type void is not allowed. There are fixed arrays and also variable length arrays. C99 inroduced variable length arrays before that arrays were only of fixed length. However, you can increase the capacity of a fixed sized array using `realloc()` function. There is single-dimensional array and then there is multi-dimensional array. We will first go through single-dimensional array then multi-dimensional.

6.1 Single-Dimensional Array

Let us first create a basic array and then see how to access it elements:

```
//Author: Shiv S. Dayal
//Description : Demo of array.

#include <stdio.h>

int main()
{
  const int MAX = 8;
  //An initialized array
  int a[8] = {0};
  //An initialized array to 0
  int b[MAX];

  for(int i=0; i<8; i++)
  {
    b[i] = i;
    printf("b[%d]=%d\n", i, b[i]);
  }
```

```
  for(int i=0; i<8; i++)
  {
    printf("a[%d]=%d\n", i, a[i]);
  }

  return 0;
}
```

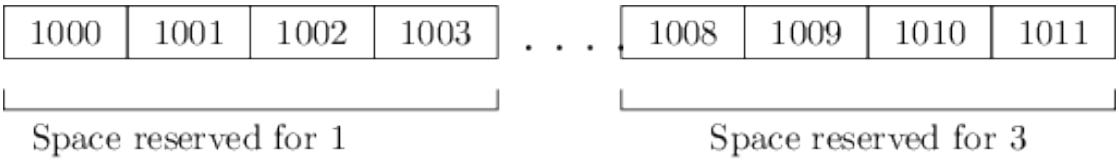
and the output is:

```
b[0]=0
b[1]=1
b[2]=2
b[3]=3
b[4]=4
b[5]=5
b[6]=6
b[7]=7
a[0]=0
a[1]=0
a[2]=0
a[3]=0
a[4]=0
a[5]=0
a[6]=0
a[7]=0
```

Here you see array subscripting operator in action. I have not covered this particular operator in fourth chapter so it becomes my duty to explain it here. There are two parts here one outside subscript and another outside. The expression which is outside will have type "const pointer to object type". This means that array's base address is fixed and cannot be changes. The expression which is inside will have integer type. The result of these two has type "type". We will see pointer arithmetic with binary + operator in the pointers section which is equivalent to subscript expression.

As you can see array a is fixed length array while array b is a variable length array. You are not allowed to initialize variable length arrays at the time of declaration. Notice that array indices do not start from 1 but 0. Never ever make the mistake of thinking that array indices start from 1. You can also initialize an array as `a[]={1, 2, 3};or a[3]={1, 2, 3};`. The array elements would be `a[0], a[1]` and `a[2]` in both the cases. Notice how assignment is done to elements of second array inside for loop one by one using the bracket operator or subscripting operator. The array elements are always in sequence in memory. A conceptual diagram is given below for first three elements of above array. Here 1 means first element.

Figure 6.1 An Array's Memory Diagram



Array elements are always(not always but most commonly. It is so common that I have used always.) accessed using their indices so order of retrieval is O(1).(This is known as big-O notation. You can find it in any Data Structure and Algorithm book. The above program will not compile using old compilers which do not support C99 standard like Turbo C++. Also, you may require to pass the flag `-std=c99` or `-std=c11` to some versions of gcc. For variable length arrays it is not necessary to declare the size in advance. Even, input to program from other sources will do.

```
//Author: Shiv S. Dayal
//Description : Demo of array.

#include <stdio.h>

int main()
{
  int i=0;

  printf("Enter the value of i:\n");
  scanf("%d\n", &i);

  char c[i];

  printf("Enter a string which contains one less no. of chars than i:\n");
  gets(c);
  puts(c);

  return 0;
}
```

and the output is:

```
Enter the value of i:
6
shiv
Enter a string which contains one less no. of chars than i:
shiv
shiv
```

As you can see variable length array should be declared after the size is known otherwise you may see strange output even though it is not compilation error. For example you could have declaraed

array immediately after `i` but you will get some garbage output. The reason for this is that at that point of time i contains garbage value. Also, note that array indices are integers. Floating-point numbers or variables cannot be indices.

Let us say you are writing a big piece of code and array is declared somewhere and you want to know how many elemnets you can fill in the array or what is the maximum size of array then you can use the following program:

```
//Author: Shiv S. Dayal
//Description : Demo of array.

#include <stdio.h>

int main()
{
  float f[10]={0.0};

  printf("Size of array f is %d.\n", sizeof(f)/sizeof(float));

  return 0;
}
```

and the output is:

```
Size of array f is 10.
```

Now an experienced programmer may ask that if we can know the size of array then why we do not have something like out of bounds exception of Java in C. My answer to that is C was written in 1970 and Java in 1990. For example, there are certain compilers with flags which help you detect this at runtime.

Feel free to experiment with arrays. Do whatever you like. Remember the more you will experiment the more you will learn. There are various ways in which you can define character arrays. For example, `char c[6]={'h', 'e', 'l', 'l', 'o', '\0'};`. Remember, you must terminate a character array with a null terminator. Another way to define the same is: `char str[6] = "hello";`. In this example you do not need to add '`\0`' explicitly as it is added automatically. Also, 6 is optional here if you want you can ommit that. Of course second example is more preferable. Note that if you declare an array of size m and data type size of array is n bytes then the array will consume m*n bytes no matter what; even when you are not using those bytes. Note that all these arrays are on stack memory area. We will see how to allocate array on heap memory area once we have studied pointers.

6.2 Multi-Dimensional Array

Arrays can be n-dmensional. There is no limit on dimensions. You can allocate as much as your memory allows. We will begin with two-dimensional array. A two-dimensional array looks like a matrix. Say a two-dimensional array has m as one dimension and n as second diemnsion. Then total

no. of elements will be m*n and size occupied is m*m*size of data type of array. There are various ways to initialize a two-dimensional array. Consider the following example:

```
//Author: Shiv S. Dayal
//Description : Demo of two-dimensional array.

#include <stdio.h>

int main()
{
  int a[2][2] = {{1,2},{3,4}};
  int b[2][2] = {1,2,3,4};

  //iterating over array
  for(int i=0;i<2;i++)
  {
    for(int j=0;j<2;j++)
      printf("%d ", a[i][j]);
    printf("\n");
  }
  for(int i=0;i<2;i++)
  {
    for(int j=0;j<2;j++)
      printf("%d ", b[i][j]);
    printf("\n");
  }

  return 0;
}
```

and the output is:

```
1 2
3 4
1 2
3 4
```

Same way you can have multi-dimensional array. I leave it up to you to find applications of different arrays. For now, try multiplying two matrices, doing a transpose, inverse of a matrix and printing a yearly calenday for any year for example. With the current information given to you, you should be able to do all these easily. As shown for array a it is not really a single array but an array of array. How we can read this is array a has two arrays each of which have two integers.

6.3 Pointers

A pointer can store an address. A pointer of some type can store address of that type and a pointer to void can store address of any type.

These are very interesting; considered to be one of the most powerful in the hands of capable programmer and most dangerous tool in the hands of an ignorant programmer. There are four standard library functions associated with them. All these are declared in **stdlib.h** which is part of standard c library. The functions are: `malloc(), calloc(), realloc()` and `free()`. Following is the contents of man pages verbatim,later in the program you can go to opengroup links as well. First signatures:

```
void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

here `size_t` is the unsigned integer type of the result of the sizeof operator. It is defined in `stddef.h`. And now descriptions:

`calloc()` allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory. The memory is set to zero. If `nmemb` or size is 0, then `calloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

`malloc()` allocates `size` bytes and returns a pointer to the allocated memory. The memory is not cleared. If size is 0, then `malloc()` returns either `NULL`, or a unique pointer value that can later be successfully passed to `free()`.

`free()` frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc(), calloc()` or `realloc()`. Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is `NULL`, no operation is performed.

`realloc()` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If `ptr` is `NULL`, then the call is equivalent to `malloc(size)`, for all values of `size`; if `size` is equal to zero, and `ptr` is not `NULL`, then the call is equivalent to `free(ptr)`. Unless `ptr` is `NULL`, it must have been returned by an earlier call to `malloc(), calloc()` or `realloc()`. If the area pointed to was moved, a `free(ptr)` is done.

Let us consider a program:

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *p = NULL;

  p = (int*)malloc(sizeof(int)*8);

  for(int i=0;i<8;i++)
  {
    *(p+i)=i;
```

```
    printf("Content at %dth location is %d.\n", i, *(p+i));
  }

  return 0;
}
```

and the output is:

```
Content at 0th location is 0.
Content at 1th location is 1.
Content at 2th location is 2.
Content at 3th location is 3.
Content at 4th location is 4.
Content at 5th location is 5.
Content at 6th location is 6.
Content at 7th location is 7.
```

There are various ways to declare a simple pointer and initialize it. For example:

```
char *c;                          //Only declaration no initialization
c = NULL;                         //Initialization
void *p = NULL;                   //declaration and initialization
void *q = malloc(sizeof(void)*10); //Declare and allocate memory for 10
```

On line number 15 and 16 you are seeing pointer arithmetic in previous program. Consider array a declared in the first example. We could have iterated in that example like `*(a+i)`.

A postfix expression followed by an expression in square brackets `[ ]` is a subscripted designation of an element of an array object. The definition of the subscript operator `[ ]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. Because of the conversion rules that apply to the binary + operator, if `E1` is an array object (equivalently, a pointer to the initial element of an array object) and `E2` is an integer, `E1[E2]` designates the `E2`-th element of of `E1` (counting from zero).

I had not covered some portion of additive operators in the chapter of operators and expression deliberately as I wanted to discuss them here. When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression `P` points to the `i`-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value `n`) point to, respectively, the `i+n`-th and `i-n`-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation will not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it will not be used as the operand of a unary `*` operator that is evaluated.

When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` defined in the <stddef.h> header. If the result is not representable in an object of that type, the behavior is undefined. In other words, if the expressions `P` and `Q` point to, respectively, the `i`-th and `j`-th elements of an array object, the expression `(P)-(Q)` has the value `i-j` provided the value fits in an object of type `ptrdiff_t`. Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object.

You can also apply increment and decrement operators on pointers. I will show you a reimplementation of previos program using increment operators:

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *p = NULL;

  p = (int*)malloc(sizeof(int)*8);
  int *q = p;

  for(int i=0;i<8;i++)
  {
    *(p+i)=i;
    printf("Content at %dth location is %d.\n", i, *(q++));
  }

  return 0;
}
```

and the output is:

```
Content at 0th location is 0.
Content at 1th location is 1.
Content at 2th location is 2.
Content at 3th location is 3.
Content at 4th location is 4.
Content at 5th location is 5.
Content at 6th location is 6.
Content at 7th location is 7.
```

6.4 Address and Indirection Operators

As is the case with subscript operator and pointer arithmetic in the fourth chapter that I have delayed these two as well for I wanted to put them here. Whenever you declare a plain variable you have an address associated with it and that variable is an lvalue. Just to repeat an lvalue is a value whose address can be taken. To take the address of an lvalue you use the address operator which is &. Now a pointer points to address of any value as we know so we can use address operator to get the address and use a pointer to store. There are several usage of storing an address. Most notable of those is pass-by-address which we will see in next chapter which will deal with functions. Let us say we take address of a variable and assign that to a pointer. Then if we change the value of the memory pointed to by the pointer then the variable whose address has been taken will get updated with this new value. Consider for example:

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int  i = 8;
  int *p = &i;

  *p = 7;

  printf("i=%d *p=%d\n", i, *p);

  return 0;
}
```

and the output is:

```
i=7 *p=7
```

So you see the power of pointers that if you have an address you can modify its contents. This is exacly what `scanf()` does. The dereference operator or indirection operator or aterisk (`*`) gives you value at address pointed to by pointer o. However, if you want to change address of some varible like that ofi by doing something like &i=&someOthervar; you cannot do that because address is not an lvalue. However, you can pass address of a pointer variable to some other function and use it using pointer to pointer notation which I will show you in next chapter. As I have shown pointers are kind of equivalent to array except the fact that they are on heap and sizeof operator will not work on them. Consider this example:

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.
```

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int  a[4] = {1,2,3,4};
  int* p    = a;
  int* q    = (int*)calloc(10, 4);

  for(int i=0; i<4; i++)
    printf("i=%d *p=%d\n", i, *(p+i));

  printf("Size of a=%d\n", sizeof(a));
  printf("Size of p=%d\n", sizeof(p));
  printf("Size of q=%d\n", sizeof(q));

  return 0;
}
```

and the output is:

```
i=0 *p=1
i=1 *p=2
i=2 *p=3
i=3 *p=4
Size of a=16
Size of p=4
Size of q=4
```

Here p acts as pointer to array. You can have a pointer to any kind of array. You can point to any element of array because array elements are lvalues whose addresses can be taken and to initialize a pointer alll you need is an address.

Advice

Complex pointer arithmetic is best avoided. Be very thoughtful that if you really really need it. Use loops to iterate arrays. Multiple levels of indirection is also bad. Typically I have not seen more than pointers to pointers. Now we will see array of pointers.

6.5 Arrays of Pointers

Pointers are just like ordinary variables so we can as well create array of pointers. Consider following for example:

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.

#include <stdio.h>
#include <stdlib.h>

int main()
{
  char* strArray[2]={"Hello", "Universe!"};

  for(int i=0; i<2; i++)
    printf("%s\n", strArray[i]);

  return 0;
}
```

and the output is:

```
Hello
Universe!
```

Note how the length of two array elements are different as they are pointers. Let us do a more complex example.

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int* intArray[2];

  intArray[0] = (int*)calloc(3, sizeof(int));
  intArray[1] = (int*)calloc(2, sizeof(int));

  *intArray[0]     = 4;
  *(intArray[0]+1) = 5;
  *(intArray[0]+2) = 6;


  *intArray[1]     = 1;
  *(intArray[1]+1) = 2;

  for(int i=0; i<3; i++)
  {
```

```
    printf("Memory location=%p Content=%d\n", intArray[0]+i, *(intArray ↩
        [0]+i));
  }

  for(int i=0; i<2; i++)
  {
    printf("Memory location=%p Content=%d\n", intArray[1]+i, *(intArray ↩
        [1]+i));
  }

  return 0;
}
```

and the output is:

```
Memory location=0x87d1008 Content=4
Memory location=0x87d100c Content=5
Memory location=0x87d1010 Content=6
Memory location=0x87d1018 Content=1
Memory location=0x87d101c Content=2
```

Note missing four bytes between 6 and 1. Memory locations may be different on your system. But see how messy pointer syntax can go even with such simple code. Array to pointers are useful for containing variables of dynamic size of same type.

Pointers to pointers are same as array of pointers. The only difference is that you can dynamically modify the number of elements.

6.6 Pointers of Pointers

Consider the following example:

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.

#include <stdio.h>
#include <stdlib.h>

int main()
{
  int** intPtr;

  intPtr = (int**)malloc(sizeof(sizeof(int*)*2));

  *intPtr = (int*)malloc(sizeof(int)*3);
  *(intPtr+1) = (int*)malloc(sizeof(int)*4);

  **intPtr     = 1;
```

```
  *(*intPtr+1) = 2;
  *(*intPtr+2) = 7;


  **(intPtr+1)      = 3;
  *(*(intPtr+1)+1)  = 5;
  *(*(intPtr+1)+2)  = 9;
  *(*(intPtr+1)+3)  = 11;

  for(int i=0; i<3; i++)
    printf("Memory location=%p content=%d\n", *intPtr+i, *(*intPtr+i));

  for(int i=0; i<4; i++)
    printf("Memory location=%p content=%d\n", *(intPtr+1)+i, *(*(intPtr+1) ↩
        +i));

  return 0;
}
```

and the output is:

```
Memory location=0x9947018 content=1
Memory location=0x994701c content=2
Memory location=0x9947020 content=7
Memory location=0x9947028 content=3
Memory location=0x994702c content=5
Memory location=0x9947030 content=9
Memory location=0x9947034 content=11
```

Again memory location may change on your system. As you can see how things can get messy with pointers. Believe me you will hate this. Also, I do not see any reason to use more than two levels of indirection. So you get the idea. If you need dynamic no. of elements with dynamic content you are going to use pointers to pointers.

6.7 realloc() Function

Once `malloc()` and `calloc()` allocate some memory you have that certain amount of memory available to you. When you have an array you have some memory but what if you want more later. `realllloc()` comes to rescue you. Here is a sample program:

```
//Author: Shiv S. Dayal
//Description : Demo of pointer.

#include <stdio.h>
#include <stdlib.h>

int main()
```

```
{
  int *p = (int*)malloc(sizeof(int)*2);

  *p     = 5;
  *(p+1) = 7;

  printf("Original 1st element=%d\n", *p);
  printf("Original 2nd element=%d\n", *(p+1));

  p = (int*)realloc(p, sizeof(int)*4);

  *(p+2) = 9;
  *(p+3) = 11;

  printf("New 1st element=%d\n", *p);
  printf("New 2nd element=%d\n", *(p+1));
  printf("New 3rd element=%d\n", *(p+2));
  printf("New 4th element=%d\n", *(p+3));

  return 0;
}
```

and the output is:

```
Original 1st element=5
Original 2nd element=7
New 1st element=5
New 2nd element=7
New 3rd element=9
New 4th element=11
```

6.8 free() Function

Whatever program we have written in this chapter related to dynamic memory allocation using `malloc()` etc are very bad code just because we are not releasing memory properly. Any call to memory allocation functions have to be matched with a corresponding `free()` call. The reason for this is that when all pointers to a memory area are lost and that memory is not freed then operating system cannot recycle that memory. In case of servers or long running processes this may eat up all the physical RAM and virtual memory and eventually freeze the system. To guard against such events you must macth all allocation calls with deallocation calls so that operating system can reclaim the freed memmory.

You must heed this warning given here with all of your focus. You got to handle heap that is dynamically allocated memory yourself. You allocate and you free it. If you miss you have a memory leak.

Warning

You must free all memory you allocate.

6.9 Constness

To make anything constant you need to associate const keyword with it. For example, `const int i;const float f;`. However, with pointers in picture scenarios change compared to two simple previous examples. When pointers are made constant there are two elements. First is the pointer itself and second is the value pointed to. Consider for example:

```
const int* i;  //constant pointer data is not
int* const i;  //constant data pointer is not
const int* const i; //both are const
```

The way to read it is you draw a vertical line where asterisk(*) is there and the value associated with const is constant. Whenever you need use a constant freely. Try to use constants more and more. Also, prefre them to following:

```
#define MAX 10
```

As told and shown to you it will replace `MAX` with 10 in the file everywhere without any concern of type-safety. Also, it does not enter in the symbol table so while debugging you will not see `MAX` anywhere. So instead you should use something like:

```
const int MAX=10;
```

I will also like to say something about volatile variables. Beginners are usually convinced that volatile variables cannot be declraed as const. Let me iterate the definitions once again. A const variable cannot be modifed by the program itself. A volatile variable can be modified by sources other than the program itself. Hence, a const volatile variable cannot be modified by the program but other sources can still modify it.

Chapter 7

Functions

I know that you will readily agree with me if I say that humans get bored if they have to do same things again and again. I know you get bored too and I too get bored. We all. We as humans have this built-in nature that repititive things are just not fit for us. Also, as a human being our capacity to understand large things at once is difficult. We understand small-small things and build large chunk based on those small things. Dennis Ritchie perhaps had known this. I am saying because C has got something called functions. C functions allow you to split a big logic into small ones and therefore facilitating modular programming. They also form the basis of strutctured programming the very base which made C popular. There is also something called recursion which is a very poewrful tool. In this chapter we will also see how to do multifile programming. I cannot emphasize much that how important it is that you master the technique of functions well and not to mention function pointers which can do the magic. I will show you the very glimpse only. I can show you the way but walking on that is your job. It is upto you to do the actual work. I have kept things simple and minimal with a pupose. I do not want you to get bogged down with a thick and heavy book. All my examples are toy examples but you have seen things can get somehwat complex.

We have already seen the special `main()` function.

7.1 Pass by Value

Here I am going to present skeleton of a function prototype and body. Consider:

```
//function prototype
return-type function-name(argument list); //here varible names may be  ↩
    ommitted

//function body
return-type function-name(argument list) //variable names cannot be  ↩
    ommitted''
{
  //your code here
```

```
  //call some other function
  function-name(arugment-list-without-type);

  return value-of-return-type;
}
```

This might be a bit abstract but please bear it a bit. In due course of time it will become clear. You will be able to see in its concrete forms soon. Consider a program which adds two numbers and let us say that you may need to add lots of them.

```
//Author: Shiv S. Dayal
//Description: Demo of function

#include <stdio.h>

void add(int firstInt, int secondInt)
{
  printf("%d+%d=%d\n", firstInt, secondInt, firstInt+secondInt);
}

int main()
{
  int a=5, b=7;

  add(a, b);

  return 0;
}
```

and the output is:

```
5+7=12
```

Note that you need function body before its use else you need at least a function prototype before use. If you do not do so you will get a compiler warnign. An example is given below:

```
//Author: Shiv S. Dayal
//Description: Demo of function

#include <stdio.h>

//not how argument names are not required
void add(int, int);

int main()
{
  int a=5, b=7;
```

```
  add(a, b);

  return 0;
}

void add(int firstInt, int secondInt)
{
  printf("%d+%d=%d\n", firstInt, secondInt, firstInt+secondInt);
}
```

output is same as above.

What you have seen just above is known as pass-by-value. In this case a copy of parameters is made and passed on to called function by caller function. So, if called function makes a change to values then those are not reflected back in the caller function. As an example I will use famous example of swapping values of two variables. First, I will show how pass-by-value works. So here is the code:

```
//Author: Shiv S. Dayal
//Description: Demo of function

#include <stdio.h>

void swap(int, int);

int main()
{
  int a=5, b=7;

  printf("Before swap a=%d and b=%d\n", a, b);
  swap(a, b);
  printf("After swap a=%d and b=%d\n", a, b);

  return 0;
}

void swap(int firstArg, int secondArg)
{
  int temp=firstArg;
  firstArg=secondArg;
  secondArg=temp;
}
```

and the output is:

```
Before swap a=5 and b=7
After swap a=5 and b=7
```

7.2 Pass by Address

Not exactly what we wanted. The solution is to pass-by-address. When you the address to a called function, it receives address in a pointer variable. Then if it modifies the value stored at that address then it is reflected back in the caller. Let us see an example to understand:

```
//Author: Shiv S. Dayal
//Description: Demo of function

#include <stdio.h>

void swap(int*, int*);

int main()
{
  int a=5, b=7;

  printf("Before swap a=%d and b=%d\n", a, b);
  swap(&a, &b);
  printf("After swap a=%d and b=%d\n", a, b);

  return 0;
}

void swap(int* firstArg, int* secondArg)
{
  int temp=*firstArg;
  *firstArg=*secondArg;
  *secondArg=temp;
}
```

and the output is:

```
Before swap a=5 and b=7
After swap a=7 and b=5
```

7.3 Recursion

In C recusion is the concept of a function calling itself. When a repeated operation has to be preformed over a variable, recursion can be used. Recursion simplifies the code a lot. Typically there is always a more effective iterative solutions are available but there are certain cases where recursion is always better than iteration. For example, traversal of trees where iteration is not so effective as compared to recursion. The first example I am going to give is that of factorials. The formula for factorial is given by $n! = \prod_{k=1}^{n} k$ and recursive definition of factorial is given by:

$$n! = \begin{cases} 1 & \text{if n=0} \\ (n-1)! * n & \text{if n>0} \end{cases}$$

Note that every recursion has to be written carefully in thse sense that it must have a termination condition and that in all the cases the termination condition must be reached. If a recursion is too deep or infinite there will be a stack overlow and the program will terminate. First, I will show you an iterative version with a function.

```
//Author: Shiv S. Dayal
//Description: Iterative factorial.

#include <stdio.h>

long long fact(int input);

int main()
{
  int input=0;

  printf("Enter a number whose input has to be computed:\n");
  scanf("%d", &input);

  printf("Factorial of %d is %lld.\n", input, fact(input));

  return 0;
}

long long fact(int input)
{
  long long output=1;
  while(input!=0)
  {
    output*=input;
    input--;
  }

  return output;
}
```

and the output is:

```
Enter a number whose factorial has to be computed:
17
Factorial of 17 is 355687428096000.
```

Now we will see recursive version:

```
//Author: Shiv S. Dayal
```

```
//Description: Recursive factorial.

#include <stdio.h>

long long fact(int input);

int main()
{
  int input=0;

  printf("Enter a number whose factorial has to be computed:\n");
  scanf("%d", &input);

  printf("Factorial of %d is %lld.\n", input, fact(input));

  return 0;
}

long long fact(int input)
{
  if(input==0)
    return 1;
  else
    return fact(input-1)*input;
}
```

and the output is:

```
Enter a number whose factorial has to be computed:
16
Factorial of 16 is 20922789888000.
```

Recursion is very simple yet may be very deceptive to understand for beginners. Let us dissect the code. Our input was 16 so if will not match and `return fact(15)*16;` will be executed. Here, before `fact(16)` can return `fact(15)` has to return. And, similarly before `fact(15)` can return `fact(14)` has to return. Now, note that for `fact(0)` there is no such condition and it can return 1 making it possible for fact(1) to return, which, in turn will make it posible for fact(2) to return and so on. So, what is happening is function is calling itself by creating more and more function frames and when the termination condition reaches the stack unwinds.

Let us consider one more famous example for recursive function, that is of computing Fibonacci numbers. The Fibonacci series is given by:

$$F_n = F_{n-1} + F_{n-2}$$

where first two numebrs are given by:

$$F_0 = 0 \text{ and } F_1 = 1$$

First consider the iterative version:

```
//Author: Shiv S. Dayal
//Description: Iterative Fibonacci series.

#include <stdio.h>

void fibonacci(int input);

int main()
{
  int input=0;

  printf("How many Fibonacci numbers you want?\n");
  scanf("%d", &input);

  fibonacci(input);

  return 0;
}

void fibonacci(input)
{
  int fib0=0, fib1=1;

  if(input==0)
  return;
  else if(input==1)
  {
    printf("%d\n", fib0);
  }
  else if(input==2)
  {
    printf("%d %d\n", fib0, fib1);
  }
  else if(input>2)
  {
    printf("%d %d", fib0, fib1);
    while(input>1)
    {
      fib1=fib1+fib0;
      fib0=fib1-fib0;
      printf(" %d", fib1);
      input--;
    }
  }
  printf("\n");
}
```

and the output is:

```
How many Fibonacci numbers you want?
16
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Now we will see recursive version:

```
//Author: Shiv S. Dayal
//Description: iRecursive Fibonacci series.

#include <stdio.h>

long long fibonacci(int input);

int main()
{
  int input=0;

  printf(”Which Fibonacci number you want?\n”);
  scanf(”%d”, &input);

  printf(”%lld\n”, fibonacci(input));

  return 0;
}

long long fibonacci(int input)
{
  long long fib0=0, fib1=1;

  if(input==0)
  {
    return fib0;
  }
  else if(input==1)
  {
    return fib1;
  }
  else
  {
    long long fib = fibonacci(input-1)+fibonacci(input-2);
    return fib;
  }
}
```

and the output is:

```
Which Fibonacci number you want?
```

```
32
2178309
```

7.4 Function like Macros

Functions are costly if they are very small. For example, let us say we want to add two integers only then it does not make sense to write a function. When you call a function a new function frame has to be created, new variables are created, when function returns things are cleaned and return value is returned. All this consume memory and CPU cycles so old C style was to use macros. For example, consider following program:

```
//Author: Shiv S. Dayal
//Description: Demo of macros.

#include <stdio.h>

#define SUM(a, b) a+b

int main()
{
  printf("%d %d\n", SUM(5, 7), SUM(8, 9));

  return 0;
}
```

and the output is:

```
12 17
```

However, such usage of macros are inappropriate, dangerous and higly adivised against. First you have to take care that you parenthesize all parameters carefully. Even then consider following:

```
#define MIN(a,b)((a)<(b))?(a):(b);
```

If it gets a call like:

```
int a=7, b=3;
MIN(a,++b)//then macro will expand to
((a)<(++b))?(a);(++b);
```

Now since `b` is less than `a` it will be incremented twice otherwise it will be incremented once. Such behavior is confusing at best. Older C programmers had no choice but only macros. But with new C99 standard we have something called inline functions. New C99 programmers have no excuse for writing macros like shown above.

7.5 Inline Functions

`inline` functions are somewhat a mix of macros and functions. It is a request to compiler to expand the code inline like macros while maintaining the type-safety of functions. Note that it is a request not a command. Compilers may choose to ignore the request of inline expansion of code if the inline function is too complicated. Also, recursive functions are not inlined. You should use inline functions to replace small functions only. The reasons are being that you may get problems mentioned in Item 33 of “Effective C++” by Scott Meyers. For smaller functions you have a much higher chance of getting your functions inlined. To use the inline function you just need to prefix the function signature and prototype declaration with keyword inline. For smaller functions code generated for inline functions will outweigh the overhead which is there for function calls. However, if you inline too much the size of your binary will become bigger and bigger and it may be a problem on systems; straved for memory; in systems like embedded systems. Typically inline functoins are declared in headers so that all source files can benefit from it. However, this may cause problems if functions are not inlined by compiler.

7.6 Function Pointers

These are very powerful but have got somewhat complex syntax. Due to their complex syntax programmers typically shun them. However, they are must if you want to do certain stuff which C typically does not allow, like, object oriented programming, generic programming, switch/if statement replacement etc. to name a few. New programmers may wonder how can we have pointers to functions as they are not variables. Well they are not varibles that I agree but still their addresses can be taken. However, their addresses lie in code segment or text segment which happens to be read-only area, hence, that address cannot be modified. Let us consider a program of a desk calculator with four operations. Addition, subtraction, multiplication and division. As a typical desk calculator I will take double as data type as it has sufficient range and precision. How would you write such a program? Well with our current knowledge we can write four functions for four operations. Then we can use a switch for choosing the function. Let us see it in action:

```
//Author: Shiv S. Dayal
//Description: Demo of function pointers.

#include <stdio.h>

int main()
{
  char op=0;
  double op1=0.0,op2=0.0,result=0.0;

  printf(”Enter operation (should be one of + - * /):”);
  scanf(”%c”, &op);

  printf(”Enter two operands separated by a space:”);
  scanf(”%lf %lf”, &op1, &op2);
```

```
  switch(op)
  {
    case '+':
      result = op1 + op2;
      break;
    case '-':
      result = op1 - op2;
      break;
    case '*':
      result = op1 * op2;
      break;
    case '/':
      result = op1 / op2;
      break;
    default:
      break;
  }

  printf("%lf%c%lf=%lf\n", op1, op, op2, result);

  return 0;
}
```

and the output is:

```
Enter operation (should be one of + - * /):+
Enter two operands separated by a space: 2.4 1.2
2.400000+1.200000=3.600000
```

As you can see depending on the operation the switch statement performs the operation on two operands. We can use function pointers to replace this swiccth statement:

```
//Author: Shiv S. Dayal
//Description: Demo of function pointers.

#include <stdio.h>
/* Since there are four arithmetic operations we need four function ↩
    pointers.*/

float plus(double op1, double op2)
{
  double result=0.0;

  result=op1+op2;
  printf("%lf+%lf=%lf\n", op1, op2, result);
}

float minus(double op1, double op2)
```

```
{
  double result=0.0;

  result=op1-op2;
  printf("%lf-%lf=%lf\n", op1, op2, result);
}

float multiply(double op1, double op2)
{
  double result=0.0;

  result=op1*op2;
  printf("%lf*%lf=%lf\n", op1, op2, result);
}

float divide(double op1, double op2)
{
  double result=0.0;

  result=op1/op2;
  printf("%lf/%lf=%lf\n", op1, op2, result);
}

void call_fp(double op1, double op2, float (*pt2Func)(double, double))
{
  pt2Func(op1, op2);
}

// Execute example code
void Switch(double op1, double op2, char op)
{
  switch(op)
  {
    case '+':
      call_fp(op1, op2, &plus);
      break;
    case '-':
      call_fp(op1, op2, &minus);
      break;
    case '*':
      call_fp(op1, op2, &multiply);
      break;
    case '/':
      call_fp(op1, op2, &divide);
      break;
    default:
      break;
```

```
  }
}

int main()
{
  char op = 0;
  double op1 = 0.0, op2 = 0.0, result=0.0;

  printf("Enter operation (should be one of + - * /):");
  scanf("%c", &op);
  printf("Enter two operands separated by a space:");
  scanf("%lf %lf", &op1, &op2);

  Switch(op1, op2, op);

  return 0;
}
```

and the output is:

```
Enter operation (should be one of + - * /):+
Enter two operands separated by a space:2.4
1.2
2.400000+1.200000=3.600000
```

So you see how a switch statement can be replaced with function pointers. The abstract declaration of a function pointer is given below:

```
return_type (*function_name)(arguments);
```

You can call these functions in two ways:

```
function_name(arguments); //shortcut call
(*function_name)(arguments); //long and correct call
```

You should always prefer the second version as it is more portable across different compilers and environments.

7.7 Passing and Receiving Function Pointers

You have already seen how to pass a function pointer as an argument to a second function in the above exercise. `call_fp(op1,op2, &plus);` is where you pass a function pointer and `void call_fp(doubleop1, double op2, float (*pt2Func)(double, double))` is where you receive it as an argument.

You can also return a function pointer from some function. Consider the following

```
return_type (*func1(arguments1))(arguments2)
{
```

```
  return &func2;
}
```

This piece of code is a function whose name is `func1`, it takes arguments1 as its arguments and returns `float`. The return value is a function pointer whose arguments are . However, this kind of declaration is messy and hard to read so we have a solution which makes things easier on us. Consider a following `typedef` and function signature:

```
typedef return_type (*function1)(arguments);
function1 function2(arguments);
```

This is much simpler and cleaner. It is also easier to understand than above.

Similarly you can declare an array of function pointers. This offers the feature of selection of a function using an index. For example, the menu bar of most of the GUI programns can be accessed using this. Similarly, there are two ways again to declare the array of function pointers. The first one is without using `typedef` and second one is using `typedef`. The choice is yours that which one you want to use. I prefer the typedef version. The syntax is given below:

```
typedef return_type (*function1)(arguments);
function1 array_of_fp[MAX];

return_type (*function2[MAX])(arguments);
```

One of the more clever usage of function pointers can be found in the library function `qsort` where you have to write the comparison function which is a callback function. Given below os the signature of `qsort` function.

```
void qsort (void * a, size_t count, size_t size, int (*comp) (
            const void *, const void * ) );
```

The brief description is given here. Sorts the count elements of the array pointed by a, each element size bytes long, using the compa function to determine the order.

The sorting algorithm used by this function compares pairs of values by calling the specified comparator function with two pointers to elements of the array.

The function does not return any value, but modifies the content of the array pointed by base reordering its elements to the newly sorted order. Let us see an example:

```
//Author: Shiv S. Dayal
//Description: Demo of qsort.

#include <stdio.h>
#include <stdlib.h>

int values[] = { 4, 1, 4, 3, 7, 10, 9, 20, 25 };

int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
```

```
}

int main ()
{
  int n=0;

  qsort (values, 6, sizeof(int), compare);

  for (n=0; n<9; n++)
    printf ("%d ",values[n]);

  return 0;
}
```

and the output is:

```
1 3 4 4 7 10 9 20 25
```

7.8 Type Generic Functions

C11 has introduced a new type of functions called type generic functions. If you read clause 2 of <tgmath.h> §(iso6.25). then you will find following.

Of the <math.h> and <complex.h> functions without an f (float) or l (long double) suffix, several have one or more parameters whose corresponding real type is double. For each such function, except modf, there is a corresponding type-generic macro.[1] The parameters whose corresponding real type is double in the function synopsis are generic parameters. Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.[2]

Since we have three different floating-point types and three different versions of complex numbers (float, double and long double) therefore we have six versions of each function which will be called based on argument type.

This is achieved through keyword _Generic which is used to create a generic selection expression. For example, consider the following code:

```
#include <stdio.h>

int main()
{
  printf("%d", _Generic((6), char: 1, int: 2, long: 3, default: 0));

  return 0;
```

[1] Like other function-like macros in Standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

[2] If the type of the argument is not compatible with the type of the parameter for the selected function, the behavior is undefined.

```
}
```

Note that you need version 4.9 of `gcc` or `clang` 3.4 to compile this. The output is:

```
2
```

as you can see we are passing 6 which is an integer and value 2 is associated with the `int` therefore the output is 2.

We can make a macro which will allow us to reuse the functionality. For example:

```
#include <stdio.h>

#define type(T) _Generic( (T), char: 1, int: 2, long: 3, default: 0)

int main()
{
  printf("%d", type(87364563853));

  return 0;
}
```

Section §(iso.6.5.1.1) of specification describes generic selection in great detail. To summarize following points can be noted:

1. A generic selection consists of a controlling expression (which is not evaluated) and one or more, comma-separated, generic associations.

Chapter 8

Structures and Unions

So far what we have seen are data types defiend by the language itself. However, there are times when these simple types are not enough. For this C has defined two types which can be defined by the user or programmer. The two keywords are struct and union. A structure or union is basically a composite type. These may consist of one more types of C. That is they may contain one or more basic types like int, char etc or other structures or unions. Consider following examples:

```
struct {
  int i;
  char c;
  int* p;
  char* s;
} mystruct;

struct {
  int x;
  struct mystruct S;
  } another_struct;

union {
  int i;
  char c;
  int* p;
  char8 s;
} myunion;

union {
  int x;
  union myunion S;
} another_union;
```

As you see from the declarations there is no difference between structures and unions. However, there is a subtle difference. Consider following program:

```
// Author: Shiv Shankar Dayal
// Description: Difference between strutures and unions.

#include <stdio.h>

typedef struct {
  int i;
  double d;
} mystruct;

typedef union {
  int i;
  double d;
} myunion;

int main()
{
  mystruct s;
  myunion u;

  printf("Size of structure is %d\n", sizeof(s));
  printf("Size of union is %d\n", sizeof(u));

  return 0;
}
```

and the output is:

```
Size of structure is 12
Size of union is 8
```

Before explaining the output let me tell you this that using typedef is mandatory here to get the `sizeof` operator working. If you try something like `sizeof(struct mystruct);` then you will get this as error. `invalid application of sizeof to incomplete type struct mystruct.` However, you can use an object of `struct mystruct`.

By using `typedef` we let the compiler recognize them as complete types. Now let us see the output. Size of structure is simple. It is equal to size of an integer plus size of a double. However, size of union is equal to size of double. Basically, size of a structure is equal to size of all its elements. For unions size of union is equal to size of biggest element. This means that elements for a union overlap on the same memory area. We can use this fact to write a very clever program. Before writing the program let me give you some background.

There is something called endianness of a machine. What it means that how bytes are stored. If a machine is little endian like most intel processors then the bytes are stoerd in reverse order. What this means that they are not in there natural order. On big endian machines like PPC architecture the bytes are in natural order. In other words if the least significant byte of an integer is stored at the lowest memory address then it is called little-endian. If the least significant byte is stored at the

highest address then it is called big-endian. For example my machine is intel so let us see what output we get from this program.

```
//Author: Shiv S. Dayal
//Description: Demo of endianness,

#include <stdio.h>

typedef union {
  short int i;
  char c;
} myunion;

int main()
{
  myunion u;

  u.i = 258;

  printf("%d\n", u.c);

  return 0;
}
```

and the output is:

```
2
```

So as you can see the extra 2 i.e. 258 - 256 is getting stored in c. Now 2 is the high order byte. Hence we can conclude that my machine is little-endian.

More ways to initialize a structure are given below:

```
//Author: Shiv S. Dayal
//Description: Structure initialization

#include <stdio.h>

typedef struct {
  short int i;
  char c;
} mystruct;

int main()
{
  mystruct s1 = {34, 'c'}, s2;

  s2.i = 43;
  s2.c = 'e';
```

```
  printf("%d %d %c %c\n", s1.i, s2.i, s1.c, s2.c);

  return 0;
}
```

and the output is:

```
34 43 c e
```

8.1 Pointer members of a Structure

Sometime structures will contain pointer members. Obviously, you will have to allocate memory to them or point them to some existing variable's address. Let us see how this is done.

```
//Author:Shiv S. Dayal
//Descrition: Pointer members of a structure

#include <stdio.h>
#include <stdlib.h>

int main()
{
  typedef struct {
    int* i;
    int j;
  }s;

  s* s1;

  s1 = (s*)malloc(sizeof(s));
  s1->i=(int*)4;
  s1->j=5;

  printf("s1->i=%p s1->j=%d\n", s1->i, s1->j);

  return 0;
}
```

and the output is:

```
s1->i=0x4 s1->j=5
```

You might be wondering why I have casted (`int*`) to 4 and used a `%p` format specifier. The reason is `*i` is a pointer to an interger and hence `s1->i` is a pointer and will accept only an integer pointer. Therefore, casting is mandatory else you will get a warning. Again, at the time of assignment I have assigned value 4 which is an address actully. Therefore `%p` is needed for conversion. However,

this program is a bad, wrong program to death. The reason is the address 4 may be out of program's segmenst and touching it in the sense of trying to read from it or write to it may doom your day. Try to burn your hands. Let us see the correct version.

```
//Author:Shiv S. Dayal
//Descrition: Pointer members of a structure

#include <stdio.h>
#include <stdlib.h>

int main()
{
  typedef struct {
    int* i;
    int j;
  }s;

  s* s1;

  s1 = (s*)malloc(sizeof(s));
  s1->i = (int*)malloc(sizeof(int));
  *(s1->i) = 4;
  s1->j = 5;

  printf("s1->i = %d s1->j = %d\n", *(s1->i), s1->j);

  return 0;
}
```

and the output is:

```
s1->i=4 s1->j=5
```

8.2 Usage of Structures and Unions

A structure can be used to represent rather complex entities. For example, a car. Consider a car. It has weight, power, cost, mileage etc. All this can be combined and represented as a structure. Structures can be categorized in two categories. One will be normal structures and second is seld-referential structures. Self-referential structures contain a pointer to a structure of its own type. You will see its usage in the book when we deal with data structures. We have already shown you normal structures. The data members of a structure are referenced using . operator or -> if they are pointer type as we have already seen. The rule of using structures or unions is simple. When you cannot represent any entity using provided data types then combine the basic entities and use them in an structure or union. Union has one distiction which you already know. Consider you know that your entity can have multiple type of values but only one at a time. Then you can use unions.

Structures and unions can be nested as well. I will just give a simple example and leave rest to your imagination and previously told facts.

```
#include <stdio.h>

typedef struct {
  int j;
}t;

typedef struct{
  int i;
  t t1;
}s;

int main()
{
  s s1;

  s1.i=4;
  s1.t1.j=5;

  printf(”%d %d”, s1.i, s1.t1.j);

return 0;
}
```

and the output is:

```
4 5
```

8.3 Structures and Arrays

There are two posiibilities here. Structures containing arrays and arrays of structure. Let us see an example which combines both:

```
//Author: Shiv S. Dayal
//Description: Arrays as structure elements

#include <stdio.h>

typedef struct {
  int i[2];
}ais;

int main()
{
  ais a[2];
```

```
  a[0].i[0] = 1;
  a[0].i[1] = 2;
  a[1].i[0] = 3;
  a[1].i[1] = 4;

  printf("%d %d %d %d\n", a[0].i[0], a[0].i[1], a[1].i[0], a[1].i[1]);

  return 0;
}
```

and the output is:

```
1 2 3 4
```

Chapter 9

Preprocessing Directives

The following comes from section 6.10 of specification. It will terminate when you see code starting. :-)

A *preprocessing directive* consists of a sequence of preprocessing tokens that begins with a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character, and is ended by the next new-line character.[1] A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

A text line shall not begin with a # preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

When in a group that is skipped (12.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

Constraints

The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

Semantics

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called preprocessing, because conceptually they occur before translation of the resulting translation unit.

The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

EXAMPLE In:

```
#define EMPTY
```

[1] Thus, preprocessing directives are commonly called “lines”. These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 12.3.2, for example).

```
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is not a preprocessing directive, because it does not begin with a # at the start of translation phase 4, even though it will do so after the macro `EMPTY` has been replaced.

9.1 Conditional Inclusion

Constraints

The expression that controls conditional inclusion shall be an integer constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below; [2] and it may contain unary operator expressions of the form:

```
defined identifier
```

or:

```
defined (identifier)
```

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject identifier), 0 if it is not.

Sematics

Preprocessing directives of the forms:

```
# if constant-expression new-line group_opt
# elif constant-expression new-line group_opt
```

check whether the controlling constant expression evaluates to nonzero.

Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the defined unary operator), just as in normal text. If the token defined is generated as a result of this replacement process or use of the defined unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the defined unary operator have been performed, all remaining identifiers are replaced with the pp-number 0, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of constant expressions. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they hav e the same representation as, respectively, the types `intmax_t` and `uintmax_t` defined in the header `<stdint.h>`.[3] This includes interpreting character constants, which may involve converting escape sequences into execution character set

[2] Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names - there simply are no keywords, enumeration constants, etc.

[3] Thus, on an implementation where `INT_MAX` is and `UINT_MAX` is `0xFFFF`, the constant `0x8000` is signed and positive within a `#if` expression even though it would be unsigned in translation phase 7.

members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.[4] Also, whether a single-character character constant may have a negative value is implementation-defined.

Preprocessing directives of the forms:

```
# ifdef identifier new-line group_opt
# ifndef identifier new-line group_opt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if` defined identifier and `#if !defined` identifier respectively.

Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a #else directive, the group controlled by the #else is processed; lacking a #else directive, all the groups until the #endif are skipped.[5]

9.2 Source File Inclusion

Constraints

A `#include` directive shall identify a header or source file that can be processed by the implementation.

Semantics

A preprocessing directive of the form:

```
# include <h-char-sequence>  new-line
```

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

A preprocessing directive of the form:

```
# include "q-char-sequence" new-line
```

[4] Thus, the constant expression in the following `#if` directive and if statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25

if ('z' - 'a' == 25)
```

[5] As indicated by the syntax, a preprocessing token shall not follow a #else or #endif directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the ” delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read:

```
# include <h-char-sequence> new-line
```

with the identical contained sequence (including > characters, if any) from the original directive.

A preprocessing directive of the form:

```
# include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after include in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms.[6] The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of ” characters is combined into a single header name preprocessing token is implementation-defined.

The implementation shall provide unique mappings for sequences consisting of one or more letters or digits followed by a period (.) and a single letter. The first character shall be a letter. The implementation may ignore the distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

Forward References: macro replacement (12.3).

9.3 Macro Replacement

Constraints

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

An identifier currently defined as an object-like macro shall not be redefined by another #define preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another `#define` preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like

[6] Note that adjacent string literals are not concatenated into a single string literal; thus, an expansion that results in two string literals is an invalid directive.

macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the ...). There shall exist a) preprocessing token that terminates the invocation.

The identifier __VA_ARGS__ shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

A parameter identifier in a function-like macro shall be uniquely declared within its scope.

Semantics

The identifier immediately following the define is called the macro name. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

A preprocessing directive of the form:

```
# define identifier replacement-list new-line
```

defines an object-like macro that causes each subsequent instance of the macro name[7] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.

A preprocessing directive of the form:

```
# define identifier lparen identifier-listopt ) replacement-list new-line
# define identifier lparen ... ) replacement-list new-line
# define identifier lparen identifier-list , ... ) replacement-list new- ↩
    line
```

defines a function-like macro with arguments, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the #define preprocessing directive. Each subsequent instance of the function-like macro name followed by a (as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching) preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,[8] the behavior is undefined.

If there is a ... in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the variable arguments. The number of arguments combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the ...).

[7] Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences, they are never scanned for macro names or parameters.

[8] Despite the name, a non-directive is a preprocessing directive.

9.3.1 Argument Substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.

An identifier `__VA_ARGS__` that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

9.3.2 The # Operator

Constraints

Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

Semantics

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters), except that it is implementation-defined whether a \ character is inserted before the \ character beginning a universal character name. If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is "". The order of evaluation of # and ## operators is unspecified.

9.3.3 The ## Operator

Constraints

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

Semantics

If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.[9]

[9] Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

9.3.4 Rescanning and Further Replacement

After all parameters in the replacement list have been substituted and # and ## processing has taken place, all placemarker preprocessing tokens are removed. Then, the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 12.9 below.

9.3.5 Scope of Macro Definitions

A macro definition lasts (independent of block structure) until a corresponding `#undef` directive is encountered or (if none is encountered) until the end of the preprocessing translation unit. Macro definitions have no significance after translation phase 4.

A preprocessing directive of the form:

```
# undef identifier new-line
```

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

9.4 Line Control

Constraints

The string literal of a `#line` directive, if present, shall be a character string literal.

Semantics

The line number of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 while processing the source file to the current token.

A preprocessing directive of the form:

```
# line digit-sequence new-line
```

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 2147483647.

A preprocessing directive of the form:

```
# line digit-sequence "s-char-sequenceopt" new-line
```

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

A preprocessing directive of the form:

```
# line pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after line on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

9.5 Error Directive

Semantics

A preprocessing directive of the form `#error pp-tokensopt new-line` causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

9.6 Pragma Directive

Semantics

A preprocessing directive of the form:

```
# pragma pp-tokensopt new-line
```

where the preprocessing token STDC does not immediately follow pragma in the directive (prior to any macro replacement)[10] causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to

[10] An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where `STDC` immediately follows pragma). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

behave in a non-conforming manner. Any such pragma that is not recognized by the implementation is ignored.

If the preprocessing token STDC does immediately follow pragma in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms whose meanings are described elsewhere:

```
#pragma STDC FP_CONTRACT on-off-switch
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

on-off-switch: on of
ON OFF DEFAULT

Forward references: the FP_CONTRACT pragma (13.12.2), the FENV_ACCESS pragma (13.6.1), the CX_LIMITED_RANGE pragma (13.3.4). 149).

9.7 Null Directive

Semantics

A preprocessing directive of the form:

```
# new-line
```

has no effect.

9.8 Predefined Macro Names

The following macro names shall be defined by the implementation:

__DATE__ The date of translation of the preprocessing translation unit: a character string literal of the form ”Mmm dd yyyy”, where the names of the months are the same as those generated by the asctime function, and the first character of dd is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

__FILE__ The presumed name of the current source file (a character string literal)[11]

__LINE__ The presumed line number (within the current source file) of the current source line (an integer constant).[12]

__STDC__ The integer constant 1, intended to indicate a conforming implementation. __STDC _HOSTED__ The integer constant 1 if the implementation is a hosted implementation or the integer constant 0 if it is not.

__STDC_VERSION__ The integer constant 199901L.[12]

__TIME__ The time of translation of the preprocessing translation unit: a character string literal of the form ”hh:mm:ss” as in the time generated by the asctime function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

[11] The presumed source file name and line number can be changed by the `#line` directive.

[12] This macro was not specified in ISO/IEC 9899:1990 and was specified as 199409L in ISO/IEC 9899/AMD1:1995. The intention is that this will remain an integer constant of type long int that is increased with each revision of International Standard.

The following macro names are conditionally defined by the implementation: __STDC_IEC_559__ The integer constant 1, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).

__STDC_IEC_559_COMPLEX__ The integer constant 1, intended to indicate adherence to the specifications in informative annex G (IEC 60559 compatible complex arithmetic).

__STDC_ISO_10646__ An integer constant of the form yyyymmL (for example, 199712L). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type wchar_t, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month.

The values of the predefined macros (except for __FILE__ and __LINE__) remain constant throughout the translation unit.

None of these macro names, nor the identifier defined, shall be the subject of a #define or a #undef preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

The implementation shall not predefine the macro __cplusplus, nor shall it define it in any standard header.

Forward references: the asctime function (13.23.3.1), standard headers (13.1.2).

9.9 Pragma Operator

Semantics

A unary operator expression of the form:

```
_Pragma ( string-literal )
```

is processed as follows: The string literal is destringized by deleting the L prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence \" by a double-quote, and replacing each escape sequence \\ by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the pp-tokens in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

At this point specification material ends here and now we will see usage of above discussed macros.

9.10 Usage

Note that for this part the compilation command should be clang -E filename.c. Let us create two files test.c and test1.c and their contents are given below respectively.

9.10.1 #include

```
#include "test1.c"
I am test.
```

```
#include "test.c"
I am test1.
```

Keep both the files in same directory and execute `clang -E test.c` you will see following:

```
# 1 "test.c"
# 1 "test.c" 1
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 143 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2
# 1 "./test1.c" 1
...
In file included from test.c:1:
In file included from ./test1.c:1:
In file included from test.c:1:
In file included from ./test1.c:1:
In file included from test.c:1:
In file included from ./test1.c:1:
...
In file included from test.c:1:
./test1.c:1:10: error: #include nested too deeply
#include "test.c"

I am test1.
# 2 "test.c" 2
I am test
# 2 "./test1.c" 2
I am test1.
# 2 "test.c" 2
I am test
# 2 "./test1.c" 2
I am test1.
# 2 "test.c" 2
```

As you can see `test.c` includes `test1.c` and `test1.c` includes `test.c`. So they are including each other which is causing nested includes. After processesing for some time preprocessor's head starts spinning as if it has drunk a full bottle of rum and it bails out. As you know headers are included in all meaningful C programs and headers include each other as well. This inclusion of each other can easily lead to nested inclusion so how do header authors circumvent this problem. Well, a technique has been devised known popularly as header guard. The lines which have the form # number text is actually `#line` directive.

Consider following code:

```
#ifndef ANYTHING
#define ANYTHING

#include "test1.c"

I am test.

#endif
```

```
#ifndef ANYTHING_ELSE
#define ANYTHING_ELSE

#include "test.c"

I am test1.

#endif
```

Now what will happen that when `test.c` is included `ANYTHING` is defined and when `test1.c` is included via it `ANYTHING_ELSE` will be defined. After first round of inclusion no more inclusion can happen as governed by the directives. Please see headers of standard library to see the conventions for `ANYTHING`.

9.10.2 Why we need headers?

Now that we have seen the `#include` directive I would like to tell that why we even need header files. Header files contain several elements of libraries which come with C. For example, function prototypes, structure/type, declarations, macros, global variable declaration etc. Actual code resides inside `*.a` or `*.so` library files on GNU/Linux or UNIX OSes. Now let us consider a case that we want to access a C function of standard library. The compilation phase requires that prototype of function should be known at compilation time. If we do not have headers we have no way to provide this function prototype at compile time. Same stands true for global variables. The declaration of these must be known at compilation time. You take any language there has to be a mechanism to include code from other files. Be it use directive of Perl or import of Python or any other mechanism of any other language.

9.10.3 #define

`#define` and `#include` are probably the most encountered macro in all C files. There are many usage of it. We will first see the text replacement and function like usage which can be avoided and should be replaced by global constants and inline functions. First let us see what text replacement functionality we get using `#define`. Consider the following code fragment:

```
#define MAX 5

MAX

I am MAX
```

Now run it though `clang -E filename.c` and you will get following output:

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "test.c"


5

I am 5
```

So as you see both the occurrences are replaced by the text 5. This is the simplest form of text replacement which people use to handle many things. Most common are array sizes and symbolic constants. Another form is the form like functions which has been shown in 10.4.

The bad part of these two is that both do not enter symbol table and make code hard to debug. The former can be replaced by const variables and latter by inline functions.

The other usage of it is to define names. For example, we revisit our old example headers. Header guards usually declare something like this:

```
#ifndef SOMETHING
#define SOMETHING

/* header code */
#endif
```

As you can see `#define` is used to define `SOMETHING` so second time the conditional inclusion `#ifndef` will fail. It can also be tested by defiend like `if(defined(SOMETHING)`. Now if `SOMETHING` has been defined if test will pass successfully. Similarly `#ifdef` can be used to test it as a shortcut i.e. `#ifdef SOMETHING`. The normal if-else statements are replaced in preprocessing directives using `#if, #elif` and `#endif`.

9.10.4 #undef

Anything defined by `#define` can be undefined by `#undef`. For example consider the following code:

```
#define test
#ifdef test

//do something
```

```
#undef test
#ifdef test

//do something else
#endif
#endif
```

If you do this then first something else will be executed while the second will not be.

9.10.5 # and

You can use following two examples and description given above to understand both of these:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); //char p[]="x ## y"

#define FIRST a # b
#define SECOND a ## b

char first[] = FIRST;
char second[] = SECOND;
```

9.10.6 #error

```
#include <stdio.h>

int main()
{
  # error MAX

  return 0;
}
```

If you try to compile this like `clang filename.c` then you will get following:

```
clang test.c
test.c:5:5: error: #error MAX
  # error MAX
  ^
1 error generated.
```

You can combine `#error` with `#if` but I have yet to see purposeful code written that way. Non-preprocessing constructs are better for handling such situations. Only if you want to test a preprocessing token then it should be used.

9.10.7 #pragma

`#pragma` is dependent on what follows it. You should consult compiler documentation as it is mostly implementation-defined. It is a way to tell compiler to do things which are not governed by specification.

9.10.8 Miscellaneous

Usage of `__LINE__`, `__FILE__`, `__DATE__` and `__TIME__` is simple and shown in following example:

```
#include <stdio.h>

int main()
{
  printf("%s:%d:%s:%s", __FILE__, __LINE__, __DATE__, __TIME__);

  return 0;
}
```

and the output is:

```
test.c:5:Jun 24 2012:11:24:57
```

This concluded our discussion on macros. Rest of the book will describe the standard library.

Chapter 10

The C Standard Library

This chapter and rest of book eflects chapter 7 of n1124.pdf. This chapter begins with assert.h. We will see three parts of each function/macro as much as possible. From specification, compiler and example point of view in that order.

10.1 Introduction

10.1.1 Definition of Terms

A *string* is a contiguous sequence of characters terminated by and including the first null character. The term *multibyte string* is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a *wide string*. A *pointer to a string* is a pointer to its initial (lowest addressed) character. The length of a string is the number of bytes preceding the null character and the *value of a string* is the sequence of the values of the contained characters, in order.

The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.[1] It is represented in the text and examples by a period, but may be changed by the `setlocale` function.

A *null wide character* is a wide character with code value zero.

A *wide string* is a contiguous sequence of wide characters terminated by and including the first null wide character. A *pointer to a wide string* is a pointer to its initial (lowest addressed) wide character. The *length of a wide string* is the number of wide characters preceding the null wide character and the *value of a wide string* is the sequence of code values of the contained wide characters, in order.

A *shift sequence* is a contiguous sequence of bytes within a multibyte string that (potentially) causes a change in shift state. A shift sequence shall not have a corresponding wide character; it is

[1] The functions that make use of the decimal-point character are the numeric conversion functions (14.20.1, 14.24.4.1) and the formatted input/output functions (14.19.6, 14.24.2).

instead taken to be an adjunct to an adjacent multibyte character.[2]

10.1.2 Standard Headers

Each library function is declared, with a type that includes a prototype, in a header,[3] whose contents are made available by the `#include` preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use. Declarations of types described in this clause shall not include type qualifiers, unless explicitly stated otherwise.

The standard headers are:

```
&lt;assert.h>  &lt;inttypes.h> &lt;signal.h>  &lt;stdlib.h>
&lt;complex.h> &lt;iso646.h>   &lt;stdarg.h>  &lt;string.h>
&lt;ctype.h>   &lt;limits.h>   &lt;stdbool.h> &lt;tgmath.h>
&lt;errno.h>   &lt;locale.h>   &lt;stddef.h>  &lt;time.h>
&lt;fenv.h>    &lt;math.h>     &lt;stdint.h>  &lt;wchar.h>
&lt;float.h>   &lt;setjmp.h>   &lt;stdio.h>   &lt;wctype.h>
```

If a file with the same name as one of the above < and > delimited sequences, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files, the behavior is undefined.

Standard headers may be included in any order; each may be included more thanonce in a giv en scope, with no effect different from being included only once, except that the effect of including `<assert.h>` depends on the definition of `NDEBUG` (see 14.2). If used, a header shall be included outside of any external declaration or definition, and it shall first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion.

Any definition of an object-like macro described in this clause shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.

Any declaration of a library function shall have external linkage.

A summary of the contents of the standard headers is given in annex B.

Forward references: diagnostics (14.2).

10.1.3 Reserved Identifiers

Each header declares or defines all identifiers listed in its associated subclause, and optionally declares or defines identifiers listed in its associated future library directions subclause and identifiers which are always reserved either for any use or for use as file scope identifiers.

[2] For state-dependent encodings, the values for `MB_CUR_MAX` and `MB_LEN_MAX` shall thus be large enough to count all the bytes in any complete multibyte character plus at least one adjacent shift sequence of maximum length. Whether these counts provide for more than one shift sequence is the implementation's choice.

[3] A header is not necessarily a source file, nor are the < and > delimited sequences in header names necessarily valid source file names.

- All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.
- All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.
- Each macro name in any of the following subclauses (including the future library directions) is reserved for use as specified if any of its associated headers is included; unless explicitly stated otherwise (see 14.1.4).
- All identifiers with external linkage in any of the following subclauses (including the future library directions) are always reserved for use as identifiers with external linkage.[4]
- Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

No other identifiers are reserved. If the program declares or defines an identifier in a context in which it is reserved (other than as allowed by 14.1.4), or defines a reserved identifier as a macro name, the behavior is undefined.

If the program removes (with `#undef`) any macro definition of an identifier in the first group listed above, the behavior is undefined.

10.1.4 Use of Library Functions

Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow: If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer, or a pointer to non-modifiable storage when the corresponding parameter is not const-qualified) or a type (after promotion) not expected by a function with variable number of arguments, the behavior is undefined. If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid. Any function declared in a header may be additionally implemented as a function-like macro defined in the header, so if a library function is declared explicitly when its header is included, one of the techniques shown below can be used to ensure the declaration is not affected by such a macro. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[5] The use of `#undef` to remove any macro definition will also ensure that an actual function is referred to. Any invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary,

[4] The list of reserved identifiers with external linkage includes `errno`, `ath_errhandling`, `setjmp`, and `va_end`.

[5] This means that an implementation shall provide an actual function for each library function, even if it also provides a macro for that function.

so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following subclauses may be invoked in an expression anywhere a function with a compatible return type could be called.[6] All object-like macros listed as expanding to integer constant expressions shall additionally be suitable for use in `#if` preprocessing directives.

Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function and use it without including its associated header.

There is a sequence point immediately before a library function returns.

The functions in the standard library are not guaranteed to be reentrant and may modify objects with static storage duration.[7]

[6] Such macros might not contain the sequence points that the corresponding function calls do.

[7] Thus, a signal handler cannot, in general, call standard library functions.

Chapter 11

Diagnostics <assert.h>

The header <assert.h> defines the assert macro and refers to another macro, NDEBUG which is not defined by code <assert.h>. If NDEBUG is defined as a macro name at the point in the source file where <assert.h> is included, the assert macro is defined simply as:

```
#define assert(ignore) ((void)0)
```

The assert macro is redefined according to the current state of NDEBUG each time that <assert.h> is included.

The Macroassert shall be implemented as a macro, not as an actual function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

11.1 Program Diagnostics

11.1.1 The assert Macro

Synopsis

```
#include <assert.h>
void assert(scalar expression);
```

Description

The assert macro puts diagnostic tests into programs; it expands to a void expression. When it is executed, if expression (which shall have a scalar type) is false (that is, compares equal to 0), the eassert macro writes information about the particular call that failed (including the text of the argument, the name of the source file, the source line number, and the name of the enclosing function - the latter are respectively the values of the preprocessing macros e__FILE__ and and of the identifier __func__) on the standard error stream in an implementation-defined format. It then calls the abort function.

Returns

The assert macro returns no value.

Forward references: the `abort` function (20.4.1).

Synopsis

```
#include &lt;assert.h>
void assert(scalar expression);
```

Description

If the macro `NDEBUG` was defined at the moment <assert.h> was last included, the macro `assert()` generates no code, and hence does nothing at all. Otherwise, the macro `assert()` prints an error message to standard error and terminates the program by calling `abort` if expression is false (i.e., compares equal to zero).

The purpose of this macro is to help the programmer find bugs in his program. The message "assertion failed in file foo.c, function do_bar(), line 1287" is of no help at all to a user.

Return Value

No value is returned.

In C89, expression is required to be of type int and undefiend behavior results if it is not, but in C99 it may have scalar type.

`assert()` is implemented as a macro; if the expression tested hasside-effects, program behavior will be different depending on whether `NDEBUG` is defined. This may create Heisenbugs which go away whendebugging is turned on.

Example

```
#include <stdio.h>
#include <assert.h>

int main()
{
  assert(0);
  printf("Not gonna happen!");

  return 0;
}
```

and the output is:

```
a.out: test.c:5: int main(): Assertion '0' failed.
Aborted (core dumped)
```

You should replace 0 inside assert() by the expression to be tested.

Chapter 12

Complex arithmetic <complex.h>

12.1 Introduction

The header `<complex.h>` defines macros and declares functions that support complex arithmetic. Each synopsis specifies a family of functions consisting of a principal function with one or more `double complex` parameters and a `double complex` or `double` return value; and other functions with the same name but with f and l suffixes which are corresponding functions with `float` and `long double` parameters and return values.

The macro complex expands to `_Complex`; the macro `_Complex_I` expands to a constant expression of type `const float _Complex`, with the value of the imaginary unit.[1].

The macro `I` expands to `_Complex_I`.

Notwithstanding the provisions of reserved identifiers, a program may undefine and perhaps then redefine the macros `complex` and `I`.

Forward references: IEC 60559-compatible complex arithmetic (annex G).

12.2 Conventions

Values are interpreted as radians, not degrees. An implementation may set errno but is not required to.

12.3 Branch Cuts

Some of the functions below have branch cuts, across which the function is discontinuous. For implementations with a signed zero (including all IEC 60559 implementations) that follow the specifications of annex G, the sign of zero distinguishes one side of a cut from another so the function is continuous (except for format limitations) as the cut is approached from either side. For example, for the square

[1] The imaginary unit is a number i such that $i^2 = -1$

root function, which has a branch cut along the negative real axis, the top of the cut, with imaginary part +0, maps to the positive imaginary axis, and the bottom of the cut, with imaginary part -0, maps to the negative imaginary axis.

Implementations that do not support a signed zero (see annex F) cannot distinguish the sides of branch cuts. These implementations shall map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction. (Branch cuts for the functions specified here have just one finite endpoint.) For example, for the square root function, coming counter clockwise around the finite endpoint of the cut along the negative real axis approaches the cut from above, so the cut maps to the positive imaginary axis.

12.4 The CX_LIMITED_RANGE Pragma

Synopsis

```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

Description

The usual mathematical formulas for complex multiply, divide, and absolute value are problematic because of their treatment of infinities and because of undue overflow and underflow. The `CX_LIMITED_RANGE` pragma can be used to inform the implementation that (where the state is "on") the usual mathematical formulas are acceptable.[2] The pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `CX_LIMITED_RANGE` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state for the pragma is "off".

12.5 Trigonometric functions

12.5.1 The cscos functions

Synopsis

```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

[2] The purpose of the pragma is to allow the implementation to use the formulas.

Descriptiotn

The `cacos` functions compute the complex arc cosine of `z`, with branch cuts outside the interval `[-1, +1]` along the real axis.

Returns

The `cacos` functions return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

Synopsis

```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

Link with `-lm`.

Description

The `cacos()` function calculates the complex arc cosine of `z`. If `y =cacos(z)`, then `z = ccos(y)`. The real part of `y` is chosen in the interval [0, π].

One has:

```
cacos(z) = -i * clog(z + i * csqrt(1 - z * z))
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cacos(z)]=%lf and Im[cacos(z)]=%lf\n", creal(cacos(z)), cimag ↩
      (cacos(z)));

  return 0;
}
```

Compile like `clang filename.c -lm`. Execution gives following output:

```
Re[cacos(z)]=0.936812 and Im[cacos(z)]=-2.305509
```

12.5.2 The casin functions

Synopsis

```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

Description

The `casin` functions compute the complex arc sine of `z`, with branch cuts outside the interval [-1, +1] along the real axis.

Returns

The functions return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

Synopsis

```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

Description

```
csin(z) = (exp(i * z) - exp(-i * z)) / (2 * i)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cacsin(z)]=%lf and Im[cacsin(z)]=%lf\n", creal(casin(z)), ↩
      cimag(casin(z)));

  return 0;
}
```

Compile like `clang filename.c -lm`. Execution gives following output:

```
Re[cacsin(z)]=0.633984 and Im[cacsin(z)]=2.305509
```

12.5.3 The cstan functions

Synopsis

```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

Description

The `catan` functions compute the complex arc tangent of `z`, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Returns

The `catan` functions return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

Synopsis

```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

Link with `-lm`.

The `catan()` function calculates the complex arc tangent of `z`. If `y =catan(z)`, then `z = ctan(y)`. The real part of `y` is chosen in the interval $[-\pi/2, \pi/2]$.

One has:

```
catan(z) = (clog(1 + i * z) - clog(1 - i * z)) / (2 * i)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cactan(z)]=%lf and Im[cactan(z)]=%lf\n", creal(catan(z)), ↩
      cimag(catan(z)));

  return 0;
}
```

Compile like . Execution`clang filename.c -lm` gives following output:

```
Re[cactan(z)]=1.448307 and Im[cactan(z)]=0.158997
```

12.5.4 The ccos functions

Synopsis

```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

Description

The `ccos` functions compute the complex cosine of `z`.

Returns

The ccos functions return the complex cosine value.

Synopsis

```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

Link with -lm.

Description

The complex cosine function is defined as:

```
ccos(z) = (exp(i * z) + exp(-i * z)) / 2
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[ccos(z)]=%lf and Im[ccos(z)]=%lf\n", creal(ccos(z)), cimag( ↩
      ccos(z)));

  return 0;
}
```

and the output is:

```
Re[ccos(z)]=-27.034946 and Im[ccos(z)]=-3.851153
```

12.5.5 The csin functions

Synopsis

```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

Description

The functions compute the complex sine of z.

Desciption

The csin functions return the complex sine value.

Synopsis

```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

Link with `-lm`.

Description

The complex sine function is defined as:

```
csin(z) = (exp(i * z) - exp(-i * z)) / (2 * i)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[csin(z)]=%lf and Im[csin(z)]=%lf\n", creal(csin(z)), cimag( ↩
      csin(z)));

  return 0;
}
```

and the output is:

```
Re[csin(z)]=3.853738 and Im[csin(z)]=-27.01681
```

12.5.6 The ctan functions

Synopsis

```
#include <complex.h>
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

Description

The `ctan` functions compute the complex tangent of `z`.

Returns

The `ctan` functions return the complex tangent value.

Synopsis

```
#include <complex.h>
double complex ctan(double complex z);
```

```
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

Link with `-lm`.

Description

The complex tangent function is defined as:

```
ctan(z) = csin(z) / ccos(z)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[ctann(z)]=%lf and Im[ctan(z)]=%lf\n", creal(ctan(z)), cimag( ↩
      ctan(z)));

  return 0;
}
```

and the output is:

```
Re[ctann(z)]=-0.000187 and Im[ctan(z)]=0.999356
```

12.6 Hyperbolic functions

12.6.1 The cscosh functions

Synopsis

```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

Description

The `cacosh` functions compute the complex arc hyperbolic cosine of `z`, with a branch cut at values less than 1 along the real axis.

Returns

The `cacosh` functions return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

Synopsis

```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

Description

The `cacosh()` function calculates the complex arc hyperpolic cosine of `z`. If `y =cacosh(z)`, then `z =ccosh(y)`. The imaginary part of `y` is chosen in the interval $[-\pi, \pi]$. The real part of `y` is chosen nonnegative.

One has:

```
cacosh(z) = 2 * clog(csqrt((z + 1) / 2) + csqrt((z - 1) /
2))
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cacosh(z)]=%lf and Im[cacosh(z)]=%lf\n", creal(cacosh(z)), ↩
      cimag(cacosh(z)));

  return 0;
}
```

and the output is:

```
Re[cacosh(z)]=2.305509 and Im[cacosh(z)]=0.93681
```

12.6.2 The casinh function

Synopsis

```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

Description

The `casinh` functions compute the complex arc hyperbolic sine of , with branch cuts outside the interval [-i, +i] along the imaginary axis.

Returns

The `casinh` functions return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

Synopsis

```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

Link with `-lm`.

Description

The `casinh()` function calculates the complex arc hyperbolic sine of `z`. If , then `z =csinh(y)`. The imaginary part of `y` is chosen in the interval $[-\pi/2, +\pi/2]$.

One has:

```
casinh(z) = clog(z + csqrt(z * z + 1))
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[casinh(z)]=%lf and Im[casinh(z)]=%lf\n", creal(casinh(z)), ↩
      cimag(casinh(z)));

  return 0;
}
```

and the output is:

```
Re[casinh(z)]=2.299914 and Im[casinh(z)]=0.917617
```

12.6.3 The catanh functions

Synopsis

```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Description

The `catanh` functions compute the complex arc hyperbolic tangent of `z`, with branch cuts outside the interval [-1, +1] along the real axis.

Rerurns

The `catanh` functions return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2$

Synopsis

```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Link with `-lm`.

Desacription

The `catanh()` function calculates the complex arc hyperbolic tangent of `z`. If `y =catanh(z)`, then `z =ctanh(y)`. The imaginary part of `y` is chosen in the interval $[-\pi/2, +\pi/2]$.

One has:

```
catanh(z) = 0.5 * (clog(1 + z) - clog(1 - z))
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[catanh(z)]=%lf and Im[catanh(z)]=%lf\n", creal(catanh(z)),  ↩
      cimag(catanh(z)));

  return 0;
}
```

and the output is:

```
Re[catanh(z)]=0.117501 and Im[catanh(z)]=1.409921
```

12.6.4 The ccosh functions

Synopsis

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

Description

The `ccosh` functions compute the complex hyperbolic cosine of `z`.

Returns
The `ccosh` functions return the complex hyperbolic cosine value.
Synopsis

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

Link with `-lm`.
Description
One has:

```
ccosh(z) = (exp(z)+exp(-z))/2
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[ccosh(z)]=%lf and Im[ccosh(z)]=%lf\n", creal(ccosh(z)), cimag ↩
      (ccosh(z)));

  return 0;
}
```

and the output is:

```
Re[ccosh(z)]=-6.580663 and Im[ccosh(z)]=-7.581553
```

12.6.5 The csinh functions

Synopsis

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

Description
The `csinh` functions compute the complex hyperbolic sine of `z`.
Returns
The `csinh` functions return the complex hyperbolic sine value.
Synopsis

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

Link with `-lm`.

Description

The complex hyperbolic sine function is defined as:

```
csinh(z) = (exp(z)-exp(-z))/2
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[csinh(z)]=%lf and Im[csinh(z)]=%lf\n", creal(csinh(z)), cimag ↩
      (csinh(z)));

  return 0;
}
```

and the output is:

```
Re[csinh(z)]=-6.548120 and Im[csinh(z)]=-7.619232
```

12.6.6 The ctanh functions

Synopsis

```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

Description

The `ctanh` functions compute the complex hyperbolic tangent of `z`.

Returns

The `ctanh` functions return the complex hyperbolic tangent value.

Synopsis

```
#include <complex.h>
double complex ctanh(double complex z);
```

```
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

Link with `-lm`.

Description

The complex hyperbolic tangent function is defined mathematically as:

```
ctanh(z) = csinh(z) / ccosh(z)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[ctanh(z)]=%lf and Im[ctanh(z)]=%lf\n", creal(ctanh(z)), cimag ↩
      (ctanh(z)));

  return 0;
}
```

and the output is:

```
Re[ctanh(z)]=1.000710 and Im[ctanh(z)]=0.004908
```

12.7 Exponential and logrithmic functions

12.7.1 The cexp functions

Synopsis

```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

Description

The `cexp` functions compute the complex base-e exponential of `z`.

Returns

The `cexp` functions return the complex base-e exponential value.

Synopsis

```
#include <complex.h>
double complex cexp(double complex z);
```

```
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

Link with `-lm`.

The function calculates e (2.71828..., the base of natural logarithms) raised to the power of `z`.

One has:

```
cexp(I * z) = ccos(z) + I * csin(z)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cexp(z)]=%lf and Im[cexp(z)]=%lf\n", creal(cexp(z)), cimag( ↩
      cexp(z)));

  return 0;
}
```

and the output is:

```
Re[cexp(z)]=-13.128783 and Im[cexp(z)]=-15.200784
```

12.7.2 The clog functions

Synopsis

```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

Description

The `clog` functions compute the complex natural (base-e) logarithm of `z`, with a branch cut along the negative real axis.

Returns

The clog functions return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval [$-i\pi, +i\pi$] along the imaginary axis.

Synopsis

```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

Link with `-lm`.

Description

The logarithm `code` is the inverse function of the exponential `cesp()`. Thus, if `y =clog(z)`, then `z =cexp(y)`. The imaginary part of `y` is chosen in the interval $[-\pi, +\pi]$.

One has:

```
clog(z) = log(cabs(z)) + I * carg(z)
```

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[clog(z)]=%lf and Im[clog(z)]=%lf\n", creal(clog(z)), cimag( ↩
      clog(z)));

  return 0;
}
```

and the output is:

```
Re[clog(z)]=1.609438 and Im[clog(z)]=0.927295
```

12.8 Power and absolute-value functions

12.8.1 The cabs functions

Synopsis

```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

Description

The `cabs` functions compute the complex absolute value (also called norm, modulus, or magnitude) of `z`.

The `cabs` functions return the complex absolute value.

Synopsis

```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

Link with `-lm`.

Description

The `cabs()` function returns the absolute value of the complex number `z`. The result is a real number.

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cabs(z)]=%lf and Im[cabs(z)]=%lf\n", creal(cabs(z)), cimag( ↩
      cabs(z)));

  return 0;
}
```

and the output is:

```
Re[cabs(z)]=5.000000 and Im[cabs(z)]=0.000000
```

12.8.2 The cpow functions

Synopsis

```
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
```

Description

The `cpow` functions compute the complex power function x^y, with a branch cut for the first parameter along the negative real axis.

Returns

The `cpow` functions return the complex power function value.

```
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
```

Link with `-lm`.

Description

The function calculates `x` raised to the power `z`. (With a branch cut for `x` along the negative real axis.)

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cpow(z, z)]=%lf and Im[cpow(z, z)]=%lf\n", creal(cpow(z, z)), ↩
       cimag(cpow(z, z)));

  return 0;
}
```

and the output is:

```
Re[cpow(z, z)]=-2.997991 and Im[cpow(z, z)]=0.623785
```

12.8.3 The csqrt functions

Synopsis

```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Description

The `csqrt` functions compute the complex square root of `z`, with a branch cut along the negative real axis.

Returns

The functions return the complex square root value, in the range of the right halfplane (including the imaginary axis).

Synopsis

```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Link with `-lm`.

Description

Calculate the square root of a given complex number, with nonnegative real part, and with a branch cut along the negative real axis. (That means that `csqrt(-1+eps*I)` will be close to `I` while `csqrt(-1-eps*I)` will be close to `-I`, if `eps` is a small positive real number.)

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[csqrt(z)]=%lf and Im[csqrt(z)]=%lf\n", creal(csqrt(z)), cimag ↩
      (csqrt(z)));

  return 0;
}
```

and the output is:

```
Re[csqrt(z)]=2.000000 and Im[csqrt(z)]=1.000000
```

12.9 Manipulation functions

12.9.1 The carg functions

Synopsis

```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

Description

The functions compute the argument (also called phase angle) of `z`, with a branch cut along the negative real axis.

Returns

The carg functions return the value of the argument in the interval $[-\pi, +\pi$

Synopsis

```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

Link with `-lm`.

Description

A complex number can be described by two real coordinates. One may use rectangular coordinates and gets

```
z = x + I * y
```

where `x =creal(z)` and `y =cimag(z)`.

Or one may use polar coordinates and gets:

```
z = r * cexp(I * a)
```

where `r =cabs(z)` is the "radius", the "modulus", the absolute value of `z`, and `a =carg(z)` is the "phase angle", the argument of `z`.

One has:

```
tan(carg(z)) = cimag(z) / creal(z)
```

Returns

The return value is the range of $[-\pi, +\pi]$.

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[carg(z)]=%lf and Im[carg(z)]=%lf\n", creal(carg(z)), cimag( ↩
      carg(z)));

  return 0;
}
```

and the output is:

```
Re[carg(z)]=0.927295 and Im[carg(z)]=0.000000
```

12.9.2 The cimag functions

Synopsis

```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

Description
The `cimag` functions compute the imaginary part of `z`.
Returns
The `cimag` functions return the imaginary part value (as a real).
Synopsis

```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

Link with `-lm`.
Description
The `cimag()` function returns the imaginary part of the complex number `z`.
One has:

```
z = creal(z) + I * cimag(z)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cimag(z)]=%lf and Im[cimag(z)]=%lf\n", creal(cimag(z)), cimag ↩
      (cimag(z)));

  return 0;
}
```

and the output is:

```
Re[cimag(z)]=4.000000 and Im[cimag(z)]=0.000000
```

12.9.3 The conj functions

Synopsis

```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

Description

The conj functions compute the complex conjugate of z, by reversing the sign of its imaginary part.

Returns

The conj functions return the complex conjugate value.

Synopsis

```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

Link with -lm.

Description

The conj() function returns the complex conjugate value of z. That is the value obtained by changing the sign of the imaginary part.

One has:

```
cabs(z) = csqrt(z * conj(z))
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[conj(z)]=%lf and Im[conj(z)]=%lf\n", creal(conj(z)), cimag( ↩
      conj(z)));

  return 0;
}
```

and the output is:

```
Re[conj(z)]=3.000000 and Im[conj(z)]=-4.000000
```

12.9.4 The cproj functions

Synopsis

```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

Description

The `cproj` functions compute a projection of `z` onto the Riemann sphere: `z` projects to `z` except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If `z` has an infinite part, then `cproj(z)` is equivalent to:

```
INFINITY + I * copysign(0.0, cimag(z))
```

Returns

The `cproj` functions return the value of the projection onto the Riemann sphere.

Synopsis

```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

Link with `-lm`.

Description

This function projects a point in the plane onto the surface of a Riemann Sphere, the one-point compactification of the complex plane. Each finite point `z` projects to `z` itself. Every complex infinite value is projected to a single infinite value, namely to positive infinity on the real axis.

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[cproj(z)]=%lf and Im[cproj(z)]=%lf\n", creal(cproj(z)), cimag ↩
         (cproj(z)));

  return 0;
}
```

and the output is:

```
Re[cproj(z)]=3.000000 and Im[cproj(z)]=4.000000
```

Since our complex number was finite, it projected itself.

12.9.5 The creal function

Synopsis

```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
```

```
long double creall(long double complex z);
```

Description
The `creal` functions compute the real part of `z`.
Returns
The `creal` functions return the real part value.
Returns

```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

Link with `-lm`.
Desciption
The `creal()` function returns the real part of the complex number `z`.
One has:

```
z = creal(z) + I * cimag(z)
```

Example

```
#include <stdio.h>
#include <complex.h>

int main()
{
  double complex z = 3.0 + 4.0i;

  printf("Re[creal(z)]=%lf and Im[creal(z)]=%lf\n", creal(creal(z)), cimag ↩
         (creal(z)));

  return 0;
}
```

and the output is:

```
Re[creal(z)]=3.000000 and Im[creal(z)]=0.000000
```

Chapter 13

Character Handling <ctype.h>

The header `<ctype.h>` declares several functions useful for classifying and mapping characters. In all cases the argument is an int, the value of which shall be representable as an unsigned char or shall equal the value of the macro EOF. If the argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the current locale. Those functions that have locale-specific aspects only when not in the ”C” locale are noted below.

The term printing character refers to a member of a locale-specific set of characters, each of which occupies one printing position on a display device; the term control character refers to a member of a locale-specific set of characters that are not printing characters.[1] All letters and digits are printing characters.

Forward references: EOF (14.19.1), localization (14.11).

13.1 Character classification functions

The functions in this subclause return nonzero (true) if and only if the value of the argument c conforms to that in the description of the function.

13.1.1 The isalnum functions

Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

Description

[1] In an implementation that uses the seven-bit US ASCII character set, the printing characters are those whose values lie from `0x20 (space)` through `0x7E (tilde)`; the control characters are those whose values lie from `0 (NUL)` through `0x1F (US)`, and the character `0x7F (DEL)`.

The `isalnum` function tests for any character for which `isalpha` or `isdigit` is true.

Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

Description

Checks for an alphanumeric character; it is equivalent to `(isalpha(c) || isdigit(c))`.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c='c', c1=9, c2='$', c3='9';

  printf("%d %d %d %d\n", isalnum(c), isalnum(c1), isalnum(c2), isalnum(c3 ↩
      ));

  return 0;
}
```

and the output is:

```
8 0 0 8
```

13.1.2 The isalpha function

Description

```
#include <ctype.h>
int isalpha(int c);
```

Description

The `isalpha` function tests for any character for which `isupper` or `islower` is true, or any character that is one of a locale-specific set of alphabetic characters for which none of `iscntrl`, `isdigit`, `ispunct` or `isspace` is true. In the "C" locale, `isalpha` returns true only for the characters for which `isupper` or `islower` is `true`.

Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

Description

Checks for an alphabetic character; in the standard ”C” locale, it is equivalent to `(isupper(c) || islower(c))`. In some locales, there may be additional characters for which `isalpha()` is true—letters which are neither upper case nor lower case.

Returns

The values returned are nonzero if the character falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c='c', c1=9, c2='$', c3='9';

  printf("%d %d %d %d\n", isalpha(c), isalpha(c1), isalpha(c2), isalpha(c3 ↩
      ));

  return 0;
}
```

and the output is:

```
1024 0 0 0
```

13.1.3 The isblank function

Synopsis

```
#include <ctype.h>
int isblank(int c);
```

Description

The `isblank` function tests for any character that is a standard blank character or is one of a locale-specific set of characters for which isspace is true and that is used to separate words within a line of text. The standard blank characters are the following: space (' '), and horizontal tab ('\t'). In the ”C” locale, isblank returns true only for the standard blank characters.

Synopsis

```
#include <ctype.h>
int isblank(int c);
```

Description

Checks for a blank character; that is, a space or a tab.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c=' ', c1='\n', c2='   ', c3='9';

  printf("%d %d %d %d\n", isblank(c), isblank(c1), isblank(c2), isblank(c3 ↩
      ));

  return 0;
}
```

and the output is:

```
1 0 1 0
```

13.1.4 The iscntrl function

Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

Description

The `iscntrl` function tests for any control character.

Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

Description

Checks for a control character.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c=0;  //NUL
```

```
  char c1=1; //SOH or ctrl-A
  char c2=3; //STX or ctrl-B
  char c3=4; //ETX or ctrl-C

  printf("%d %d %d %d\n", iscntrl(c), iscntrl(c1), iscntrl(c2), iscntrl(c3 ↩
      ));

  return 0;
}
```

and the output is:

```
2 2 2 2
```

13.1.5 The isdigit function

Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

Description
The `isdigit` function tests for any decimal-digit character.
Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

Description
Checks for a digit (0 through 9).
Returns
The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.
Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c1='a', c2='1';

  printf("%d %d\n", isdigit(c1), isdigit(c2));

  return 0;
}
```

and the output is:

```
0 2048
```

13.1.6 The isgraph function

Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

Description

The `isgraph` function tests for any printing character except space (‘ ‘).

Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

Description

Checks for any printable character except space.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c1='a', c2=' ';

  printf("%d %d\n", isgraph(c1), isgraph(c2));

  return 0;
}
```

and the output is:

```
32768 0
```

13.1.7 The islower function

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

The `islower` function tests for any character that is a lowercase letter or is one of a locale-specific set of characters for which none of `iscntrl, isdigit, ispunct` or `isspace` is true. In the ”C” locale, `islower` returns true only for the lowercase letters

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

Checks for a lower-case character.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c1='a', c2=' ';

  printf("%d %d\n", islower(c1), islower(c2));

  return 0;
}
```

and the output is:

```
512 0
```

13.1.8 The isprint function

Synopsis

```
#include <ctype.h>
int isprint(int c);
```

Description

The `isprint` function tests for any printing character including space (' ').

Synopsis

```
#include <ctype.h>
int isprint(int c);
```

Description

Checks for any printable character including space.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c1='a', c2=0;

  printf("%d %d\n", isprint(c1), isprint(c2));

  return 0;
}
```

and the output is:

```
16384 0
```

13.1.9 The ispunct function

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Description

The `ispunct` function tests for any printing character that is one of a locale-specific set of punctuation characters for which neither isspace nor isalnum is true. In the "C" locale, `ispunct` returns true for every printing character for which neither `isspace` nor `isalnum` is true.

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Description

Checks for any printable character which is not a space or an alphanumeric character.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c1='a', c2=';';

  printf("%d %d\n", ispunct(c1), ispunct(c2));

  return 0;
}
```

and the output is:

```
0 4
```

13.1.10 The isspace function

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

Description

The `isspace` function tests for any character that is a standard white-space character or is one of a locale-specific set of characters for which isalnum is false. The standard white-space characters are the following: space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). In the "C" locale, `isspace` returns true only for the standard white-space characters.

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

Description

Checks for white-space characters. In the "C" and "POSIX" locales, these are: space, form-feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v').

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
```

```
{
  char c1='\n', c2=';';

  printf("%d %d\n", isspace(c1), isspace(c2));

  return 0;
}
```

and the output is:

```
8192 0
```

13.1.11 The isupper function

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

Description

The `isupper` function tests for any character that is an uppercase letter or is one of a locale-specific set of characters for which none of `iscntrl, isdigit, ispunct` or `isspace` is true. In the "C" locale, `isupper` returns true only for the uppercase letters.

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

Description

Checks for an uppercase letter.

Returns

The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.

Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c1='A', c2=';';

  printf("%d %d\n", isupper(c1), isupper(c2));

  return 0;
}
```

and the output is:

```
256 0
```

13.1.12 The isxdigit function

Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Description
The `isxdigit` function tests for any hexadecimal-digit character.
Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Description
Checks for a hexadecimal digits, that is, one of:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.
```

Returns
The values returned are nonzero if the character `c` falls into the tested class, and a zero value if not.
Example

```
#include <stdio.h>
#include <ctype.h>

int main()
{
  char c1='a', c2=';';

  printf("%d %d\n", isxdigit(c1), isxdigit(c2));

  return 0;
}
```

and the output is:

```
4096 0
```

Chapter 14

Errors <errno.h>

The header <errno.h> defines several macros, all relating to the reporting of error conditions.

The macros are:

EDOM EILSEQ ERANGE

which expand to integer constant expressions with type `int`, distinct positive values, and which are suitable for use in `#if` preprocessing directives; and `errno` which expands to a modifiable lvalue[1] that has type `int`, the value of which is set to a positive error number by several library functions. It is unspecified whether `errno` is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name errno, the behavior is undefined.

The value of `errno` is zero at program startup, but is never set to zero by any library function.[2] The value of `errno` may be set to nonzero by a library function call whether or not there is an error, provided the use of `errno` is not documented in the description of the function in this International Standard.

Additional macro definitions, beginning with `E` and a digit or `E` and an uppercase letter, may also be specified by the implementation.

[1] The macro errno need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, `*errno()`).

[2] Thus, a program that uses `errno` for error checking should set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of `errno` on entry and then set it to zero, as long as the original value is restored if `errno`'s value is still zero just before the return.

Chapter 15

Floating-point environment <fenv.h>

The header <fenv.h> declares two types and several macros and functions to provide access to the floating-point environment. The floating-point environment refers collectively to any floating-point status flags and control modes supported by the implementation.[1] A floating-point status flag is a system variable whose value is set (but never cleared) when a floating-point exception is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information. A floating-point control mode is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.

Certain programming conventions support the intended model of use for the floating-point environment:[2]

- a function call does not alter its caller's floating-point control modes, clear its caller's floating-point status flags, nor depend on the state of its caller's floating-point status flags unless the function is so documented;
- a function call is assumed to require default floating-point control modes, unless its documentation promises otherwise;
- a function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise.

The type `fenv_t` represents the entire floating-point environment.

The type `fexcept_t` represents the floating-point status flags collectively, including any status the implementation associates with the flags.

[1] This header is designed to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559, and other similar floating-point state information. Also it is designed to facilitate code portability among all systems.

[2] With these conventions, a programmer can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

Each of the macros:

FE_DIVBYZERO FE_INEXACT FE_INVALID FE_OVERFLOW FE_UNDERFLOW

is defined if and only if the implementation supports the floating-point exception by means of the functions in 13.6.2.[3] Additional implementation-defined floating-point exceptions, with macro definitions beginning with FE_ and an uppercase letter, may also be specified by the implementation. The defined macros expand to integer constant expressions with values such that bitwise ORs of all combinations of the macros result in distinct values.

The macro FE_ALL_EXCEPT is simply the bitwise OR of all floating-point exception macros defined by the implementation. If no such macros are defined, FE_ALL_EXCEPT shall be defined as 0.

Each of the macros:

```
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
```

is defined if and only if the implementation supports getting and setting the represented rounding direction by means of the fegetround and fesetround functions. Additional implementation-defined rounding directions, with macro definitions beginning with FE_ and an uppercase letter, may also be specified by the implementation. The defined macros expand to integer constant expressions whose values are distinct nonnegative values.[4]

The macro FE_DFL_ENV represents the default floating-point environment - the one installed at program startup - and has type ”pointer to const-qualified fenv_t”. It can be used as an argument to <fenv.h> functions that manage the floating-point environment.

Additional implementation-defined environments, with macro definitions beginning with FE_ and an uppercase letter, and having type ”pointer to const-qualified fenv_t”, may also be specified by the implementation.

15.1 The FENV_ACCESS pragma

Synopsis

```
#include <fenv.h>
#pragma STDC FENV_ACCESS on-off-switch
```

Description

The FENV_ACCESS pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default

[3] The implementation supports an exception if there are circumstances where a call to at least one of the functions in 13.6.2, using the macro as the appropriate argument, will succeed. It is not necessary for all the functions to succeed all the time.

[4] Even though the rounding direction macros may expand to constants corresponding to the values of FLT_ROUNDS, they are not required to do so.

floating-point control modes.[5] The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another FENV_ACCESSpragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another FENV_ACCESS pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. If part of a program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the FENV_ACCESS pragma ”off”, the behavior is undefined. The default state (”on” or ”off”) for the pragma is implementation-defined. (When execution passes from a part of the program translated with FENV_ACCESS ”off” to a part translated with FENV_ACCESS ”on”, the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)

EXAMPLE

```
#include <fenv.h>
void f(double x)
{
  #pragma STDC FENV_ACCESS ON
  void g(double);
  void h(double);
  /* ... */
  g(x + 1);
  h(x + 1);
  /* ... */
}
```

If the function g might depend on status flags set as a side effect of the first x + 1, or if the second x + 1 might depend on control modes set as a side effect of the call to function g, then the program shall contain an appropriately placed invocation of #pragma STDC FENV_ACCESS ON.[6]

15.2 Floating-point exceptions

The following functions provide access to the floating-point status flags.[7] The int input argument for the functions represents a subset of floating-point exceptions, and can be zero or the bitwise OR

[5] The purpose of the FENV_ACCESS pragma is to allow certain optimizations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of FENV_ACCESS is ”off”, the translator can assume that default modes are in effect and the flags are not tested.

[6] The side effects impose a temporal ordering that requires two evaluations of x + 1. On the other hand, without the #pragma STDC FENV_ACCESS ON pragma, and assuming the default state is ”off”, just one evaluation of x + 1 would suffice.

[7] The functions fetestexcept, feraiseexcept and feclearexcept support the basic abstraction of flags that are either set or clear. An implementation may endow floating-point status flags with more information - for example, the address of the code which first raised the floating-point exception; the functions fegetexceptflag and fesetexceptflag deal with the full content of flags.

of one or more floating-point exception macros, for example `FE_OVERFLOW | FE_INEXACT`. For other argument values the behavior of these functions is undefined.

15.2.1 The feclearexcept function

Synopsis

```
#include <fenv.h>
int feclearexcept(int excepts);
```

Description

The `feclearexcept` function attempts to clear the supported floating-point exceptions represented by its argument.

Returns

The `feclearexcept` function returns zero if the excepts argument is zero or if all the specified exceptions were successfully cleared. Otherwise, it returns a nonzero value.

Synopsis

```
#include <fenv.h>
int feclearexcept(int excepts);
```

Link with `-lm`.

Following description, exceptions, rounding mode, floating-point environment and return values are applicable to all functions of this header

Description

This function was defined in C99, and describe the handling of floating-point rounding and exceptions (overflow, zero-divide, etc.).

Exceptions

The *divide-by-zero* exception occurs when an operation on finite numbers produces infinity as exact answer.

The *overflow* exception occurs when a result has to be represented as a floating-point number, but has (much) larger absolute value than the largest (finite) floating-point number that is representable.

The *underflow* exception occurs when a result has to be represented as a floating-point number, but has smaller absolute value than the smallest positive normalized floating-point number (and would lose much accuracy when represented as a denormalized number).

The *inexact* exception occurs when the rounded result of an operation is not equal to the infinite precision result. It may occur whenever *overflow* or *underflow* occurs.

The *invalid* exception occurs when there is no well-defined result for an operation, as for 0/0 or infinity - infinity or sqrt(-1).

Exception Handling

Exceptions are represented in two ways: as a single bit (exception present/absent), and these bits correspond in some implementation- defined way with bit positions in an integer, and also as an opaque structure that may contain more information about the exception (perhaps the code address where it occurred).

Each of the macros FE_DIVBYZERO, FE_INEXACT, FE_INVALID, FE_OVERFLOW, FE_UNDERFLOW is defined when the implementation supports handling of the corresponding exception, and if so then defines the corresponding bit(s), so that one can call exception handling functions, for example, using the integer argument FE_OVERFLOW | FE_UNDERFLOW. Other exceptions may be supported. The macro FE_ALL_EXCEPT is the bitwise OR of all bits corresponding to supported exceptions.

The feclearexcept() function clears the supported exceptions represented by the bits in its argument.

The fegetexceptflag() function stores a representation of the state of the exception flags represented by the argument excepts in the opaque object *flagp.

The feraiseexcept() function raises the supported exceptions represented by the bits in excepts.

The fesetexceptflag() function sets the complete status for the exceptions represented by excepts to the value *flagp. This value must have been obtained by an earlier call of fegetexceptflag() with a last argument that contained all bits in excepts.

The fetestexcept() function returns a word in which the bits are set that were set in the argument excepts and for which the corresponding exception is currently set.

Rounding Mode

The rounding mode determines how the result of floating-point operations is treated when the result cannot be exactly represented in the significand. Various rounding modes may be provided: round to nearest (the default), round up (toward positive infinity), round down (toward negative infinity), and round toward zero.

Each of the macros FE_TONEAREST, FE_UPWARD, FE_DOWNWARD and FE_TOWARDZERO is defined when the implementation supports getting and setting the corresponding rounding direction.

The fegetround() function returns the macro corresponding to the current rounding mode.

The fesetround() function sets the rounding mode as specified by its argument and returns zero when it was successful.

C99 and POSIX.1-2008 specify an identifier, FLT_ROUNDS, defined in <float.h>, which indicates the implementation-defined rounding behavior for floating-point addition. This identifier has one of the following values:

-1 The rounding mode is not determinable.

1 Rounding is toward nearest number.

2 Rounding is toward positive infinity.

3 Rounding is toward negative infinity.

Other values represent machine-dependent, nonstandard rounding modes.

The value of FLT_ROUNDS should reflect the current rounding mode as set by fesetround().

Floating-point environment

The entire floating-point environment, including control modes and status flags, can be handled as one opaque object, of type fenv_t. The default environment is denoted by FE_DFL_ENV (of type const fenv_t *). This is the environment setup at program start and it is defined by ISO C to have round to nearest, all exceptions cleared and a nonstop (continue on exceptions) mode.

The `fegetenv()` function saves the current floating-point environment in the object `*envp`.

The `feholdexcept()` function does the same, then clears all exception flags, and sets a non-stop (continue on exceptions) mode, if available. It returns zero when successful.

The `fesetenv()` function restores the floating-point environment from the object `*envp`. This object must be known to be valid, for example, the result of a call to `fegetenv()` or `feholdexcept()` or equal to `FE_DFL_ENV`. This call does not raise exceptions.

The `feupdateenv()` function installs the floating-point environment represented by the object `*envp`, except that currently raised exceptions are not cleared. After calling this function, the raised exceptions will be a bitwise OR of those previously set with those in `*envp`. As before, the object `*envp` must be known to be valid.

RETURN VALUE

These functions return zero on success and nonzero if an error occurred.

15.2.2 The fegetexceptflag function

Synopsis

```
#include <fenv.h>
int fegetexceptflag(fexcept_t *flagp, int excepts);
```

Description

The `fegetexceptflag` function attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument excepts in the object pointed to by the argument flagp.

Returns

The `fegetexceptflag` function returns zero if the representation was successfully stored. Otherwise, it returns a nonzero value.

15.2.3 The feraiseexcept function

Synopsis

```
#include <fenv.h>
int feraiseexcept(int excepts);
```

Description

The `feraiseexcept` function attempts to raise the supported floating-point exceptions represented by its argument.[8] The order in which these floating-point exceptions are raised is unspecified, except as stated in F.7.6 of specification. Whether the feraiseexcept function additionally raises the ”inexact” floating-point exception whenever it raises the ”overflow” or ”underflow” floating-point exception is implementation-defined.

Return

[8] The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, enabled traps for floating-point exceptions raised by this function are taken. The specification in F.7.6 of specification is in the same spirit.

The `feraiseexcept` function returns zero if the excepts argument is zero or if all the specified exceptions were successfully raised. Otherwise, it returns a nonzero value.

15.2.4 The fesetexceptflag function

Synopsis

```
#include <fenv.h>
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

Description

The `fesetexceptflag` function attempts to set the floating-point status flags indicated by the argument excepts to the states stored in the object pointed to by `flagp`. The value of `*flagp` shall have been set by a previous call to `fegetexceptflag` whose second argument represented at least those floating-point exceptions represented by the argument excepts. This function does not raise floating-point exceptions, but only sets the state of the flags.

Returns

The `fesetexceptflag` function returns zero if the excepts argument is zero or if all the specified flags were successfully set to the appropriate state. Otherwise, it returns a nonzero value.

15.2.5 The fetestexcept function

Synopsis

```
#include <fenv.h>
int fetestexcept(int excepts);
```

Description

The fetestexcept function determines which of a specified subset of the floating-point exception flags are currently set. The excepts argument specifies the floating-point status flags to be queried.[9]

Returns

[9] This mechanism allows testing several floating-point exceptions with just one function call.
EXAMPLE Call `f` if "invalid" is set, then `g` if "overflow" is set:

```
#include <fenv.h>
/* ... */
{
  #pragma STDC FENV_ACCESS ON
  int set_excepts;
  feclearexcept(FE_INVALID | FE_OVERFLOW);
  // maybe raise exceptions
  set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
  if (set_excepts & FE_INVALID) f();
  if (set_excepts & FE_OVERFLOW) g();
  /* ... */
}
```

The `fetestexcept` function returns the value of the bitwise OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in `excepts`.

15.3 Rounding

The `fegetround` and `fesetround` functions provide control of rounding direction modes.

15.3.1 The fegetround function

Synopsis

```
#include <fenv.h>
int fegetround(void);
```

Description

The `fegetround` function gets the current rounding direction.

Returns

The `fegetround` function returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

15.3.2 The fesetround function

Synopsis

```
#include <fenv.h>
int fesetround(int round);
```

Description

The `fesetround` function establishes the rounding direction represented by its argument round. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

Returns

The `fesetround` function returns zero if and only if the requested rounding direction was established.

EXAMPLE Save, set, and restore the rounding direction. Report an error and abort if setting the rounding direction fails.

```
#include <fenv.h>
#include <assert.h>
void f(int round_dir)
{
  #pragma STDC FENV_ACCESS ON
  int save_round;
  int setround_ok;
```

```
  save_round = fegetround();
  setround_ok = fesetround(round_dir);
  assert(setround_ok == 0);
  /* ... */
  fesetround(save_round);
  /* ... */
}
```

15.4 Environment

The functions in this section manage the floating-point environment - status flags and control modes - as one entity.

15.4.1 The fegetenv function

Synopsis

```
#include <fenv.h>
int fegetenv(fenv_t *envp);
```

Description

The `fegetenv` function attempts to store the current floating-point environment in the object pointed to by envp.

Returns

The `fegetenv` function returns zero if the environment was successfully stored. Otherwise, it returns a nonzero value.

15.4.2 The feholdexcept function

Synopsis

```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

Description

The `feholdexcept` function saves the current floating-point environment in the object pointed to by `envp`, clears the floating-point status flags, and then installs a non-stop (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.[10]

Returns

The `feholdexcept` function returns zero if and only if non-stop floating-point exception handling was successfully installed.

[10] IEC 60559 systems have a default non-stop mode, and typically at least one other mode for trap handling or aborting; if the system provides only the non-stop mode then installing it is trivial. For such systems, the `feholdexcept` function can be used in conjunction with the `feupdateenv` function to write routines that hide spurious floating-point exceptions from their callers.

15.4.3 The fesetenv function

Synopsis

```
#include <fenv.h>
int fesetenv(const fenv_t *envp);
```

Description

The `fesetenv` function attempts to establish the floating-point environment represented by the object pointed to by `envp`. The argument envp shall point to an object set by a call to `fegetenv` or `feholdexcept`, or equal a floating-point environment macro. Note that fesetenv merely installs the state of the floating-point status flags represented through its argument, and does not raise these floating-point exceptions.

Returns

The `fesetenv` function returns zero if the environment was successfully established. Otherwise, it returns a nonzero value.

15.4.4 The feupdateenv function

Synopsis

```
#include <fenv.h>
int feupdateenv(const fenv_t *envp);
```

Description

The `feupdateenv` function attempts to save the currently raised floating-point exceptions in its automatic storage, install the floating-point environment represented by the object pointed to by `envp`, and then raise the saved floating-point exceptions. The argument `envp` shall point to an object set by a call to `feholdexcept` or `fegetenv`, or equal a floating-point environment macro.

Returns

The `feupdateenv` function returns zero if all the actions were successfully carried out. Otherwise, it returns a nonzero value.

EXAMPLE Hide spurious underflow floating-point exceptions:

```
#include <fenv.h>
double f(double x)
{
  #pragma STDC FENV_ACCESS ON
  double result;
  fenv_t save_env;
  if (feholdexcept(&save_env))
    return /* indication of an environmental problem */;
  // compute result
  if (/* test spurious underflow */)
    if (feclearexcept(FE_UNDERFLOW))
      return /* indication of an environmental problem */;
  if (feupdateenv(&save_env))
```

```
    return /* indication of an environmental problem */;
  return result;
}
```

Chapter 16

Characterictics of floating types <float.h>

The header <float.h> defines several macros that expand to various limits and parameters of the standard floating-point types.

The macros, their meanings, and the constraints (or restrictions) on their values are listed in Numerical limits.

Chapter 17

Format conversion of integer types <inttypes.h>

The header <inttypes.h> includes the header <stdint.h> and extends it with additional facilities provided by hosted implementations.

It declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers, and it declares the type `imaxdiv_t` which is a structure type that is the type of the value returned by the `imaxdiv` function.

For each type declared in <stdint.h>, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.

Forward references: integer types <stdint.h> (Integer types <stdint.h>), formatted input/output functions (Formatted input/output functions), formatted wide character input/output functions (Formatted wide character input/output functions).

17.1 Macros for format specifiers

Each of the following object-like macros[1] expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the format argument of a formatted input/output function when converting the corresponding integer type. These macro names have the general form of `PRI` (character string literals for the `fprintf` and `fwprintf` family) or `SCN` (character string literals for the `fscanf` and `fwscanf` family),[2] followed by the conversion specifier, followed by a name corresponding to a similar type name in Integer types. In these names, `N` represents the width of the type as described in Integer types. For example, `PRIdFAST32` can be used in a format string to print the value of an integer of type `int_fast32_t`.

[1] C++ implementations should define these macros only when `__STDC_FORMAT_MACROS` is defined before <inttypes.h> is included.

[2] Separate macros are given for use with `fprintf` and `fscanf` functions because, in the general case, different format specifiers may be required for `fprintf` and `fscanf`, even when the type is the same.

The `fprintf` macros for signed integers are:

PRIdN PRIdLEASTN PRIdFASTN PRIdMAX PRIdPTR

PRIiN PRIiLEASTN PRIiFASTN PRIiMAX PRIiPTR

The `fprintf` macros for unsigned integers are:

PRIoN PRIoLEASTN PRIoFASTN PRIoMAX PRIoPTR

PRIuN PRIuLEASTN PRIuFASTN PRIuMAX PRIuPTR

PRIxN PRIxLEASTN PRIxFASTN PRIxMAX PRIxPTR

PRIXN PRIXLEASTN PRIXFASTN PRIXMAX PRIXPTR

The `fscanf` macros for signed integers are:

SCNdN SCNdLEASTN SCNdFASTN SCNdMAX SCNdPTR

SCNiN SCNiLEASTN SCNiFASTN SCNiMAX SCNiPTR

The `fscanf` macros for unsigned integers are:

SCNoN SCNoLEASTN SCNoFASTN SCNoMAX SCNoPTR

SCNuN SCNuLEASTN SCNuFASTN SCNuMAX SCNuPTR

SCNxN SCNxLEASTN SCNxFASTN SCNxMAX SCNxPTR

For each type that the implementation provides in <stdint.h>, the corresponding fprintf macros shall be defined and the corresponding fscanf macros shall be defined unless the implementation does not have a suitable fscanf length modifier for the type.

EXAMPLE

```
#include <inttypes.h>
#include <wchar.h>

int main(void)
{
  uintmax_t i = UINTMAX_MAX; // this type always exists

  wprintf(L"The largest integer value is %020" PRIxMAX "\n", i);

  return 0;
}
```

17.2 Functions for greatest-width integer types

17.2.1 The imaxabs function

Synopsis

```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
```

Description

The `imaxabs` function computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.[3]

Returns

The `imaxabs` function returns the absolute value.

17.2.2 The imaxdiv function

Synopsis

```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Description

The `imaxdiv` function computes `numer /denom` and `numer % denom` in a single operation.

Returns

The `imaxdiv` function returns a structure of type `imaxdiv_t` comprising both the quotient and the remainder. The structure shall contain (in either order) the members quot (the quotient) and rem (the remainder), each of which has type `intmax_t`. If either part of the result cannot be represented, the behavior is undefined.

17.2.3 The strtoimax and strtoumax functions

Synopsis

```
#include <inttypes.h>
intmax_t strtoimax(const char * restrict nptr,
char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr,
char ** restrict endptr, int base);
```

Description

The `strtoimax` and `strtoumax` functions are equivalent to the `strtol, strtoll, strtoul` and `strtoull` functions, except that the initial portion of the string is converted to `intmax_t` and `uintmax_t` representation, respectively.

Returns

The `strtoimax` and `strtoumax` functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `INTMAX_MAX, INTMAX_MIN` or `UINTMAX_MAX` is returned (according to the return type and sign of the value, if any), and the value of the macro `ERANGE` is stored in `errno`.

[3] The absolute value of the most negative number cannot be represented in two's complement.

Forward references: the `strtol, strtoll, strtoul` and `strtoull` functions (The strtol, strtoll, strtoul and strtoull functions).

17.2.4 The wcstoimax and wcstoumax functions

Synopsis

```
#include <stddef.h> // for wchar_t
#include <inttypes.h>
intmax_t wcstoimax(const wchar_t * restrict nptr,
wchar_t ** restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr,
wchar_t ** restrict endptr, int base);
```

Description

The `wcstoimax` and `wcstoumax` functions are equivalent to the `wcstol, wcstoll, wcstoul` and `wcstoull` functions except that the initial portion of the wide string is converted to `intmax_t` and `uintmax_t` representation, respectively.

Returns

The `wcstoimax` function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `INTMAX_MAX, INTMAX_MIN` or `UINTMAX_MAX` is returned (according to the return type and sign of the value, if any), and the value of the macro `ERANGE` is stored in `errno`.

Chapter 18

Alternative spellings <iso646.h>

The header <iso646.h> defines the following eleven macros (on the left) that expand to the corresponding tokens (on the right):

```
and &&
and_eq &=
bitand &
bitor |
compl ~
not !
not_eq !=
or ||
or_eq |=
xor ^
xor_eq ^=
```

Chapter 19

Sizes of integer types <limits.h>

The header <limits.h> defines several macros that expand to various limits and parameters of the standard integer types.

The macros, their meanings, and the constraints (or restrictions) on their values are listed in Numerical limits.

Chapter 20

Localization <locale.h>

The header <locale.h> declares two functions, one type, and defines several macros.

The type is

```
struct lconv
```

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges are explained in The localeconv function. In the ”C” locale, the members shall have the values specified in the comments.

```
char *decimal_point;     // "."
char *thousands_sep;     // ""
char *grouping;          // ""
char *mon_decimal_point; // ""
char *mon_thousands_sep; // ""
char *mon_grouping;      // ""
char *positive_sign;     // ""
char *negative_sign;     // ""
char *currency_symbol;   // ""
char frac_digits;        // CHAR_MAX
char p_cs_precedes;      // CHAR_MAX
char n_cs_precedes;      // CHAR_MAX
char p_sep_by_space;     // CHAR_MAX
char n_sep_by_space;     // CHAR_MAX
char p_sign_posn;        // CHAR_MAX
char n_sign_posn;        // CHAR_MAX
char *int_curr_symbol;   // ""
char int_frac_digits;    // CHAR_MAX
char int_p_cs_precedes;  // CHAR_MAX
char int_n_cs_precedes;  // CHAR_MAX
char int_p_sep_by_space; // CHAR_MAX
char int_n_sep_by_space; // CHAR_MAX
```

```
char int_p_sign_posn;    // CHAR_MAX
char int_n_sign_posn;    // CHAR_MAX
```

The macros defined are NULL (described in Common definitions <stddef.h>); and

LC_ALL

LC_COLLATE

LC_CTYPE

LC_MONETARY

LC_NUMERIC

LC_TIME

which expand to integer constant expressions with distinct values, suitable for use as the first argument to the setlocale function.[1] Additional macro definitions, beginning with the characters LC_ and an uppercase letter.

20.1 Locale Control

20.1.1 The setlocale function

Synopsis

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Description

The setlocale function selects the appropriate portion of the program's locale as specified by the category and locale arguments. The setlocale function may be used to change or query the programs entire current locale or portions thereof. The value LC_ALL for category names the program's entire locale; the other values for category name only a portion of the program's locale. LC_COLLATE affects the behavior of the strcoll and strxfrm functions. eLC_CTYPE affects the behavior of the character handling functions[2] and the multibyte and wide character functions. LC_MONETARY affects the monetary formatting information returned by the localeconv function. LC_NUMERIC affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the localeconv function. LC_TIME affects the behavior of the strftime and wcsftime functions.

A value of "C" for locale specifies the minimal environment for C translation; a value of "" for locale specifies the locale-specific native environment. Other implementation-defined strings may be passed as the second argument to setlocale.

At program startup, the equivalent of

[1] ISO/IEC 9945-2 specifies locale and charmap formats that may be used to specify locales for C.

[2] The only functions in Character Handling <ctype.h> whose behavior is not affected by the current locale are isdigit and isxdigit.

```
setlocale(LC_ALL, "C");
```

is executed.

The implementation shall behave as if no library function calls the `setlocale` function.

Returns

If a pointer to a string is given for `locale` and the selection can be honored, the `setlocale` function returns a pointer to the string associated with the specified category for the new `locale`. If the selection cannot be honored, the `setlocale` function returns a null pointer and the program's `locale` is not changed.

A null pointer for `locale` causes the `setlocale` function to return a pointer to the string associated with the category for the program's current `locale`; the program's `locale` is not changed.

The pointer to string returned by the `setlocale` function is such that a subsequent call with that string value and its associated category will restore that part of the program's `locale`. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the `setlocale` function.

20.2 Numeric formatting convention inquiry

20.2.1 The localeconv function

Synopsis

```
#include <locale.h>
struct lconv *localeconv(void);
```

Description

The `localeconv` function sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current `locale`.

The members of the structure with type `char *` are pointers to strings, any of which (except decimal_point) can point to "", to indicate that the value is not available in the current `locale` or is of zero length. Apart from grouping and mon_grouping, the strings shall start and end in the initial shift state. The members with type char are nonnegative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current `locale`. The members include the following:

```
char *decimal_point
```

The decimal-point character used to format nonmonetary quantities.

```
char *thousands_sep
```

The character used to separate groups of digits before the decimal-point. character in formatted nonmonetary quantities.

```
char *grouping
```

A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

```
char *mon_decimal_point
```

The decimal-point used to format monetary quantities.

```
char *mon_thousands_sep
```

The separator for groups of digits before the decimal-point in formatted monetary quantities.

```
char *mon_grouping
```

A string whose elements indicate the size of each group of digits in formatted monetary quantities.

```
char *positive_sign
```

The string used to indicate a nonnegative-valued formatted monetary quantity.

```
char *negative_sign
```

The string used to indicate a negative-valued formatted monetary quantity.

```
char *currency_symbol
```

The local currency symbol applicable to the current locale.

```
char frac_digits
```

The number of fractional digits (those after the decimal-point) to be displayed in a locally formatted monetary quantity.

```
char p_cs_precedes
```

Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a nonnegative locally formatted monetary quantity.

```
char n_cs_precedes
```

Set to 1 or 0 if the `currency_symbol` respectively precedes or succeeds the value for a negative locally formatted monetary quantity.

```
char p_sep_by_space
```

Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a nonnegative locally formatted monetary quantity.

```
char n_sep_by_space
```

Set to a value indicating the separation of the `currency_symbol`, the sign string, and the value for a negative locally formatted monetary quantity.

```
char p_sign_posn
```

Set to a value indicating the positioning of the `positive_sign` for a nonnegative locally formatted monetary quantity.

```
char n_sign_posn
```

Set to a value indicating the positioning of the `negative_sign` for a negative locally formatted monetary quantity.

```
char *int_curr_symbol
```

The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

```
char int_frac_digits
```

The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

```
char int_p_cs_precedes
```

Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a nonnegative internationally formatted monetary quantity.

```
char int_n_cs_precedes
```

Set to 1 or 0 if the `int_curr_symbol` respectively precedes or succeeds the value for a negative internationally formatted monetary quantity.

```
char int_p_sep_by_space
```

Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a nonnegative internationally formatted monetary quantity.

```
char int_n_sep_by_space
```

Set to a value indicating the separation of the `int_curr_symbol`, the sign string, and the value for a negative internationally formatted monetary quantity.

```
char int_p_sign_posn
```

Set to a value indicating the positioning of the `positive_sign` for a nonnegative internationally formatted monetary quantity.

```
char int_n_sign_posn
```

Set to a value indicating the positioning of the `negative_sign` for a negative internationally formatted monetary quantity.

The elements of grouping and mon_grouping are interpreted according to the following:

CHAR_MAX No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that compose the current group.

The next element is examined to determine the size of the next group of digits before the current group.

The values of p_sep_by_space, n_sep_by_space, int_p_sep_by_space and int_n_sep_by_space are interpreted according to the following:

0 No space separates the currency symbol and value.

1 If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

For int_p_sep_by_space and int_n_sep_by_space, the fourth character of int_curr_symbol is used instead of a space.

The values of p_sign_posn, n_sign_posn, int_p_sign_posn and int_n_sign_posn are interpreted according to the following:

0 Parentheses surround the quantity and currency symbol.

1 The sign string precedes the quantity and currency symbol.

2 The sign string succeeds the quantity and currency symbol.

3 The sign string immediately precedes the currency symbol.

4 The sign string immediately succeeds the currency symbol.

The implementation shall behave as if no library function calls the localeconv function.

Returns

The localeconv function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the localeconv function. In addition, calls to the setlocale function with categories LC_ALL, LC_MONETARY or LC_NUMERIC may overwrite the contents of the structure.

Please refer to three tables in specification for reference examples.

Chapter 21

Mathematics <math.h>

The header <math.h> declares two types and many mathematical functions and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more double parameters, a double return value, or both; and other functions with the same name but with f and l suffixes, which are corresponding functions with float and long double parameters, return values, or both.[1] Integer arithmetic functions and conversion functions are discussed later.

The types

float_t

double_t

are floating types at least as wide as float and double, respectively, and such that double_t is at least as wide as float_t. If FLT_EVAL_METHOD equals 0, float_t and double_t are float and double, respectively; if FLT_EVAL_METHOD equals 1, they are both double; if FLT_EVAL_METHOD equals 2, they are both long double; and for other values of FLT_EVAL_METHOD, they are otherwise implementation-defined.[2]

The macro

HUGE_VAL

expands to a positive double constant expression, not necessarily representable as a float. The macros

HUGE_VALF

HUGE_VALL

[1] Particularly on systems with wide expression evaluation, a <math.h> function might pass arguments and return values in wider format than the synopsis prototype indicates.

[2] The types float_t and double_t are intended to be the implementation's most efficient types at least as wide as float and double, respectively. For FLT_EVAL_METHOD equal 0, 1 or 2, the type float_t is the narrowest type used by the implementation to evaluate floating expressions.

are respectively `float` and `long double` analogs of `HUGE_VAL`.[3]
The macro

`INFINITY`

expands to a constant expression of type `float` representing positive or unsigned infinity, if available; else to a positive constant of type `float` that overflows at translation time.[4]
The macro

`NAN`

is defined if and only if the implementation supports quiet NaNs for the `float` type. It expands to a constant expression of type float representing a quiet NaN.
The *number classification macros*

`FP_INFINITE`

`FP_NAN`

`FP_NORMAL`

`FP_SUBNORMAL`

`FP_ZERO`

represent the mutually exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floatingpoint classifications, with macro definitions beginning with `FP_1` and an uppercase letter, may also be specified by the implementation.
The macro

FP_FAST_FMA

is optionally defined. If defined, it indicates that the fma function generally executes about as fast as, or faster than, a multiply and an add of double operands.[5] The macros

`FP_FAST_FMAF`

`FP_FAST_FMAL`

are, respectively, `float` and `long double` analogs of `FP_FAST_FMA`. If defined, these macros expand to the integer constant 1.
The macros

`FP_ILOGB0`

`FP_ILOGBNAN`

[3] `HUGE_VAL, HUGE_VALF` and `HUGE_VALL` can be positive infinities in an implementation that supports infinities.
[4] In this case, using `INFINITY` will violate the constraint in Constants and thus require a diagnostic.
[5] Typically, the `FP_FAST_FMA` macro is defined if and only if the `fma` function is implemented directly with a hardware multiply-add instruction. Software implementations are expected to be substantially slower.

expand to integer constant expressions whose values are returned by `ilogb(x)` if `x` is zero or NaN, respectively. The value of `FP_ILOGB0` shall be either `INT_MIN` or `-INT_MAX`. The value of `FP_ILOGBNAN` shall be either `INT_MAX` or `INT_MIN`.

The macros

`MATH_ERRNO`

`MATH_ERREXCEPT`

expand to the integer constants 1 and 2, respectively; the macro

`math_errhandling`

expands to an expression that has type int and the value `MATH_ERRNO, MATH_ERREXCEPT` or the bitwise OR of both. The value of `math_errhandling` is constant for the duration of the program. It is unspecified whether `math_errhandling` is a macro or an identifier with external linkage. If a macro definition is suppressed or a program defines an identifier with the name `math_errhandling`, the behavior is undefined. If the expression math_errhandling & `MATH_ERREXCEPT` can be nonzero, the implementation shall define the macros `FE_DIVBYZERO, FE_INVALID` and in <fenv.h>.

21.1 Treatment of error conditions

The behavior of each of the functions in <math.h> is specified for all representable values of its input arguments, except where stated otherwise. Each function shall execute as if it were a single operation without generating any externally visible exceptional conditions.

For all functions, a domain error occurs if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.[6] On a domain error, the function returns an implementation-defined value; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression errno acquires the value `EDOM`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the ”invalid” floating-point exception is raised.

Similarly, a range error occurs if the mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude.

A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, or if the mathematical result is an exact infinity (for example `log(0.0)`), then the function returns the value of the macro `HUGE_VAL, HUGE_VALF` or `HUGE_VALL` according to the return type, with the same sign as the correct value of the function; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, the integer expression errno acquires the value `ERANGE`; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, the ”divide-by-zero” floating-point exception

[6] In an implementation that supports infinities, this allows an infinity as an argument to be a domain error if the mathematical domain of the function does not include the infinity.

is raised if the mathematical result is an exact infinity and the ”overflow” floating-point exception is raised otherwise.

The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type.[7] If the result underflows, the function returns an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression `math_errhandling & MATH_ERRNO` is nonzero, whether errno acquires the value `ERANGE` is implementation-defined; if the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero, whether the ”underflow” floating-point exception is raised is implementation-defined.

21.2 The FP_CONTRACT pragma

Synopsis

```
#include <math.h>
#pragma STDC FP_CONTRACT on-off-switch
```

Description

The `FP_CONTRACT` pragma can be used to allow (if the state is ”on”) or disallow (if the state is ”off”) the implementation to contract expressions (Expressions). Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state (”on” or ”off”) for the pragma is implementation-defined.

21.3 Classification macros

In the synopses in this subclause, real-floating indicates that the argument shall be an expression of real floating type.

21.3.1 The fpclassify macro

Synopsis

```
#include <math.h>
int fpclassify(real-floating x);
```

[7] The term underflow here is intended to encompass both ”gradual underflow” as in IEC 60559 and also ”flush-to-zero” underflow.

Description

The `fpclassify` macro classifies its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.[8]

Returns

The `fpclassify` macro returns the value of the number classification macro appropriate to the value of its argument.

EXAMPLE The `fpclassify` macro might be implemented in terms of ordinary functions as

```
#define fpclassify(x) \
        ((sizeof (x) == sizeof (float)) ? _ _fpclassifyf(x) : \
        (sizeof (x) == sizeof (double)) ? _ _fpclassifyd(x) : \
        _ _fpclassifyl(x))
```

21.3.2 The isfinite macro

Synopsis

```
#include <math.h>
int isfinite(real-floating x);
```

Description

The `isfinite` macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

The `isfinite` macro returns a nonzero value if and only if its argument has a finite value.

21.3.3 The isinf macro

Synopsis

```
#include <math.h>
int isinf(real-floating x);
```

Description

The `isinf` macro determines whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

The `isinf` macro returns a nonzero value if and only if its argument has an infinite value.

[8] Since an expression can be evaluated with more range and precision than its type has, it is important to know the type that classification is based on. For example, a normal `long double` value might become subnormal when converted to `double`, and zero when converted to `float`.

21.3.4 The isnan macro

Synopsis

```
#include <math.h>
int isnan(real-floating x);
```

Description

The `isnan` macro determines whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.[9]

Returns

The `isnan` macro returns a nonzero value if and only if its argument has a NaN value.

21.3.5 The isnormal macro

Synopsis

```
#include <math.h>
int isnormal(real-floating x);
```

Description

The `isnormal` macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

The `isnormal` macro returns a nonzero value if and only if its argument has a normal value.

21.3.6 The signbit macro

Synopsis

```
#include <math.h>
int signbit(real-floating x);
```

Description

The `signbit` macro determines whether the sign of its argument value is negative.[10]

Returns

The `signbit` macro returns a nonzero value if and only if the sign of its argument value is negative.

[9] For the isnan macro, the type for determination does not matter unless the implementation supports NaNs in the evaluation type but not in the semantic type.

[10] The `signbit` macro reports the sign of all values, including infinities, zeros, and NaNs. If zero is unsigned, it is treated as positive.

21.4 Trigonometric functions

21.4.1 The acos functions

Synopsis

```
#include <math.h>
double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Description

The `acos` functions compute the principal value of the arc cosine of `x`. A domain error occurs for arguments not in the interval [-1, +1].

Returns

The acos functions return arccos x in the interval $[0, \pi]$ radians.

21.4.2 The asin functions

Synopsis

```
#include <math.h>
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Description

The `asin` functions compute the principal value of the arc sine of `x`. A domain error occurs for arguments not in the interval [-1, +1].

Returns

The `asin` functions return arcsin `x` in the interval $[-\pi/2, +\pi/2]$ radians.

21.4.3 The atan functions

Synopsis

```
#include <math.h>
double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

Description

The `atan` functions compute the principal value of the arc tangent of `x`.

Returns

The `atan` functions return arctan `x` in the interval $[-\pi/2, +\pi/2]$ radians.

21.4.4 The atan2 functions

Synopsis

```
#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Description

The `atan2` functions compute the value of the arc tangent of `y/x`, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

Returns

The `atan2` functions return arctan `y/x` in the interval $[-\pi, +\pi]$ radians.

21.4.5 The cos functions

Synopsis

```
#include <math.h>
double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

Description

The `cos` functions compute the cosine of `x` (measured in radians).

Returns

The `cos` functions return cos `x`.

21.4.6 The sin functions

Synopsis

```
#include <math.h>
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Description

The `sin` functions compute the sine of `x` (measured in radians).

Returns

The `sin` functions return sin `x`.

21.4.7 The tan functions

Synopsis

```
#include <math.h>
double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

Description

The `tan` functions return the tangent of `x` (measured in radians).

Returns

The `tan` functions return tan `x`.

21.5 Hyperbolic functions

21.5.1 The acosh functions

Synopsis

```
#include <math.h>
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Description

The `acosh` functions compute the (nonnegative) arc hyperbolic cosine of `x`. A domain error occurs for arguments less than 1.

Returns

The acosh functions return arcosh `x` in the interval $[0, +\infty]$.

21.5.2 The asinh functions

Synopsis

```
#include <math.h>
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

Description

The `asinh` functions compute the arc hyperbolic sine of `x`.

Returns

The `asinh` functions return arsinh `x`.

21.5.3 The atanh functions

Synopsis

```
#include <math.h>
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

Description

The `atanh` functions compute the arc hyperbolic tangent of `x`. A domain error occurs for arguments not in the interval [-1, +1]. A range error may occur if the argument equals -1 or +1.

Returns

The `atanh` functions return atanh `x`.

21.5.4 The cosh functions

Synopsis

```
#include <math.h>
double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

Description

The `cosh` functions compute the hyperbolic cosine of `x`. A range error occurs if the magnitude of `x` is too large.

Returns

The `cosh` functions return cosh `x`.

21.5.5 The sinh functions

Synopsis

```
#include <math.h>
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

Description

The `sinh` functions compute the hyperbolic sine of `x`. A range error occurs if the magnitude of `x` is too large.

Returns

The `sinh` functions return sinh `x`.

21.5.6 The tanh functions

Synopsis

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

Description

The `tanh` functions compute the hyperbolic tangent of `x`.

Returns

The `tanh` functions return tanh `x`.

21.6 Exponential and logrithmic functions

21.6.1 The exp functions

Synopsis

```
#include <math.h>
double exp(double x);
float expf(float x);
long double expl(long double x);
```

Description

The `exp` functions compute the base-e exponential of `x`. A range error occurs if the magnitude of `x` is too large.

Returns

The `exp` functions return e^x.

21.6.2 The exp2 functions

Synopsis

```
#include <math.h>
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

Description

The `exp2` functions compute the base-2 exponential of `x`. A range error occurs if the magnitude of `x` is too large.

Returns

The exp2 functions return 2^x.

21.6.3 The expml functions

Synopsis

```
#include <math.h>
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

Description

The `expm1` functions compute the base-e exponential of the argument, minus 1. A range error occurs if `x` is too large.[11]

Returns

The expm1 functions return $e^x - 1$

21.6.4 The frexp functions

Synopsis

```
#include <math.h>
double frexp(double value, int *exp);
float frexpf(float value, int *exp);
long double frexpl(long double value, int *exp);
```

Description

The `frexp` functions break a floating-point number into a normalized fraction and an integral power of 2. They store the integer in the int object pointed to by `exp`.

Returns

If `value` is not a floating-point number, the results are unspecified. Otherwise, the `frexp` functions return the value `x`, such that `x` has a magnitude in the interval [1/2, 1) or zero, and value equals $x * 2^{*exp}$.

21.6.5 The ilogb functions

Synopsis

```
#include <math.h>
int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

Description

The `ilogb` functions extract the exponent of `x` as a signed int value. If `x` is zero they compute the value `FP_ILOGB0`; if `x` is infinite they compute the value `INT_MAX`; if `x` is a NaN they compute the value `FP_ILOGBNAN`; otherwise, they are equivalent to calling the corresponding `logb` function and

[11] For small magnitude `x`, `expm1(x)` is expected to be more accurate than `exp(x) -1`.

casting the returned value to type `int`. A domain error or range error may occur if `x` is zero, infinite, or NaN. If the correct value is outside the range of the return type, the numeric result is unspecified.

Returns

The `ilogb` functions return the exponent of `x` as a signed int value.

Forward references: the `logb` functions (The logb functions).

21.6.6 The ldexp functions

Synopsis

```
#include <math.h>
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

Description

The `ldexp` functions multiply a floating-point number by an integral power of 2. A range error may occur.

Returns

The ldexp functions return $x * 2^{exp}$.

21.6.7 The log functions

Synopsis

```
#include <math.h>
double log(double x);
float logf(float x);
long double logl(long double x);
```

Description

The `log` functions compute the base-e (natural) logarithm of `x`. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

Returns

The log functions return $log_e x$.

21.6.8 The log10 functions

Synopsis

```
#include <math.h>
double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

Description

The `log10` functions compute the base-10 (natural) logarithm of `x`. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

Returns

The log10 functions return $log_{10}x$.

21.6.9 The log1p functions

Synopsis

```
#include <math.h>
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

Description

The `log1p` functions compute the base-e (natural) logarithm of 1 plus the argument.[12] A domain error occurs if the argument is less than -1. A range error may occur if the argument equals 11.

Returns

The log1p functions return $log_e(1 + x)$

21.6.10 The log2 functions

Synopsis

```
#include <math.h>
double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

Description

The `log2` functions compute the base-2 logarithm of `x`. A domain error occurs if the argument is less than zero. A range error may occur if the argument is zero.

Returns

The `log2` functions return log_2x

21.6.11 The logb functions

Synopsis

```
#include <math.h>
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

[12] For small magnitude , `log1p(x)` is expected to be more accurate than `log(1 + x)`.

Description

The `logb` functions extract the exponent of `x`, as a signed integer value in floating-point format. If `x` is subnormal it is treated as though it were normalized; thus, for positive finite `x`,

$$1 \leq x * FLT\_RADIX^{-logb(x)} < FLT\_RADIX$$

A domain error or range error may occur if the argument is zero.

Returns

The `logb` functions return the signed exponent of `x`.

21.6.12 The modf functions

Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

Description

The `modf` functions break the argument value into integral and fractional parts, each of which has the same type and sign as the argument. They store the integral part (in floating-point format) in the object pointed to by `iptr`.

Returns

The `modf` functions return the signed fractional part of value.

21.6.13 The scalbn and scalbln functions

Synopsis

```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

Description

The `scalbn` and `scalbln` functions compute $x * FLT\_RADIX^n$ efficiently, not normally by computin FLT_RADIX^n explicitly. A range error may occur.

Returns

The `scalbn` and `scalbln` functions return $x * FLT\_RADIX^n$.

21.7 Power and absolute-value functions

21.7.1 The cbrt functions

Synopsis

```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Description

The `cbrt` functions compute the real cube root of `x`.

Returns

The `cbrt` functions return $x^{1/3}$.

21.7.2 The fabs function

Synopsis

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

Description

The `fabs` functions compute the absolute value of a floating-point number `x`.

Returns

The `fabs` functions return $|x|$.

21.7.3 The hypot functions

Synopsis

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

Description

The `hypot` functions compute the square root of the sum of the squares of `x` and `y`, without undue overflow or underflow. A range error may occur.

Returns

The `hypot` functions return $\sqrt{x^2 + y^2}$

21.7.4 The pow functions

Synopsis

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

Description

The pow functions compute x raised to the power y. A domain error occurs if x is finite and negative and y is finite and not an integer value. A range error may occur. A domain error may occur if x is zero and y is zero. A domain error or range error may occur if x is zero and y is less than zero.

Returns

The pow functions return x^y.

21.7.5 The sqrt functions

Synopsis

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

Description

The sqrt functions compute the nonnegative square root of x. A domain error occurs if the argument is less than zero.

Returns

The sqrt functions return $\sqrt{x}$

21.8 Error and gamma functions

21.8.1 The erf functions

Synopsis

```
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);
```

Description

The erf functions compute the error function of .

Returns

The functions return erf $x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$

21.8.2 The erfc functions

Synopsis

```
#include <math.h>
double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

Description

The `erfc` functions compute the complementary error function of `x`. A range error occurs if `x` is too large.

Returns

The `erfc` functions return `erfc` $x = 1 - erfx = \frac{2}{\sqrt{\pi}} \int_0^\infty e^{-t^2} dt$

21.8.3 The lgamma functions

Synopsis

```
#include <math.h>
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
```

Description

The `lgamma` functions compute the natural logarithm of the absolute value of gamma of `x`. A range error occurs if `x` is too large. A range error may occur if `x` is a negative integer or zero.

Returns

The `lgamma` functions return $log_e|(x)|$

21.8.4 The tgamma functions

Synopsis

```
#include <math.h>
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

Description

The functions compute the gamma function of `x`. A domain error or range error may occur if `x` is a negative integer or zero. A range error may occur if the magnitude of `x` is too large or too small.

Returns

The `tgamma` functions return (x)

21.9 Nearest integer functions

21.9.1 The ceil functions

Synopsis

```
#include <math.h>
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

Description

The `ceil` functions compute the smallest integer value not less than `x`.

Returns

The functions return $\lceil x \rceil$, expressed as a floating-point number.

21.9.2 The floor functions

Synopsis

```
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

Description

The `floor` functions compute the largest integer value not greater than `x`.

Returns

The `floor` functions return $\lfloor x \rfloor$, expressed as a floating-point number.

21.9.3 The nearbyint functions

Synopsis

```
#include <math.h>
double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

Description

The `nearbyint` functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the ”inexact” floatingpoint exception.

Returns

The `nearbyint` functions return the rounded integer value.

21.9.4 The rint functions

Synopsis

```
#include <math.h>
double rint(double x);
float rintf(float x);
long double rintl(long double x);
```

Description

The `rint` functions differ from the nearbyint functions (The nearbyint functions) only in that the rint functions may raise the ”inexact” floating-point exception if the result differs in value from the argument.

Returns

The `rint` functions return the rounded integer value.

21.9.5 The lrint and llrint functions

Synopsis

```
#include <math.h>
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

Description

The `lrint` and `llrint` functions round their argument to the nearest integer value, rounding according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

The `lrint` and `llrint` functions return the rounded integer value.

21.9.6 The round functions

Synopsis

```
#include <math.h>
double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Description

The `round` functions round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

Returns

The `round` functions return the rounded integer value.

21.9.7 The lround and llround functions

Synopsis

```
#include <math.h>
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Description

The `lround` and `llround` functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

The `lround` and `llround` functions return the rounded integer value.

21.9.8 The trunc functions

Synopsis

```
#include <math.h>
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

Description

The `trunc` functions round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

The `trunc` functions return the truncated integer value.

21.10 Remainder functions

21.10.1 The fmod functions

Synopsis

```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

Description

The `fmod` functions compute the floating-point remainder of `x/y`.

Returns

The `fmod` functions return the value `x-ny`, for some integer `n` such that, if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`. If `y` is zero, whether a domain error occurs or the `fmod` functions return zero is implementationdefined.

21.10.2 The remainder functions

Synopsis

```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

Description

The `remainder` functions compute the remainder $xREMy$ required by IEC 60559.[13]

Returns

The `remainder` functions return $xREMy$. If y is zero, whether a domain error occurs or the functions return zero is implementation defined.

21.10.3 The remquo functions

Synopsis

```
#include <math.h>
double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

Description

The `remquo` functions compute the same remainder as the remainder`remainder` functions. In the object pointed to by `quo` they store a value whose sign is the sign of `x/y` and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of `x/y`, where `n` is an implementation-defined integer greater than or equal to 3.

Returns

[13] "When $y \neq 0$, the remainder r $= xREMy$ is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$ then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x".This definition is applicable for all implementations.

The remquo functions return $xREMy$. If y is zero, the value stored in the object pointed to by quo is unspecified and whether a domain error occurs or the functions return zero is implementation defined.

21.11 Manipulation functions

21.11.1 The copysign function

Synopsis

```
#include <math.h>
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

Description

The copysign functions produce a value with the magnitude of x and the sign of y. They produce a NaN (with the sign of y) if x is a NaN. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, the copysign functions regard the sign of zero as positive.

Returns

The copysign functions return a value with the magnitude of x and the sign of y.

21.11.2 The nan functions

Synopsis

```
#include <math.h>
double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

Description

The call nan("*n-char-sequence*") is equivalent to strtod("NAN(*n-char-sequence*)", (char**) NULL); the call nan("") is equivalent to strtod("NAN()", (char**) NULL). If tagp does not point to an n-char sequence or an empty string, the call is equivalent to strtod("NAN", (char**) NULL). Calls to nanf and nanl are equivalent to the corresponding calls to strtof and strtold.

Returns

The nan functions return a quiet NaN, if available, with content indicated through tagp. If the implementation does not support quiet NaNs, the functions return zero.

Forward references: the strtod, strtof and strtold functions (The strtod, strtof and strtold functions).

21.11.3 The nextafter functions

Synopsis

```
#include <math.h>
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

Description

The nextafter functions determine the next representable value, in the type of the function, after `x` in the direction of `y`, where `x` and `y` are first converted to the type of the function.[14] The nextafter functions return `y` if `x` equals `y`. A range error may occur if the magnitude of `x` is the largest finite value representable in the type and the result is infinite or not representable in the type.

Returns

The `nextafter` functions return the next representable value in the specified format after `x` in the direction of `y`.

21.11.4 The nexttoward functions

Synopsis

```
#include <math.h>
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

Description

The `nexttoward` functions are equivalent to the `nextafter` functions except that the second parameter has type `long double` and the functions return `y` converted to the type of the function if `x` equals `y`.[15]

21.12 Maximum, minimum and positive difference functions

21.12.1 The fdim funcitons

Synopsis

```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

[14] The argument values are converted to the type of the function, even by a macro implementation of the function.

[15] The result of the `nexttoward` functions is determined in the type of the function, without loss of range or precision in a floating second argument.

Description
The `fdim` functions determine the positive difference between their arguments:

$$\begin{cases} x - y & \text{if } x \geq y \\ +0 & \text{if } x \leq y \end{cases}$$

A range error may occur.
Returns
The `fdim` functions return the positive difference value.

21.12.2 The fmax function

Synopsis

```
#include <math.h>
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

Description
The `fmax` functions determine the maximum numeric value of their arguments.[16]
Returns
The `fmax` functions return the maximum numeric value of their arguments.

21.12.3 The fmin functions

Synopsis

```
#include <math.h>
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

Description
The `fmin` functions determine the minimum numeric value of their arguments.[17]
Returns
The `fmin` functions return the minimum numeric value of their arguments.

21.13 Floating multiply-add

21.13.1 The fma functions

Synopsis

[16] NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the `fmax` functions choose the numeric value. See F.9.9.2.
[17] The `fmin` functions are analogous to the `fmax` functions in their treatment of NaNs.

```
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

Description

The `fma` functions compute `(x * y) + z`, rounded as one ternary operation: they compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of `FLT_ROUNDS`. A range error may occur.

Returns

The `fma` functions return `(x * y) + z`, rounded as one ternary operation.

21.14 Comparison macros

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values exactly one of the relationships -- *less, greater* and *equal* -- is true. Relational operators may raise the ”invalid” floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true.[18] The following subclauses provide macros that are quiet (non floating-point exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the ”invalid” floating-point exception. In the synopses in this subclause, *real-floating* indicates that the argument shall be an expression of real floating type.

21.14.1 The isgreater macro

Synopsis

```
#include <math.h>
int isgreater(real-floating x, real-floating y);
```

Description

The `isgreater` macro determines whether its first argument is greater than its second argument. The value of `isgreater(x, y)` is always equal to `(x) > (y)`; however, unlike `(x) > (y)`, `isgreater(x, y)` does not raise the ”invalid” floating-point exception when `x` and `y` are unordered.

Returns

The `isgreater` macro returns the value of `(x) > (y)`.

21.14.2 The isgreaterequal macro

Synopsis

[18] IEC 60559 requires that the built-in relational operators raise the ”invalid” floating-point exception if the operands compare unordered, as an error indicator for programs written without consideration of NaNs; the result in these cases is false.

```
#include <math.h>
int isgreaterequal(real-floating x, real-floating y);
```

Description

The `isgreaterequal` macro determines whether its first argument is greater than or equal to its second argument. The value of `isgreaterequal(x, y)` is always equal to `(x) >=(y)`; however, unlike `(x) >=(y), isgreaterequal(x, y)` does not raise the ”invalid” floating-point exception when `x` and `y` are unordered.

Returns

The `isgreaterequal` macro returns the value of `(x) >=(y)`.

21.14.3 The isless macro

Synopsis

```
#include <math.h>
int isless(real-floating x, real-floating y);
```

Description

The `isless` macro determines whether its first argument is less than its second argument. The value of `isless(x, y)` is always equal to `(x) <(y)`; however, unlike `(x) <(y), isless(x, y)` does not raise the ”invalid” floating-point exception when `x` and `y` are unordered.

Returns

The `isless` macro returns the value of `(x) <(y)`.

21.14.4 The islessequal macro

Synopsis

```
#include <math.h>
int islessequal(real-floating x, real-floating y);
```

Description

The `islessequal` macro determines whether its first argument is less than or equal to its second argument. The value of `islessequal(x, y)` is always equal to `(x) <=(y)`; however, unlike `(x) <isunordered` macro returns 1 if its arguments are unordered and 0 otherwise.

Chapter 22

Nonlocal jumps <setjmp.h>

The header `<setjmp.h>` defines the macro `setjmp`, and declares one function and one type, for bypassing the normal function call and return discipline.[1]

The type declared is

jmp_buf

which is an array type suitable for holding the information needed to restore a calling environment. The environment of a call to the setjmp macro consists of information sufficient for a call to the longjmp function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point status flags, of open files, or of any other component of the abstract machine.

It is unspecified whether `setjmp` is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name setjmp, the behavior is undefined.

22.1 Save calling environment

22.1.1 The setjmp macro

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

The `setjmp` macro saves its calling environment in its `jmp_buf` argument for later use by the `longjmp` function.

Returns

[1] These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.

If the return is from a direct invocation, the `setjmp` macro returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` macro returns a nonzero value.

Environmental limits

An invocation of the `setjmp` macro shall appear only in one of the following contexts:

- the entire controlling expression of a selection or iteration statement;
- one operand of a relational or equality operator with the other operand an integer constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement;
- the operand of a unary ! operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or
- the entire expression of an expression statement (possibly cast to `void`).

If the invocation appears in any other context, the behavior is undefined.

22.2 Restore calling environment

22.2.1 The longjmp function

Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Description

The `longjmp` function restores the environment saved by the most recent invocation of the `setjmp` macro in the same invocation of the program with the corresponding `jmp_buf` argument. If there has been no such invocation, or if the function containing the invocation of the setjmp macro has terminated execution[2] in the interim, or if the invocation of the `setjmp` macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

All accessible objects have values, and all other components of the abstract machine[3] have state, as of the time the `longjmp` function was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding setjmp macro that do not have volatile-qualified type and have been changed between the `setjmp` invocation and `longjmp` call are indeterminate.

Returns

After `longjmp` is completed, program execution continues as if the corresponding invocation of the `setjmp` macro had just returned the value specified by `val`. The `longjmp` function cannot cause the `setjmp` macro to return the value 0; if val is 0, the `setjmp` macro returns the value 1.

[2] For example, by executing a `return` statement or because another `longjmp` call has caused a transfer to a `setjmp` invocation in a function earlier in the set of nested calls.

[3] This includes, but is not limited to, the floating-point status flags and the state of open files.

EXAMPLE The `longjmp` function that returns control back to the point of the `setjmp` invocation might cause memory associated with a variable length array object to be squandered.

```
#include <1setjmp.h>

jmp_buf buf;
void g(int n);
void h(int n);
int n = 6;

void f(void)
{
  int x[n]; // valid: f is not terminated
  setjmp(buf);
  g(n);
}

void g(int n)
{
  int a[n]; // a may remain allocated
  h(n);
}

void h(int n)
{
  int b[n]; // b may remain allocated
  longjmp(buf, 2); // might cause memory loss
}
```

Chapter 23

Signal handling <signal.h>

The header <signal.h> declares a type and two functions and defines several macros, for handling various signals (conditions that may be reported during program execution).

The type defined is

sig_atomic_t

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

The macros defiend are

SIG_DFL

SIG_ERR

SIG_IGN

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the signal function, and whose values compare unequal to the address of any declarable function; and the following, which expand to positive integer constant expressions with type int and distinct values that are the signal numbers, each corresponding to the specified condition:

SIGABRT abnormal termination, such as is initiated by the abort function

SIGFPE an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

SIGILL detection of an invalid function image, such as an invalid instruction

SIGINT receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

An implementation need not generate any of these signals, except as a result of explicit calls to the `raise` function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters SIG and an uppercase letter or with `SIG_` and an uppercase letter,[1] may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

23.1 Specify signal handling

23.1.1 The signal function

Synopsis

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Description

The `signal` function chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of func is `SIG_DFL`, default handling for that signal will occur. If the value of func is `SIG_IGN`, the signal will be ignored. Otherwise, func shall point to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library), is called a signal handler.

When a signal occurs and `func` points to a function, it is implementation-defined whether the equivalent of `signal(sig, SIG_DFL);` is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed; in the case of `SIGILL`, the implementation may alternatively define that no action is taken. Then the equivalent of `(*func)(sig);` is executed. If and when the function returns, if the value of sig is `SIGFPE, SIGILL, SIGSEGV` or any other implementation-defined value corresponding to a computational exception, the behavior is undefined; otherwise the program will resume execution at the point it was interrupted.

If the signal occurs as the result of calling the `abort` or `raise` function, the signal handler shall not call the raise function.

If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler refers to any object with static storage duration other than by assigning a value to an object declared as `volatile sig_atomic_t` or the signal handler calls any function in the standard library other than the abort function, the `_Exit` function or the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the `signal` function results in a `SIG_ERR` return, the value of `errno` is indeterminate.[2]

At program startup, the equivalent of

[1] The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

[2] If any signal is generated by an asynchronous signal handler, the behavior is undefined.

```
signal(sig, SIG_IGN);
```

may be executed for some signals selected in an implementation-defined manner; the equivalent of

```
signal(sig, SIG_DFL);
```

is executed for all other signals defined by the implementation.

The implementation shall behave as if no library function calls the `signal` function.

Returns

If the request can be honored, the `signal` function returns the value of `func` for the most recent successful call to `signal` for the specified signal `sig`. Otherwise, a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

Forward references: the `abort` function (The abort function), the `exit` function (The exit function), the `_Exit` function (The _Exit function).

23.2 Send signal

23.2.1 The raise function

Synopsis

```
#include <signal.h>
int raise(int sig);
```

Description

The `raise` function carries out the actions described in The signal function for the signal `sig`. If a signal handler is called, the `raise` function shall not return until after the signal handler does.

Returns

The `raise` function returns zero if successful, nonzero if unsuccessful.

Chapter 24

Variable arguments <stdarg.h>

The header `<stdarg.h>` declares a type and defines four macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

A function may be called with a variable number of arguments of varying types. As described in Function definitions, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated `parmN` in this description.

The type declared is

va_list

which is an object type suitable for holding information needed by the macros `va_start, va_arg, va_end` and `va_copy`. If access to the varying arguments is desired, the called function shall declare an object (generally referred to as ap in this subclause) having type `va_list`. The object `ap` may be passed as an argument to another function; if that function invokes the `va_arg` macro with parameter `ap`, the value of `ap` in the calling function is indeterminate and shall be passed to the `va_end` macro prior to any further reference to `ap`.[1]

24.1 Variable argument list access macros

The `va_start` and `va_arg` macros described in this subclause shall be implemented as macros, not functions. It is unspecified whether `va_copy` and `va_end` are macros or identifiers declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the same name, the behavior is undefined. Each invocation of the `va_start` and `va_copy` macros shall be matched by a corresponding invocation of the `va_end` macro in the same function.

[1] It is permitted to create a pointer to a `va_list` and pass that pointer to another function, in which case the original function may make further use of the original list after the other function returns.

24.1.1 The va_arg macro

Synopsis

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

The `va_arg` macro expands to an expression that has the specified type and the value of the next argument in the call. The parameter `ap` shall have been initialized by the `va_start` or `va_copy` macro (without an intervening invocation of the `va_end` macro for the same `ap`). Each invocation of the `va_arg` macro modifies `ap` so that the values of successive arguments are returned in turn. The parameter type shall be a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a `*` to *type*. If there is no actual next argument, or if type is not compatible with the *type* of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined, except for the following cases:

- one type is a signed integer type, the other type is the corresponding unsigned integer type, and the value is representable in both types;
- one type is pointer to void and the other is a pointer to a character type.

Returns

The first invocation of the `va_arg` macro after that of the `va_start` macro returns the value of the argument after that specified by `parmN`. Successive invocations return the values of the remaining arguments in succession.

24.1.2 The va_copy macro

Synopsis

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src);
```

Description

The `va_copy` macro initializes dest as a copy of `src`, as if the `va_start` macro had been applied to `dest` followed by the same sequence of uses of the `va_arg` macro as had previously been used to reach the present state of `src`. Neither the `va_copy` nor `va_start` macro shall be invoked to reinitialize dest without an intervening invocation of the `va_end` macro for the same `dest`.

Returns

The `va_copy` macro returns no value.

24.1.3 The va_end macro

Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

The va_end macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of the va_start macro, or the function containing the expansion of the va_copy macro, that initialized the va_list ap. The va_end macro may modify ap so that it is no longer usable (without being reinitialized by the va_start or va_copy macro). If there is no corresponding invocation of the va_start or va_copy macro or if the va_end macro is not invoked before the return, the behavior is undefined.

Returns

The va_end macro returns no value.

24.1.4 The va_start macro

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Description

The va_start macro shall be invoked before any access to the unnamed arguments.

The va_start macro initializes ap for subsequent use by the va_arg and va_end macros. Neither the va_start nor va_copy macro shall be invoked to reinitialize ap without an intervening invocation of the va_end macro for the same ap.

The parameter parmN is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). If the parameter parmN is declared with the register storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

Returns

The va_start macro returns no value.

EXAMPLE 1 The function f1 gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments), then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.

```
#include <stdarg.h>

#define MAXARGS 31

void f1(int n_ptrs, ...)
{
  va_list ap;
  char *array[MAXARGS];
  int ptr_no = 0;
```

```
    if (n_ptrs > MAXARGS)
      n_ptrs = MAXARGS;

    va_start(ap, n_ptrs);
    while (ptr_no < n_ptrs)
      array[ptr_no++] = va_arg(ap, char *);

    va_end(ap);
    f2(n_ptrs, array);
}
```

Each call to `f1` is required to have visible the definition of the function or a declaration such as

```
void f1(int, ...);
```

EXAMPLE 2 The function `f3` is similar, but saves the status of the variable argument list after the indicated number of arguments; after `f2` has been called once with the whole list, the trailing part of the list is gathered again and passed to function `f4`.

```
#include <stdarg.h>

#define MAXARGS 31

void f3(int n_ptrs, int f4_after, ...)
{
  va_list ap, ap_save;
  char *array[MAXARGS];
  int ptr_no = 0;

  if (n_ptrs > MAXARGS)
    n_ptrs = MAXARGS;

  va_start(ap, f4_after);

  while (ptr_no < n_ptrs) {
    array[ptr_no++] = va_arg(ap, char *);
    if (ptr_no == f4_after)
      va_copy(ap_save, ap);
  }
  va_end(ap);
  f2(n_ptrs, array);
  // Now process the saved copy.
  n_ptrs -= f4_after;
  ptr_no = 0;
  while (ptr_no < n_ptrs)
    array[ptr_no++] = va_arg(ap_save, char *);
  va_end(ap_save);
  f4(n_ptrs, array);
```

```
}
```

Chapter 25

Boolean types and values <stdbool.h>

The header `<stdbool.h>` defines four macros. The macro

`bool`

expands to `_Bool`

The remaining three macros are suitable for use in `#if` preprocessing directives. They are

`true`

which expands to the integer constant 1,

`false`

which expands to the integer constant 0, and

`__bool_true_false_are_defined`

which expands to the integer constant 1.

Notwithstanding the provisions of macro Replacement, a program may undefine and perhaps then redefine the macros `bool, true` and `false`.

Chapter 26

Common definitions <stddef.h>

The following types and macros are defined in the standard header `<stddef.h>`. Some are also defined in other headers, as noted in their respective subclauses.

The tyeps are

`ptrdiff_t`

which is the signed integer type of the result of subtracting two pointers;

`size_t`

which is the unsigned integer type of the result of the sizeof operator; and

`wchar_t`

which is an integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero.

The macros are

`NULL`

which expands to an implementation-defined null pointer constant; and

`offsetof` *(type, member-designator)*

which expands to an integer constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by member-designator), from the beginning of its structure (designated by type). The type and member designator shall be such that given

```
static type t;
```

then the expression & (t.*member-designator*) evaluates to an address constant. (If the specified member is a bit-field, the behavior is undefined.)

Recommended practice

The types used for `size_t` and `ptrdiff_t` should not have an integer conversion rank greater than that of `signed long int` unless the implementation supports objects large enough to make this necessary.

Chapter 27

Integer types <stdint.h>

The header <stdint.h> declares sets of integer types having specified widths, and defines corresponding sets of macros. It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.

Types are defined in the following categories:

- integer types having certain exact widths;
- integer types having at least certain specified widths;
- fastest integer types having at least certain specified widths;
- integer types wide enough to hold pointers to objects;
- integer types having greatest width.

(Some of these types may denote the same type.)

Corresponding macros specify limits of the declared types and construct suitable constants.

For each type described herein that the implementation provides,[1] `<stdint.h>` shall declare that `typedef` name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, `<stdint.h>` shall not declare that `typedef` name nor shall it define the associated macros. An implementation shall provide those types described as ”required”, but need not provide any of the others (described as ”optional”).

27.1 Integer types

When typedef names differing only in the absence or presence of the initial `u` are defined, they shall denote corresponding signed and unsigned types as described in Types; an implementation providing one of these corresponding types shall also provide the other.

In the following descriptions, the symbol N represents an unsigned decimal integer with no leading zeros (e.g., 8 or 24, but not 04 or 048).

[1] Some of these types may denote implementation-defined extended integer types.

27.1.1 Exact-width integer types

The typedef name `intN_t` designates a signed integer type with width `N`, no padding bits and a two's complement representation. Thus, `int8_t` denotes a signed integer type with a width of exactly 8 bits.

The typedef name `uintN_t` designates an unsigned integer type with width `N`. Thus, `uint24_t` denotes an unsigned integer type with a width of exactly 24 bits.

These types are optional. However, if an implementation provides integer types with widths of 8, 16, 32 or 64 bits, no padding bits, and (for the signed types) that have a two's complement representation, it shall define the corresponding typedef names.

27.1.2 Minimum-width integer types

The typedef name `int_leastN_t` designates a signed integer type with a width of at least `N`, such that no signed integer type with lesser size has at least the specified width. Thus, `int_least32_t` denotes a signed integer type with a width of at least 32 bits.

The typedef name `uint_leastN_t` designates an unsigned integer type with a width of at least `N`, such that no unsigned integer type with lesser size has at least the specified width. Thus, xcode`uint_least16_t` denotes an unsigned integer type with a width of at least 16 bits.

The following types are required:

```
int_least8_t uint_least8_t
int_least16_t uint_least16_t
int_least32_t uint_least32_t
int_least64_t uint_least64_t
```

All other types of this form are optional.

27.1.3 Fastest minimum-width integer types

Each of the following types designates an integer type that is usually fastest[2] to operate with among all integer types that have at least the specified width.

The typedef name `int_fastN_t` designates the fastest signed integer type with a width of at least `N`. The typedef name `uint_fastN_t` designates the fastest unsigned integer type with a width of at least `N`.

The following types are required:

```
int_fast8_t uint_fast8_t
int_fast16_t uint_fast16_t
int_fast32_t uint_fast32_t
int_fast64_t uint_fast64_t
```

All other types of this form are optional.

[2] The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.

27.1.4 Integer types capable of holding object pointers

The following type designates a signed integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer:

```
intptr_t
```

The following type designates an unsigned integer type with the property that any valid pointer to void can be converted to this type, then converted back to pointer to void, and the result will compare equal to the original pointer:

```
uintptr_t
```

These types are optional.

27.1.5 Greatest-width integer types

The following type designates a signed integer type capable of representing any value of any signed integer type:

```
intmax_t
```

The following type designates an unsigned integer type capable of representing any value of any unsigned integer type:

```
uintmax_t
```

These types are required.

27.2 Limits of specific-width integer types

The following object-like macros[3] specify the minimum and maximum limits of the types declared in `<stdint.h>`. Each macro name corresponds to a similar type name in Integer types.

Each instance of any defined macro shall be replaced by a constant expression suitable for use in `#if` preprocessing directives, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign, except where stated to be exactly the given value.

[3] C++ implementations should define these macros only when `__STDC_LIMIT_MACROS` is defined before `<stdint.h>` is included.

27.2.1 Limits of exact-width integer types

- minimum values of exact-width signed integer types
  `INTN_MIN` exactly $-(2^{N-1})$
- maximum values of exact-width signed integer types
  `INTN_MAX` exactly $2^{N-1} - 1$
- maximum values of exact-width unsigned integer types
  `UINT_MAX` exactly $2^{N} - 1$

27.2.2 Limits of minimum-width integer types

- minimum values of minimum-width signed integer types
  `INT_LEASTN_MIN` $-(2^{N-1} - 1)$
- maximum values of minimum-width signed integer types
  `INT_LEASTN_MAX` $2^{N-1} - 1$
- maximum values of minimum-width unsigned integer types
  `UINT_LEASTN_MAX` $2^{N} - 1$

27.2.3 Limits of fastest minimum-width integer types

- minimum values of fastest minimum-width signed integer types
  `INT_FASTN_MIN` $-(2^{N-1} - 1)$
- maximum values of fastest minimum-width signed integer types
  `INT_FASTN_MAX` $2^{N-1} - 1$
- maximum values of fastest minimum-width unsigned integer types
  `UINT_FASTN_MAX` $2^{N} - 1$

27.2.4 Limits of integer types capable of holding object pointers

- minimum value of pointer-holding signed integer type
  `INTPTR_MIN` $-(2^{15} - 1)$
- maximum value of pointer-holding signed integer type
  `INTPTR_MAX` $2^{15} - 1$
- maximum value of pointer-holding unsigned integer type
  `UINTPTR_MAX` $2^{16} - 1$

27.2.5 Limits of greatest-width integer types

- minimum value of greatest-width signed integer type
  `INTMAX_MIN` $-(2^{63} - 1)$
- maximum value of greatest-width signed integer type
  `INTMAX_MAX` $2^{63} - 1$
- maximum value of greatest-width unsigned integer type
  `UINTMAX_MAX` $2^{64} - 1$

27.3 Limits of other integer types

The following object-like macros[4] specify the minimum and maximum limits of integer types corresponding to types defined in other standard headers.

Each instance of these macros shall be replaced by a constant expression suitable for use in `#if` preprocessing directives, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign. An implementation shall define only the macros corresponding to those typedef names it actually provides.[5]

- limits of `ptrdiff_t`
  `PTRDIFF_MIN` -65535
  `PTRDIFF_MAX` +65535
- limits of `sig_atomic_t`
  `SIG_ATOMIC_MIN` see below
  `SIG_ATOMIC_MAX` see below
- limits of `size_t`
  `SIZE_MAX` 65535
- limits of `wchar_t`
  `WCHAR_MIN` see below
  `WCHAR_MAX` see below
- limits of `wint_t`
  `WINT_MIN` see below
  `WINT_MAX` see below

[4] C++ implementations should define these macros only when is defined before `<stdint.h>` is included.
[5] A freestanding implementation need not provide all of these types.

If sig_atomic_t (see Signal handling <signal.h>) is defined as a signed integer type, the value of SIG_ATOMIC_MIN shall be no greater than -127 and the value of SIG_ATOMIC_MAX shall be no less than 127; otherwise, sig_atomic_t is defined as an unsigned integer type, and the value of SIG_ATOMIC_MIN shall be 0 and the value of SIG_ATOMIC_MAX shall be no less than 255.

If wchar_t (see Common definitions <stddef.h>) is defined as a signed integer type, the value of WCHAR_MIN shall be no greater than -127 and the value of WCHAR_MAX shall be no less than 127; otherwise, wchar_t is defined as an unsigned integer type, and the value of WCHAR_MIN shall be 0 and the value of WCHAR_MAX shall be no less than 255.

If wint_t (see Extended multibyte and wide character utilities <wchar.h>) is defined as a signed integer type, the value of WINT_MIN shall be no greater than -32767 and the value of WINT_MAX shall be no less than 32767; otherwise, wint_t is defined as an unsigned integer type, and the value of WINT_MIN shall be 0 and the value of WINT_MAX shall be no less than 65535.

27.4 Macros for integer constants

The following function-like macros[6] expand to integer constants suitable for initializing objects that have integer types corresponding to types defined in <stdint.h>. Each macro name corresponds to a similar type name in Minimum-width integer types or Greatest-width integer types.

The argument in any instance of these macros shall be a decimal, octal, or hexadecimal constant (as defined in Integer Constants) with a value that does not exceed the limits for the corresponding type.

Each invocation of one of these macros shall expand to an integer constant expression suitable for use in #if preprocessing directives. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument.

27.4.1 Macros for minimum-width integer constants

The macro INTN_C (value) shall expand to an integer constant expression corresponding to the type int_leastN_t. The macro UINTN_C (value) shall expand to an integer constant expression corresponding to the type uint_leastN_t. For example, if uint_least64_t is a name for the type unsigned long long int, then UINT64_C(0x123) might expand to the integer constant 0x123ULL.

27.4.2 Macros for greatest-width integer constants

The following macro expands to an integer constant expression having the value specified by its argument and the type intmax_t:

INTMAX_C (value)

[6] C++ implementations should define these macros only when __STDC_CONSTANT_MACROS is defined before <stdint.h> is included.

The following macro expands to an integer constant expression having the value specified by its argument and the type `uintmax_t`:

`UINTMAX_C` (value)

Chapter 28

Input/Output <stdio.h>

28.1 Introduction

The header `<stdio.h>` declares three types, several macros, and many functions for performing input and output.

The types declared are `size_t` (described in Common definitions <stddef.h>);

`FILE`

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an error indicator that records whether a read/write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached; and

`fpos_t`

which is an object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.

The macros are NULL (described in Common definitions <stddef.h>);

`_IOFBF`

`_IOLBF`

`_IONBF`

which expand to integer constant expressions with distinct values, suitable for use as the third argument to the `setvbuf` function;

`BUFSIZ`

which expands to an integer constant expression that is the size of the buffer used by the `setbuf` function;

`EOF`

which expands to an integer constant expression, with type int and a negative value, that is returned by several functions to indicate end-of-file, that is, no more input from a stream;

`FOPEN_MAX`

which expands to an integer constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

`FILENAME_MAX`

which expands to an integer constant expression that is the size needed for an array of char large enough to hold the longest file name string that the implementation guarantees can be opened;[1]

`L_tmpnam`

which expands to an integer constant expression that is the size needed for an array of char large enough to hold a temporary file name string generated by the `tmpnam` function;

`stderr`

`stdin`

`stdout`

which are expressions of type ”pointer to `FILE`” that point to the `FILE` objects associated, respectively, with the standard error, input, and output streams.

The header `<wchar.h>` declares a number of functions useful for wide character input and output. The wide character input/output functions described in that subclause provide operations analogous to most of those described here, except that the fundamental units internal to the program are wide characters. The external representation (in the file) is a sequence of ”generalized” multibyte characters, as described further in Files.

The input/output functions are given the following collective terms:

- The *wide character input functions* - those functions described in Extended multibyte and wide character utilities <wchar.h> that perform input into wide characters and wide strings: `fget wc, fgetws, getwc, getwchar, fwscanf, wscanf, vfwscanf` and `vwscanf`.
- The *wide character output functions* - those functions described in Extended multibyte and wide character utilities <wchar.h> that perform output from wide characters and wide strings: `fputwc, fputws, putwc, putwchar, fwprintf, wprintf, vfwprintf` and `vwprintf`.
- The *wide character input/output functions* the union of the `ungetwc` function, the wide character input functions, and the wide character output functions.

[1] If the implementation imposes no practical limit on the length of file name strings, the value of `FILENAME_MAX` should instead be the recommended size of an array intended to hold a file name string. Of course, file name string contents are subject to other system-specific constraints; therefore all possible strings of length `FILENAME_MAX` cannot be expected to be opened successfully.

- The *byte input/output functions* - those functions described in this subclause that perform input/output: `fgetc, fgets, fprintf, fputc, fputs, fread, fscanf, fwrite, getc, getchar, gets, perror, printf, putc, putchar, puts, scanf, ungetc, vfprintf, vfscanf, vprintf` and `vscanf`.

28.2 Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for text streams and for binary streams.[2]

A text stream is an ordered sequence of characters composed into lines, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a oneto- one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printing characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, hav e an implementation-defined number of null characters appended to the end of the stream.

Each stream has an orientation. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide character input/output function has been applied to a stream without orientation, the stream becomes a *wide-oriented* stream. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream becomes a byte-oriented stream. Only a call to the `freopen` function or the fwide function can otherwise alter the orientation of a stream. (A successful call to `freopen` removes any orientation.)[3]

Byte input/output functions shall not be applied to a wide-oriented stream and wide character input/output functions shall not be applied to a byte-oriented stream. The remaining stream operations do not affect, and are not affected by, a stream's orientation, except for the following additional restrictions:

- Binary wide-oriented streams have the file-positioning restrictions ascribed to both text and binary streams.

[2] An implementation need not distinguish between text streams and binary streams. In such an implementation, there need be no new-line characters in a text stream nor any limit to the length of a line.

[3] The three predefined streams `stdin, stdout` and `stderr` are unoriented at program startup.

- For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide character output function can overwrite a partial multibyte character; any file contents beyond the byte(s) written are henceforth indeterminate.

Each wide-oriented stream has an associated `mbstate_t` object that stores the current parse state of the stream. A successful call to fgetpos stores a representation of the value of this mbstate_t object as part of the value of the `fpos_t` object. A later successful call to fsetpos using the same stored `fpos_t` value restores the value of the associated `mbstate_t` object as well as the position within the controlled stream.

Environmental limits

An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro `BUFSIZ` shall be at least 256.

28.3 Files

A stream is associated with an external file (which may be a physical device) by opening a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as opposed to a terminal), then a file position indicator associated with the stream is positioned at the start (character number zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file *position* indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file.

Binary files are not truncated, except as defined in The fopen function. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

When a stream is *unbuffered*, characters are intended to appear from the source or at the destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the setbuf and setvbuf functions.

A file may be disassociated from a controlling stream by `closing` the file. Output streams are flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a `FILE` object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the `main` function returns to its original caller, or if the `exit` function is called, all open files are closed (hence all output streams are

flushed) before program termination. Other paths to program termination, such as calling the abort function, need not close all files properly.

The address of the `FILE` object used to control a stream may be significant; a copy of a `FILE` object need not serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly - *standard input* (for reading conventional input), *standard output* (for writing conventional output) and standard error (for writing diagnostic output). As initially opened, the *y* stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of multibyte characters, generalized as follows:

- Multibyte encodings within files may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program).
- A file need not begin nor end in the initial shift state.[4]

Moreover, the encodings used for multibyte characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

The wide character input functions read multibyte characters from the stream and convert them to wide characters as if they were read by successive calls to the `fgetwc` function. Each conversion occurs as if by a call to the `mbrtowc` function, with the conversion state described by the stream's own `mbstate_t` object. The byte input functions read characters from the stream as if by successive calls to the `fgetc` function.

The wide character output functions convert wide characters to multibyte characters and write them to the stream as if they were written by successive calls to the `fputwc` function. Each conversion occurs as if by a call to the wcrtomb function, with the conversion state described by the stream's own `mbstate_t` object. The byte output functions write characters to the stream as if by successive calls to the `fputc` function.

In some cases, some of the byte input/output functions also perform conversions between multibyte characters and wide characters. These conversions also occur as if by calls to the `mbrtowc` and `wcrtomb` functions.

An *encoding error* occurs if the character sequence presented to the underlying `mbrtowc` function does not form a valid (generalized) multibyte character, or if the code value passed to the underlying `wcrtomb` does not correspond to a valid (generalized) multibyte character. The wide character input/output functions and the byte input/output functions store the value of the macro `EILSEQ` in errno if and only if an encoding error occurs.

Enviromental limits

[4] Setting the file position indicator to end-of-file, as with `fseek(file, 0, SEEK_END)`, has undefined behavior for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state.

The value of FOPEN_MAX shall be at least eight, including the three standard text streams.

Forward references: the exit function (The exit function), the fgetc function (The fgetc function), the fopen function (The fopen function), the fputc function (The fputc function), the setbuf function (The setbuf function), the setvbuf function (The setvbuf function), the fgetwc function (The fgetwc function), the fputwc function (The fputwc function), conversion state (Extended multibyte/wide character conversion utilities), the mbrtowc function (The mbrtowc function), the wcrtomb function (The wcrtomb function)

28.4 Operations on files

28.4.1 The remove function

Synopsis

```
#include <stdio.h>
int remove(const char *filename);
```

Description

The remove function causes the file whose name is the string pointed to by filename to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the remove function is implementation-defined.

Returns

The remove function returns zero if the operation succeeds, nonzero if it fails.

28.4.2 The rename function

Synopsis

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description

The rename function causes the file whose name is the string pointed to by old to be henceforth known by the name given by the string pointed to by new. The file named old is no longer accessible by that name. If a file named by the string pointed to by new exists prior to the call to the rename function, the behavior is implementation-defined.

Returns

The rename function returns zero if the operation succeeds, nonzero if it fails,[5] in which case if the file existed previously it is still known by its original name.

[5] Among the reasons the implementation may cause the rename function to fail are that the file is open or that it is necessary to copy its contents to effectuate its renaming.

28.4.3 The tmpfile function

Synopsis

```
#include <stdio.h>
FILE *tmpfile(void);
```

Description

The `tmpfile` function creates a temporary binary file that is different from any other existing file and that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with ”wb+” mode.

Recommended Practice

It should be possible to open at least `TMP_MAX` temporary files during the lifetime of the program (this limit may be shared with `tmpnam`) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files (`FOPEN_MAX`).

Returns

The `tmpfile` function returns a pointer to the stream of the file that it created. If the file cannot be created, the `tmpfile` function returns a null pointer.

`Forward references:` the `fopen` function (The fopen function).

28.4.4 The tmpnam function

Synopsis

```
#include <stdio.h>
char *tmpnam(char *s);
```

Description

The `tmpnam` function generates a string that is a valid file name and that is not the same as the name of an existing file.[6] The function is potentially capable of generating `TMP_MAX` different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

The `tmpnam` function generates a different string each time it is called.

The implementation shall behave as if no library function calls the `tmpnam` function.

Returns

If no suitable string can be generated, the `tmpnam` function returns a null pointer. Otherwise, if the argument is a null pointer, the tmpnam function leaves its result in an internal static object and returns a pointer to that object (subsequent calls to the tmpnam function may modify the same object). If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` chars; the `tmpnam` function writes its result in that array and returns the argument as its value.

Environemental limits

The value of the macro `TMP_MAX` shall be at least 25.

[6] Files created using strings generated by the tmpnam function are temporary only in the sense that their names should not collide with those generated by conventional naming rules for the implementation. It is still necessary to use the remove function to remove such files when their use is ended, and before program termination.

28.5 File acces functions

28.5.1 The fclose functions

Synopsis

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

A successful call to the `fclose` function causes the stream pointed to by stream to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. Whether or not the call succeeds, the stream is disassociated from the file and any buffer set by the `setbuf` or `setvbuf` function is disassociated from the stream (and deallocated if it was automatically allocated).

Returns

The `fclose` function returns zero if the stream was successfully closed, or `EOF` if any errors were detected.

28.5.2 The fflush function

Synopsis

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

If stream points to an output stream or an update stream in which the most recent operation was not input, the `fflush` function causes any unwritten data for that stream to be delivered to the host environment to be written to the file; otherwise, the behavior is undefined.

If stream is a null pointer, the fflush function performs this flushing action on all streams for which the behavior is defined above.

Returns

The `fflush` function sets the error indicator for the stream and returns EOF if a write error occurs, otherwise it returns zero.

`Forward references:` the `fopen` function (The fopen function).

28.5.3 The fopen function

Synopsis

```
#include <stdio.h>
FILE *fopen(const char * restrict filename, const char * restrict mode);
```

Description

The `fopen` function opens the file whose name is the string pointed to by filename, and associates a stream with it.

The argument mode points to a string. If the string is one of the following, the file is open in the indicated mode. Otherwise, the behavior is undefined.[7]

`r` open test file for reading

`w` truncate zero length or create text file for writing

`a` append; open or create text file for writing at end-of-file

`rb` open binary file for reading

`wb` truncate to zero length or create binary file for writing

`ab` append; open or create binary file for writing at end-of-file

`r+` open text file for update (reading and writing)

`w+` truncate to zero length or create text file for update

`a+` append; open or create text file for update, writing at end-of-file

`r+b` or `rb+` open binary file for update (reading and writing)

`w+b` or `wb+` truncate to zero length or create binary file for update

`a+b` or `ab+` append; open or create binary file for update, writing at end-of-file

Opening a file with read mode ('`r`' as the first character in the mode argument) fails if the file does not exist or cannot be read.

Opening a file with append mode ('`a`' as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to the fseek function. In some implementations, opening a binary file with append mode ('`b`' as the second or third character in the above list of mode argument values) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

When a file is opened with update mode ('+' as the second or third character in the above list of mode argument values), both input and output may be performed on the associated stream. However, output shall not be directly followed by input without an intervening call to the `fflush` function or to a file positioning function (`fseek, fsetpos` or `rewind`), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns

The `fopen` function returns a pointer to the object controlling the stream. If the open operation fails, fopen returns a null pointer.

Forward references: file positioning functions (File positioning functions).

[7] If the string begins with one of the above sequences, the implementation might choose to ignore the remaining characters, or it might use them to select different kinds of a file (some of which might not conform to the properties in Streams).

28.5.4 The freopen function

Synopsis

```
#include <stdio.h>
FILE *freopen(const char * restrict filename, const char * restrict mode,
              FILE * restrict stream);
```

Description

The `freopen` function opens the file whose name is the string pointed to by filename and associates the stream pointed to by stream with it. The mode argument is used just as in the `fopen` function.[8]

If filename is a null pointer, the `freopen` function attempts to change the mode of the stream to that specified by mode, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

The `freopen` function first attempts to close any file that is associated with the specified stream. Failure to close the file is ignored. The error and end-of-file indicators for the stream are cleared.

Returns

The `freopen` function returns a null pointer if the open operation fails. Otherwise, `freopen` returns the value of stream.

28.5.5 The setbuf function

Synopsis

```
#include <stdio.h>
void setbuf(FILE * restrict stream, char * restrict buf);
```

Description

Except that it returns no value, the `setbuf` function is equivalent to the `setvbuf` function invoked with the values `_IOFBF` for mode and `BUFSIZ` for size, or (if `buf` is a null pointer), with the value `_IONBF` for mode.

Returns

The `setbuf` function returns no value.

Forward references: the `setvbuf` function (The setvbuf function).

28.5.6 The setvbuf function

Synopsis

[8] The primary use of the `freopen` function is to change the file associated with a standard text stream (`stderr, stdin` or `stdout`), as those identifiers need not be modifiable lvalues to which the value returned by the fopen function may be assigned.

```
#include <stdio.h>
int setvbuf(FILE * restrict stream, char * restrict buf, int mode, size_t ↩
    size);
```

Description

The `setvbuf` function may be used only after the stream pointed to by stream has been associated with an open file and before any other operation (other than an unsuccessful call to `setvbuf`) is performed on the stream. The argument mode determines how stream will be buffered, as follows: `_IOFBF` causes input/output to be fully buffered; `_IOLBF` causes input/output to be line buffered; `_IONBF` causes input/output to be unbuffered. If `buf` is not a null pointer, the array it points to may be used instead of a buffer allocated by the `setvbuf` function[9] and the argument size specifies the size of the array; otherwise, size may determine the size of a buffer allocated by the `setvbuf` function. The contents of the array at any time are indeterminate.

Returns

The `setvbuf` function returns zero on success, or nonzero if an invalid value is given for mode or if the request cannot be honored.

28.6 Formatted input/output function

The formatted input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[10]

28.6.1 The fprintf function

Synopsis

```
#include <stdio.h>
int fprintf(FILE * restrict stream, const char * restrict format, ...);
```

Description

The `fprintf` function writes output to the stream pointed to by stream, under control of the string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The `fprintf` function returns when the end of the format string is encountered.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not `%`), which are copied unchanged to the output stream; and conversion pecifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

[9] The buffer has to have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.

[10] The `fprintf` functions perform writes to memory for the `%n` specifier.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a nonnegative decimal integer.[11]
- An optional precision that gives the minimum number of digits to appear for the `d, i, o, u, x` and `X` conversions, the number of digits to appear after the decimal-point character for `a, A, e, E, f` and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of bytes to be written for `s` conversions. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an int argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)[12]

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and + flags both appear, the space flag is ignored.

The result is converted to an "alternative form". For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversion, a nonzero result has `0x` (or `0X`) prefixed to it. For `a, A, e, E, f, F, g` and `G` conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

0 For `d, i, o, u, x, X, a, A, e, E, f, F, g` and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing

[11] Note that 0 is taken as a flag, not as the beginning of a field width.

[12] The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

space padding, except when converting an infinity or NaN. If the 0 and - flags both appear, the 0 flag is ignored. For `d, i, o, u, x` and `X` conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

`hh` Specifies that a following `d, i, o, u, x` or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `signed char` or `unsigned char` before printing); or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.

`h` Specifies that a following `d, i, o, u, x` or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short int` or `unsigned short int` before printing); or that a following `n` conversion specifier applies to a pointer to a `short int` argument.

`l` (ell) Specifies that a following `d, i, o, u, x` or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; that a following `n` conversion specifier applies to a pointer to a `long int` argument; that a following c conversion specifier applies to a `wint_t` argument; that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following `a, A, e, E, f, F, g` or `G` conversion specifier.

`ll` (ell-ell) Specifies that a following `d, i, o, u, x` or `X` conversion specifier applies to a `long long int` or `unsigned long long` int argument; or that a following n conversion specifier applies to a pointer to a `long long int` argument.

`j` Specifies that a following `d, i, o, u, x` or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following `en` conversion specifier applies to a pointer to an `int max_t` argument.

`z` Specifies that a following `d, i, o, u, x` or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to `size_t` argument.

`t` Specifies that a following `d, i, o, u, x` or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument; or that a following n conversion specifier applies to a pointer to a `ptrdiff_t` argument.

`L` Specifies that a following `a, A, e, E, f, F, g` or `G` conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d, i` The `int` argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`o, u, x, X` The `unsigned int` argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (`x` or `X`) in the style `dddd`; the letters `abcdef` are used for `y` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of

zero is no characters.

`f, F` A double argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

A `double` argument representing an infinity is converted in one of the styles **[-]inf** or —- which style is implementation-defined. A double argument representing a NaN is converted in one of the styles **[-]nan** or **[-]nan** (*n-char-sequence*) — which style, and the meaning of any n-char-sequence, is implementation-defined. The `F` conversion specifier produces `INF, INFINITY` or `NAN` instead of `inf, infinity` or `nan` respectively.[13]

`e, E` A double argument representing a floating-point number is converted in the style *[-]d.ddd* e$\pm dd$, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The E conversion specifier produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.

A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`g, G` A double argument representing a floating-point number is converted in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), depending on the value converted and the precision. Let P equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style `E` would have an exponent of X:

- If $P > X \geq -4$. the conversion is with style `f` (or `F`) and precision $P - (X + 1)$
- otherwise, the conversion is with style `e` (or `E`) and precision $P - 1$.

Finally, unless the # flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining.

A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`a, A` A `double` argument representing a floating-point number is converted in the style *[-]0xh.hhhh* $p \pm d$, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character[14] nd the number of hexadecimal digits after it is equal to the precision; if the precision is missing and `FLT_RADIX` is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and `FLT_RADIX` is not a power of 2, then the precision is sufficient to

[13] When applied to infinite and NaN values, the -, + and space flag characters have their usual meaning; the # and 0 flag characters have no effect.

[14] Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.

distinguish[15] values of type double, except that trailing zeros may be omitted; if the precision is zero and the # flag is not specified, no decimal-point character appears. The letters abcdef are used for a conversion and the letters ABCDEF for A conversion. The A conversion specifier produces a number with X and P instead of x and p. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero. A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

c If no l length modifier is present, the int argument is converted to an unsigned char, and the resulting character is written.

If an l length modifier is present, the wint_t argument is converted as if by an ls conversion specification with no precision and an argument that points to the initial element of a two-element array of wchar_t, the first element containing the wint_t argument to the lc conversion specification and the second a null wide character.

s If no l length modifier is present, the argument shall be a pointer to the initial element of an array of character type.[16] Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

If an l length modifier is present, the argument shall be a pointer to the initial element of an array of wchar_t type. Wide characters from the array are converted to multibyte characters (each as if by a call to the wcrtomb function, with the conversion state described by an mbstate_t object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many bytes are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written.[17]

p The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printing characters, in an implementation-defined manner.

n The argument shall be a pointer to signed integer into which is written the number of characters written to the output stream so far by this call to fprintf. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

% A % character is written. No argument is converted. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

[15] The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is FLT_RADIX and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.

[16] No special provisions are made for multibyte characters.

[17] Redundant shift sequences may result if multibyte characters have a state-dependent encoding.

For a and A conversions, if FLT_RADIX is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

Recommended practice

For a and A conversions, if FLT_RADIX is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For e, E, f, F, g and G conversions, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded.[18] If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

Returns

The fprintf function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

Environmental limits

The number of characters that can be produced by any single conversion shall be at least 4095.

EXAMPLE 1 To print a date and time in the form ”Sunday, July 3, 10:02” followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
/* ... */
char *weekday, *month; // pointers to strings
int day, hour, min;
fprintf(stdout, ”%s, %s %d, %.2d:%.2d\n”,
weekday, month, day, hour, min);
fprintf(stdout, ”pi = %.5f\n”, 4 * atan(1.0));
```

EXAMPLE 2 In this example, multibyte characters do not have a state-dependent encoding, and the members of the extended character set that consist of more than one byte each consist of exactly two bytes, the first of which is denoted here by a and the second by an uppercase letter.

Forward references: conversion state (Extended multibyte/wide character conversion utilities), the wcrtomb function (The wcrtomb function).

28.6.2 The fscanf function

Synopsis

```
#include <stdio.h>
```

[18] For binary-to-decimal conversion, the result format’s values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.

```
int fscanf(FILE * restrict stream, const char * restrict format, ...);
```

Description

The `fscanf` function reads input from the stream pointed to by stream, under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters, an ordinary multibyte character (neither `%` nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional decimal integer greater than zero that specifies the maximum field width (in characters).
- An optional *length modifier* that specifies the size of the receiving object.
- A *conversion specifier* character that specifies the type of conversion to be applied.

The `fscanf` function executes each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the occurrence of an encoding error or the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If any of those characters differ from the ones composing the directive, the directive fails and the differing and subsequent characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive fails.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a `[`, `c` or `n` specifier.[19]

An input item is read from the stream, unless the specification includes an n specifier. An input item is defined as the longest sequence of input characters which does not exceed any specified field width and which is, or is a prefix of, a matching input sequence.[20] The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

[19] These white-space characters are not counted against a specified field width.

[20] `fscanf` pushes back at most one input character onto the input stream. Therefore, some sequences that are acceptable to `strtod`, `strtol` etc. are unacceptable to `fscanf`.

Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the object, the behavior is undefined.

The length modifiers and their meanings are:

`hh` Specifies that a following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument with type pointer to `signed char` or `unsigned char`.

`h` Specifies that a following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument with type pointer to `short int` or `unsigned short int`.

`l` (ell) Specifies that a following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument with type pointer to `long int` or `unsigned long int`; that a following `a, A, e, E, f, F, g` or `G` conversion specifier applies to an argument with type pointer to `double`; or that a following `c, s` or `[` conversion specifier applies to an argument with type pointer to `wchar_t`.

`ll` (ell-ell) Specifies that a following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument with type pointer to `long long int` or `unsigned long long int`.

`j` Specifies that a following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument with type pointer to `intmax_t` or `uintmax_t`.

`z` Specifies that a following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument with type pointer to `size_t` or the corresponding signed integer type.

`t` Specifies that a following `d, i, o, u, x, X` or `n` conversion specifier applies to an argument with type pointer to `ptrdiff_t` or the corresponding unsigned integer type.

`L` Specifies that a following `a, A, e, E, f, F, g` or `G` conversion specifier applies to an argument with type pointer to `long double`.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the base argument. The corresponding argument shall be a pointer to signed integer.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 0 for the base argument. The corresponding argument shall be a pointer to signed integer.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 8 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the base argument. The corresponding argument shall be a pointer to unsigned integer.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the base argument. The corre-

sponding argument shall be a pointer to unsigned integer.

`a, e, f, g` Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of the `strtod` function. The corresponding argument shall be a pointer to floating.

`c` Matches a sequence of characters of exactly the number specified by the field width (1 if no field width is present in the directive).[21] If no `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the `mbrtowc` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of `wchar_t` large enough to accept the resulting sequence of wide characters. No null wide character is added.

`s` Matches a sequence of non-white-space characters.[22]

If no `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the `mbrtowc` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which will be added automatically.

`[` Matches a nonempty sequence of characters from a set of expected characters (the *scanset*).[22]

If no `l` length modifier is present, the corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an `l` length modifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the `mbrtowc` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first multibyte character is converted. The corresponding argument shall be a pointer to the initial element of an array of `wchar_t` large enough to accept the sequence and the terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent characters in the format string, up to and including the matching right bracket (`]`). The characters between the brackets (the scanlist) compose the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right bracket character is in the scanlist and

[21] No special provisions are made for multibyte characters in the matching rules used by the `c, s` and `[` conversion specifiers — the extent of the input field is determined on a byte-by-byte basis. The resulting field is nevertheless a sequence of multibyte characters that begins in the initial shift state.

[22] As the functions `vfprintf, vfscanf, vprintf, vscanf, vsnprintf, vsprintf` and `vsscanf` invoke the `va_arg` macro, the value of `arg` after the return is indeterminate.

the next following right bracket character is the matching right bracket that ends the specification; otherwise the first following right bracket character is the one that ends the specification. If a - character is in the scanlist and is not the first, nor the second where the first character is a ^, nor the last character, the behavior is implementation-defined.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the

%p conversion of the fprintf function. The corresponding argument shall be a pointer to a pointer to void. The input item is converted to a pointer value in an implementation-defined manner. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to signed integer into which is to be written the number of characters read from the input stream so far by this call to the fscanf function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the fscanf function. No argument is converted, but one is consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

% Matches a single % character; no conversion or assignment occurs. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers A, E, F, G and X are also valid and behave the same as, respectively, a, e, f, g and x.

Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

Returns

The fscanf function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

EXAMPLE 1 The call:

```
#include <stdio.h>
/* ... */
int n, i; float x; char name[50];
n = fscanf(stdin, ”%d%f%s”, &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to n the value 3, to i the value 25, to x the value 5.432, and to name the sequence thompson\0.

EXAMPLE 2 The call:

```
#include <stdio.h>
/* ... */
int i; float x; char name[50];
```

```
fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign to `i` the value 56 and to `x` the value 789.0, will skip 0123, and will assign to name the sequence `56\0`. The next character read from the input stream will be `a`.

EXAMPLE 3 To accept repeatedly from stdin a quantity, a unit of measure, and an item name:

```
#include <stdio.h>
/* ... */
int count; float quant; char units[21], item[21];
do {
  count = fscanf(stdin, "%f%20s of %20s", &quant, units, item);
  fscanf(stdin,"%*[^\n]");
} while (!feof(stdin) && !ferror(stdin));
```

If the stdin stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS
of
dirt
100ergs of energy
```

the execution of the above example will be analogous to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; // "C" fails to match "o"
count = 0; // "l" fails to match "%f"
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "dirt");
count = 3;
count = 0; // "100e" fails to match "%f"
count = EOF;
```

EXAMPLE 4 In:

```
#include <stdio.h>
/* ... */
int d1, d2, n1, n2, i;
i = sscanf("123", "%d%n%n%d", &d1, &n1, &n2, &d2);
```

the value 123 is assigned to `d1` and the value 3 to `n1`. Because `%n` can never get an input failure the value of 3 is also assigned to `n2`. The value of `d2` is not affected. The value 1 is assigned to `i`.

Forward references: the `strtod, strtof` and `strtold` functions (The strtod, strtof and strtold functions), the `strtol, strtoll, strtoul` and `strtoull` functions (The strtol, strtoll, strtoul and strtoull functions), conversion state (Extended multibyte/wide character conversion utilities), the `wcrtomb` function (The wcrtomb function).

28.6.3 The printf function

Synopsis

```
#include <stdio.h>
int printf(const char * restrict format, ...);
```

Description

The `printf` function is equivalent to `fprintf` with the argument `stdout` interposed before the arguments to `printf`.

Returns

The `printf` function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

28.6.4 The scanf function

Synopsis

```
#include <stdio.h>
int scanf(const char * restrict format, ...);
```

Description

The `scanf` function is equivalent to `fscanf` with the argument `stdin` interposed before the arguments to `scanf`.

Returns

The `scanf` function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Index

D
Dennis
    Ritchie, 2

H
history, 2