GNU C Language Introduction and Reference Manual

Edition 0.0

Richard Stallman
and
Trevis Rothwell
plus Nelson Beebe
on floating point

This is Edition 0.0.

Copyright © 2022, 2023 Richard Stallman and Free Software Foundation, Inc.

(The work of Trevis Rothwell and Nelson Beebe has been assigned to the FSF.)

> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being “GNU General Public License,” with the Front-Cover Texts being “A GNU Manual,” and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled “GNU Free Documentation License.”
>
> (a) The FSF’s Back-Cover Text is: “You have the freedom to copy and modify this GNU manual.”

Short Contents

Preface 1
1 The First Example 3
2 A Complete Program 9
3 Storage and Data 12
4 Beyond Integers 13
5 Lexical Syntax 17
6 Arithmetic 22
7 Assignment Expressions 30
8 Execution Control Expressions 36
9 Binary Operator Grammar 41
10 Order of Execution 43
11 Primitive Data Types 46
12 Constants 52
13 Type Size 60
14 Pointers 62
15 Structures 74
16 Arrays 91
17 Enumeration Types 99
18 Defining Typedef Names 101
19 Statements 103
20 Variables 120
21 Type Qualifiers 128
22 Functions 132
23 Compatible Types 154
24 Type Conversions 155
25 Scope 158
26 Preprocessing 160
27 Integers in Depth 191
28 Floating Point in Depth 193
29 Compilation 211
30 Directing Compilation 212
A Type Alignment 217
B Aliasing 218

C Digraphs 221
D Attributes in Declarations 222
E Signals 224
F GNU Free Documentation License 225
G GNU General Public License 233
Index of Symbols and Keywords 244
Concept Index 246

Table of Contents

Preface 1

1 The First Example 3
1.1 Example: Recursive Fibonacci 3
1.1.1 Function Header 4
1.1.2 Function Body 4
1.2 The Stack, And Stack Overflow 5
1.3 Example: Iterative Fibonacci 6

2 A Complete Program 9
2.1 Complete Program Example 9
2.2 Complete Program Explanation 9
2.3 Complete Program, Line by Line 10
2.4 Compiling the Example Program 11

3 Storage and Data 12

4 Beyond Integers 13
4.1 An Example with Non-Integer Numbers 13
4.2 An Example with Arrays 14
4.3 Calling the Array Example 14
4.4 Variations for Array Example 15

5 Lexical Syntax 17
5.1 Write Programs in English! 17
5.2 Characters 17
5.3 Whitespace 18
5.4 Comments 18
5.5 Identifiers 19
5.6 Operators and Punctuation 20
5.7 Line Continuation 21

6 Arithmetic 22
6.1 Basic Arithmetic 22
6.2 Integer Arithmetic 22
6.3 Integer Overflow 23
6.3.1 Overflow with Unsigned Integers 23
6.3.2 Overflow with Signed Integers 24
6.4 Mixed-Mode Arithmetic 25
6.5 Division and Remainder 25

6.6 Numeric Comparisons 26
6.7 Shift Operations 26
  6.7.1 Shifting Makes New Bits 27
  6.7.2 Caveats for Shift Operations 27
  6.7.3 Shift Hacks 28
6.8 Bitwise Operations 28

7 Assignment Expressions 30
7.1 Simple Assignment 30
7.2 Lvalues 31
7.3 Modifying Assignment 31
7.4 Increment and Decrement Operators 32
7.5 Postincrement and Postdecrement 33
7.6 Pitfall: Assignment in Subexpressions 34
7.7 Write Assignments in Separate Statements 35

8 Execution Control Expressions 36
8.1 Logical Operators 36
8.2 Logical Operators and Comparisons 36
8.3 Logical Operators and Assignments 37
8.4 Conditional Expression 37
  8.4.1 Rules for the Conditional Operator 37
  8.4.2 Conditional Operator Branches 38
8.5 Comma Operator 39
  8.5.1 The Uses of the Comma Operator 39
  8.5.2 Clean Use of the Comma Operator 39
  8.5.3 When Not to Use the Comma Operator 40

9 Binary Operator Grammar 41

10 Order of Execution 43
10.1 Reordering of Operands 43
10.2 Associativity and Ordering 43
10.3 Sequence Points 44
10.4 Postincrement and Ordering 44
10.5 Ordering of Operands 45
10.6 Optimization and Ordering 45

11 Primitive Data Types 46
11.1 Integer Data Types 46
  11.1.1 Basic Integers 46
  11.1.2 Signed and Unsigned Types 46
  11.1.3 Narrow Integers 47
  11.1.4 Conversion among Integer Types 47
  11.1.5 Boolean Type 48
  11.1.6 Integer Variations 48

11.2 Floating-Point Data Types 48
11.3 Complex Data Types 49
11.4 The Void Type 50
11.5 Other Data Types 50
11.6 Type Designators 51

12 Constants 52
12.1 Integer Constants 52
12.2 Integer Constant Data Types 52
12.3 Floating-Point Constants 53
12.4 Imaginary Constants 54
12.5 Invalid Numbers 55
12.6 Character Constants 55
12.7 String Constants 56
12.8 UTF-8 String Constants 57
12.9 Unicode Character Codes 57
12.10 Wide Character Constants 58
12.11 Wide String Constants 59

13 Type Size 60

14 Pointers 62
14.1 Address of Data 62
14.2 Pointer Types 62
14.3 Pointer-Variable Declarations 62
14.4 Pointer-Type Designators 63
14.5 Dereferencing Pointers 63
14.6 Null Pointers 64
14.7 Dereferencing Null or Invalid Pointers 65
14.8 Void Pointers 65
14.9 Pointer Comparison 66
14.10 Pointer Arithmetic 67
14.11 Pointers and Arrays 69
14.12 Pointer Arithmetic at Low-Level 69
14.13 Pointer Increment and Decrement 70
14.14 Drawbacks of Pointer Arithmetic 72
14.15 Pointer-Integer Conversion 72
14.16 Printing Pointers 73

15 Structures 74
15.1 Referencing Structure Fields 75
15.2 Arrays as Fields 75
15.3 Dynamic Memory Allocation 76
15.4 Field Offset 77
15.5 Structure Layout 77
15.6 Packed Structures 78

15.7 Bit Fields 79
15.8 Bit Field Packing 79
15.9 `const` Fields 80
15.10 Arrays of Length Zero 81
15.11 Flexible Array Fields 81
15.12 Overlaying Different Structures 82
15.13 Structure Assignment 82
15.14 Unions 83
15.15 Packing With Unions 84
15.16 Cast to a Union Type 85
15.17 Structure Constructors 86
15.18 Unnamed Types as Fields 86
15.19 Incomplete Types 87
15.20 Intertwined Incomplete Types 88
15.21 Type Tags 88

16 Arrays 91
16.1 Accessing Array Elements 91
16.2 Declaring an Array 91
16.3 Strings 92
16.4 Array Type Designators 93
16.5 Incomplete Array Types 93
16.6 Limitations of C Arrays 94
16.7 Multidimensional Arrays 95
16.8 Constructing Array Values 96
16.9 Arrays of Variable Length 97

17 Enumeration Types 99

18 Defining Typedef Names 101

19 Statements 103
19.1 Expression Statement 103
19.2 `if` Statement 103
19.3 `if-else` Statement 104
19.4 Blocks 104
19.5 `return` Statement 105
19.6 Loop Statements 105
  19.6.1 `while` Statement 105
  19.6.2 `do-while` Statement 106
  19.6.3 `break` Statement 106
  19.6.4 `for` Statement 107
  19.6.5 Example of `for` 107
  19.6.6 Omitted `for`-Expressions 108
  19.6.7 `for`-Index Declarations 109
  19.6.8 `continue` Statement 109

19.7 `switch` Statement 110
19.8 Example of `switch` 111
19.9 Duff’s Device 112
19.10 Case Ranges 113
19.11 Null Statement 113
19.12 `goto` Statement and Labels 114
19.13 Locally Declared Labels 116
19.14 Labels as Values 117
  19.14.1 Label Value Uses 117
  19.14.2 Label Value Caveats 118
19.15 Statements and Declarations in Expressions 118

20 Variables 120

20.1 Variable Declarations 120
  20.1.1 Declaring Arrays and Pointers 120
  20.1.2 Combining Variable Declarations 121
20.2 Initializers 121
20.3 Designated Initializers 122
20.4 Referring to a Type with `__auto_type` 124
20.5 Local Variables 124
20.6 File-Scope Variables 125
20.7 Static Local Variables 125
20.8 `extern` Declarations 126
20.9 Allocating File-Scope Variables 126
20.10 `auto` and `register` 127
20.11 Omitting Types in Declarations 127

21 Type Qualifiers 128

21.1 `const` Variables and Fields 128
21.2 `volatile` Variables and Fields 129
21.3 `restrict`-Qualified Pointers 130
21.4 `restrict` Pointer Example 130

22 Functions 132

22.1 Function Definitions 132
  22.1.1 Function Parameter Variables 132
  22.1.2 Forward Function Declarations 133
  22.1.3 Static Functions 134
  22.1.4 Arrays as Parameters 134
    22.1.4.1 Array parameters are pointers 134
    22.1.4.2 Passing array arguments 135
    22.1.4.3 Type qualifiers on array parameters 136
  22.1.5 Functions That Accept Structure Arguments 137
22.2 Function Declarations 137
22.3 Function Calls 139
22.4 Function Call Semantics 139
22.5 Function Pointers 140

22.5.1 Declaring Function Pointers 140
22.5.2 Assigning Function Pointers 141
22.5.3 Calling Function Pointers 141
22.6 The `main` Function 142
22.6.1 Returning Values from `main` 142
22.6.2 Accessing Command-line Parameters 143
22.6.3 Accessing Environment Variables 143
22.7 Advanced Function Features 144
22.7.1 Variable-Length Array Parameters 144
22.7.2 Variable-Length Parameter Lists 145
22.7.3 Nested Functions 147
22.7.4 Inline Function Definitions 149
22.8 Obsolete Function Features 151
22.8.1 Older GNU C Inlining 151
22.8.2 Old-Style Function Definitions 151

23 Compatible Types 154

24 Type Conversions 155

24.1 Explicit Type Conversion 155
24.2 Assignment Type Conversions 155
24.3 Argument Promotions 156
24.4 Operand Promotions 157
24.5 Common Type 157

25 Scope 158

26 Preprocessing 160

26.1 Preprocessing Overview 160
26.2 Directives 160
26.3 Preprocessing Tokens 161
26.4 Header Files 162
26.4.1 `#include` Syntax 163
26.4.2 `#include` Operation 164
26.4.3 Search Path 164
26.4.4 Once-Only Headers 165
26.4.5 Computed Includes 166
26.5 Macros 167
26.5.1 Object-like Macros 167
26.5.2 Function-like Macros 169
26.5.3 Macro Arguments 170
26.5.4 Stringification 171
26.5.5 Concatenation 173
26.5.6 Variadic Macros 174
26.5.7 Predefined Macros 175
26.5.8 Undefining and Redefining Macros 178

26.5.9 Directives Within Macro Arguments 178
26.5.10 Macro Pitfalls 179
26.5.10.1 Misnesting 179
26.5.10.2 Operator Precedence Problems 179
26.5.10.3 Swallowing the Semicolon 180
26.5.10.4 Duplication of Side Effects 181
26.5.10.5 Using `__auto_type` for Local Variables 182
26.5.10.6 Self-Referential Macros 182
26.5.10.7 Argument Prescan 183
26.6 Conditionals 184
26.6.1 Uses of Conditional Directives 185
26.6.2 Syntax of Preprocessing Conditionals 185
26.6.2.1 The `#ifdef` directive 185
26.6.2.2 The `#if` directive 186
26.6.2.3 The `defined` test 187
26.6.2.4 The `#else` directive 188
26.6.2.5 The `#elif` directive 188
26.6.3 Deleted Code 188
26.7 Diagnostics 189
26.8 Line Control 189
26.9 Null Directive 190

27 Integers in Depth 191

27.1 Integer Representations 191
27.2 Maximum and Minimum Values 192

28 Floating Point in Depth 193

28.1 Floating-Point Representations 193
28.2 Floating-Point Type Specifications 193
28.3 Special Floating-Point Values 194
28.4 Invalid Optimizations 195
28.5 Floating Arithmetic Exception Flags 195
28.6 Exact Floating-Point Arithmetic 196
28.7 Rounding 196
28.8 Rounding Issues 197
28.9 Significance Loss 198
28.10 Fused Multiply-Add 199
28.11 Error Recovery 200
28.12 Exact Floating-Point Constants 201
28.13 Handling Infinity 201
28.14 Handling NaN 202
28.15 Signed Zeros 203
28.16 Scaling by Powers of the Base 203
28.17 Rounding Control 204
28.18 Machine Epsilon 204
28.19 Complex Arithmetic 206
28.20 Round-Trip Base Conversion 208
28.21 Further Reading 208

29 Compilation 211

30 Directing Compilation 212

- 30.1 Pragmas 212
  - 30.1.1 Pragma Basics 212
  - 30.1.2 Severity Pragmas 213
  - 30.1.3 Optimization Pragmas 215
- 30.2 Static Assertions 216

Appendix A Type Alignment 217

Appendix B Aliasing 218

- B.1 Aliasing and Alignment 218
- B.2 Aliasing and Length 218
- B.3 Type Rules for Aliasing 219

Appendix C Digraphs 221

Appendix D Attributes in Declarations 222

Appendix E Signals 224

Appendix F GNU Free Documentation License 225

Appendix G GNU General Public License 233

Index of Symbols and Keywords 244

Concept Index 246

Preface

This manual explains the C language for use with the GNU Compiler Collection (GCC) on the GNU/Linux operating system and other systems. We refer to this dialect as GNU C. If you already know C, you can use this as a reference manual.

If you understand basic concepts of programming but know nothing about C, you can read this manual sequentially from the beginning to learn the C language.

If you are a beginner in programming, we recommend you first learn a language with automatic garbage collection and no explicit pointers, rather than starting with C. Good choices include Lisp, Scheme, Python and Java. Because of C's explicit pointers, programmers must be careful to avoid certain kinds of errors in memory usage.

C is a venerable language; it was first used in 1973. The GNU C Compiler, which was subsequently extended into the GNU Compiler Collection, was first released in 1987. Other important languages were designed based on C: once you know C, it gives you a useful base for learning C++, C#, Java, Scala, D, Go, and more.

The special advantage of C is that it is fairly simple while allowing close access to the computer's hardware, which previously required writing in assembler language to describe the individual machine instructions. Some have called C a "high-level assembler language" because of its explicit pointers and lack of automatic management of storage. As one wag put it, "C combines the power of assembler language with the convenience of assembler language." However, C is far more portable, and much easier to read and write, than assembler language.

This manual describes the GNU C language supported by the GNU Compiler Collection, as of roughly 2017. Please inform us of any changes needed to match the current version of GNU C.

When a construct may be absent or work differently in other C compilers, we say so. When it is not part of ISO standard C, we say it is a "GNU C extension," because it is useful to know that. However, standards and other dialects are secondary topics for this manual. For simplicity's sake, we keep those notes short, unless it is vital to say more.

Some aspects of the meaning of C programs depend on the target platform: which computer, and which operating system, the compiled code will run on. Where this is the case, we say so.

When compiling for a "real computer", one that is a reasonable platform for running the GNU/Linux system, the type `int` is always 32 bits in size. This manual assumes you are compiling for the computer where you are running the compiler, which implies `int` has that size. GNU C can also compile code for some microprocessors on which type `int` has fewer bits, but this manual does not try to cover the complications of those peculiar platforms.

We hardly mention C++ or other languages that the GNU Compiler Collection supports. We hope this manual will serve as a base for writing manuals for those languages, but languages so different can't share one common manual.

The C language provides no built-in facilities for performing such common operations as input/output, memory management, string manipulation, and the like. Instead, these facilities are provided by functions defined in the standard library, which is automatically available in every C program. See *The GNU C Library Reference Manual*.

Most GNU/Linux systems use the GNU C Library to provide those facilities. It is itself written in C, so once you know C you can read its source code and see how its library functions do their jobs. Some fraction of the functions are implemented as *system calls*, which means they contain a special instruction that asks the system kernel (Linux) to do a specific task. To understand how those are implemented, you'd need to read Linux source code. Whether a library function is a system call is an internal implementation detail that makes no difference for how to call the function.

This manual incorporates the former GNU C Preprocessor Manual, which was among the earliest GNU manuals. It also uses some text from the earlier GNU C Manual that was written by Trevis Rothwell and James Youngman.

GNU C has many obscure features, each one either for historical compatibility or meant for very special situations. We have left them to a companion manual, the GNU C Obscurities Manual, which will be published digitally later.

Please report errors and suggestions to c-manual@gnu.org.

1 The First Example

This chapter presents the source code for a very simple C program and uses it to explain a few features of the language. If you already know the basic points of C presented in this chapter, you can skim it or skip it.

We present examples of C source code (other than comments) using a fixed-width typeface, since that's the way they look when you edit them in an editor such as GNU Emacs.

1.1 Example: Recursive Fibonacci

To introduce the most basic features of C, let's look at code for a simple mathematical function that does calculations on integers. This function calculates the *n*th number in the Fibonacci series, in which each number is the sum of the previous two: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

```
int
fib (int n)
{
  if (n <= 2)  /* This avoids infinite recursion.  */
    return 1;
  else
    return fib (n - 1) + fib (n - 2);
}
```

This very simple program illustrates several features of C:

- A function definition, whose first two lines constitute the function header. See Section 22.1 [Function Definitions], page 132.
- A function parameter `n`, referred to as the variable `n` inside the function body. See Section 22.1.1 [Function Parameter Variables], page 132. A function definition uses parameters to refer to the argument values provided in a call to that function.
- Arithmetic. C programs add with '`+`' and subtract with '`-`'. See Chapter 6 [Arithmetic], page 22.
- Numeric comparisons. The operator '`<=`' tests for "less than or equal." See Section 6.6 [Numeric Comparisons], page 26.
- Integer constants written in base 10. See Section 12.1 [Integer Constants], page 52.
- A function call. The function call `fib (n - 1)` calls the function `fib`, passing as its argument the value `n - 1`. See Section 22.3 [Function Calls], page 139.
- A comment, which starts with '`/*`' and ends with '`*/`'. The comment has no effect on the execution of the program. Its purpose is to provide explanations to people reading the source code. Including comments in the code is tremendously important—they provide background information so others can understand the code more quickly. See Section 5.4 [Comments], page 18.

  In this manual, we present comment text in the variable-width typeface used for the text of the chapters, not in the fixed-width typeface used for the rest of the code. That is to make comments easier to read. This distinction of typeface does not exist in a real file of C source code.

- Two kinds of statements, the `return` statement and the `if...else` statement. See Chapter 19 [Statements], page 103.
- Recursion. The function `fib` calls itself; that is called a *recursive call*. These are valid in C, and quite common.

  The `fib` function would not be useful if it didn't return. Thus, recursive definitions, to be of any use, must avoid *infinite recursion*.

  This function definition prevents infinite recursion by specially handling the case where `n` is two or less. Thus the maximum depth of recursive calls is less than `n`.

1.1.1 Function Header

In our example, the first two lines of the function definition are the *header*. Its purpose is to state the function's name and say how it is called:

```
int
fib (int n)
```

says that the function returns an integer (type `int`), its name is `fib`, and it takes one argument named `n` which is also an integer. (Data types will be explained later, in Chapter 11 [Primitive Types], page 46.)

1.1.2 Function Body

The rest of the function definition is called the *function body*. Like every function body, this one starts with '`{`', ends with '`}`', and contains zero or more *statements* and *declarations*. Statements specify actions to take, whereas declarations define names of variables, functions, and so on. Each statement and each declaration ends with a semicolon ('`;`').

Statements and declarations often contain *expressions*; an expression is a construct whose execution produces a *value* of some data type, but may also take actions through "side effects" that alter subsequent execution. A statement, by contrast, does not have a value; it affects further execution of the program only through the actions it takes.

This function body contains no declarations, and just one statement, but that one is a complex statement in that it contains nested statements. This function uses two kinds of statements:

`return`

The `return` statement makes the function return immediately. It looks like this:

```
return value;
```

Its meaning is to compute the expression *value* and exit the function, making it return whatever value that expression produced. For instance,

```
return 1;
```

returns the integer 1 from the function, and

```
return fib (n - 1) + fib (n - 2);
```

returns a value computed by performing two function calls as specified and adding their results.

`if...else`

The `if...else` statement is a *conditional*. Each time it executes, it chooses one of its two substatements to execute and ignores the other. It looks like this:

```
if (condition)
```

```
  if-true-statement
else
  if-false-statement
```

Its meaning is to compute the expression *condition* and, if it's "true," execute *if-true-statement*. Otherwise, execute *if-false-statement*. See Section 19.3 [if-else Statement], page 104.

Inside the `if...else` statement, *condition* is simply an expression. It's considered "true" if its value is nonzero. (A comparison operation, such as `n <= 2`, produces the value 1 if it's "true" and 0 if it's "false." See Section 6.6 [Numeric Comparisons], page 26.) Thus,

```
if (n <= 2)
  return 1;
else
  return fib (n - 1) + fib (n - 2);
```

first tests whether the value of `n` is less than or equal to 2. If so, the expression `n <= 2` has the value 1. So execution continues with the statement

```
return 1;
```

Otherwise, execution continues with this statement:

```
return fib (n - 1) + fib (n - 2);
```

Each of these statements ends the execution of the function and provides a value for it to return. See Section 19.5 [return Statement], page 105.

Calculating `fib` using ordinary integers in C works only for $n < 47$ because the value of `fib (47)` is too large to fit in type `int`. In GNU C, type `int` holds 32 bits (see Section 11.1 [Integer Types], page 46), so the addition operation that tries to add `fib (46)` and `fib (45)` cannot deliver the correct result. This occurrence is called *integer overflow*.

Overflow can manifest itself in various ways, but one thing that can't possibly happen is to produce the correct value, since that can't fit in the space for the value. See Section 6.3 [Integer Overflow], page 23, for more details about this situation.

See Chapter 22 [Functions], page 132, for a full explanation about functions.

1.2 The Stack, And Stack Overflow

Recursion has a drawback: there are limits to how many nested levels of function calls a program can make. In C, each function call allocates a block of memory which it uses until the call returns. C allocates these blocks consecutively within a large area of memory known as the *stack*, so we refer to the blocks as *stack frames*.

The size of the stack is limited; if the program tries to use too much, that causes the program to fail because the stack is full. This is called *stack overflow*.

Stack overflow on GNU/Linux typically manifests itself as the *signal* named `SIGSEGV`, also known as a "segmentation fault." By default, this signal terminates the program immediately, rather than letting the program try to recover, or reach an expected ending point. (We commonly say in this case that the program "crashes.") See Appendix E [Signals], page 224.

It is inconvenient to observe a crash by passing too large an argument to recursive Fibonacci, because the program would run a long time before it crashes. This algorithm is simple but ridiculously slow: in calculating `fib (n)`, the number of (recursive) calls `fib (1)` or `fib (2)` that it makes equals the final result.

However, you can observe stack overflow very quickly if you use this function instead:

```
int
fill_stack (int n)
{
  if (n <= 1)  /* This limits the depth of recursion.  */
    return 1;
  else
    return fill_stack (n - 1);
}
```

Under gNewSense GNU/Linux on the Lemote Yeeloong, without optimization and using the default configuration, an experiment showed there is enough stack space to do 261906 nested calls to that function. One more, and the stack overflows and the program crashes. On another platform, with a different configuration, or with a different function, the limit might be bigger or smaller.

1.3 Example: Iterative Fibonacci

Here's a much faster algorithm for computing the same Fibonacci series. It is faster for two reasons. First, it uses *iteration* (that is, repetition or looping) rather than recursion, so it doesn't take time for a large number of function calls. But mainly, it is faster because the number of repetitions is small—only *n*.

```
int
fib (int n)
{
  int last = 1;   /* Initial value is fib (1).  */
  int prev = 0;   /* Initial value controls fib (2).  */
  int i;

  for (i = 1; i < n; ++i)
    /* If n is 1 or less, the loop runs zero times,  */
    /* since in that case i < n is false the first time.  */
    {
      /* Now last is fib (i)
         and prev is fib (i - 1).  */
      /* Compute fib (i + 1).  */
      int next = prev + last;
      /* Shift the values down.  */
      prev = last;
      last = next;
      /* Now last is fib (i + 1)
         and prev is fib (i).
         But that won't stay true for long,
             because we are about to increment i.  */
        }

      return last;
    }
```

This definition computes `fib (n)` in a time proportional to *n*. The comments in the definition explain how it works: it advances through the series, always keeps the last two values in `last` and `prev`, and adds them to get the next value.

Here are the additional C features that this definition uses:

Internal blocks

Within a function, wherever a statement is called for, you can write a *block*. It looks like `{ ... }` and contains zero or more statements and declarations. (You can also use additional blocks as statements in a block.)

The function body also counts as a block, which is why it can contain statements and declarations.

See Section 19.4 [Blocks], page 104.

Declarations of local variables

This function body contains declarations as well as statements. There are three declarations directly in the function body, as well as a fourth declaration in an internal block. Each starts with `int` because it declares a variable whose type is integer. One declaration can declare several variables, but each of these declarations is simple and declares just one variable.

Variables declared inside a block (either a function body or an internal block) are *local variables*. These variables exist only within that block; their names are not defined outside the block, and exiting the block deallocates their storage. This example declares four local variables: `last`, `prev`, `i`, and `next`.

The most basic local variable declaration looks like this:

```
type variablename;
```

For instance,

```
int i;
```

declares the local variable `i` as an integer. See Section 20.1 [Variable Declarations], page 120.

Initializers

When you declare a variable, you can also specify its initial value, like this:

```
type variablename = value;
```

For instance,

```
int last = 1;
```

declares the local variable `last` as an integer (type `int`) and starts it off with the value 1. See Section 20.2 [Initializers], page 121.

Assignment

Assignment: a specific kind of expression, written with the ‘`=`’ operator, that stores a new value in a variable or other place. Thus,

```
variable = value
```

is an expression that computes *`value`* and stores the value in *`variable`*. See Chapter 7 [Assignment Expressions], page 30.

Expression statements

An expression statement is an expression followed by a semicolon. That computes the value of the expression, then ignores the value.

An expression statement is useful when the expression changes some data or has other side effects—for instance, with function calls, or with assignments as in this example. See Section 19.1 [Expression Statement], page 103.

Using an expression with no side effects in an expression statement is pointless; for instance, the expression statement `x;` would examine the value of `x` and ignore it. That is not useful.[1]

Increment operator

The increment operator is '`++`'. `++i` is an expression that is short for `i = i + 1`. See Section 7.4 [Increment/Decrement], page 32.

`for` statements

A `for` statement is a clean way of executing a statement repeatedly—a *loop* (see Section 19.6 [Loop Statements], page 105). Specifically,

```
for (i = 1; i < n; ++i)
  body
```

means to start by doing `i = 1` (set `i` to one) to prepare for the loop. The loop itself consists of

- Testing `i < n` and exiting the loop if that's false.
- Executing *body*.
- Advancing the loop (executing `++i`, which increments `i`).

The net result is to execute *body* with 1 in `i`, then with 2 in `i`, and so on, stopping just before the repetition where `i` would equal `n`. If `n` is less than 1, the loop will execute the body zero times.

The body of the `for` statement must be one and only one statement. You can't write two statements in a row there; if you try to, only the first of them will be treated as part of the loop.

The way to put multiple statements in such a place is to group them with a block, and that's what we do in this example.

---

[1] Computing an expression and ignoring the result can be useful in peculiar cases. For instance, dereferencing a pointer and ignoring the value is a way to cause a fault if a pointer value is invalid. See Appendix E [Signals], page 224. But you may need to declare the pointer target `volatile` or the dereference may be optimized away. See Section 21.2 [volatile], page 129.

2 A Complete Program

It's all very well to write a Fibonacci function, but you cannot run it by itself. It is a useful program, but it is not a complete program.

In this chapter we present a complete program that contains the `fib` function. This example shows how to make the program start, how to make it finish, how to do computation, and how to print a result.

2.1 Complete Program Example

Here is the complete program that uses the simple, recursive version of the `fib` function (see Section 1.1 [Recursive Fibonacci], page 3):

```
#include <stdio.h>

int
fib (int n)
{
  if (n <= 2)  /* This avoids infinite recursion.  */
    return 1;
  else
    return fib (n - 1) + fib (n - 2);
}

int
main (void)
{
  printf ("Fibonacci series item %d is %d\n",
          20, fib (20));
  return 0;
}
```

This program prints a message that shows the value of `fib (20)`.

Now for an explanation of what that code means.

2.2 Complete Program Explanation

This sample program prints a message that shows the value of `fib (20)`, and exits with code 0 (which stands for successful execution).

Every C program is started by running the function named `main`. Therefore, the example program defines a function named `main` to provide a way to start it. Whatever that function does is what the program does. See Section 22.6 [The main Function], page 142.

The `main` function is the first one called when the program runs, but it doesn't come first in the example code. The order of the function definitions in the source code makes no difference to the program's meaning.

The initial call to `main` always passes certain arguments, but `main` does not have to pay attention to them. To ignore those arguments, define `main` with `void` as the parameter list.

(`void` as a function's parameter list normally means "call with no arguments," but `main` is a special case.)

The function `main` returns 0 because that is the conventional way for `main` to indicate successful execution. It could instead return a positive integer to indicate failure, and some utility programs have specific conventions for the meaning of certain numeric *failure codes*. See Section 22.6.1 [Values from main], page 142.

The simplest way to print text in C is by calling the `printf` function, so here we explain very briefly what that function does. For a full explanation of `printf` and the other standard I/O functions, see Section "Input/Output on Streams" in *The GNU C Library Reference Manual*.

The first argument to `printf` is a *string constant* (see Section 12.7 [String Constants], page 56) that is a template for output. The function `printf` copies most of that string directly as output, including the newline character at the end of the string, which is written as '`\n`'. The output goes to the program's *standard output* destination, which in the usual case is the terminal.

'`%`' in the template introduces a code that substitutes other text into the output. Specifically, '`%d`' means to take the next argument to `printf` and substitute it into the text as a decimal number. (The argument for '`%d`' must be of type `int`; if it isn't, `printf` will malfunction.) So the output is a line that looks like this:

```
Fibonacci series item 20 is 6765
```

This program does not contain a definition for `printf` because it is defined by the C library, which makes it available in all C programs. However, each program does need to *declare* `printf` so it will be called correctly. The `#include` line takes care of that; it includes a *header file* called `stdio.h` into the program's code. That file is provided by the operating system and it contains declarations for the many standard input/output functions in the C library, one of which is `printf`.

Don't worry about header files for now; we'll explain them later in Section 26.4 [Header Files], page 162.

The first argument of `printf` does not have to be a string constant; it can be any string (see Section 16.3 [Strings], page 92). However, using a constant is the most common case.

2.3 Complete Program, Line by Line

Here's the same example, explained line by line. **Beginners, do you find this helpful or not? Would you prefer a different layout for the example? Please tell rms@gnu.org.**

```
#include <stdio.h>      /* Include declaration of usual */
                        /*   I/O functions such as printf. */
                        /* Most programs need these. */

int                     /* This function returns an int. */
fib (int n)             /* Its name is fib; */
                        /*   its argument is called n. */
{                       /* Start of function body. */
  /* This stops the recursion from being infinite. */
  if (n <= 2)           /* If n is 1 or 2, */
```

```
    return 1;           /*    make fib return 1.  */
  else                  /* Otherwise, add the two previous  */
                        /* Fibonacci numbers.  */
    return fib (n - 1) + fib (n - 2);
}

int                     /* This function returns an int.  */
main (void)             /* Start here; ignore arguments.  */
{                       /* Print message with numbers in it.  */
  printf ("Fibonacci series item %d is %d\n",
          20, fib (20));
  return 0;             /* Terminate program, report success.  */
}
```

2.4 Compiling the Example Program

To run a C program requires converting the source code into an *executable file*. This is called *compiling* the program, and the command to do that using GNU C is `gcc`.

This example program consists of a single source file. If we call that file `fib1.c`, the complete command to compile it is this:

```
gcc -g -O -o fib1 fib1.c
```

Here, `-g` says to generate debugging information, `-O` says to optimize at the basic level, and `-o fib1` says to put the executable program in the file `fib1`.

To run the program, use its file name as a shell command. For instance,

```
./fib1
```

However, unless you are sure the program is correct, you should expect to need to debug it. So use this command,

```
gdb fib1
```

which starts the GDB debugger (see Section “A Sample GDB Session” in *Debugging with GDB*) so you can run and debug the executable program `fib1`.

Richard Stallman’s advice, from personal experience, is to turn to the debugger as soon as you can reproduce the problem. Don’t try to avoid it by using other methods instead—occasionally they are shortcuts, but usually they waste an unbounded amount of time. With the debugger, you will surely find the bug in a reasonable time; overall, you will get your work done faster. The sooner you get serious and start the debugger, the sooner you are likely to find the bug.

See Chapter 29 [Compilation], page 211, for an introduction to compiling more complex programs which consist of more than one source file.

3 Storage and Data

Storage in C programs is made up of units called *bytes*. A byte is the smallest unit of storage that can be used in a first-class manner.

On nearly all computers, a byte consists of 8 bits. There are a few peculiar computers (mostly "embedded controllers" for very small systems) where a byte is longer than that, but this manual does not try to explain the peculiarity of those computers; we assume that a byte is 8 bits.

Every C data type is made up of a certain number of bytes; that number is the data type's *size*. See Chapter 13 [Type Size], page 60, for details. The types `signed char` and `unsigned char` are one byte long; use those types to operate on data byte by byte. See Section 11.1.2 [Signed and Unsigned Types], page 46. You can refer to a series of consecutive bytes as an array of `char` elements; that's what a character string looks like in memory. See Section 12.7 [String Constants], page 56.

4 Beyond Integers

So far we've presented programs that operate on integers. In this chapter we'll present examples of handling non-integral numbers and arrays of numbers.

4.1 An Example with Non-Integer Numbers

Here's a function that operates on and returns *floating point* numbers that don't have to be integers. Floating point represents a number as a fraction together with a power of 2. (For more detail, see Section 11.2 [Floating-Point Data Types], page 48.) This example calculates the average of three floating point numbers that are passed to it as arguments:

```
double
average_of_three (double a, double b, double c)
{
  return (a + b + c) / 3;
}
```

The values of the parameter *a*, *b* and *c* do not have to be integers, and even when they happen to be integers, most likely their average is not an integer.

`double` is the usual data type in C for calculations on floating-point numbers.

To print a `double` with `printf`, we must use '`%f`' instead of '`%d`':

```
printf ("Average is %f\n",
        average_of_three (1.1, 9.8, 3.62));
```

The code that calls `printf` must pass a `double` for printing with '`%f`' and an `int` for printing with '`%d`'. If the argument has the wrong type, `printf` will produce meaningless output.

Here's a complete program that computes the average of three specific numbers and prints the result:

```
double
average_of_three (double a, double b, double c)
{
  return (a + b + c) / 3;
}

int
main (void)
{
    printf ("Average is %f\n",
            average_of_three (1.1, 9.8, 3.62));
    return 0;
}
```

From now on we will not present examples of calls to `main`. Instead we encourage you to write them for yourself when you want to test executing some code.

4.2 An Example with Arrays

A function to take the average of three numbers is very specific and limited. A more general function would take the average of any number of numbers. That requires passing the numbers in an array. An array is an object in memory that contains a series of values of the same data type. This chapter presents the basic concepts and use of arrays through an example; for the full explanation, see Chapter 16 [Arrays], page 91.

Here's a function definition to take the average of several floating-point numbers, passed as type `double`. The first parameter, `length`, specifies how many numbers are passed. The second parameter, `input_data`, is an array that holds those numbers.

```
double
avg_of_double (int length, double input_data[])
{
  double sum = 0;
  int i;

  for (i = 0; i < length; i++)
    sum = sum + input_data[i];

  return sum / length;
}
```

This introduces the expression to refer to an element of an array: `input_data[i]` means the element at index `i` in `input_data`. The index of the element can be any expression with an integer value; in this case, the expression is `i`. See Section 16.1 [Accessing Array Elements], page 91.

The lowest valid index in an array is 0, *not* 1, and the highest valid index is one less than the number of elements. (This is known as *zero-origin indexing*.)

This example also introduces the way to declare that a function parameter is an array. Such declarations are modeled after the syntax for an element of the array. Just as `double foo` declares that `foo` is of type `double`, `double input_data[]` declares that each element of `input_data` is of type `double`. Therefore, `input_data` itself has type "array of `double`."

When declaring an array parameter, it's not necessary to say how long the array is. In this case, the parameter `input_data` has no length information. That's why the function needs another parameter, `length`, for the caller to provide that information to the function `avg_of_double`.

4.3 Calling the Array Example

To call the function `avg_of_double` requires making an array and then passing it as an argument. Here is an example.

```
{
  /* The array of values to compute the average of.  */
  double nums_to_average[5];
  /* The average, once we compute it.  */
  double average;
  /* Fill in elements of nums_to_average.  */

  nums_to_average[0] = 58.7;
  nums_to_average[1] = 5.1;
  nums_to_average[2] = 7.7;
  nums_to_average[3] = 105.2;
  nums_to_average[4] = -3.14159;

  average = avg_of_double (5, nums_to_average);

  /* ...now make use of average... */
}
```

This shows an array subscripting expression again, this time on the left side of an assignment, storing a value into an element of an array.

It also shows how to declare a local variable that is an array: `double nums_to_average[5];`. Since this declaration allocates the space for the array, it needs to know the array's length. You can specify the length with any expression whose value is an integer, but in this declaration the length is a constant, the integer 5.

The name of the array, when used by itself as an expression, stands for the address of the array's data, and that's what gets passed to the function `avg_of_double` in `avg_of_double (5, nums_to_average)`.

We can make the code easier to maintain by avoiding the need to write 5, the array length, when calling `avg_of_double`. That way, if we change the array to include more elements, we won't have to change that call. One way to do this is with the `sizeof` operator:

```
average = avg_of_double ((sizeof (nums_to_average)
                          / sizeof (nums_to_average[0])),
                         nums_to_average);
```

This computes the number of elements in `nums_to_average` by dividing its total size by the size of one element. See Chapter 13 [Type Size], page 60, for more details of using `sizeof`.

We don't show in this example what happens after storing the result of `avg_of_double` in the variable `average`. Presumably more code would follow that uses that result somehow. (Why compute the average and not use it?) But that isn't part of this topic.

4.4 Variations for Array Example

The code to call `avg_of_double` has two declarations that start with the same data type:

```
/* The array of values to average.  */
double nums_to_average[5];
/* The average, once we compute it.  */
double average;
```

In C, you can combine the two, like this:

```
double nums_to_average[5], average;
```

This declares `nums_to_average` so each of its elements is a `double`, and `average` itself as a `double`.

However, while you *can* combine them, that doesn't mean you *should.* If it is useful to write comments about the variables, and usually it is, then it's clearer to keep the declarations separate so you can put a comment on each one. That also helps with using textual tools to find occurrences of a variable in source files.

We set all of the elements of the array `nums_to_average` with assignments, but it is more convenient to use an initializer in the declaration:

```
{
  /* The array of values to average.  */
  double nums_to_average[]
    = { 58.7, 5.1, 7.7, 105.2, -3.14159 };

  /* The average, once we compute it.  */
  average = avg_of_double ((sizeof (nums_to_average)
                            / sizeof (nums_to_average[0])),
                           nums_to_average);

  /* ...now make use of average... */
}
```

The array initializer is a comma-separated list of values, delimited by braces. See Section 20.2 [Initializers], page 121.

Note that the declaration does not specify a size for `nums_to_average`, so the size is determined from the initializer. There are five values in the initializer, so `nums_to_average` gets length 5. If we add another element to the initializer, `nums_to_average` will have six elements.

Because the code computes the number of elements from the size of the array, using `sizeof`, the program will operate on all the elements in the initializer, regardless of how many those are.

5 Lexical Syntax

To start the full description of the C language, we explain the lexical syntax and lexical units of C code. The lexical units of a programming language are known as *tokens*. This chapter covers all the tokens of C except for constants, which are covered in a later chapter (see Chapter 12 [Constants], page 52). One vital kind of token is the *identifier* (see Section 5.5 [Identifiers], page 19), which is used for names of any kind.

5.1 Write Programs in English!

In principle, you can write the function and variable names in a program, and the comments, in any human language. C allows any kinds of Unicode characters in comments, and you can put them into identifiers with a special prefix (see Section 12.9 [Unicode Character Codes], page 57). However, to enable programmers in all countries to understand and develop the program, it is best under today's circumstances to write all identifiers and comments in English.

English is the common language of programmers; in all countries, programmers generally learn English. If names and comments in a program are written in English, most programmers in Bangladesh, Belgium, Bolivia, Brazil, Bulgaria and Burundi can understand them. In all those countries, most programmers can speak English, or at least read it, but they do not read each other's languages at all. In India, with so many languages, two programmers may have no common language other than English.

If you don't feel confident in writing English, do the best you can, and follow each English comment with a version in a language you write better; add a note asking others to translate that to English. Someone will eventually do that.

The program's user interface is a different matter. We don't need to choose one language for that; it is easy to support multiple languages and let each user choose the language for display. This requires writing the program to support localization of its interface. (The `gettext` package exists to support this; see Section "Message Translation" in *The GNU C Library Reference Manual*.) Then a community-based translation effort can provide support for all the languages users want to use.

5.2 Characters

GNU C source files are usually written in the ASCII character set, which was defined in the 1960s for English. However, they can also include Unicode characters represented in the UTF-8 multibyte encoding. This makes it possible to represent accented letters such as '`á`', as well as other scripts such as Arabic, Chinese, Cyrillic, Hebrew, Japanese, and Korean.[1]

In C source code, non-ASCII characters are valid in comments, in wide character constants (see Section 12.10 [Wide Character Constants], page 58), and in string constants (see Section 12.7 [String Constants], page 56).

Another way to specify non-ASCII characters in constants (character or string) and identifiers is with an escape sequence starting with backslash, specifying the intended Unicode character. (See Section 12.9 [Unicode Character Codes], page 57.) This specifies non-ASCII characters without putting a real non-ASCII character in the source file itself.

---

[1] On some obscure systems, GNU C uses UTF-EBCDIC instead of UTF-8, but that is not worth describing in this manual.

C accepts two-character aliases called *digraphs* for certain characters. See Appendix C [Digraphs], page 221.

5.3 Whitespace

Whitespace means characters that exist in a file but appear blank in a printed listing of a file (or traditionally did appear blank, several decades ago). The C language requires whitespace in order to separate two consecutive identifiers, or to separate an identifier from a numeric constant. Other than that, and a few special situations described later, whitespace is optional; you can put it in when you wish, to make the code easier to read.

Space and tab in C code are treated as whitespace characters. So are line breaks. You can represent a line break with the newline character (also called *linefeed* or LF), CR (carriage return), or the CRLF sequence (two characters: carriage return followed by a newline character).

The *formfeed* character, Control-L, was traditionally used to divide a file into pages. It is still used this way in source code, and the tools that generate nice printouts of source code still start a new page after each "formfeed" character. Dividing code into pages separated by formfeed characters is a good way to break it up into comprehensible pieces and show other programmers where they start and end.

The *vertical tab* character, Control-K, was traditionally used to make printing advance down to the next section of a page. We know of no particular reason to use it in source code, but it is still accepted as whitespace in C.

Comments are also syntactically equivalent to whitespace.

5.4 Comments

A comment encapsulates text that has no effect on the program's execution or meaning.

The purpose of comments is to explain the code to people that read it. Writing good comments for your code is tremendously important—they should provide background information that helps programmers understand the reasons why the code is written the way it is. You, returning to the code six months from now, will need the help of these comments to remember why you wrote it this way.

Outdated comments that become incorrect are counterproductive, so part of the software developer's responsibility is to update comments as needed to correspond with changes to the program code.

C allows two kinds of comment syntax, the traditional style and the C++ style. A traditional C comment starts with '`/*`' and ends with '`*/`'. For instance,

```
/* This is a comment in traditional C syntax. */
```

A traditional comment can contain '`/*`', but these delimiters do not nest as pairs. The first '`*/`' ends the comment regardless of whether it contains '`/*`' sequences.

```
/* This /* is a comment */ But this is not! */
```

A *line comment* starts with '`//`' and ends at the end of the line. For instance,

```
// This is a comment in C++ style.
```

Line comments do nest, in effect, because '`//`' inside a line comment is part of that comment:

```
// this whole line is // one comment
This is code, not comment.
```

It is safe to put line comments inside block comments, or vice versa.

```
/* traditional comment
   // contains line comment
   more traditional comment
 */ text here is not a comment

// line comment /* contains traditional comment */
```

But beware of commenting out one end of a traditional comment with a line comment. The delimiter '`/*`' doesn't start a comment if it occurs inside an already-started comment.

```
 // line comment  /* That would ordinarily begin a block comment.
    Oops! The line comment has ended;
    this isn't a comment any more.  */
```

Comments are not recognized within string constants. `"/* blah */"` is the string constant '`/* blah */`', not an empty string.

In this manual we show the text in comments in a variable-width font, for readability, but this font distinction does not exist in source files.

A comment is syntactically equivalent to whitespace, so it always separates tokens. Thus,

```
int/* comment */foo;
```

is equivalent to

```
int foo;
```

but clean code always uses real whitespace to separate the comment visually from surrounding code.

5.5 Identifiers

An *identifier* (name) in C is a sequence of letters and digits, as well as '`_`', that does not start with a digit. Most C compilers also allow '`$`'; GNU C allows it. An identifier can be as long as you like; for example,

```
int anti_dis_establishment_arian_ism;
```

Letters in identifiers are case-sensitive in C; thus, `a` and `A` are two different identifiers.

Identifiers in C are used as variable names, function names, typedef names, enumeration constants, type tags, field names, and labels. Certain identifiers in C are *keywords*, which means they have specific syntactic meanings. Keywords in C are *reserved words*, meaning you cannot use them in any other way. For instance, you can't define a variable or function named `return` or `if`.

You can also include other characters, even non-ASCII characters, in identifiers by writing their Unicode character names, which start with '`\u`' or '`\U`', in the identifier name. See Section 12.9 [Unicode Character Codes], page 57. However, it is usually a bad idea to use non-ASCII characters in identifiers, and when the names are written in English, they never need non-ASCII characters. See Section 5.1 [English], page 17.

As stated above, whitespace is required to separate two consecutive identifiers, or to separate an identifier from a preceding or following numeric constant.

5.6 Operators and Punctuation

Here we describe the lexical syntax of operators and punctuation in C. The specific operators of C and their meanings are presented in subsequent chapters.

Some characters that are generally considered punctuation have a different sort of meaning in the C language. C uses double-quote ‘`"`’ to delimit string constants (see Section 12.7 [String Constants], page 56) and ‘`'`’ to delimit character constants (see Section 12.7 [String Constants], page 56). The characters ‘`$`’ and ‘`_`’ can be part of an identifier or a keyword.

Most operators in C consist of one or two characters that can’t be used in identifiers. The characters used for such operators in C are ‘`!~^&|*/%+-=<>,.?:`’. (C preprocessing uses *preprocessing operators*, based on ‘`#`’, which are entirely different from these operators; Chapter 26 [Preprocessing], page 160.)

Some operators are a single character. For instance, ‘`-`’ is the operator for negation (with one operand) and the operator for subtraction (with two operands).

Some operators are two characters. For example, ‘`++`’ is the increment operator. Recognition of multicharacter operators works by reading and grouping as many successive characters as can constitute one operator, and making them one token.

For instance, the character sequence ‘`++`’ is always interpreted as the increment operator; therefore, if we want to write two consecutive instances of the operator ‘`+`’, we must separate them with a space so that they do not combine as one token. Applying the same rule, `a+++++b` is always tokenized as `a++ ++ + b`, not as `a++ + ++b`, even though the latter could be part of a valid C program and the former could not (since `a++` is not an lvalue and thus can’t be the operand of `++`).

A few C operators are keywords rather than special characters. They include `sizeof` (see Chapter 13 [Type Size], page 60) and `_Alignof` (see Appendix A [Type Alignment], page 217).

The characters ‘`;{}[]()`’ are used for punctuation and grouping. Semicolon (‘`;`’) ends a statement. Braces (‘`{`’ and ‘`}`’) begin and end a block at the statement level (see Section 19.4 [Blocks], page 104), and surround the initializer (see Section 20.2 [Initializers], page 121) for a variable with multiple elements or fields (such as arrays or structures).

Square brackets (‘`[`’ and ‘`]`’) do array indexing, as in `array[5]`.

Parentheses are used in expressions for explicit nesting of expressions (see Section 6.1 [Basic Arithmetic], page 22), around the parameter declarations in a function declaration or definition, and around the arguments in a function call, as in `printf ("Foo %d\n", i)` (see Section 22.3 [Function Calls], page 139). Several kinds of statements also use parentheses as part of their syntax—for instance, `if` statements, `for` statements, `while` statements, and `switch` statements. See Section 19.2 [if Statement], page 103, and following sections.

Parentheses are also required around the operand of the operator keywords `sizeof` and `_Alignof` when the operand is a data type rather than a value. See Chapter 13 [Type Size], page 60.

5.7 Line Continuation

The sequence of a backslash and a newline is ignored absolutely anywhere in a C program. This makes it possible to split a single source line into multiple lines in the source file. GNU C tolerates and ignores other whitespace between the backslash and the newline. In particular, it always ignores a CR (carriage return) character there, in case some text editor decided to end the line with the CRLF sequence.

The main use of line continuation in C is for macro definitions that would be inconveniently long for a single line (see Section 26.5 [Macros], page 167).

It is possible to continue a line comment onto another line with backslash-newline. You can put backslash-newline in the middle of an identifier, even a keyword, or an operator. You can even split '`/*`', '`*/`', and '`//`' onto multiple lines with backslash-newline. Here's an ugly example:

```
/\
*
*/ fo\
o +\
= 1\
0;
```

That's equivalent to '`/* */ foo += 10;`'.

Don't do those things in real programs, since they make code hard to read.

Note: For the sake of using certain tools on the source code, it is wise to end every source file with a newline character which is not preceded by a backslash, so that it really ends the last line.

6 Arithmetic

Arithmetic operators in C attempt to be as similar as possible to the abstract arithmetic operations, but it is impossible to do this perfectly. Numbers in a computer have a finite range of possible values, and non-integer values have a limit on their possible accuracy. Nonetheless, except when results are out of range, you will encounter no surprises in using '`+`' for addition, '`-`' for subtraction, and '`*`' for multiplication.

Each C operator has a *precedence*, which is its rank in the grammatical order of the various operators. The operators with the highest precedence grab adjoining operands first; these expressions then become operands for operators of lower precedence. We give some information about precedence of operators in this chapter where we describe the operators; for the full explanation, see Chapter 9 [Binary Operator Grammar], page 41.

The arithmetic operators always *promote* their operands before operating on them. This means converting narrow integer data types to a wider data type (see Section 24.4 [Operand Promotions], page 157). If you are just learning C, don't worry about this yet.

Given two operands that have different types, most arithmetic operations convert them both to their *common type*. For instance, if one is `int` and the other is `double`, the common type is `double`. (That's because `double` can represent all the values that an `int` can hold, but not vice versa.) For the full details, see Section 24.5 [Common Type], page 157.

6.1 Basic Arithmetic

Basic arithmetic in C is done with the usual binary operators of algebra: addition ('`+`'), subtraction ('`-`'), multiplication ('`*`') and division ('`/`'). The unary operator '`-`' is used to change the sign of a number. The unary `+` operator also exists; it yields its operand unaltered.

'`/`' is the division operator, but dividing integers may not give the result you expect. Its value is an integer, which is not equal to the mathematical quotient when that is a fraction. Use '`%`' to get the corresponding integer remainder when necessary. See Section 6.5 [Division and Remainder], page 25. Floating-point division yields a value as close as possible to the mathematical quotient.

These operators use algebraic syntax with the usual algebraic precedence rule (see Chapter 9 [Binary Operator Grammar], page 41) that multiplication and division are done before addition and subtraction, but you can use parentheses to explicitly specify how the operators nest. They are left-associative (see Section 10.2 [Associativity and Ordering], page 43). Thus,

```
-a + b - c + d * e / f
```

is equivalent to

```
(((-a) + b) - c) + ((d * e) / f)
```

6.2 Integer Arithmetic

Each of the basic arithmetic operations in C has two variants for integers: *signed* and *unsigned*. The choice is determined by the data types of their operands.

Each integer data type in C is either *signed* or *unsigned*. A signed type can hold a range of positive and negative numbers, with zero near the middle of the range. An unsigned type can hold only nonnegative numbers; its range starts with zero and runs upward.

The most basic integer types are `int`, which normally can hold numbers from −2,147,483,648 to 2,147,483,647, and `unsigned int`, which normally can hold numbers from 0 to 4,294,967,295. (This assumes `int` is 32 bits wide, always true for GNU C on real computers but not always on embedded controllers.) See Section 11.1 [Integer Types], page 46, for full information about integer types.

When a basic arithmetic operation is given two signed operands, it does signed arithmetic. Given two unsigned operands, it does unsigned arithmetic.

If one operand is `unsigned int` and the other is `int`, the operator treats them both as unsigned. More generally, the common type of the operands determines whether the operation is signed or not. See Section 24.5 [Common Type], page 157.

Printing the results of unsigned arithmetic with `printf` using '`%d`' can produce surprising results for values far away from zero. Even though the rules above say that the computation was done with unsigned arithmetic, the printed result may appear to be signed!

The explanation is that the bit pattern resulting from addition, subtraction or multiplication is actually the same for signed and unsigned operations. The difference is only in the data type of the result, which affects the *interpretation* of the result bit pattern, and whether the arithmetic operation can overflow (see the next section).

But '`%d`' doesn't know its argument's data type. It sees only the value's bit pattern, and it is defined to interpret that as `signed int`. To print it as unsigned requires using '`%u`' instead of '`%d`'. See Section "Formatted Output" in *The GNU C Library Reference Manual*.

Arithmetic in C never operates directly on narrow integer types (those with fewer bits than `int`; Section 11.1.3 [Narrow Integers], page 47). Instead it "promotes" them to `int`. See Section 24.4 [Operand Promotions], page 157.

6.3 Integer Overflow

When the mathematical value of an arithmetic operation doesn't fit in the range of the data type in use, that's called *overflow*. When it happens in integer arithmetic, it is *integer overflow*.

Integer overflow happens only in arithmetic operations. Type conversion operations, by definition, do not cause overflow, not even when the result can't fit in its new type. See Section 11.1.4 [Integer Conversion], page 47.

Signed numbers use two's-complement representation, in which the most negative number lacks a positive counterpart (see Chapter 27 [Integers in Depth], page 191). Thus, the unary '`-`' operator on a signed integer can overflow.

6.3.1 Overflow with Unsigned Integers

Unsigned arithmetic in C ignores overflow; it produces the true result modulo the nth power of 2, where n is the number of bits in the data type. We say it "truncates" the true result to the lowest n bits.

A true result that is negative, when taken modulo the nth power of 2, yields a positive number. For instance,

```
unsigned int x = 1;
unsigned int y;

y = -x;
```

causes overflow because the negative number −1 can't be stored in an unsigned type. The actual result, which is −1 modulo the nth power of 2, is one less than the nth power of 2. That is the largest value that the unsigned data type can store. For a 32-bit `unsigned int`, the value is 4,294,967,295. See Section 27.2 [Maximum and Minimum Values], page 192.

Adding that number to itself, as here,

```
unsigned int z;

z = y + y;
```

ought to yield 8,489,934,590; however, that is again too large to fit, so overflow truncates the value to 4,294,967,294. If that were a signed integer, it would mean −2, which (not by coincidence) equals −1 `+` −1.

6.3.2 Overflow with Signed Integers

For signed integers, the result of overflow in C is *in principle* undefined, meaning that anything whatsoever could happen. Therefore, C compilers can do optimizations that treat the overflow case with total unconcern. (Since the result of overflow is undefined in principle, one cannot claim that these optimizations are erroneous.)

Watch out: These optimizations can do surprising things. For instance,

```
int i;
...
if (i < i + 1)
  x = 5;
```

could be optimized to do the assignment unconditionally, because the `if`-condition is always true if `i + 1` does not overflow.

GCC offers compiler options to control handling signed integer overflow. These options operate per module; that is, each module behaves according to the options it was compiled with.

These two options specify particular ways to handle signed integer overflow, other than the default way:

`-fwrapv` Make signed integer operations well-defined, like unsigned integer operations: they produce the n low-order bits of the true result. The highest of those n bits is the sign bit of the result. With `-fwrapv`, these out-of-range operations are not considered overflow, so (strictly speaking) integer overflow never happens.

The option `-fwrapv` enables some optimizations based on the defined values of out-of-range results. In GCC 8, it disables optimizations that are based on assuming signed integer operations will not overflow.

`-ftrapv` Generate a signal `SIGFPE` when signed integer overflow occurs. This terminates the program unless the program handles the signal. See Appendix E [Signals], page 224.

One other option is useful for finding where overflow occurs:

`-fsanitize=signed-integer-overflow`
: Output a warning message at run time when signed integer overflow occurs. This checks the '`+`', '`*`', and '`-`' operators. This takes priority over `-ftrapv`.

6.4 Mixed-Mode Arithmetic

Mixing integers and floating-point numbers in a basic arithmetic operation converts the integers automatically to floating point. In most cases, this gives exactly the desired results. But sometimes it matters precisely where the conversion occurs.

If `i` and `j` are integers, `(i + j) * 2.0` adds them as an integer, then converts the sum to floating point for the multiplication. If the addition causes an overflow, that is not equivalent to converting each integer to floating point and then adding the two floating point numbers. You can get the latter result by explicitly converting the integers, as in `((double) i + (double) j) * 2.0`. See Section 24.1 [Explicit Type Conversion], page 155.

Adding or multiplying several values, including some integers and some floating point, performs the operations left to right. Thus, `3.0 + i + j` converts `i` to floating point, then adds 3.0, then converts `j` to floating point and adds that. You can specify a different order using parentheses: `3.0 + (i + j)` adds `i` and `j` first and then adds that sum (converted to floating point) to 3.0. In this respect, C differs from other languages, such as Fortran.

6.5 Division and Remainder

Division of integers in C rounds the result to an integer. The result is always rounded towards zero.

```
 16 / 3  ⇒ 5
-16 / 3  ⇒ -5
 16 / -3 ⇒ -5
-16 / -3 ⇒ 5
```

To get the corresponding remainder, use the '`%`' operator:

```
 16 % 3  ⇒ 1
-16 % 3  ⇒ -1
 16 % -3 ⇒ 1
-16 % -3 ⇒ -1
```

'`%`' has the same operator precedence as '`/`' and '`*`'.

From the rounded quotient and the remainder, you can reconstruct the dividend, like this:

```
int
original_dividend (int divisor, int quotient, int remainder)
{
  return divisor * quotient + remainder;
}
```

To do unrounded division, use floating point. If only one operand is floating point, '`/`' converts the other operand to floating point.

```
16.0 / 3   ⇒ 5.333333333333333
```

```
16   / 3.0 ⇒ 5.333333333333333
16.0 / 3.0 ⇒ 5.333333333333333
16   / 3   ⇒ 5
```

The remainder operator '`%`' is not allowed for floating-point operands, because it is not needed. The concept of remainder makes sense for integers because the result of division of integers has to be an integer. For floating point, the result of division is a floating-point number, in other words a fraction, which will differ from the exact result only by a very small amount.

There are functions in the standard C library to calculate remainders from integral-values division of floating-point numbers. See Section "Remainder Functions" in *The GNU C Library Reference Manual*.

Integer division overflows in one specific case: dividing the smallest negative value for the data type (see Section 27.2 [Maximum and Minimum Values], page 192) by −1. That's because the correct result, which is the corresponding positive number, does not fit (see Section 6.3 [Integer Overflow], page 23) in the same number of bits. On some computers now in use, this always causes a signal `SIGFPE` (see Appendix E [Signals], page 224), the same behavior that the option `-ftrapv` specifies (see Section 6.3.2 [Signed Overflow], page 24).

Division by zero leads to unpredictable results—depending on the type of computer, it might cause a signal `SIGFPE`, or it might produce a numeric result.

Watch out: Make sure the program does not divide by zero. If you can't prove that the divisor is not zero, test whether it is zero, and skip the division if so.

6.6 Numeric Comparisons

There are two kinds of comparison operators: *equality* and *ordering*. Equality comparisons test whether two expressions have the same value. The result is a *truth value*: a number that is 1 for "true" and 0 for "false."

```
a == b   /* Test for equal.  */
a != b   /* Test for not equal.  */
```

The equality comparison is written `==` because plain `=` is the assignment operator.

Ordering comparisons test which operand is greater or less. Their results are truth values. These are the ordering comparisons of C:

```
a < b   /* Test for less-than.  */
a > b   /* Test for greater-than.  */
a <= b  /* Test for less-than-or-equal.  */
a >= b  /* Test for greater-than-or-equal.  */
```

For any integers `a` and `b`, exactly one of the comparisons `a < b`, `a == b` and `a > b` is true, just as in mathematics. However, if `a` and `b` are special floating point values (not ordinary numbers), all three can be false. See Section 28.3 [Special Float Values], page 194, and Section 28.4 [Invalid Optimizations], page 195.

6.7 Shift Operations

Shifting an integer means moving the bit values to the left or right within the bits of the data type. Shifting is defined only for integers. Here's the way to write it:

```
/* Left shift.  */
```

```
5 << 2 ⇒ 20

/* Right shift.  */
5 >> 2 ⇒ 1
```

The left operand is the value to be shifted, and the right operand says how many bits to shift it (the *shift count*). The left operand is promoted (see Section 24.4 [Operand Promotions], page 157), so shifting never operates on a narrow integer type; it's always either `int` or wider. The result of the shift operation has the same type as the promoted left operand.

The examples in this section use binary constants, starting with '`0b`' (see Section 12.1 [Integer Constants], page 52). They stand for 32-bit integers of type `int`.

6.7.1 Shifting Makes New Bits

A shift operation shifts towards one end of the number and has to generate new bits at the other end.

Shifting left one bit must generate a new least significant bit. It always brings in zero there. It is equivalent to multiplying by the appropriate power of 2. For example,

```
5 << 3      is equivalent to   5 * 2*2*2
-10 << 4    is equivalent to   -10 * 2*2*2*2
```

The meaning of shifting right depends on whether the data type is signed or unsigned (see Section 11.1.2 [Signed and Unsigned Types], page 46). For a signed data type, GNU C performs "arithmetic shift," which keeps the number's sign unchanged by duplicating the sign bit. For an unsigned data type, it performs "logical shift," which always shifts in zeros at the most significant bit.

In both cases, shifting right one bit is division by two, rounding towards negative infinity. For example,

```
(unsigned) 19 >> 2 ⇒ 4
(unsigned) 20 >> 2 ⇒ 5
(unsigned) 21 >> 2 ⇒ 5
```

For a negative left operand `a`, `a >> 1` is not equivalent to `a / 2`. Both operations divide by 2, but '`/`' rounds toward zero.

The shift count must be zero or greater. Shifting by a negative number of bits gives machine-dependent results.

6.7.2 Caveats for Shift Operations

Warning: If the shift count is greater than or equal to the width in bits of the promoted first operand, the results are machine-dependent. Logically speaking, the "correct" value would be either -1 (for right shift of a negative number) or 0 (in all other cases), but the actual result is whatever the machine's shift instruction does in that case. So unless you can prove that the second operand is not too large, write code to check it at run time.

Warning: Never rely on how the shift operators relate in precedence to other arithmetic binary operators. Programmers don't remember these precedences, and won't understand the code. Always use parentheses to explicitly specify the nesting, like this:

```
a + (b << 5)   /* Shift first, then add.  */
(a + b) << 5   /* Add first, then shift.  */
```

Note: according to the C standard, shifting of signed values isn't guaranteed to work properly when the value shifted is negative, or becomes negative during shifting. However, only pedants have a reason to be concerned about this; only computers with strange shift instructions could plausibly do this wrong. In GNU C, the operation always works as expected.

6.7.3 Shift Hacks

You can use the shift operators for various useful hacks. For example, given a date specified by day of the month `d`, month `m`, and year `y`, you can store the entire date in a single integer `date`:

The examples in this section use binary constants, starting with '`0b`' (see Section 12.1 [Integer Constants], page 52). They stand for 32-bit integers of type `int`.

```
unsigned int d = 12; /* 12 in binary is 0b1100.  */
unsigned int m = 6;             /* 6 in binary is 0b110.  */
unsigned int y = 1983;          /* 1983 in binary is 0b11110111111.  */
unsigned int date = (((y << 4) + m) << 5) + d;
                                /* Add 0b11110111111000000000
                                   and 0b11000000 and 0b1100.
                                   Sum is 0b11110111111011001100.  */
```

To extract the day, month, and year out of `date`, use a combination of shift and remainder:

```
/* 32 in binary is 0b100000.  */
/* Remainder dividing by 32 gives lowest 5 bits, 0b1100.  */
d = date % 32;
/* Shifting 5 bits right discards the day, leaving 0b111101111110110.
   Remainder dividing by 16 gives lowest remaining 4 bits, 0b110.  */
m = (date >> 5) % 16;
/* Shifting 9 bits right discards day and month,
   leaving 0b11110111111.  */
y = date >> 9;
```

`-1 << LOWBITS` is a clever way to make an integer whose `LOWBITS` lowest bits are all 0 and the rest are all 1. `-(1 << LOWBITS)` is equivalent to that, since negating a value is equivalent to multiplying it by −1.

6.8 Bitwise Operations

Bitwise operators operate on integers, treating each bit independently. They are not allowed for floating-point types.

As in the previous section, the examples in this section use binary constants, starting with '`0b`' (see Section 12.1 [Integer Constants], page 52). They stand for 32-bit integers of type `int`.

`~a` Unary operator for bitwise negation; this changes each bit of `a` from 1 to 0 or from 0 to 1.

```
~0b10101000 ⇒ 0b11111111111111111111111101010111
~0 ⇒ 0b11111111111111111111111111111111
~0b11111111111111111111111111111111 ⇒ 0
```

```
~ (-1) ⇒ 0
```

It is useful to remember that `~x + 1` equals `-x`, for integers, and `~x` equals `-x - 1`. The last example above shows this with −1 as *x*.

`a & b`
Binary operator for bitwise "and" or "conjunction." Each bit in the result is 1 if that bit is 1 in both `a` and `b`.

```
0b10101010 & 0b11001100 ⇒ 0b10001000
```

`a | b`
Binary operator for bitwise "or" ("inclusive or" or "disjunction"). Each bit in the result is 1 if that bit is 1 in either `a` or `b`.

```
0b10101010 | 0b11001100 ⇒ 0b11101110
```

`a ^ b`
Binary operator for bitwise "xor" ("exclusive or"). Each bit in the result is 1 if that bit is 1 in exactly one of `a` and `b`.

```
0b10101010 ^ 0b11001100 ⇒ 0b01100110
```

To understand the effect of these operators on signed integers, keep in mind that all modern computers use two's-complement representation (see Section 27.1 [Integer Representations], page 191) for negative integers. This means that the highest bit of the number indicates the sign; it is 1 for a negative number and 0 for a positive number. In a negative number, the value in the other bits *increases* as the number gets closer to zero, so that `0b111...111` is −1 and `0b100...000` is the most negative possible integer.

Warning: C defines a precedence ordering for the bitwise binary operators, but you should never rely on it. Likewise, you should never rely on how bitwise binary operators relate in precedence to the arithmetic and shift binary operators. Other programmers don't remember these aspects of C's precedence ordering; to make your programs clear, always use parentheses to explicitly specify the nesting among these operators.

For example, suppose `offset` is an integer that specifies the offset within shared memory of a table, except that its bottom few bits (`LOWBITS` says how many) are special flags. Here's how to get just that offset and add it to the base address.

```
shared_mem_base + (offset & (-1 << LOWBITS))
```

Thanks to the outer set of parentheses, we don't need to know whether '`&`' has higher precedence than '`+`'. Thanks to the inner set, we don't need to know whether '`&`' has higher precedence than '`<<`'. But we can rely on all unary operators to have higher precedence than any binary operator, so we don't need parentheses around the left operand of '`<<`'.

7 Assignment Expressions

As a general concept in programming, an *assignment* is a construct that stores a new value into a place where values can be stored—for instance, in a variable. Such places are called *lvalues* (see Section 7.2 [Lvalues], page 31) because they are locations that hold a value.

An assignment in C is an expression because it has a value; we call it an *assignment expression*. A simple assignment looks like

```
lvalue = value-to-store
```

We say it assigns the value of the expression *value-to-store* to the location *lvalue*, or that it stores *value-to-store* there. You can think of the "l" in "lvalue" as standing for "left," since that's what you put on the left side of the assignment operator.

However, that's not the only way to use an lvalue, and not all lvalues can be assigned to. To use the lvalue in the left side of an assignment, it has to be *modifiable*. In C, that means it was not declared with the type qualifier `const` (see Section 21.1 [const], page 128).

The value of the assignment expression is that of *lvalue* after the new value is stored in it. This means you can use an assignment inside other expressions. Assignment operators are right-associative so that

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

is equivalent to

```
x = (y = (z = 0));
```

This is the only useful way for them to associate; the other way,

```
((x = y) = z) = 0;
```

would be invalid since an assignment expression such as `x = y` is not a valid lvalue.

Warning: Write parentheses around an assignment if you nest it inside another expression, unless that containing expression is a comma-separated series or another assignment.

7.1 Simple Assignment

A *simple assignment expression* computes the value of the right operand and stores it into the lvalue on the left. Here is a simple assignment expression that stores 5 in `i`:

```
i = 5
```

We say that this is an *assignment to* the variable `i` and that it *assigns* `i` the value 5. It has no semicolon because it is an expression (so it has a value). Adding a semicolon at the end would make it a statement (see Section 19.1 [Expression Statement], page 103).

Here is another example of a simple assignment expression. Its operands are not simple, but the kind of assignment done here is simple assignment.

```
x[foo ()] = y + 6
```

A simple assignment with two different numeric data types converts the right operand value to the lvalue's type, if possible. It can convert any numeric type to any other numeric type.

Simple assignment is also allowed on some non-numeric types: pointers (see Chapter 14 [Pointers], page 62), structures (see Section 15.13 [Structure Assignment], page 82), and unions (see Section 15.14 [Unions], page 83).

Warning: Assignment is not allowed on arrays because there are no array values in C; C variables can be arrays, but these arrays cannot be manipulated as wholes. See Section 16.6 [Limitations of C Arrays], page 94.

See Section 24.2 [Assignment Type Conversions], page 155, for the complete rules about data types used in assignments.

7.2 Lvalues

An expression that identifies a memory space that holds a value is called an *lvalue*, because it is a location that can hold a value.

The standard kinds of lvalues are:

- A variable.
- A pointer-dereference expression (see Section 14.5 [Pointer Dereference], page 63) using unary '`*`', if its type is not a non-function type.
- A structure field reference (see Chapter 15 [Structures], page 74) using '`.`', if the structure value is an lvalue.
- A structure field reference using '`->`'. This is always an lvalue since '`->`' implies pointer dereference.
- A union alternative reference (see Section 15.14 [Unions], page 83), on the same conditions as for structure fields.
- An array-element reference using '`[...]`', if the array is an lvalue.
- A string constant (see Section 12.7 [String Constants], page 56).
- An array constructor (see Section 16.8 [Constructing Array Values], page 96).
- A structure or union constructor (see Section 15.17 [Structure Constructors], page 86).

If an expression's outermost operation is any other operator, that expression is not an lvalue. Thus, the variable `x` is an lvalue, but `x + 0` is not, even though these two expressions compute the same value (assuming `x` is a number).

It is rare that a structure value or an array value is not an lvalue, but that does happen—for instance, the result of a function call or a conditional operator can have a structure or array type, but is never an lvalue.

If an array is an lvalue, using the array in an expression still converts it automatically to a pointer to the zeroth element. The result of this conversion is not an lvalue. Thus, if the variable `a` is an array, you can't use `a` by itself as the left operand of an assignment. But you can assign to an element of `a`, such as `a[0]`. That is an lvalue since `a` is an lvalue.

7.3 Modifying Assignment

You can abbreviate the common construct

```
lvalue = lvalue + expression
```

as

```
lvalue += expression
```

This is known as a *modifying assignment*. For instance,

```
i = i + 5;
```

```
i += 5;
```

shows two statements that are equivalent. The first uses simple assignment; the second uses modifying assignment.

Modifying assignment works with any binary arithmetic operator. For instance, you can subtract something from an lvalue like this,

```
lvalue -= expression
```

or multiply it by a certain amount like this,

```
lvalue *= expression
```

or shift it by a certain amount like this.

```
lvalue <<= expression
lvalue >>= expression
```

In most cases, this feature adds no power to the language, but it provides substantial convenience. Also, when *lvalue* contains code that has side effects, the simple assignment performs those side effects twice, while the modifying assignment performs them once. For instance, suppose that the function `foo` has a side effect, perhaps changing static storage. This statement

```
x[foo ()] = x[foo ()] + 5;
```

calls `foo` twice. If `foo` operates on static variables, it could return a different value each time. If `foo ()` will return 1 the first time and 3 the second time, the effect could be to add `x[3]` and 5 and store the result in `x[1]`, or to add `x[1]` and 5 and store the result in `x[3]`. We don't know which of the two it will do, because C does not specify which call to `foo` is computed first.

Such a statement is not well defined, and shouldn't be used.

By contrast,

```
x[foo ()] += 5;
```

is well defined: it calls `foo` only once to determine which element of `x` to adjust, and it adjusts that element by adding 5 to it.

7.4 Increment and Decrement Operators

The operators '`++`' and '`--`' are the *increment* and *decrement* operators. When used on a numeric value, they add or subtract 1. We don't consider them assignments, but they are equivalent to assignments.

Using '`++`' or '`--`' as a prefix, before an lvalue, is called *preincrement* or *predecrement*. This adds or subtracts 1 and the result becomes the expression's value. For instance,

```
#include <stdio.h>   /* Declares printf. */

int
main (void)
{
  int i = 5;
  printf ("%d\n", i);
  printf ("%d\n", ++i);
  printf ("%d\n", i);
```

```
    return 0;
  }
```

prints lines containing ‘`5`’, ‘`6`’, and ‘`6`’ again. The expression `++i` increments `i` from 5 to 6, and has the value 6, so the output from `printf` on that line says ‘`6`’.

Using ‘`--`’ instead, for predecrement,

```
#include <stdio.h>   /* Declares printf. */

int
main (void)
{
  int i = 5;
  printf ("%d\n", i);
  printf ("%d\n", --i);
  printf ("%d\n", i);
  return 0;
}
```

prints three lines that contain (respectively) ‘`5`’, ‘`4`’, and again ‘`4`’.

7.5 Postincrement and Postdecrement

Using ‘`++`’ or ‘`--`’ *after* an lvalue does something peculiar: it gets the value directly out of the lvalue and *then* increments or decrements it. Thus, the value of `i++` is the same as the value of `i`, but `i++` also increments `i` “a little later.” This is called *postincrement* or *postdecrement*.

For example,

```
#include <stdio.h>   /* Declares printf. */

int
main (void)
{
  int i = 5;
  printf ("%d\n", i);
  printf ("%d\n", i++);
  printf ("%d\n", i);
  return 0;
}
```

prints lines containing ‘`5`’, again ‘`5`’, and ‘`6`’. The expression `i++` has the value 5, which is the value of `i` at the time, but it increments `i` from 5 to 6 just a little later.

How much later is “just a little later”? The compiler has some flexibility in deciding that. The rule is that the increment has to happen by the next *sequence point*; in simple cases, that means by the end of the statement. See Section 10.3 [Sequence Points], page 44.

Regardless of precisely where the compiled code increments the value of `i`, the crucial thing is that the value of `i++` is the value that `i` has *before* incrementing it.

If a unary operator precedes a postincrement or postdecrement expression, the post-whatever expression nests inside:

```
-a++    is equivalent to    -(a++)
```

In some cases, for instance this one, the other order would not even make sense; `-a` is not an lvalue, so it can't be incremented.

The most common use of postincrement is with arrays. Here's an example of using postincrement to access one element of an array and advance the index for the next access. Compare this with the example `avg_of_double` (see Section 4.2 [Array Example], page 14), which is almost the same but doesn't use postincrement for that.

```
double
avg_of_double_alt (int length, double input_data[])
{
  double sum = 0;
  int i;

  /* Fetch each element and add it into sum.  */
  for (i = 0; i < length;)
    /* Use the index i, then increment it.  */
    sum += input_data[i++];

  return sum / length;
}
```

7.6 Pitfall: Assignment in Subexpressions

In C, the order of computing parts of an expression is not fixed. Aside from a few special cases, the operations can be computed in any order. If one part of the expression has an assignment to `x` and another part of the expression uses `x`, the result is unpredictable because that use might be computed before or after the assignment.

Here's an example of ambiguous code:

```
x = 20;
printf ("%d %d\n", x, x = 4);
```

If the second argument, `x`, is computed before the third argument, `x = 4`, the second argument's value will be 20. If they are computed in the other order, the second argument's value will be 4.

Here's one way to make that code unambiguous:

```
y = 20;
printf ("%d %d\n", y, x = 4);
```

Here's another way, with the other meaning:

```
x = 4;
printf ("%d %d\n", x, x);
```

This issue applies to all kinds of assignments, and to the increment and decrement operators, which are equivalent to assignments. See Chapter 10 [Order of Execution], page 43, for more information about this.

However, it can be useful to write assignments inside an `if`-condition or `while`-test along with logical operators. See Section 8.3 [Logicals and Assignments], page 37.

7.7 Write Assignments in Separate Statements

It is often convenient to write an assignment inside an `if`-condition, but that can reduce the readability of the program. Here's an example of what to avoid:

```
if (x = advance (x))
  ...
```

The idea here is to advance `x` and test if the value is nonzero. However, readers might miss the fact that it uses '`=`' and not '`==`'. In fact, writing '`=`' where '`==`' was intended inside a condition is a common error, so GNU C can give warnings when '`=`' appears in a way that suggests it's an error.

It is much clearer to write the assignment as a separate statement, like this:

```
x = advance (x);
if (x != 0)
  ...
```

This makes it unmistakably clear that `x` is assigned a new value.

Another method is to use the comma operator (see Section 8.5 [Comma Operator], page 39), like this:

```
if (x = advance (x), x != 0)
  ...
```

However, putting the assignment in a separate statement is usually clearer (unless the assignment is very short), because it reduces nesting.

8 Execution Control Expressions

This chapter describes the C operators that combine expressions to control which of those expressions execute, or in which order.

8.1 Logical Operators

The *logical operators* combine truth values, which are normally represented in C as numbers. Any expression with a numeric value is a valid truth value: zero means false, and any other value means true. A pointer type is also meaningful as a truth value; a null pointer (which is zero) means false, and a non-null pointer means true (see Section 14.2 [Pointer Types], page 62). The value of a logical operator is always 1 or 0 and has type `int` (see Section 11.1 [Integer Types], page 46).

The logical operators are used mainly in the condition of an `if` statement, or in the end test in a `for` statement or `while` statement (see Chapter 19 [Statements], page 103). However, they are valid in any context where an integer-valued expression is allowed.

‘`! *exp*`’
Unary operator for logical “not.” The value is 1 (true) if *exp* is 0 (false), and 0 (false) if *exp* is nonzero (true).

Warning: If *exp* is anything but an lvalue or a function call, you should write parentheses around it.

‘`*left* && *right*`’
The logical “and” binary operator computes *left* and, if necessary, *right*. If both of the operands are true, the ‘`&&`’ expression gives the value 1 (true). Otherwise, the ‘`&&`’ expression gives the value 0 (false). If *left* yields a false value, that determines the overall result, so *right* is not computed.

‘`*left* || *right*`’
The logical “or” binary operator computes *left* and, if necessary, *right*. If at least one of the operands is true, the ‘`||`’ expression gives the value 1 (which is true). Otherwise, the ‘`||`’ expression gives the value 0 (false). If *left* yields a true value, that determines the overall result, so *right* is not computed.

Warning: Never rely on the relative precedence of ‘`&&`’ and ‘`||`’. When you use them together, always use parentheses to specify explicitly how they nest, as shown here:

```
if ((r != 0 && x % r == 0)
    ||
    (s != 0 && x % s == 0))
```

8.2 Logical Operators and Comparisons

The most common thing to use inside the logical operators is a comparison. Conveniently, ‘`&&`’ and ‘`||`’ have lower precedence than comparison operators and arithmetic operators, so we can write expressions like this without parentheses and get the nesting that is natural: two comparison operations that must both be true.

```
if (r != 0 && x % r == 0)
```

This example also shows how it is useful that ‘`&&`’ guarantees to skip the right operand if the left one turns out false. Because of that, this code never tries to divide by zero.

This is equivalent:

```
if (r && x % r == 0)
```

A truth value is simply a number, so using `r` as a truth value tests whether it is nonzero. But `r`'s meaning as an expression is not a truth value—it is a number to divide by. So it is clearer style to write the explicit `!= 0`.

Here's another equivalent way to write it:

```
if (!(r == 0) && x % r == 0)
```

This illustrates the unary '`!`' operator, as well as the need to write parentheses around its operand.

8.3 Logical Operators and Assignments

There are cases where assignments nested inside the condition can actually make a program *easier* to read. Here is an example using a hypothetical type `list` which represents a list; it tests whether the list has at least two links, using hypothetical functions, `nonempty` which is true if the argument is a nonempty list, and `list_next` which advances from one list link to the next. We assume that a list is never a null pointer, so that the assignment expressions are always "true."

```
if (nonempty (list)
    && (temp1 = list_next (list))
    && nonempty (temp1)
    && (temp2 = list_next (temp1)))
  ...  /* use temp1 and temp2 */
```

Here we take advantage of the '`&&`' operator to avoid executing the rest of the code if a call to `nonempty` returns "false." The only natural place to put the assignments is among those calls.

It would be possible to rewrite this as several statements, but that could make it much more cumbersome. On the other hand, when the test is even more complex than this one, splitting it into multiple statements might be necessary for clarity.

If an empty list is a null pointer, we can dispense with calling `nonempty`:

```
if (list
    && (temp1 = list_next (list))
    && (temp2 = list_next (temp1)))
  ...
```

8.4 Conditional Expression

C has a conditional expression that selects one of two expressions to compute and get the value from. It looks like this:

```
condition ? iftrue : iffalse
```

8.4.1 Rules for the Conditional Operator

The first operand, *condition*, should be a value that can be compared with zero—a number or a pointer. If it is true (nonzero), then the conditional expression computes *iftrue* and its value becomes the value of the conditional expression. Otherwise the conditional expression computes *iffalse* and its value becomes the value of the conditional expression. The conditional expression always computes just one of *iftrue* and *iffalse*, never both of them.

Here's an example: the absolute value of a number `x` can be written as `(x >= 0 ? x : -x)`.

Warning: The conditional expression has rather low syntactic precedence. Except when the conditional expression is used as an argument in a function call, write parentheses around it. For clarity, always write parentheses around it if it extends across more than one line.

Warning: Assignment operators and the comma operator (see Section 8.5 [Comma Operator], page 39) have lower precedence than conditional expressions, so write parentheses around those when they appear inside a conditional expression. See Chapter 10 [Order of Execution], page 43.

Warning: When nesting a conditional expression within another conditional expression, unless a pair of matching delimiters surrounds the inner conditional expression for some other reason, write parentheses around it:

```
((foo > 0 ? test1 : test2) ? (ifodd (foo) ? 5 : 10)
                           : (ifodd (whatever) ? 5 : 10));
```

In the first operand, those parentheses are necessary to prevent incorrect parsing. In the second and third operands, the computer may not need the parentheses, but they will help human beings.

8.4.2 Conditional Operator Branches

We call *iftrue* and *iffalse* the *branches* of the conditional.

The two branches should normally have the same type, but a few exceptions are allowed. If they are both numeric types, the conditional converts both to their common type (see Section 24.5 [Common Type], page 157).

With pointers (see Chapter 14 [Pointers], page 62), the two values can be pointers to nearly compatible types (see Chapter 23 [Compatible Types], page 154). In this case, the result type is a similar pointer whose target type combines all the type qualifiers (see Chapter 21 [Type Qualifiers], page 128) of both branches.

If one branch has type `void *` and the other is a pointer to an object (not to a function), the conditional converts the latter to `void *`.

If one branch is an integer constant with value zero and the other is a pointer, the conditional converts zero to the pointer's type.

In GNU C, you can omit *iftrue* in a conditional expression. In that case, if *condition* is nonzero, its value becomes the value of the conditional expression, after conversion to the common type. Thus,

```
x ? : y
```

has the value of `x` if that is nonzero; otherwise, the value of `y`.

Omitting *iftrue* is useful when *condition* has side effects. In that case, writing that expression twice would carry out the side effects twice, but writing it once does them just once. For example, if we suppose that the function `next_element` advances a pointer variable to point to the next element in a list and returns the new pointer,

```
next_element () ? : default_pointer
```

is a way to advance the pointer and use its new value if it isn't null, but use `default_pointer` if that is null. We cannot do it this way,

```
next_element () ? next_element () : default_pointer
```

because that would advance the pointer a second time.

8.5 Comma Operator

The comma operator stands for sequential execution of expressions. The value of the comma expression comes from the last expression in the sequence; the previous expressions are computed only for their side effects. It looks like this:

```
exp1, exp2 ...
```

You can bundle any number of expressions together this way, by putting commas between them.

8.5.1 The Uses of the Comma Operator

With commas, you can put several expressions into a place that allows one expression—for example, in the header of a `for` statement. This statement

```
for (i = 0, j = 10, k = 20; i < n; i++)
```

contains three assignment expressions, to initialize `i`, `j` and `k`. The syntax of `for` requires just one expression for initialization; to include three assignments, we use commas to bundle them into a single larger expression, `i = 0, j = 10, k = 20`. This technique is also useful in the loop-advance expression, the last of the three inside the `for` parentheses.

In the `for` statement and the `while` statement (see Section 19.6 [Loop Statements], page 105), a comma provides a way to perform some side effect before the loop-exit test. For example,

```
while (printf ("At the test, x = %d\n", x), x != 0)
```

8.5.2 Clean Use of the Comma Operator

Always write parentheses around a series of comma operators, except when it is at top level in an expression statement, or within the parentheses of an `if`, `for`, `while`, or `switch` statement (see Chapter 19 [Statements], page 103). For instance, in

```
for (i = 0, j = 10, k = 20; i < n; i++)
```

the commas between the assignments are clear because they are between a parenthesis and a semicolon.

The arguments in a function call are also separated by commas, but that is not an instance of the comma operator. Note the difference between

```
foo (4, 5, 6)
```

which passes three arguments to `foo` and

```
foo ((4, 5, 6))
```

which uses the comma operator and passes just one argument (with value 6).

Warning: Don't use the comma operator within an argument of a function unless it makes the code more readable. When you do so, don't put part of another argument on the same line. Instead, add a line break to make the parentheses around the comma operator easier to see, like this.

```
foo ((mumble (x, y), frob (z)),
     *p)
```

8.5.3 When Not to Use the Comma Operator

You can use a comma in any subexpression, but in most cases it only makes the code confusing, and it is clearer to raise all but the last of the comma-separated expressions to a higher level. Thus, instead of this:

```
x = (y += 4, 8);
```

it is much clearer to write this:

```
y += 4, x = 8;
```

or this:

```
y += 4;
x = 8;
```

Use commas only in the cases where there is no clearer alternative involving multiple statements.

By contrast, don't hesitate to use commas in the expansion in a macro definition. The trade-offs of code clarity are different in that case, because the *use* of the macro may improve overall clarity so much that the ugliness of the macro's *definition* is a small price to pay. See Section 26.5 [Macros], page 167.

9 Binary Operator Grammar

Binary operators are those that take two operands, one on the left and one on the right.

All the binary operators in C are syntactically left-associative. This means that `a op b op c` means `(a op b) op c`. However, the only operators you should repeat in this way without parentheses are '`+`', '`-`', '`*`' and '`/`', because those cases are clear from algebra. So it is OK to write `a + b + c` or `a - b - c`, but never `a == b == c` or `a % b % c`. For those operators, use explicit parentheses to show how the operations nest.

Each C operator has a *precedence*, which is its rank in the grammatical order of the various operators. The operators with the highest precedence grab adjoining operands first; these expressions then become operands for operators of lower precedence.

The precedence order of operators in C is fully specified, so any combination of operations leads to a well-defined nesting. We state only part of the full precedence ordering here because it is bad practice for C code to depend on the other cases. For cases not specified in this chapter, always use parentheses to make the nesting explicit.[1]

Clean code can depend on this subsequence of the precedence ordering (stated from highest precedence to lowest):

1. Postfix operations: access to a field or alternative ('`.`' and '`->`'), array subscripting, function calls, and unary postfix operators.
2. Unary prefix operations.
3. Multiplication, division, and remainder (they have the same precedence).
4. Addition and subtraction (they have the same precedence).
5. Comparisons—but watch out!
6. Logical operations '`&&`' and '`||`'—but watch out!
7. Conditional expression with '`?`' and '`:`'.
8. Assignments.
9. Sequential execution (the comma operator, '`,`').

Two of the lines in the above list say "but watch out!" That means that the line covers operations with subtly different precedence. When you use two comparison operations together, don't depend on the grammar of C to control how they nest. Instead, always use parentheses to show their nesting.

You can let several '`&&`' operations associate, or several '`||`' operations, but always use parentheses to show how '`&&`' and '`||`' nest with each other. See Section 8.1 [Logical Operators], page 36.

There is one other precedence ordering that clean code can depend on:

1. Unary postfix operations.
2. Bitwise and shift operations—but watch out!
3. Conditional expression with '`?`' and '`:`'.

---

[1] Personal note from Richard Stallman: I wrote GCC without remembering anything about the C precedence order beyond what's stated here. I studied the full precedence table to write the parser, and promptly forgot it again. If you need to look up the full precedence order to understand some C code, add enough parentheses so nobody else needs to do that.

The caveat for bitwise and shift operations is like that for logical operators: you can let multiple uses of one bitwise operation associate, but always use parentheses to control nesting of dissimilar operations.

These lists do not specify any precedence ordering between the bitwise and shift operations of the second list and the binary operations above conditional expressions in the first list. When they come together, parenthesize them. See Section 6.8 [Bitwise Operations], page 28.

10 Order of Execution

The order of execution of a C program is not always obvious, and not necessarily predictable. This chapter describes what you can count on.

10.1 Reordering of Operands

The C language does not necessarily carry out operations within an expression in the order they appear in the code. For instance, in this expression,

```
foo () + bar ()
```

`foo` might be called first or `bar` might be called first. If `foo` updates a datum and `bar` uses that datum, the results can be unpredictable.

The unpredictable order of computation of subexpressions also makes a difference when one of them contains an assignment. We already saw this example of bad code,

```
x = 20;
printf ("%d %d\n", x, x = 4);
```

in which the second argument, `x`, has a different value depending on whether it is computed before or after the assignment in the third argument.

10.2 Associativity and Ordering

The bitwise binary operators, `&`, `|` and `^`, are associative. The arithmetic binary operators `+` and `*` are associative if the operand type is unsigned. An associative binary operator, when used repeatedly, can combine any number of operands. The operands' values may be computed in any order, and since the operation is associative, they can be combined in any order too.

Thus, given four functions that return `unsigned int`, calling them and adding their results as here

```
(foo () + bar ()) + (baz () + quux ())
```

may add up the results in any order.

By contrast, arithmetic on signed integers is not always associative because there is the possibility of overflow (see Section 6.3 [Integer Overflow], page 23). Thus, the additions must be done in the order specified, obeying parentheses (or left-association in the absence of parentheses). That means computing `(foo () + bar ())` and `(baz () + quux ())` first (in either order), then adding the two.

The same applies to arithmetic on floating-point values, since that too is not really associative. However, the GCC option `-funsafe-math-optimizations` allows the compiler to change the order of calculation when an associative operation (associative in exact mathematics) combines several operands. The option takes effect when compiling a module (see Chapter 29 [Compilation], page 211). Changing the order of association can enable GCC to optimize the floating-point operations better.

In all these examples, the four function calls can be done in any order. There is no right or wrong about that.

10.3 Sequence Points

There are some points in the code where C makes limited guarantees about the order of operations. These are called *sequence points*. Here is where they occur:

- At the end of a *full expression*; that is to say, an expression that is not part of a larger expression. All side effects specified by that expression are carried out before execution moves on to subsequent code.
- At the end of the first operand of certain operators: '`,`', '`&&`', '`||`', and '`?:`'. All side effects specified by that expression are carried out before any execution of the next operand.

  The commas that separate arguments in a function call are *not* comma operators, and they do not create sequence points. The sequence-point rule for function arguments and the rule for operands (see Section 10.5 [Ordering of Operands], page 45) are different.
- Just before calling a function. All side effects specified by the argument expressions are carried out before calling the function.

  If the function to be called is not constant—that is, if it is computed by an expression—all side effects in that expression are carried out before calling the function.

The ordering imposed by a sequence point applies locally to a limited range of code, as stated above in each case. For instance, the ordering imposed by the comma operator does not apply to code outside the operands of that comma operator. Thus, in this code,

```
(x = 5, foo (x)) + x * x
```

the sequence point of the comma operator orders `x = 5` before `foo (x)`, but `x * x` could be computed before or after them.

10.4 Postincrement and Ordering

The ordering requirements for the postincrement and postdecrement operations (see Section 7.5 [Postincrement/Postdecrement], page 33) are loose: those side effects must happen "a little later," before the next sequence point. That still leaves room for various orders that give different results. In this expression,

```
z = x++ - foo ()
```

it's unpredictable whether `x` gets incremented before or after calling the function `foo`. If `foo` refers to `x`, it might see the old value or it might see the incremented value.

In this perverse expression,

```
x = x++
```

`x` will certainly be incremented but the incremented value may be replaced with the old value. That's because the incrementation and the assignment may occur in either order. If the incrementation of `x` occurs after the assignment to `x`, the incremented value will remain in place. But if the incrementation happens first, the assignment will put the not-yet-incremented value back into `x`, so the expression as a whole will leave `x` unchanged.

The conclusion: **avoid such expressions**. Take care, when you use postincrement and postdecrement, that the specific expression you use is not ambiguous as to order of execution.

10.5 Ordering of Operands

Operands and arguments can be computed in any order, but there are limits to this intermixing in GNU C:

- The operands of a binary arithmetic operator can be computed in either order, but they can't be intermixed: one of them has to come first, followed by the other. Any side effects in the operand that's computed first are executed before the other operand is computed.
- That applies to assignment operators too, except that, in simple assignment, the previous value of the left operand is unused.
- The arguments in a function call can be computed in any order, but they can't be intermixed. Thus, one argument is fully computed, then another, and so on until they have all been done. Any side effects in one argument are executed before computation of another argument begins.

These rules don't cover side effects caused by postincrement and postdecrement operators—those can be deferred up to the next sequence point.

If you want to get pedantic, the fact is that GCC can reorder the computations in many other ways provided that it doesn't alter the result of running the program. However, because it doesn't alter the result of running the program, it is negligible, unless you are concerned with the values in certain variables at various times as seen by other processes. In those cases, you should use `volatile` to prevent optimizations that would make them behave strangely. See Section 21.2 [volatile], page 129.

10.6 Optimization and Ordering

Sequence points limit the compiler's freedom to reorder operations arbitrarily, but optimizations can still reorder them if the compiler concludes that this won't alter the results. Thus, in this code,

```
x++;
y = z;
x++;
```

there is a sequence point after each statement, so the code is supposed to increment `x` once before the assignment to `y` and once after. However, incrementing `x` has no effect on `y` or `z`, and setting `y` can't affect `x`, so the code could be optimized into this:

```
y = z;
x += 2;
```

Normally that has no effect except to make the program faster. But there are special situations where it can cause trouble due to things that the compiler cannot know about, such as shared memory. To limit optimization in those places, use the `volatile` type qualifier (see Section 21.2 [volatile], page 129).

11 Primitive Data Types

This chapter describes all the primitive data types of C—that is, all the data types that aren't built up from other types. They include the types `int` and `double` that we've already covered.

These types are all made up of bytes (see Chapter 3 [Storage], page 12).

11.1 Integer Data Types

Here we describe all the integer types and their basic characteristics. See Chapter 27 [Integers in Depth], page 191, for more information about the bit-level integer data representations and arithmetic.

11.1.1 Basic Integers

Integer data types in C can be signed or unsigned. An unsigned type can represent only positive numbers and zero. A signed type can represent both positive and negative numbers, in a range spread almost equally on both sides of zero.

Aside from signedness, the integer data types vary in size: how many bytes long they are. The size determines the range of integer values the type can hold.

Here's a list of the signed integer data types, with the sizes they have on most computers. Each has a corresponding unsigned type; see Section 11.1.2 [Signed and Unsigned Types], page 46.

`signed char`	One byte (8 bits). This integer type is used mainly for integers that represent characters, usually as elements of arrays or fields of other data structures.
`short` `short int`	Two bytes (16 bits).
`int`	Four bytes (32 bits).
`long` `long int`	Four bytes (32 bits) or eight bytes (64 bits), depending on the platform. Typically it is 32 bits on 32-bit computers and 64 bits on 64-bit computers, but there are exceptions.
`long long` `long long int`	Eight bytes (64 bits). Supported in GNU C in the 1980s, and incorporated into standard C as of ISO C99.

You can omit `int` when you use `long` or `short`. This is harmless and customary.

11.1.2 Signed and Unsigned Types

An unsigned integer type can represent only positive numbers and zero. A signed type can represent both positive and negative numbers, in a range spread almost equally on both sides of zero. For instance, `unsigned char` holds numbers from 0 to 255 (on most computers), while `signed char` holds numbers from -128 to 127. Each of these types holds 256 different possible values, since they are both 8 bits wide.

Write `signed` or `unsigned` before the type keyword to specify a signed or an unsigned type. However, the integer types other than `char` are signed by default; with them, `signed` is a no-op.

Plain `char` may be signed or unsigned; this depends on the compiler, the machine in use, and its operating system. It is not *the same type* as either `signed char` or `unsigned char`, but it is always equivalent to one of those two.

In many programs, it makes no difference whether the type `char` is signed. When signedness does matter for a certain vslue, don't leave it to chance; declare it as `signed char` or `unsigned char` instead.[1]

11.1.3 Narrow Integers

The types that are narrower than `int` are rarely used for ordinary variables—we declare them `int` instead. This is because C converts those narrower types to `int` for any arithmetic. There is literally no reason to declare a local variable `char`, for instance.

In particular, if the value is really a character, you should declare the variable `int`. Not `char`! Using that narrow type can force the compiled code to truncate values to `char` before conversion, which is a waste. Furthermore, some functions return either a character value or −1 for "no character." Using type `int` makes it possible to distinguish −1 from any character, by sign.

The narrow integer types are useful as parts of other objects, such as arrays and structures. Compare these array declarations, whose sizes on 32-bit processors are shown:

```
signed char ac[1000];    /* 1000 bytes */
short as[1000];          /* 2000 bytes */
int ai[1000];            /* 4000 bytes */
long long all[1000];     /* 8000 bytes */
```

In addition, character strings must be made up of `char`s, because that's what all the standard library string functions expect. Thus, array `ac` could be used as a character string, but the others could not be.

11.1.4 Conversion among Integer Types

C converts between integer types implicitly in many situations. It converts the narrow integer types, `char` and `short`, to `int` whenever they are used in arithmetic. Assigning a new value to an integer variable (or other lvalue) converts the value to the variable's type.

You can also convert one integer type to another explicitly with a *cast* operator. See Section 24.1 [Explicit Type Conversion], page 155.

The process of conversion to a wider type is straightforward: the value is unchanged. The only exception is when converting a negative value (in a signed type, obviously) to a wider unsigned type. In that case, the result is a positive value with the same bits (see Chapter 27 [Integers in Depth], page 191), padded on the left with zeros.

Converting to a narrower type, also called *truncation*, involves discarding some of the value's bits. This is not considered overflow (see Section 6.3 [Integer Overflow], page 23) because loss of significant bits is a normal consequence of truncation. Likewise for conversion between signed and unsigned types of the same width.

---

[1] Personal note from Richard Stallman: Eating with hackers at a fish restaurant, I ordered arctic char. When my meal arrived, I noted that the chef had not signed it. So I told other hackers, "This char is unsigned—I wanted a signed char!"

More information about conversion for assignment is in Section 24.2 [Assignment Type Conversions], page 155. For conversion for arithmetic, see Section 24.3 [Argument Promotions], page 156.

11.1.5 Boolean Type

The unsigned integer type `bool` holds truth values: its possible values are 0 and 1. Converting any nonzero value to `bool` results in 1. For example:

```
bool a = 0;
bool b = 1;
bool c = 4; /* Stores the value 1 in c.  */
```

Unlike `int`, `bool` is not a keyword. It is defined in the header file `stdbool.h`.

11.1.6 Integer Variations

The integer types of C have standard *names*, but what they *mean* varies depending on the kind of platform in use: which kind of computer, which operating system, and which compiler. It may even depend on the compiler options used.

Plain `char` may be signed or unsigned; this depends on the platform, too. Even for GNU C, there is no general rule.

In theory, all of the integer types' sizes can vary. `char` is always considered one "byte" for C, but it is not necessarily an 8-bit byte; on some platforms it may be more than 8 bits. `short int` and `int` are at least two bytes long (it may be longer). `long int` is at least four bytes long, and `long long int` at least eight bytes long.

It is possible that in the future GNU C will support platforms where `int` is 64 bits long. In practice, however, on today's real computers, there is little variation; you can rely on the table given previously (see Section 11.1.1 [Basic Integers], page 46).

To be completely sure of the size of an integer type, use the types `int16_t`, `int32_t` and `int64_t`. Their corresponding unsigned types add '`u`' at the front: `uint16_t`, `uint32_t` and `uint64_t`. To define all these types, include the header file `stdint.h`.

The GNU C Compiler can compile for some embedded controllers that use two bytes for `int`. On some, `int` is just one "byte," and so is `short int`—but that "byte" may contain 16 bits or even 32 bits. These processors can't support an ordinary operating system (they may have their own specialized operating systems), and most C programs do not try to support them.

11.2 Floating-Point Data Types

Floating point is the binary analogue of scientific notation: internally it represents a number as a fraction and a binary exponent; the value is that fraction multiplied by the specified power of 2. (The C standard nominally permits other bases, but in GNU C the base is always 2.)

For instance, to represent 6, the fraction would be 0.75 and the exponent would be 3; together they stand for the value $0.75 * 2^3$, meaning `0.75 * 8`. The value 1.5 would use 0.75

as the fraction and 1 as the exponent. The value 0.75 would use 0.75 as the fraction and 0 as the exponent. The value 0.375 would use 0.75 as the fraction and −1 as the exponent.

These binary exponents are used by machine instructions. You can write a floating-point constant this way if you wish, using hexadecimal; but normally we write floating-point numbers in decimal (base 10). See Section 12.3 [Floating Constants], page 53.

C has three floating-point data types:

`double` “Double-precision” floating point, which uses 64 bits. This is the normal floating-point type, and modern computers normally do their floating-point computations in this type, or some wider type. Except when there is a special reason to do otherwise, this is the type to use for floating-point values.

`float` “Single-precision” floating point, which uses 32 bits. It is useful for floating-point values stored in structures and arrays, to save space when the full precision of `double` is not needed. In addition, single-precision arithmetic is faster on some computers, and occasionally that is useful. But not often—most programs don’t use the type `float`.

C would be cleaner if `float` were the name of the type we use for most floating-point values; however, for historical reasons, that’s not so.

`long double`

“Extended-precision” floating point is either 80-bit or 128-bit precision, depending on the machine in use. On some machines, which have no floating-point format wider than `double`, this is equivalent to `double`.

Floating-point arithmetic raises many subtle issues. See Chapter 28 [Floating Point in Depth], page 193, for more information.

11.3 Complex Data Types

Complex numbers can include both a real part and an imaginary part. The numeric constants covered above have real-numbered values. An imaginary-valued constant is an ordinary real-valued constant followed by ‘`i`’.

To declare numeric variables as complex, use the `_Complex` keyword.[2] The standard C complex data types are floating point,

```
_Complex float foo;
_Complex double bar;
_Complex long double quux;
```

but GNU C supports integer complex types as well.

Since `_Complex` is a keyword just like `float` and `double` and `long`, the keywords can appear in any order, but the order shown above seems most logical.

GNU C supports constants for complex values; for instance, `4.0 + 3.0i` has the value 4 + 3i as type `_Complex double`. ‘`j`’ is equivalent to ‘`i`’, as a numeric suffix. See Section 12.4 [Imaginary Constants], page 54.

---

[2] For compatibility with older versions of GNU C, the keyword `__complex__` is also allowed. Going forward, however, use the new `_Complex` keyword as defined in ISO C11.

To pull the real and imaginary parts of the number back out, GNU C provides the keywords `__real__` and `__imag__`:

```
_Complex double foo = 4.0 + 3.0i;

double a = __real__ foo; /* a is now 4.0. */
double b = __imag__ foo; /* b is now 3.0. */
```

Standard C does not include these keywords, and instead relies on functions defined in `complex.h` for accessing the real and imaginary parts of a complex number: `crealf`, `creal`, and `creall` extract the real part of a float, double, or long double complex number, respectively; `cimagf`, `cimag`, and `cimagl` extract the imaginary part.

GNU C also defines '`~`' as an operator for complex conjugation, which means negating the imaginary part of a complex number:

```
_Complex double foo = 4.0 + 3.0i;
_Complex double bar = ~foo; /* bar is now 4.0 − 3.0i. */
```

For standard C compatibility, you can use the appropriate library function: `conjf`, `conj`, or `conjl`.

11.4 The Void Type

The data type `void` is a dummy—it allows no operations. It really means "no value at all." When a function is meant to return no value, we write `void` for its return type. Then `return` statements in that function should not specify a value (see Section 19.5 [return Statement], page 105). Here's an example:

```
void
print_if_positive (double x, double y)
{
  if (x <= 0)
    return;
  if (y <= 0)
    return;
  printf ("Next point is (%f,%f)\n", x, y);
}
```

A `void`-returning function is comparable to what some other languages (for instance, Fortran and Pascal) call a "procedure" instead of a "function."

11.5 Other Data Types

Beyond the primitive types, C provides several ways to construct new data types. For instance, you can define *pointers*, values that represent the addresses of other data (see Chapter 14 [Pointers], page 62). You can define *structures*, as in many other languages (see Chapter 15 [Structures], page 74), and *unions*, which define multiple ways to interpret the contents of the same memory space (see Section 15.14 [Unions], page 83). *Enumerations* are collections of named integer codes (see Chapter 17 [Enumeration Types], page 99).

Array types in C are used for allocating space for objects, but C does not permit operating on an array value as a whole. See Chapter 16 [Arrays], page 91.

11.6 Type Designators

Some C constructs require a way to designate a specific data type independent of any particular variable or expression which has that type. The way to do this is with a *type designator*. The constructs that need one include casts (see Section 24.1 [Explicit Type Conversion], page 155) and `sizeof` (see Chapter 13 [Type Size], page 60).

We also use type designators to talk about the type of a value in C, so you will see many type designators in this manual. When we say, “The value has type `int`,” `int` is a type designator.

To make the designator for any type, imagine a variable declaration for a variable of that type and delete the variable name and the final semicolon.

For example, to designate the type of full-word integers, we start with the declaration for a variable `foo` with that type, which is this:

```
int foo;
```

Then we delete the variable name `foo` and the semicolon, leaving `int`—exactly the keyword used in such a declaration. Therefore, the type designator for this type is `int`.

What about long unsigned integers? From the declaration

```
unsigned long int foo;
```

we determine that the designator is `unsigned long int`.

Following this procedure, the designator for any primitive type is simply the set of keywords which specifies that type in a declaration. The same is true for structure types, union types, and enumeration types.

Designators for pointer types do follow the rule of deleting the variable name and semicolon, but the result is not so simple. See Section 14.4 [Pointer Type Designators], page 63, as part of the chapter about pointers. See Section 16.4 [Array Type Designators], page 93, for designators for array types.

To understand what type a designator stands for, imagine a variable name inserted into the right place in the designator to make a valid declaration. What type would that variable be declared as? That is the type the designator designates.

12 Constants

A *constant* is an expression that stands for a specific value by explicitly representing the desired value. C allows constants for numbers, characters, and strings. We have already seen numeric and string constants in the examples.

12.1 Integer Constants

An integer constant consists of a number to specify the value, followed optionally by suffix letters to specify the data type.

The simplest integer constants are numbers written in base 10 (decimal), such as `5`, `77`, and `403`. A decimal constant cannot start with the character '`0`' (zero) because that makes the constant octal.

You can get the effect of a negative integer constant by putting a minus sign at the beginning. In grammatical terms, that is an arithmetic expression rather than a constant, but it behaves just like a true constant.

Integer constants can also be written in octal (base 8), hexadecimal (base 16), or binary (base 2). An octal constant starts with the character '`0`' (zero), followed by any number of octal digits ('`0`' to '`7`'):

```
0      // zero
077    // 63
0403   // 259
```

Pedantically speaking, the constant `0` is an octal constant, but we can think of it as decimal; it has the same value either way.

A hexadecimal constant starts with '`0x`' (upper or lower case) followed by hex digits ('`0`' to '`9`', as well as '`a`' through '`f`' in upper or lower case):

```
0xff   // 255
0XA0   // 160
0xffFF // 65535
```

A binary constant starts with '`0b`' (upper or lower case) followed by bits (each represented by the characters '`0`' or '`1`'):

```
0b101  // 5
```

Binary constants are a GNU C extension, not part of the C standard.

Sometimes a space is needed after an integer constant to avoid lexical confusion with the following tokens. See Section 12.5 [Invalid Numbers], page 55.

12.2 Integer Constant Data Types

The type of an integer constant is normally `int`, if the value fits in that type, but here are the complete rules. The type of an integer constant is the first one in this sequence that can properly represent the value,

1. `int`
2. `unsigned int`
3. `long int`

4. `unsigned long int`
5. `long long int`
6. `unsigned long long int`

and that isn't excluded by the following rules.

If the constant has '`l`' or '`L`' as a suffix, that excludes the first two types (those that are not `long`).

If the constant has '`ll`' or '`LL`' as a suffix, that excludes first four types (those that are not `long long`).

If the constant has '`u`' or '`U`' as a suffix, that excludes the signed types.

Otherwise, if the constant is decimal (not binary, octal, or hexadecimal), that excludes the unsigned types.

Here are some examples of the suffixes.

```
3000000000u      // three billion as unsigned int.
0LL              // zero as a long long int.
0403l            // 259 as a long int.
2147483648       // This is of type long long int.
                 // on typical 32-bit machines,
                 // since it won't fit in 32 bits as a signed number.
2147483648U      // This is of type unsigned int,
                 // since it fits in 32 unsigned bits.
```

Suffixes in integer constants are rarely used. When the precise type is important, it is cleaner to convert explicitly (see Section 24.1 [Explicit Type Conversion], page 155).

See Section 11.1 [Integer Types], page 46.

12.3 Floating-Point Constants

A floating-point decimal constant must have either a decimal point, an exponent-of-ten, or both; they distinguish it from an integer constant. Just adding the floating-point suffix, '`f`', to an integer does not make a valid floating-point constant, and adding '`l`' would instead make it a long integer.

To indicate an exponent, write '`e`' or '`E`'. The exponent value follows. It is always written as a decimal number; it can optionally start with a sign. The exponent *n* means to multiply the constant's value by ten to the *n*th power.

Thus, '`1500.0`', '`15e2`', '`15e+2`', '`15.0e2`', '`1.5e+3`', '`.15e4`', and '`15000e-1`' are six ways of writing a floating-point number whose value is 1500. They are all equivalent in principle.

Here are more examples with decimal points:

```
1.0
1000.
3.14159
.05
.0005
```

For each of them, here are some equivalent constants written with exponents:

```
1e0, 1.0000e0
```

```
100e1, 100e+1, 100E+1, 1e3, 10000e-1
3.14159e0
5e-2, .0005e+2, 5E-2, .0005E2
.05e-2
```

A floating-point constant normally has type `double`. You can force it to type `float` by adding '`f`' or '`F`' at the end. For example,

```
3.14159f
3.14159e0f
1000.f
100E1F
.0005f
.05e-2f
```

Likewise, '`l`' or '`L`' at the end forces the constant to type `long double`.

There are also *hexadecimal floating constants*. These *must* have an exponent, but since '`e`' would be interpreted as a hexadecimal digit, the character '`p`' or '`P`' (for "power") indicates the exponent.

The exponent in a hexadecimal floating constant is an optionally signed decimal integer that specifies a power of 2 (*not* 10 or 16) to multiply into the number.

Here are some examples:

```
0xAp2         // 40 in decimal
0xAp-1        // 5 in decimal
0x2.0Bp4      // 32.6875 decimal
0xE.2p3       // 113 decimal
0x123.ABCp0   // 291.6708984375 in decimal
0x123.ABCp4   // 4666.734375 in decimal
0x100p-8      // 1
0x10p-4       // 1
0x1p+4        // 16
0x1p+8        // 256
```

See Section 11.2 [Floating-Point Data Types], page 48.

12.4 Imaginary Constants

A complex number consists of a real part plus an imaginary part. (You may omit one part if it is zero.) This section explains how to write numeric constants with imaginary values. By adding these to ordinary real-valued numeric constants, we can make constants with complex values.

The simple way to write an imaginary-number constant is to attach the suffix '`i`' or '`I`', or '`j`' or '`J`', to an integer or floating-point constant. For example, `2.5fi` has type `_Complex float` and `3i` has type `_Complex int`. The four alternative suffix letters are all equivalent.

The other way to write an imaginary constant is to multiply a real constant by `_Complex_I`, which represents the imaginary number i. Standard C doesn't support suffixes for imaginary constants, so this clunky method is needed.

To write a complex constant with a nonzero real part and a nonzero imaginary part, write the two separately and add them, like this:

```
4.0 + 3.0i
```

That gives the value 4 + 3i, with type `_Complex double`.

Such a sum can include multiple real constants, or none. Likewise, it can include multiple imaginary constants, or none. For example:

```
_Complex double foo, bar, quux;

foo = 2.0i + 4.0 + 3.0i; /* Imaginary part is 5.0. */
bar = 4.0 + 12.0; /* Imaginary part is 0.0. */
quux = 3.0i + 15.0i; /* Real part is 0.0. */
buux = 3.0i + 15.0j; /* Equal to quux. */
```

See Section 11.3 [Complex Data Types], page 49.

12.5 Invalid Numbers

Some number-like constructs which are not really valid as numeric constants are treated as numbers in preprocessing directives. If these constructs appear outside of preprocessing, they are erroneous. See Section 26.3 [Preprocessing Tokens], page 161.

Sometimes we need to insert spaces to separate tokens so that they won't be combined into a single number-like construct. For example, `0xE+12` is a preprocessing number that is not a valid numeric constant, so it is a syntax error. If what we want is the three tokens `0xE + 12`, we have to insert spaces as separators.

12.6 Character Constants

A *character constant* is written with single quotes, as in `'c'`. In the simplest case, *c* is a single ASCII character that the constant should represent. The constant has type `int`, and its value is the character code of that character. For instance, `'a'` represents the character code for the letter '`a`': 97, that is.

To put the '`'`' character (single quote) in the character constant, *escape* it with a backslash ('\'). This character constant looks like `'\''`. The backslash character here functions as an *escape character*, and such a sequence, starting with '\', is called an *escape sequence*.

To put the '\' character (backslash) in the character constant, escape it with '\' (another backslash). This character constant looks like `'\\'`.

Here are all the escape sequences that represent specific characters in a character constant. The numeric values shown are the corresponding ASCII character codes, as decimal numbers. The comments give the characters' conventional or traditional namss, as well as the appearance for graphical characters.

```
'\a' ⇒ 7        /* alarm, bell, CTRL-g */
'\b' ⇒ 8        /* backspace, BS, CTRL-h */
'\t' ⇒ 9        /* tab, TAB, CTRL-i */
'\n' ⇒ 10       /* newline, CTRL-j */
'\v' ⇒ 11       /* vertical tab, CTRL-k */
'\f' ⇒ 12       /* formfeed, CTRL-l */
'\r' ⇒ 13       /* carriage return, RET, CTRL-m */
'\e' ⇒ 27       /* escape character, ESC, CTRL-[ */
'\\' ⇒ 92       /* backslash character, \ */
```

```
'\'' ⇒ 39       /* single quote character, ' */
'\"' ⇒ 34       /* double quote character, " */
'\?' ⇒ 63       /* question mark, ? */
```

‘`\e`’ is a GNU C extension; to stick to standard C, write ‘`\33`’. (The number after ‘`\`’ is octal.) To specify a character constant using decimal, use a cast; for instance, `(unsigned char) 27`.

You can also write octal and hex character codes as ‘`\octalcode`’ or ‘`\xhexcode`’. Decimal is not an option here, so octal codes do not need to start with ‘`0`’. An octal code is limited to three octal digits, and any non-octal character terminates it.

The character constant’s value has type `int`. However, the character code is treated initially as a `char` value, which is then converted to `int`. If the character code is greater than 127 (`0177` in octal), the resulting `int` may be negative on a platform where the type `char` is 8 bits long and signed.

12.7 String Constants

A *string constant* represents a series of characters. It starts with ‘`"`’ and ends with ‘`"`’; in between are the contents of the string. Quoting special characters such as ‘`"`’, ‘`\`’ and newline in the contents works in string constants as in character constants. In a string constant, ‘`'`’ does not need to be quoted.

A string constant defines an array of characters which contains the specified characters followed by the null character (code 0). Using the string constant is equivalent to using the name of an array with those contents. In simple cases, where there are no backslash escape sequences, the length in bytes of the string constant is one greater than the number of characters written in it.

As with any array in C, using the string constant in an expression converts the array to a pointer (see Chapter 14 [Pointers], page 62) to the array’s zeroth element (see Section 16.1 [Accessing Array Elements], page 91). This pointer will have type `char *` because it points to an element of type `char`. `char *` is an example of a type designator for a pointer type (see Section 14.4 [Pointer Type Designators], page 63). That type is used for operating on strings generally, not just the strings expressed as constants.

Thus, the string constant `"Foo!"` is almost equivalent to declaring an array like this

```
char string_array_1[] = {'F', 'o', 'o', '!', '\0' };
```

and then using `string_array_1` in the program (which converts it to type `char *`). There are two differences, however:

- The string constant doesn’t define a name for the array.
- The string constant is probably stored in a read-only area of memory.

Newlines are not allowed in the text of a string constant. The motive for this prohibition is to catch the error of omitting the closing ‘`"`’. To put a newline in a constant string, write it as ‘`\n`’ in the string constant.

A real null character in the source code inside a string constant causes a warning. To put a null character in the middle of a string constant, write ‘`\0`’ or ‘`\000`’.

Consecutive string constants are effectively concatenated. Thus,

```
"Fo" "o!"     is equivalent to     "Foo!"
```

This is useful for writing a string containing multiple lines, like this:

```
"This message is so long that it needs more than\n"
"a single line of text.  C does not allow a newline\n"
"to represent itself in a string constant, so we have to\n"
"write \\n to put it in the string.  For readability of\n"
"the source code, it is advisable to put line breaks in\n"
"the source where they occur in the contents of the\n"
"constant.\n"
```

The sequence of a backslash and a newline is ignored anywhere in a C program, and that includes inside a string constant. Thus, you can write multi-line string constants this way:

```
"This is another way to put newlines in a string constant\n\
and break the line after them in the source code."
```

However, concatenation is the recommended way to do this.

You can also write perverse string constants like this,

```
"Fo\
o!"
```

but don't do that—write it like this instead:

```
"Foo!"
```

Be careful to avoid passing a string constant to a function that modifies the string it receives. The memory where the string constant is stored may be read-only, which would cause a fatal `SIGSEGV` signal that normally terminates the function (see Appendix E [Signals], page 224). Even worse, the memory may not be read-only. Then the function might modify the string constant, thus spoiling the contents of other string constants that are supposed to contain the same value and are unified by the compiler.

12.8 UTF-8 String Constants

Writing '`u8`' immediately before a string constant, with no intervening space, means to represent that string in UTF-8 encoding as a sequence of bytes. UTF-8 represents ASCII characters with a single byte, and represents non-ASCII Unicode characters (codes 128 and up) as multibyte sequences. Here is an example of a UTF-8 constant:

```
u8"A cónstàñt"
```

This constant occupies 13 bytes plus the terminating null, because each of the accented letters is a two-byte sequence.

Concatenating an ordinary string with a UTF-8 string conceptually produces another UTF-8 string. However, if the ordinary string contains character codes 128 and up, the results cannot be relied on.

12.9 Unicode Character Codes

You can specify Unicode characters using escape sequences called *universal character names* that start with '`\u`' and '`\U`'. They are valid in C for individual character constants, inside string constants (see Section 12.7 [String Constants], page 56), and even in identifiers. These escape sequence includes a hexadecimal Unicode character code, also called a *code point* in Unicode terminology.

Use the ‘`\u`’ escape sequence with a 16-bit hexadecimal Unicode character code. If the character’s numeric code is too big for 16 bits, use the ‘`\U`’ escape sequence with a 32-bit hexadecimal Unicode character code. Here are some examples.

```
\u6C34      /* 16-bit code (Chinese for “water”), UTF-16 */
\U0010ABCD  /* 32-bit code, UTF-32 */
```

One way to use these is in UTF-8 string constants (see Section 12.8 [UTF-8 String Constants], page 57). For instance, here we use two of them, each preceded by a space.

```
u8"fóó \u6C34 \U0010ABCD"
```

You can also use them in wide character constants (see Section 12.10 [Wide Character Constants], page 58), like this:

```
u'\u6C34'       /* 16-bit code (water) */
U'\U0010ABCD'   /* 32-bit code */
```

and in wide string constants (see Section 12.11 [Wide String Constants], page 59), like this:

```
u"\u6C34\u706B"  /* 16-bit codes (water, fire) */
U"\U0010ABCD"    /* 32-bit code */
```

And in an identifier:

```
int foo\u6C34bar = 0;
```

Codes in the range of D800 through DFFF are invalid in universal character names. Trying to write them using ‘`\u`’ causes an error. Unicode calls them “surrogate code points” and uses them in UTF-16 for purposes too specialized to explain here.

Codes less than 00A0 are likewise invalid in universal character names, and likewise cause errors, except for 0024 (‘`$`’), 0040 (‘`@`’), and 0060 (‘`` ` ``’). Characters which can’t be represented with universal character names can be specified with octal or hexadecimal escape sequences (see Section 12.6 [Character Constants], page 55).

12.10 Wide Character Constants

A *wide character constant* represents characters with more than 8 bits of character code. This is an obscure feature that we need to document but that you probably won’t ever use. If you’re just learning C, you may as well skip this section.

The original C wide character constant looks like ‘`L`’ (upper case!) followed immediately by an ordinary character constant (with no intervening space). Its data type is `wchar_t`, which is an alias defined in `stddef.h` for one of the standard integer types. Depending on the platform, it could be 16 bits or 32 bits. If it is 16 bits, these character constants use the UTF-16 form of Unicode; if 32 bits, UTF-32.

There are also Unicode wide character constants which explicitly specify the width. These constants start with ‘`u`’ or ‘`U`’ instead of ‘`L`’. ‘`u`’ specifies a 16-bit Unicode wide character constant, and ‘`U`’ a 32-bit Unicode wide character constant. Their types are, respectively, `char16_t` and `char32_t`; they are declared in the header file `uchar.h`. These character constants are valid even if `uchar.h` is not included, but some uses of them may be inconvenient without including it to declare those type names.

The character represented in a wide character constant can be an ordinary ASCII character. `L'a'`, `u'a'` and `U'a'` are all valid, and they are all equal to `'a'`.

In all three kinds of wide character constants, you can write a non-ASCII Unicode character in the constant itself; the constant's value is the character's Unicode character code. Or you can specify the Unicode character with an escape sequence (see Section 12.9 [Unicode Character Codes], page 57).

12.11 Wide String Constants

A *wide string constant* stands for an array of 16-bit or 32-bit characters. They are rarely used; if you're just learning C, you may as well skip this section.

There are three kinds of wide string constants, which differ in the data type used for each character in the string. Each wide string constant is equivalent to an array of integers, but the data type of those integers depends on the kind of wide string. Using the constant in an expression will convert the array to a pointer to its zeroth element, as usual for arrays in C (see Section 16.1 [Accessing Array Elements], page 91). For each kind of wide string constant, we state here what type that pointer will be.

`char16_t` This is a 16-bit Unicode wide string constant: each element is a 16-bit Unicode character code with type `char16_t`, so the string has the array type `char16_t[]`. (That is a type designator; see Section 14.4 [Pointer Type Designators], page 63.) The constant is written as '`u`' (which must be lower case) followed (with no intervening space) by a string constant with the usual syntax.

`char32_t` This is a 32-bit Unicode wide string constant: each element is a 32-bit Unicode character code, and the string has type `char32_t[]`. It's written as '`U`' (which must be upper case) followed (with no intervening space) by a string constant with the usual syntax.

`wchar_t` This is the original kind of wide string constant. It's written as '`L`' (which must be upper case) followed (with no intervening space) by a string constant with the usual syntax, and the string has type `wchar_t[]`.

The width of the data type `wchar_t` depends on the target platform, which makes this kind of wide string somewhat less useful than the newer kinds.

`char16_t` and `char32_t` are declared in the header file `uchar.h`. `wchar_t` is declared in `stddef.h`.

Consecutive wide string constants of the same kind concatenate, just like ordinary string constants. A wide string constant concatenated with an ordinary string constant results in a wide string constant. You can't concatenate two wide string constants of different kinds. In addition, you can't concatenate a wide string constant (of any kind) with a UTF-8 string constant.

13 Type Size

Each data type has a *size*, which is the number of bytes (see Chapter 3 [Storage], page 12) that it occupies in memory. To refer to the size in a C program, use `sizeof`. There are two ways to use it:

`sizeof` *`expression`*

This gives the size of *expression*, based on its data type. It does not calculate the value of *expression*, only its size, so if *expression* includes side effects or function calls, they do not happen. Therefore, `sizeof` with an expression as argument is always a compile-time operation that has zero run-time cost, unless it applies to a variable-size array.

A value that is a bit field (see Section 15.7 [Bit Fields], page 79) is not allowed as an operand of `sizeof`.

For example,

```
double a;

i = sizeof a + 10;
```

sets `i` to 18 on most computers because `a` occupies 8 bytes.

Here's how to determine the number of elements in an array `arr`:

```
(sizeof arr / sizeof arr[0])
```

The expression `sizeof arr` gives the size of the array, not the size of a pointer to an element. However, if *expression* is a function parameter that was declared as an array, that variable really has a pointer type (see Section 22.1.4.1 [Array Params are Ptrs], page 134), so the result is the size of that pointer.

`sizeof (`*`type`*`)`

This gives the size of *type*. For example,

```
i = sizeof (double) + 10;
```

is equivalent to the previous example.

Warning: If *type* contains expressions which have side effects, those expressions are actually computed and any side effects in them do occur.

You can't apply `sizeof` to an incomplete type (see Section 15.19 [Incomplete Types], page 87). Using it on a function type or `void` gives 1 in GNU C, which makes adding an integer to these pointer types work as desired (see Section 14.10 [Pointer Arithmetic], page 67).

Warning: When you use `sizeof` with a type instead of an expression, you must write parentheses around the type.

Warning: When applying `sizeof` to the result of a cast (see Section 24.1 [Explicit Type Conversion], page 155), you must write parentheses around the cast expression to avoid an ambiguity in the grammar of C. Specifically,

```
sizeof (int) -x
```

parses as

```
(sizeof (int)) - x
```

If what you want is

```
sizeof ((int) -x)
```

you must write it that way, with parentheses.

The data type of the value of the `sizeof` operator is always an unsigned integer type; which one of those types depends on the machine. The header file `stddef.h` defines `size_t` as a name for such a type. See Chapter 18 [Defining Typedef Names], page 101.

14 Pointers

Among high-level languages, C is rather low-level, close to the machine. This is mainly because it has explicit *pointers*. A pointer value is the numeric address of data in memory. The type of data to be found at that address is specified by the data type of the pointer itself. Nothing in C can determine the "correct" data type of data in memory; it can only blindly follow the data type of the pointer you use to access the data.

The unary operator '`*`' gets the data that a pointer points to—this is called *dereferencing the pointer*. Its value always has the type that the pointer points to.

C also allows pointers to functions, but since there are some differences in how they work, we treat them later. See Section 22.5 [Function Pointers], page 140.

14.1 Address of Data

The most basic way to make a pointer is with the "address-of" operator, '`&`'. Let's suppose we have these variables available:

```
int i;
double a[5];
```

Now, `&i` gives the address of the variable `i`—a pointer value that points to `i`'s location—and `&a[3]` gives the address of the element 3 of `a`. (By the usual 1-origin numbering convention of ordinary English, it is actually the fourth element in the array, since the element at the start has index 0.)

The address-of operator is unusual because it operates on a place to store a value (an lvalue, see Section 7.2 [Lvalues], page 31), not on the value currently stored there. (The left argument of a simple assignment is unusual in the same way.) You can use it on any lvalue except a bit field (see Section 15.7 [Bit Fields], page 79) or a constructor (see Section 15.17 [Structure Constructors], page 86).

14.2 Pointer Types

For each data type *t*, there is a type for pointers to type *t*. For these variables,

```
int i;
double a[5];
```

- `i` has type `int`; we say `&i` is a "pointer to `int`."
- `a` has type `double[5]`; we say `&a` is a "pointer to an array of five `doubles`."
- `a[3]` has type `double`; we say `&a[3]` is a "pointer to `double`."

14.3 Pointer-Variable Declarations

The way to declare that a variable `foo` points to type *t* is

```
t *foo;
```

To remember this syntax, think "if you dereference `foo`, using the '`*`' operator, what you get is type *t*. Thus, `foo` points to type *t*."

Thus, we can declare variables that hold pointers to these three types, like this:

```
int *ptri;            /* Pointer to int. */
```

```
double *ptrd;          /* Pointer to double. */
double (*ptrda)[5];    /* Pointer to double[5]. */
```

'`int *ptri;`' means, "if you dereference `ptri`, you get an `int`." '`double (*ptrda)[5];`' means, "if you dereference `ptrda`, then subscript it by an integer less than 5, you get a `double`." The parentheses express the point that you would dereference it first, then subscript it.

Contrast the last one with this:

```
double *aptrd[5];      /* Array of five pointers to double. */
```

Because '`*`' has lower syntactic precedence than subscripting, '`double *aptrd[5]`' means, "if you subscript `aptrd` by an integer less than 5, then dereference it, you get a `double`." Therefore, `*aptrd[5]` declares an array of pointers, not a pointer to an array.

14.4 Pointer-Type Designators

Every type in C has a designator; you make it by deleting the variable name and the semicolon from a declaration (see Section 11.6 [Type Designators], page 51). Here are the designators for the pointer types of the example declarations in the previous section:

```
int *           /* Pointer to int. */
double *        /* Pointer to double. */
double (*)[5]   /* Pointer to double[5]. */
```

Remember, to understand what type a designator stands for, imagine the corresponding variable declaration with a variable name in it, and figure out what type that variable would have. Thus, the type designator `double (*)[5]` corresponds to the variable declaration `double (*`*`variable`*`)[5]`. That declares a pointer variable which, when dereferenced, gives an array of 5 `double`s. So the type designator means, "pointer to an array of 5 `double`s."

14.5 Dereferencing Pointers

The main use of a pointer value is to *dereference it* (access the data it points at) with the unary '`*`' operator. For instance, `*&i` is the value at `i`'s address—which is just `i`. The two expressions are equivalent, provided `&i` is valid.

A pointer-dereference expression whose type is data (not a function) is an lvalue.

Pointers become really useful when we store them somewhere and use them later. Here's a simple example to illustrate the practice:

```
{
  int i;
  int *ptr;

  ptr = &i;

  i = 5;

  ...

  return *ptr;   /* Returns 5, fetched from i.  */
}
```

This shows how to declare the variable `ptr` as type `int *` (pointer to `int`), store a pointer value into it (pointing at `i`), and use it later to get the value of the object it points at (the value in `i`).

Here is another example of using a pointer to a variable.

```
/* Define global variable i.  */
int i = 2;

int
foo (void)
{
  /* Save global variable i's address.  */
  int *global_i = &i;

  /* Declare local i, shadowing the global i.  */
  int i = 5;

  /* Print value of global i and value of local i.  */
  printf ("global i: %d\nlocal i: %d\n", *global_i, i);
  return i;
}
```

Of course, in a real program it would be much cleaner to use different names for these two variables, rather than calling both of them `i`. But it is hard to illustrate this syntaxtical point with clean code. If anyone can provide a useful example to illustrate this point with, that would be welcome.

14.6 Null Pointers

A pointer value can be *null*, which means it does not point to any object. The cleanest way to get a null pointer is by writing `NULL`, a standard macro defined in `stddef.h`. You can also do it by casting 0 to the desired pointer type, as in `(char *) 0`. (The cast operator performs explicit type conversion; See Section 24.1 [Explicit Type Conversion], page 155.)

You can store a null pointer in any lvalue whose data type is a pointer type:

```
char *foo;
foo = NULL;
```

These two, if consecutive, can be combined into a declaration with initializer,

```
char *foo = NULL;
```

You can also explicitly cast `NULL` to the specific pointer type you want—it makes no difference.

```
char *foo;
foo = (char *) NULL;
```

To test whether a pointer is null, compare it with zero or `NULL`, as shown here:

```
if (p != NULL)
  /* p is not null.  */
  operate (p);
```

Since testing a pointer for not being null is basic and frequent, all but beginners in C will understand the conditional without need for `!= NULL`:

```
if (p)
  /* p is not null.  */
  operate (p);
```

14.7 Dereferencing Null or Invalid Pointers

Trying to dereference a null pointer is an error. On most platforms, it generally causes a signal, usually `SIGSEGV` (see Appendix E [Signals], page 224).

```
char *foo = NULL;
c = *foo;    /* This causes a signal and terminates.  */
```

Likewise a pointer that has the wrong alignment for the target data type (on most types of computer), or points to a part of memory that has not been allocated in the process's address space.

The signal terminates the program, unless the program has arranged to handle the signal (see Section "Signal Handling" in *The GNU C Library Reference Manual*).

However, the signal might not happen if the dereference is optimized away. In the example above, if you don't subsequently use the value of `c`, GCC might optimize away the code for `*foo`. You can prevent such optimization using the `volatile` qualifier, as shown here:

```
volatile char *p;
volatile char c;
c = *p;
```

You can use this to test whether `p` points to unallocated memory. Set up a signal handler first, so the signal won't terminate the program.

14.8 Void Pointers

The peculiar type `void *`, a pointer whose target type is `void`, is used often in C. It represents a pointer to we-don't-say-what. Thus,

```
void *numbered_slot_pointer (int);
```

declares a function `numbered_slot_pointer` that takes an integer parameter and returns a pointer, but we don't say what type of data it points to.

The functions for dynamic memory allocation (see Section 15.3 [Dynamic Memory Allocation], page 76) use type `void *` to refer to blocks of memory, regardless of what sort of data the program stores in those blocks.

With type `void *`, you can pass the pointer around and test whether it is null. However, dereferencing it gives a `void` value that can't be used (see Section 11.4 [The Void Type], page 50). To dereference the pointer, first convert it to some other pointer type.

Assignments convert `void *` automatically to any other pointer type, if the left operand has a pointer type; for instance,

```
{
  int *p;
  /* Converts return value to int *.  */
  p = numbered_slot_pointer (5);
  ...
}
```

Passing an argument of type `void *` for a parameter that has a pointer type also converts. For example, supposing the function `hack` is declared to require type `float *` for its parameter, this call to `hack` will convert the argument to that type.

```
/* Declare hack that way.
   We assume it is defined somewhere else.  */
void hack (float *);
...
/* Now call hack.  */
{
  /* Converts return value of numbered_slot_pointer
     to float * to pass it to hack.  */
  hack (numbered_slot_pointer (5));
  ...
}
```

You can also convert to another pointer type with an explicit cast (see Section 24.1 [Explicit Type Conversion], page 155), like this:

```
(int *) numbered_slot_pointer (5)
```

Here is an example which decides at run time which pointer type to convert to:

```
void
extract_int_or_double (void *ptr, bool its_an_int)
{
  if (its_an_int)
    handle_an_int (*(int *)ptr);
  else
    handle_a_double (*(double *)ptr);
}
```

The expression `*(int *)ptr` means to convert `ptr` to type `int *`, then dereference it.

14.9 Pointer Comparison

Two pointer values are equal if they point to the same location, or if they are both null. You can test for this with `==` and `!=`. Here's a trivial example:

```
{
  int i;
  int *p, *q;

  p = &i;
  q = &i;
  if (p == q)
    printf ("This will be printed.\n");
  if (p != q)
    printf ("This won't be printed.\n");
}
```

Ordering comparisons such as `>` and `>=` operate on pointers by converting them to unsigned integers. The C standard says the two pointers must point within the same object in memory, but on GNU/Linux systems these operations simply compare the numeric values of the pointers.

The pointer values to be compared should in principle have the same type, but they are allowed to differ in limited cases. First of all, if the two pointers' target types are nearly compatible (see Chapter 23 [Compatible Types], page 154), the comparison is allowed.

If one of the operands is `void *` (see Section 14.8 [Void Pointers], page 65) and the other is another pointer type, the comparison operator converts the `void *` pointer to the other type so as to compare them. (In standard C, this is not allowed if the other type is a function pointer type, but it works in GNU C.)

Comparison operators also allow comparing the integer 0 with a pointer value. This works by converting 0 to a null pointer of the same type as the other operand.

14.10 Pointer Arithmetic

Adding an integer (positive or negative) to a pointer is valid in C. It assumes that the pointer points to an element in an array, and advances or retracts the pointer across as many array elements as the integer specifies. Here is an example, in which adding a positive integer advances the pointer to a later element in the same array.

```
void
incrementing_pointers ()
{
  int array[5] = { 45, 29, 104, -3, 123456 };
  int elt0, elt1, elt4;

  int *p = &array[0];
  /* Now p points at element 0.  Fetch it.  */
  elt0 = *p;

  ++p;
  /* Now p points at element 1.  Fetch it.  */
  elt1 = *p;

  p += 3;
  /* Now p points at element 4 (the last).  Fetch it.  */
  elt4 = *p;

  printf ("elt0 %d  elt1 %d  elt4 %d.\n",
          elt0, elt1, elt4);
  /* Prints elt0 45  elt1 29  elt4 123456.  */
}
```

Here's an example where adding a negative integer retracts the pointer to an earlier element in the same array.

```
void
```

```
decrementing_pointers ()
{
  int array[5] = { 45, 29, 104, -3, 123456 };
  int elt0, elt3, elt4;

  int *p = &array[4];
  /* Now p points at element 4 (the last).  Fetch it.  */
  elt4 = *p;

  --p;
  /* Now p points at element 3.  Fetch it.  */
  elt3 = *p;

  p -= 3;
  /* Now p points at element 0.  Fetch it.  */
  elt0 = *p;

  printf ("elt0 %d  elt3 %d  elt4 %d.\n",
          elt0, elt3, elt4);
  /* Prints elt0 45  elt3 -3  elt4 123456.  */
}
```

If one pointer value was made by adding an integer to another pointer value, it should be possible to subtract the pointer values and recover that integer. That works too in C.

```
void
subtract_pointers ()
{
  int array[5] = { 45, 29, 104, -3, 123456 };
  int *p0, *p3, *p4;

  int *p = &array[4];
  /* Now p points at element 4 (the last).  Save the value.  */
  p4 = p;

  --p;
  /* Now p points at element 3.  Save the value.  */
  p3 = p;

  p -= 3;
  /* Now p points at element 0.  Save the value.  */
  p0 = p;

  printf ("%d, %d, %d, %d\n",
          p4 - p0, p0 - p0, p3 - p0, p0 - p3);
  /* Prints 4, 0, 3, -3.  */
}
```

The addition operation does not know where arrays begin or end in memory. All it does is add the integer (multiplied by target object size) to the numeric value of the pointer. When the initial pointer and the result point into the same array, the result is well-defined.

Warning: Only experts should do pointer arithmetic involving pointers into different memory objects.

The difference between two pointers has type `int`, or `long` if necessary (see Section 11.1 [Integer Types], page 46). The clean way to declare it is to use the typedef name `ptrdiff_t` defined in the file `stddef.h`.

C defines pointer subtraction to be consistent with pointer-integer addition, so that `(p3 - p1) + p1` equals `p3`, as in ordinary algebra. Pointer subtraction works by subtracting `p1`'s numeric value from `p3`'s, and dividing by target object size. The two pointer arguments should point into the same array.

In standard C, addition and subtraction are not allowed on `void *`, since the target type's size is not defined in that case. Likewise, they are not allowed on pointers to function types. However, these operations work in GNU C, and the "size of the target type" is taken as 1 byte.

14.11 Pointers and Arrays

The clean way to refer to an array element is `array[index]`. Another, complicated way to do the same job is to get the address of that element as a pointer, then dereference it: `* (&array[0] + index)` (or equivalently `* (array + index)`). This first gets a pointer to element zero, then increments it with `+` to point to the desired element, then gets the value from there.

That pointer-arithmetic construct is the *definition* of square brackets in C. `a[b]` means, by definition, `*(a + b)`. This definition uses *a* and *b* symmetrically, so one must be a pointer and the other an integer; it does not matter which comes first.

Since indexing with square brackets is defined in terms of addition and dereferencing, that too is symmetrical. Thus, you can write `3[array]` and it is equivalent to `array[3]`. However, it would be foolish to write `3[array]`, since it has no advantage and could confuse people who read the code.

It may seem like a discrepancy that the definition `*(a + b)` requires a pointer, while `array[3]` uses an array value instead. Why is this valid? The name of the array, when used by itself as an expression (other than in `sizeof`), stands for a pointer to the array's zeroth element. Thus, `array + 3` converts `array` implicitly to `&array[0]`, and the result is a pointer to element 3, equivalent to `&array[3]`.

Since square brackets are defined in terms of such an addition, `array[3]` first converts `array` to a pointer. That's why it works to use an array directly in that construct.

14.12 Pointer Arithmetic at Low-Level

The behavior of pointer arithmetic is theoretically defined only when the pointer values all point within one object allocated in memory. But the addition and subtraction operators can't tell whether the pointer values are all within one object. They don't know where objects start and end. So what do they really do?

Adding pointer *p* to integer *i* treats *p* as a memory address, which is in fact an integer—call it *pint*. It treats *i* as a number of elements of the type that *p* points to. These elements' sizes add up to `i * sizeof (*p)`. So the sum, as an integer, is `pint + i * sizeof (*p)`. This value is reinterpreted as a pointer of the same type as *p*.

If the starting pointer value *p* and the result do not point at parts of the same object, the operation is not officially legitimate, and C code is not "supposed" to do it. But you can do it anyway, and it gives precisely the results described by the procedure above. In some special situations it can do something useful, but non-wizards should avoid it.

Here's a function to offset a pointer value *as if* it pointed to an object of any given size, by explicitly performing that calculation:

```
#include <stdint.h>

void *
ptr_add (void *p, int i, int objsize)
{
  intptr_t p_address = (long) p;
  intptr_t totalsize = i * objsize;
  intptr_t new_address = p_address + totalsize;
  return (void *) new_address;
}
```

This does the same job as `p + i` with the proper pointer type for *p*. It uses the type `intptr_t`, which is defined in the header file `stdint.h`. (In practice, `long long` would always work, but it is cleaner to use `intptr_t`.)

14.13 Pointer Increment and Decrement

The '`++`' operator adds 1 to a variable. We have seen it for integers (see Section 7.4 [Increment/Decrement], page 32), but it works for pointers too. For instance, suppose we have a series of positive integers, terminated by a zero, and we want to add them up. Here is a simple way to step forward through the array by advancing a pointer.

```
int
sum_array_till_0 (int *p)
{
  int sum = 0;

  for (;;)
    {
      /* Fetch the next integer.  */
      int next = *p++;
      /* Exit the loop if it's 0.  */
      if (next == 0)
        break;
      /* Add it into running total.  */
      sum += next;
    }
  return sum;
}
```

The statement ‘`break;`’ will be explained further on (see Section 19.6.3 [break Statement], page 106). Used in this way, it immediately exits the surrounding `for` statement.

`*p++` uses postincrement (`++`; see Section 7.5 [Postincrement/Postdecrement], page 33) on the pointer `p`. That expression parses as `*(p++)`, because a postfix operator always takes precedence over a prefix operator. Therefore, it dereferences the entering value of `p`, then increments `p` afterwards.

Incrementing a variable means adding 1 to it, as in `p = p + 1`. Since `p` is a pointer, adding 1 to it advances it by the width of the datum it points to—in this case, `sizeof (int)`. Therefore, each iteration of the loop picks up the next integer from the series and puts it into `next`.

This `for`-loop has no initialization expression since `p` and `sum` are already initialized, has no end-test since the ‘`break;`’ statement will exit it, and needs no expression to advance it since that’s done within the loop by incrementing `p` and `sum`. Thus, those three expressions after `for` are left empty.

Another way to write this function is by keeping the parameter value unchanged and using indexing to access the integers in the table.

```
int
sum_array_till_0_indexing (int *p)
{
  int i;
  int sum = 0;

  for (i = 0; ; i++)
    {
      /* Fetch the next integer.  */
      int next = p[i];
      /* Exit the loop if it’s 0.  */
      if (next == 0)
        break;
      /* Add it into running total.  */
      sum += next;
    }

  return sum;
}
```

In this program, instead of advancing `p`, we advance `i` and add it to `p`. (Recall that `p[i]` means `*(p + i)`.) Either way, it uses the same address to get the next integer.

It makes no difference in this program whether we write `i++` or `++i`, because the value *of that expression* is not used. We use it for its effect, to increment `i`.

The ‘`--`’ operator also works on pointers; it can be used to step backwards through an array, like this:

```
int
after_last_nonzero (int *p, int len)
```

```
{
  /* Set up q to point just after the last array element.  */
  int *q = p + len;

  while (q != p)
    /* Step q back until it reaches a nonzero element.  */
    if (*--q != 0)
      /* Return the index of the element after that nonzero.  */
      return q - p + 1;

  return 0;
}
```

That function returns the length of the nonzero part of the array specified by its arguments; that is, the index of the first zero of the run of zeros at the end.

14.14 Drawbacks of Pointer Arithmetic

Pointer arithmetic is clean and elegant, but it is also the cause of a major security flaw in the C language. Theoretically, it is only valid to adjust a pointer within one object allocated as a unit in memory. However, if you unintentionally adjust a pointer across the bounds of the object and into some other object, the system has no way to detect this error.

A bug which does that can easily result in clobbering (overwriting) part of another object. For example, with `array[-1]` you can read or write the nonexistent element before the beginning of an array—probably part of some other data.

Combining pointer arithmetic with casts between pointer types, you can create a pointer that fails to be properly aligned for its type. For example,

```
int a[2];
char *pa = (char *)a;
int *p = (int *)(pa + 1);
```

gives `p` a value pointing to an "integer" that includes part of `a[0]` and part of `a[1]`. Dereferencing that with `*p` can cause a fatal `SIGSEGV` signal or it can return the contents of that badly aligned `int` (see Appendix E [Signals], page 224. If it "works," it may be quite slow. It can also cause aliasing confusions (see Appendix B [Aliasing], page 218).

Warning: Using improperly aligned pointers is risky—don't do it unless it is really necessary.

14.15 Pointer-Integer Conversion

On modern computers, an address is simply a number. It occupies the same space as some size of integer. In C, you can convert a pointer to the appropriate integer types and vice versa, without losing information. The appropriate integer types are `uintptr_t` (an unsigned type) and `intptr_t` (a signed type). Both are defined in `stdint.h`.

For instance,

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

```
void
print_pointer (void *ptr)
{
  uintptr_t converted = (uintptr_t) ptr;

  printf ("Pointer value is 0x%x\n",
          (unsigned int) converted);
}
```

The specification '`%x`' in the template (the first argument) for `printf` means to represent this argument using hexadecimal notation. It's cleaner to use `uintptr_t`, since hexadecimal printing treats the number as unsigned, but it won't actually matter: all `printf` gets to see is the series of bits in the number.

Warning: Converting pointers to integers is risky—don't do it unless it is really necessary.

14.16 Printing Pointers

To print the numeric value of a pointer, use the '`%p`' specifier. For example:

```
void
print_pointer (void *ptr)
{
  printf ("Pointer value is %p\n", ptr);
}
```

The specification '`%p`' works with any pointer type. It prints '`0x`' followed by the address in hexadecimal, printed as the appropriate unsigned integer type.

15 Structures

A *structure* is a user-defined data type that holds various *fields* of data. Each field has a name and a data type specified in the structure's definition. Because a structure combines various fields, each of its own type, we call a structure type a *compound type*. (Union types are also compound types; see Section 15.14 [Unions], page 83.)

Here we define a structure suitable for storing a linked list of integers. Each list item will hold one integer, plus a pointer to the next item.

```
struct intlistlink
  {
    int datum;
    struct intlistlink *next;
  };
```

The structure definition has a *type tag* so that the code can refer to this structure. The type tag here is `intlistlink`. The definition refers recursively to the same structure through that tag.

You can define a structure without a type tag, but then you can't refer to it again. That is useful only in some special contexts, such as inside a `typedef` or a `union`.

The contents of the structure are specified by the *field declarations* inside the braces. Each field in the structure needs a declaration there. The fields in one structure definition must have distinct names, but these names do not conflict with any other names in the program.

A field declaration looks just like a variable declaration. You can combine field declarations with the same beginning, just as you can combine variable declarations.

This structure has two fields. One, named `datum`, has type `int` and will hold one integer in the list. The other, named `next`, is a pointer to another `struct intlistlink` which would be the rest of the list. In the last list item, it would be `NULL`.

This structure definition is recursive, since the type of the `next` field refers to the structure type. Such recursion is not a problem; in fact, you can use the type `struct intlistlink *` before the definition of the type `struct intlistlink` itself. That works because pointers to all kinds of structures really look the same at the machine level.

After defining the structure, you can declare a variable of type `struct intlistlink` like this:

```
struct intlistlink foo;
```

The structure definition itself can serve as the beginning of a variable declaration, so you can declare variables immediately after, like this:

```
struct intlistlink
  {
    int datum;
    struct intlistlink *next;
  } foo;
```

But that is ugly. It is almost always clearer to separate the definition of the structure from its uses.

Declaring a structure type inside a block (see Section 19.4 [Blocks], page 104) limits the scope of the structure type name to that block. That means the structure type is recognized only within that block. Declaring it in a function parameter list, as here,

```
int f (struct foo {int a, b} parm);
```

(assuming that `struct foo` is not already defined) limits the scope of the structure type `struct foo` to that parameter list; that is basically useless, so it triggers a warning.

Standard C requires at least one field in a structure. GNU C does not require this.

15.1 Referencing Structure Fields

To make a structure useful, there has to be a way to examine and store its fields. The '`.`' (period) operator does that; its use looks like *`object.field`*.

Given this structure and variable,

```
struct intlistlink
  {
    int datum;
    struct intlistlink *next;
  };

struct intlistlink foo;
```

you can write `foo.datum` and `foo.next` to refer to the two fields in the value of `foo`. These fields are lvalues, so you can store values into them, and read the values out again.

Most often, structures are dynamically allocated (see the next section), and we refer to the objects via pointers. `(*p).`*`field`* is somewhat cumbersome, so there is an abbreviation: `p->`*`field`*. For instance, assume the program contains this declaration:

```
struct intlistlink *ptr;
```

You can write `ptr->datum` and `ptr->next` to refer to the two fields in the object that `ptr` points to.

If a unary operator precedes an expression using '`->`', the '`->`' nests inside:

`-ptr->datum` is equivalent to `-(ptr->datum)`

You can intermix '`->`' and '`.`' without parentheses, as shown here:

```
struct { double d; struct intlistlink l; } foo;

...foo.l.next->next->datum...
```

15.2 Arrays as Fields

When you declare field in a structure as an array, as here:

```
struct record
  {
    char *name;
    int data[4];
  };
```

Each `struct record` object holds one string (a pointer, of course) and four integers, all part of a field called `data`. If `recptr` is a pointer of type `struct record *`, then it points to a `struct record` which contains those things; you can access the second integer in that record with `recptr->data[1]`.

If you have two objects of type `struct record`, each one contains an array. With this declaration,

```
struct record r1, r2;
```

`r1.data` holds space for 4 `int`s, and `r2.data` holds space for another 4 `int`s,

15.3 Dynamic Memory Allocation

To allocate an object dynamically, call the library function `malloc` (see Section "Basic Allocation" in *The GNU C Library Reference Manual*). Here is how to allocate an object of type `struct intlistlink`. To make this code work, include the file `stdlib.h`, like this:

```
#include <stddef.h>  /* Defines NULL. */
#include <stdlib.h>  /* Declares malloc.  */

...

struct intlistlink *
alloc_intlistlink ()
{
  struct intlistlink *p;

  p = malloc (sizeof (struct intlistlink));

  if (p == NULL)
    fatal ("Ran out of storage");

  /* Initialize the contents. */
  p->datum = 0;
  p->next = NULL;

  return p;
}
```

`malloc` returns `void *`, so the assignment to `p` will automatically convert it to type `struct intlistlink *`. The return value of `malloc` is always sufficiently aligned (see Appendix A [Type Alignment], page 217) that it is valid for any data type.

The test for `p == NULL` is necessary because `malloc` returns a null pointer if it cannot get any storage. We assume that the program defines the function `fatal` to report a fatal error to the user.

Here's how to add one more integer to the front of such a list:

```
struct intlistlink *my_list = NULL;

void
add_to_mylist (int my_int)
{
  struct intlistlink *p = alloc_intlistlink ();

  p->datum = my_int;
  p->next = mylist;
  mylist = p;
}
```

The way to free the objects is by calling `free`. Here's a function to free all the links in one of these lists:

```
void
free_intlist (struct intlistlink *p)
{
  while (p)
    {
      struct intlistlink *q = p;
      p = p->next;
      free (q);
    }
}
```

We must extract the `next` pointer from the object before freeing it, because `free` can clobber the data that was in the object. For the same reason, the program must not use the list any more after freeing its elements. To make sure it won't, it is best to clear out the variable where the list was stored, like this:

```
free_intlist (mylist);

mylist = NULL;
```

15.4 Field Offset

To determine the offset of a given field *field* in a structure type *type*, use the macro `offsetof`, which is defined in the file `stddef.h`. It is used like this:

```
offsetof (type, field)
```

Here is an example:

```
struct foo
{
  int element;
  struct foo *next;
};

offsetof (struct foo, next)
/* On most machines that is 4.  It may be 8.  */
```

15.5 Structure Layout

The rest of this chapter covers advanced topics about structures. If you are just learning C, you can skip it.

The precise layout of a `struct` type is crucial when using it to overlay hardware registers, to access data structures in shared memory, or to assemble and disassemble packets for network communication. It is also important for avoiding memory waste when the program makes many objects of that type. However, the layout depends on the target platform. Each platform has conventions for structure layout, which compilers need to follow.

Here are the conventions used on most platforms.

The structure's fields appear in the structure layout in the order they are declared. When possible, consecutive fields occupy consecutive bytes within the structure. However, if a field's type demands more alignment than it would get that way, C gives it the alignment it requires by leaving a gap after the previous field.

Once all the fields have been laid out, it is possible to determine the structure's alignment and size. The structure's alignment is the maximum alignment of any of the fields in it. Then the structure's size is rounded up to a multiple of its alignment. That may require leaving a gap at the end of the structure.

Here are some examples, where we assume that `char` has size and alignment 1 (always true), and `int` has size and alignment 4 (true on most kinds of computers):

```
struct foo
{
  char a, b;
  int c;
};
```

This structure occupies 8 bytes, with an alignment of 4. `a` is at offset 0, `b` is at offset 1, and `c` is at offset 4. There is a gap of 2 bytes before `c`.

Contrast that with this structure:

```
struct foo
{
  char a;
  int c;
  char b;
};
```

This structure has size 12 and alignment 4. `a` is at offset 0, `c` is at offset 4, and `b` is at offset 8. There are two gaps: three bytes before `c`, and three bytes at the end.

These two structures have the same contents at the C level, but one takes 8 bytes and the other takes 12 bytes due to the ordering of the fields. A reliable way to avoid this sort of wastage is to order the fields by size, biggest fields first.

15.6 Packed Structures

In GNU C you can force a structure to be laid out with no gaps by adding `__attribute__((packed))` after `struct` (or at the end of the structure type declaration). Here's an example:

```
struct __attribute__((packed)) foo
{
  char a;
  int c;
```

```
    char b;
  };
```

Without `__attribute__((packed))`, this structure occupies 12 bytes (as described in the previous section), assuming 4-byte alignment for `int`. With `__attribute__ ((packed))`, it is only 6 bytes long—the sum of the lengths of its fields.

Use of `__attribute__((packed))` often results in fields that don't have the normal alignment for their types. Taking the address of such a field can result in an invalid pointer because of its improper alignment. Dereferencing such a pointer can cause a `SIGSEGV` signal on a machine that doesn't, in general, allow unaligned pointers.

See Appendix D [Attributes], page 222.

15.7 Bit Fields

A structure field declaration with an integer type can specify the number of bits the field should occupy. We call that a *bit field*. These are useful because consecutive bit fields are packed into a larger storage unit. For instance,

```
unsigned char opcode: 4;
```

specifies that this field takes just 4 bits. Since it is unsigned, its possible values range from 0 to 15. A signed field with 4 bits, such as this,

```
signed char small: 4;
```

can hold values from -8 to 7.

You can subdivide a single byte into those two parts by writing

```
unsigned char opcode: 4;
signed char small: 4;
```

in the structure. With bit fields, these two numbers fit into a single `char`.

Here's how to declare a one-bit field that can hold either 0 or 1:

```
unsigned char special_flag: 1;
```

You can also use the `bool` type for bit fields:

```
bool special_flag: 1;
```

Except when using `bool` (which is always unsigned, see Section 11.1.5 [Boolean Type], page 48), always specify `signed` or `unsigned` for a bit field. There is a default, if that's not specified: the bit field is signed if plain `char` is signed, except that the option `-funsigned-bitfields` forces unsigned as the default. But it is cleaner not to depend on this default.

Bit fields are special in that you cannot take their address with '`&`'. They are not stored with the size and alignment appropriate for the specified type, so they cannot be addressed through pointers to that type.

15.8 Bit Field Packing

Programs to communicate with low-level hardware interfaces need to define bit fields laid out to match the hardware data. This section explains how to do that.

Consecutive bit fields are packed together, but each bit field must fit within a single object of its specified type. In this example,

```
unsigned short a : 3, b : 3, c : 3, d : 3, e : 3;
```

all five fields fit consecutively into one two-byte `short`. They need 15 bits, and one `short` provides 16. By contrast,

```
unsigned char a : 3, b : 3, c : 3, d : 3, e : 3;
```

needs three bytes. It fits `a` and `b` into one `char`, but `c` won't fit in that `char` (they would add up to 9 bits). So `c` and `d` go into a second `char`, leaving a gap of two bits between `b` and `c`. Then `e` needs a third `char`. By contrast,

```
unsigned char a : 3, b : 3;
unsigned int c : 3;
unsigned char d : 3, e : 3;
```

needs only two bytes: the type `unsigned int` allows `c` to straddle bytes that are in the same word.

You can leave a gap of a specified number of bits by defining a nameless bit field. This looks like *`type`* `:` *`nbits`*`;`. It is allocated space in the structure just as a named bit field would be allocated.

You can force the following bit field to advance to the following aligned memory object with *`type`* `: 0;`.

Both of these constructs can syntactically share *type* with ordinary bit fields. This example illustrates both:

```
unsigned int a : 5, : 3, b : 5, : 0, c : 5, : 3, d : 5;
```

It puts `a` and `b` into one `int`, with a 3-bit gap between them. Then `: 0` advances to the next `int`, so `c` and `d` fit into that one.

These rules for packing bit fields apply to most target platforms, including all the usual real computers. A few embedded controllers have special layout rules.

15.9 `const` Fields

A structure field declared `const` cannot be assigned to (see Section 21.1 [const], page 128). For instance, let's define this modified version of `struct intlistlink`:

```
struct intlistlink_ro  /* “ro” for read-only.  */
  {
    const int datum;
    struct intlistlink *next;
  };
```

This structure can be used to prevent part of the code from modifying the `datum` field:

```
/* p has type struct intlistlink *.
   Convert it to struct intlistlink_ro *.  */
struct intlistlink_ro *q
  = (struct intlistlink_ro *) p;

q->datum = 5;      /* Error! */
p->datum = 5;      /* Valid since *p is
                      not a struct intlistlink_ro.  */
```

A `const` field can get a value in two ways: by initialization of the whole structure, and by making a pointer-to-structure point to an object in which that field already has a value.

Any `const` field in a structure type makes assignment impossible for structures of that type (see Section 15.13 [Structure Assignment], page 82). That is because structure assignment works by assigning the structure's fields, one by one.

15.10 Arrays of Length Zero

GNU C allows zero-length arrays. They are useful as the last field of a structure that is really a header for a variable-length object. Here's an example, where we construct a variable-size structure to hold a line which is `this_length` characters long:

```
struct line {
  int length;
  char contents[0];
};

struct line *thisline
  = ((struct line *)
     malloc (sizeof (struct line)
             + this_length));
thisline->length = this_length;
```

In ISO C90, we would have to give `contents` a length of 1, which means either wasting space or complicating the argument to `malloc`.

15.11 Flexible Array Fields

The C99 standard adopted a more complex equivalent of zero-length array fields. It's called a *flexible array*, and it's indicated by omitting the length, like this:

```
struct line
{
  int length;
  char contents[];
};
```

The flexible array has to be the last field in the structure, and there must be other fields before it.

Under the C standard, a structure with a flexible array can't be part of another structure, and can't be an element of an array.

GNU C allows static initialization of flexible array fields. The effect is to "make the array long enough" for the initializer.

```
struct f1 { int x; int y[]; } f1
  = { 1, { 2, 3, 4 } };
```

This defines a structure variable named `f1` whose type is `struct f1`. In C, a variable name or function name never conflicts with a structure type tag.

Omitting the flexible array field's size lets the initializer determine it. This is allowed only when the flexible array is defined in the outermost structure and you declare a variable of that structure type. For example:

```
struct foo { int x; int y[]; };
struct bar { struct foo z; };

struct foo a = { 1, { 2, 3, 4 } };         // Valid.
struct bar b = { { 1, { 2, 3, 4 } } };     // Invalid.
struct bar c = { { 1, { } } };             // Valid.
struct foo d[1] = { { 1 { 2, 3, 4 } } };  // Invalid.
```

15.12 Overlaying Different Structures

Be careful about using different structure types to refer to the same memory within one function, because GNU C can optimize code assuming it never does that. See Appendix B [Aliasing], page 218. Here's an example of the kind of aliasing that can cause the problem:

```
struct a { int size; char *data; };
struct b { int size; char *data; };
struct a foo;
struct a *p = &foo;
struct b *q = (struct b *) &foo;
```

Here `q` points to the same memory that the variable `foo` occupies, but they have two different types. The two types `struct a` and `struct b` are defined alike, but they are not the same type. Interspersing references using the two types, like this,

```
p->size = 0;
q->size = 1;
x = p->size;
```

allows GNU C to assume that `p->size` is still zero when it is copied into `x`. The GNU C compiler "knows" that `q` points to a `struct b` and this is not supposed to overlap with a `struct a`. Other compilers might also do this optimization.

The ISO C standard considers such code erroneous, precisely so that this optimization will not be incorrect.

15.13 Structure Assignment

Assignment operating on a structure type copies the structure. The left and right operands must have the same type. Here is an example:

```
#include <stddef.h>  /* Defines NULL. */
#include <stdlib.h>  /* Declares malloc.  */
...

struct point { double x, y; };

struct point *
copy_point (struct point point)
{
  struct point *p
    = (struct point *) malloc (sizeof (struct point));
  if (p == NULL)
    fatal ("Out of memory");
    *p = point;
    return p;
  }
```

Notionally, assignment on a structure type works by copying each of the fields. Thus, if any of the fields has the `const` qualifier, that structure type does not allow assignment:

```
struct point { const double x, y; };

struct point a, b;

a = b;              /* Error! */
```

See Chapter 7 [Assignment Expressions], page 30.

When a structure type has a field which is an array, as here,

```
struct record
  {
    char *name;
    int data[4];
  };

struct record r1, r2;
```

structure assignment such as `r1 = r2` copies array fields' contents just as it copies all the other fields.

This is the only way in C that you can operate on the whole contents of a array with one operation: when the array is contained in a `struct`. You can't copy the contents of the `data` field as an array, because

```
r1.data = r2.data;
```

would convert the array objects (as always) to pointers to the zeroth elements of the arrays (of type `int *`), and the assignment would be invalid because the left operand is not an lvalue.

15.14 Unions

A *union type* defines alternative ways of looking at the same piece of memory. Each alternative view is defined with a data type, and identified by a name. Because a union combines various types, it is considered a *compound type*, like structures (see Chapter 15 [Structures], page 74). A union definition looks like this:

```
union name
{
  alternative declarations...
};
```

Each alternative declaration looks like a structure field declaration, except that it can't be a bit field. For instance,

```
union number
{
  long int integer;
  double float;
}
```

lets you store either an integer (type `long int`) or a floating point number (type `double`) in the same place in memory. The length and alignment of the union type are the maximum of all the alternatives—they do not have to be the same. In this union example, `double` probably takes more space than `long int`, but that doesn't cause a problem in programs that use the union in the normal way.

The members don't have to be different in data type. Sometimes each member pertains to a way the data will be used. For instance,

```
union datum
{
  double latitude;
  double longitude;
  double height;
  double weight;
  int continent;
}
```

This union holds one of several kinds of data; most kinds are floating points, but the value can also be a code for a continent which is an integer. You *could* use one member of type `double` to access all the values which have that type, but the different member names will make the program clearer.

The alignment of a union type is the maximum of the alignments of the alternatives. The size of the union type is the maximum of the sizes of the alternatives, rounded up to a multiple of the alignment (because every type's size must be a multiple of its alignment).

All the union alternatives start at the address of the union itself. If an alternative is shorter than the union as a whole, it occupies the first part of the union's storage, leaving the last part unused *for that alternative*.

Warning: If the code stores data using one union alternative and accesses it with another, the results depend on the kind of computer in use. Only wizards should try to do this. However, when you need to do this, a union is a clean way to do it.

Assignment works on any union type by copying the entire value.

15.15 Packing With Unions

Sometimes we design a union with the intention of packing various kinds of objects into a certain amount of memory space. For example.

```
union bytes8
{
  long long big_int_elt;
  double double_elt;
  struct { int first, second; } two_ints;
  struct { void *first, *second; } two_ptrs;
};

union bytes8 *p;
```

This union makes it possible to look at 8 bytes of data that `p` points to as a single 8-byte integer (`p->big_int_elt`), as a single floating-point number (`p->double_elt`), as a pair of integers (`p->two_ints.first` and `p->two_ints.second`), or as a pair of pointers (`p->two_ptrs.first` and `p->two_ptrs.second`).

To pack storage with such a union makes assumptions about the sizes of all the types involved. This particular union was written expecting a pointer to have the same size as `int`. On a machine where one pointer takes 8 bytes, the code using this union probably won't work as expected. The union, as such, will function correctly—if you store two values through `two_ints` and extract them through `two_ints`, you will get the same integers back—but the part of the program that expects the union to be 8 bytes long could malfunction, or at least use too much space.

The above example shows one case where a `struct` type with no tag can be useful. Another way to get effectively the same result is with arrays as members of the union:

```
union eight_bytes
{
  long long big_int_elt;
  double double_elt;
  int two_ints[2];
  void *two_ptrs[2];
};
```

15.16 Cast to a Union Type

In GNU C, you can explicitly cast any of the alternative types to the union type; for instance,

```
(union eight_bytes) (long long) 5
```

makes a value of type `union eight_bytes` which gets its contents through the alternative named `big_int_elt`.

The value being cast must exactly match the type of the alternative, so this is not valid:

```
(union eight_bytes) 5  /* Error!  5 is int. */
```

A cast to union type looks like any other cast, except that the type specified is a union type. You can specify the type either with `union` *`tag`* or with a typedef name (see Chapter 18 [Defining Typedef Names], page 101).

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in an alternative of the union:

```
union foo u;

u = (union foo) x   means   u.i = x

u = (union foo) y   means   u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
...
hack ((union foo) x);
```

15.17 Structure Constructors

You can construct a structure value by writing its type in parentheses, followed by an initializer that would be valid in a declaration for that type. For instance, given this declaration,

```
struct foo {int a; char b[2];} structure;
```

you can create a `struct foo` value as follows:

```
((struct foo) {x + y, 'a', 0})
```

This specifies `x + y` for field `a`, the character '`a`' for field `b`'s element 0, and the null character for field `b`'s element 1.

The parentheses around that constructor are not necessary, but we recommend writing them to make the nesting of the containing expression clearer.

You can also show the nesting of the two by writing it like this:

```
((struct foo) {x + y, {'a', 0} })
```

Each of those is equivalent to writing the following statement expression (see Section 19.15 [Statement Exprs], page 118):

```
({
  struct foo temp = {x + y, 'a', 0};
  temp;
})
```

You can also use field labels in the structure constructor to indicate which fields you're specifying values for, instead of using the order of the fields to specify that:

```
(struct foo) {.a = x + y, .b = {'a', 0}}
```

You can also create a union value this way, but it is not especially useful since that is equivalent to doing a cast:

```
  ((union whosis) {value})
is equivalent to
  ((union whosis) (value))
```

15.18 Unnamed Types as Fields

A structure or a union can contain, as fields, unnamed structures and unions. Here's an example:

```
struct
{
  int a;
  union
  {
    int b;
    float c;
  };
  int d;
} foo;
```

You can access the fields of the unnamed union within `foo` as if they were individual fields at the same level as the union definition:

```
foo.a = 42;
foo.b = 47;
foo.c = 5.25; // Overwrites the value in foo.b.
foo.d = 314;
```

Avoid using field names that could cause ambiguity. For example, with this definition:

```
struct
{
  int a;
  struct
  {
    int a;
    float b;
  };
} foo;
```

it is impossible to tell what `foo.a` refers to. GNU C reports an error when a definition is ambiguous in this way.

15.19 Incomplete Types

A type that has not been fully defined is called an *incomplete type*. Structure and union types are incomplete when the code makes a forward reference, such as `struct foo`, before defining the type. An array type is incomplete when its length is unspecified.

You can't use an incomplete type to declare a variable or field, or use it for a function parameter or return type. The operators `sizeof` and `_Alignof` give errors when used on an incomplete type.

However, you can define a pointer to an incomplete type, and declare a variable or field with such a pointer type. In general, you can do everything with such pointers except dereference them. For example:

```
extern void bar (struct mysterious_value *);

void
foo (struct mysterious_value *arg)
{
  bar (arg);
}

...

{
  struct mysterious_value *p, **q;

  p = *q;
  foo (p);
}
```

These examples are valid because the code doesn't try to understand what `p` points to; it just passes the pointer around. (Presumably `bar` is defined in some other file that really does have a definition for `struct mysterious_value`.) However, dereferencing the pointer would get an error; that requires a definition for the structure type.

15.20 Intertwined Incomplete Types

When several structure types contain pointers to each other, you can define the types in any order because pointers to types that come later are incomplete types. Thus, Here is an example.

```
/* An employee record points to a group.  */
struct employee
{
  char *name;
  ...
  struct group *group;  /* incomplete type.  */
  ...
};

/* An employee list points to employees.  */
struct employee_list
{
  struct employee *this_one;
  struct employee_list *next;  /* incomplete type.  */
  ...
};

/* A group points to one employee_list.  */
struct group
{
  char *name;
  ...
  struct employee_list *employees;
  ...
};
```

15.21 Type Tags

The name that follows `struct` (see Chapter 15 [Structures], page 74), `union` (see Section 15.14 [Unions], page 83, or `enum` (see Chapter 17 [Enumeration Types], page 99) is called a *type tag*. In C, a type tag never conflicts with a variable name or function name; the type tags have a separate *name space*. Thus, there is no name conflict in this code:

```
struct pair { int a, b; };
int pair = 1;
```

nor in this one:

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

where `pair` is both a structure type tag and a variable name.

However, `struct`, `union`, and `enum` share the same name space of tags, so this is a conflict:

```
struct pair { int a, b; };
enum pair { c, d };
```

and so is this:

```
struct pair { int a, b; };
struct pair { int c, d; };
```

When the code defines a type tag inside a block, the tag's scope is limited to that block (as for local variables). Two definitions for one type tag do not conflict if they are in different scopes; rather, each is valid in its scope. For example,

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

void
pair_up_doubles (int len, double array[])
{
  struct pair { double a, b; };
  ...
}
```

has two definitions for `struct pair` which do not conflict. The one inside the function applies only within the definition of `pair_up_doubles`. Within its scope, that definition *shadows* the outer definition.

If `struct pair` appears inside the function body, before the inner definition, it refers to the outer definition—the only one that has been seen at that point. Thus, in this code,

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

void
pair_up_doubles (int len, double array[])
{
  struct two_pairs { struct pair *p, *q; };
  struct pair { double a, b; };
  ...
}
```

the structure `two_pairs` has pointers to the outer definition of `struct pair`, which is probably not desirable.

To prevent that, you can write `struct pair;` inside the function body as a variable declaration with no variables. This is a *forward declaration* of the type tag `pair`: it makes the type tag local to the current block, with the details of the type to come later. Here's an example:

```
void
pair_up_doubles (int len, double array[])
{
  /* Forward declaration for pair.  */
  struct pair;
  struct two_pairs { struct pair *p, *q; };
```

```
    /* Give the details.  */
    struct pair { double a, b; };
    ...
  }
```

However, the cleanest practice is to avoid shadowing type tags.

16 Arrays

An *array* is a data object that holds a series of *elements*, all of the same data type. Each element is identified by its numeric *index* within the array.

We presented arrays of numbers in the sample programs early in this manual (see Section 4.2 [Array Example], page 14). However, arrays can have elements of any data type, including pointers, structures, unions, and other arrays.

If you know another programming language, you may suppose that you know all about arrays, but C arrays have special quirks, so in this chapter we collect all the information about arrays in C.

The elements of a C array are allocated consecutively in memory, with no gaps between them. Each element is aligned as required for its data type (see Appendix A [Type Alignment], page 217).

16.1 Accessing Array Elements

If the variable `a` is an array, the *n*th element of `a` is `a[n]`. You can use that expression to access an element's value or to assign to it:

```
x = a[5];
a[6] = 1;
```

Since the variable `a` is an lvalue, `a[n]` is also an lvalue.

The lowest valid index in an array is 0, *not* 1, and the highest valid index is one less than the number of elements.

The C language does not check whether array indices are in bounds, so if the code uses an out-of-range index, it will access memory outside the array.

Warning: Using only valid index values in C is the programmer's responsibility.

Array indexing in C is not a primitive operation: it is defined in terms of pointer arithmetic and dereferencing. Now that we know *what* `a[i]` does, we can ask *how* `a[i]` does its job.

In C, `x[y]` is an abbreviation for `*(x+y)`. Thus, `a[i]` really means `*(a+i)`. See Section 14.11 [Pointers and Arrays], page 69.

When an expression with array type (such as `a`) appears as part of a larger C expression, it is converted automatically to a pointer to element zero of that array. For instance, `a` in an expression is equivalent to `&a[0]`. Thus, `*(a+i)` is computed as `*(&a[0]+i)`.

Now we can analyze how that expression gives us the desired element of the array. It makes a pointer to element 0 of `a`, advances it by the value of `i`, and dereferences that pointer.

Another equivalent way to write the expression is `(&a[0])[i]`.

16.2 Declaring an Array

To make an array declaration, write `[length]` after the name being declared. This construct is valid in the declaration of a variable, a function parameter, a function value type (the value can't be an array, but it can be a pointer to one), a structure field, or a union alternative.

The surrounding declaration specifies the element type of the array; that can be any type of data, but not `void` or a function type. For instance,

```
double a[5];
```

declares `a` as an array of 5 `double`s.

```
struct foo bstruct[length];
```

declares `bstruct` as an array of `length` objects of type `struct foo`. A variable array size like this is allowed when the array is not file-scope.

Other declaration constructs can nest within the array declaration construct. For instance:

```
struct foo *b[length];
```

declares `b` as an array of `length` pointers to `struct foo`. This shows that the length need not be a constant (see Section 16.9 [Arrays of Variable Length], page 97).

```
double (*c)[5];
```

declares `c` as a pointer to an array of 5 `double`s, and

```
char *(*f (int))[5];
```

declares `f` as a function taking an `int` argument and returning a pointer to an array of 5 strings (pointers to `char`s).

```
double aa[5][10];
```

declares `aa` as an array of 5 elements, each of which is an array of 10 `double`s. This shows how to declare a multidimensional array in C (see Section 16.7 [Multidimensional Arrays], page 95).

All these declarations specify the array's length, which is needed in these cases in order to allocate storage for the array.

16.3 Strings

A string in C is a sequence of elements of type `char`, terminated with the null character, the character with code zero. However, the C code that operates on strings normally uses the pointer type `char *` to do it.

Programs often need to use strings with specific, fixed contents. To write one in a C program, use a *string constant* such as `"Take me to your leader!"`. The data type of a string constant is `char *`. For the full syntactic details of writing string constants, Section 12.7 [String Constants], page 56.

To declare a place to store a non-constant string, declare an array of `char`. Keep in mind that it must include one extra `char` for the terminating null. For instance,

```
char text[] = { 'H', 'e', 'l', 'l', 'o', 0 };
```

declares an array named '`text`' with six elements—five letters and the terminating null character. An equivalent way to get the same result is this,

```
char text[] = "Hello";
```

which copies the elements of the string constant, including *its* terminating null character.

```
char message[200];
```

declares an array long enough to hold a string of 199 ASCII characters plus the terminating null character.

When you store a string into `message` be sure to check or prove that the length does not exceed its size. For example,

```
void
set_message (char *text)
{
  int i;
  /* Recall that message is declared above.  */
  for (i = 0; i < sizeof (message); i++)
    {
      message[i] = text[i];
      if (text[i] == 0)
        return;
    }
  fatal_error ("Message is too long for `message'\n");
}
```

It's easy to do this with the standard library function `strncpy`, which fills out the whole destination array (up to a specified length) with null characters. Thus, if the last character of the destination is not null, the string did not fit. Many system libraries, including the GNU C library, hand-optimize `strncpy` to run faster than an explicit `for`-loop.

Here's what the code looks like:

```
void
set_message (char *text)
{
  strncpy (message, text, sizeof (message));
  if (message[sizeof (message) - 1] != 0)
    fatal_error ("Message is too long for `message');
}
```

See Section "String and Array Utilities" in *The GNU C Library Reference Manual*, for more information about the standard library functions for operating on strings.

You can avoid putting a fixed length limit on strings you construct or operate on by allocating the space for them dynamically. See Section 15.3 [Dynamic Memory Allocation], page 76.

16.4 Array Type Designators

Every C type has a type designator, which you make by deleting the variable name and the semicolon from a declaration (see Section 11.6 [Type Designators], page 51). The designators for array types follow this rule, but they may appear surprising.

type	`int a[5];`	designator	`int [5]`
type	`double a[5][3];`	designator	`double [5][3]`
type	`struct foo *a[5];`	designator	`struct foo *[5]`

16.5 Incomplete Array Types

An array is equivalent, for most purposes, to a pointer to its zeroth element. When that is true, the length of the array is irrelevant. The length needs to be known only for allocating space for the array, or for `sizeof` and `typeof` (see Section 20.4 [Auto Type], page 124). Thus, in some contexts C allows

- An `extern` declaration says how to refer to a variable allocated elsewhere. It does not need to allocate space for the variable, so if it is an array, you can omit the length. For example,

  ```
  extern int foo[];
  ```

- When declaring a function parameter as an array, the argument value passed to the function is really a pointer to the array's zeroth element. This value does not say how long the array really is, there is no need to declare it. For example,

  ```
  int
  func (int foo[])
  ```

These declarations are examples of *incomplete* array types, types that are not fully specified. The incompleteness makes no difference for accessing elements of the array, but it matters for some other things. For instance, `sizeof` is not allowed on an incomplete type.

With multidimensional arrays, only the first dimension can be omitted. For example, suppose we want to represent the positions of pieces on a chessboard which has the usual 8 files (columns), but more (or fewer) ranks (rows) than the usual 8. This declaration could hold a pointer to a two-dimensional array that can hold that data. Each element of the array holds one row.

```
struct chesspiece *funnyboard[][8];
```

Since it is just a pointer to the start of an array, its type can be incomplete, but it must state how big each array element is—the number of elements in each row.

16.6 Limitations of C Arrays

Arrays have quirks in C because they are not "first-class objects": there is no way in C to operate on an array as a unit.

The other composite objects in C, structures and unions, are first-class objects: a C program can copy a structure or union value in an assignment, or pass one as an argument to a function, or make a function return one. You can't do those things with an array in C. That is because a value you can operate on never has an array type.

An expression in C can have an array type, but that doesn't produce the array as a value. Instead it is converted automatically to a pointer to the array's element at index zero. The code can operate on the pointer, and through that on individual elements of the array, but it can't get and operate on the array as a unit.

There are three exceptions to this conversion rule, but none of them offers a way to operate on the array as a whole.

First, '`&`' applied to an expression with array type gives you the address of the array, as an array type. However, you can't operate on the whole array that way—if you apply '`*`' to get the array back, that expression converts, as usual, to a pointer to its zeroth element.

Second, the operators `sizeof`, `_Alignof`, and `typeof` do not convert the array to a pointer; they leave it as an array. But they don't operate on the array's data—they only give information about its type.

Third, a string constant used as an initializer for an array is not converted to a pointer—rather, the declaration copies the *contents* of that string in that one special case.

You *can* copy the contents of an array, just not with an assignment operator. You can do it by calling the library function `memcpy` or `memmove` (see Section "Copying and Concatenation" in *The GNU C Library Reference Manual*). Also, when a structure contains just an array, you can copy that structure.

An array itself is an lvalue if it is a declared variable, or part of a structure or union that is an lvalue. When you construct an array from elements (see Section 16.8 [Constructing Array Values], page 96), that array is not an lvalue.

16.7 Multidimensional Arrays

Strictly speaking, all arrays in C are unidimensional. However, you can create an array of arrays, which is more or less equivalent to a multidimensional array. For example,

```
struct chesspiece *board[8][8];
```

declares an array of 8 arrays of 8 pointers to `struct chesspiece`. This data type could represent the state of a chess game. To access one square's contents requires two array index operations, one for each dimension. For instance, you can write `board[row][column]`, assuming `row` and `column` are variables with integer values in the proper range.

How does C understand `board[row][column]`? First of all, `board` is converted automatically to a pointer to the zeroth element (at index zero) of `board`. Adding `row` to that makes it point to the desired element. Thus, `board[row]`'s value is an element of `board`—an array of 8 pointers.

However, as an expression with array type, it is converted automatically to a pointer to the array's zeroth element. The second array index operation, `[column]`, accesses the chosen element from that array.

As this shows, pointer-to-array types are meaningful in C. You can declare a variable that points to a row in a chess board like this:

```
struct chesspiece *(*rowptr)[8];
```

This points to an array of 8 pointers to `struct chesspiece`. You can assign to it as follows:

```
rowptr = &board[5];
```

The dimensions don't have to be equal in length. Here we declare `statepop` as an array to hold the population of each state in the United States for each year since 1900:

```
#define NSTATES 50
{
  int nyears = current_year - 1900 + 1;
  int statepop[NSTATES][nyears];
  ...
}
```

The variable `statepop` is an array of `NSTATES` subarrays, each indexed by the year (counting from 1900). Thus, to get the element for a particular state and year, we must subscript it first by the number that indicates the state, and second by the index for the year:

```
statepop[state][year - 1900]
```

The subarrays within the multidimensional array are allocated consecutively in memory, and within each subarray, its elements are allocated consecutively in memory. The most efficient way to process all the elements in the array is to scan the last subscript in the innermost loop. This means consecutive accesses go to consecutive memory locations, which optimizes use of the processor's memory cache. For example:

```
int total = 0;
float average;

for (int state = 0; state < NSTATES, ++state)
  {
    for (int year = 0; year < nyears; ++year)
      {
        total += statepop[state][year];
      }
  }

average = total / nyears;
```

C's layout for multidimensional arrays is different from Fortran's layout. In Fortran, a multidimensional array is not an array of arrays; rather, multidimensional arrays are a primitive feature, and it is the first index that varies most rapidly between consecutive memory locations. Thus, the memory layout of a 50x114 array in C matches that of a 114x50 array in Fortran.

16.8 Constructing Array Values

You can construct an array from elements by writing them inside braces, and preceding all that with the array type's designator in parentheses. There is no need to specify the array length, since the number of elements determines that. The constructor looks like this:

```
(elttype[]) { elements };
```

Here is an example, which constructs an array of string pointers:

```
(char *[]) { "x", "y", "z" };
```

That's equivalent in effect to declaring an array with the same initializer, like this:

```
char *array[] = { "x", "y", "z" };
```

and then using the array.

If all the elements are simple constant expressions, or made up of such, then the compound literal can be coerced to a pointer to its zeroth element and used to initialize a file-scope variable (see Section 20.6 [File-Scope Variables], page 125), as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

The data type of `foo` is `char **`, which is a pointer type, not an array type. The declaration is equivalent to defining and then using an array-type variable:

```
char *nameless_array[] = { "x", "y", "z" };
char **foo = &nameless_array[0];
```

16.9 Arrays of Variable Length

In GNU C, you can declare variable-length arrays like any other arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the block scope containing the declaration exits. For example:

```
#include <stdio.h>  /* Defines FILE. */
#include <string.h> /* Declares str. */

FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

(This uses some standard library functions; see Section "String and Array Utilities" in *The GNU C Library Reference Manual*.)

The length of an array is computed once when the storage is allocated and is remembered for the scope of the array in case it is used in `sizeof`.

Warning: Don't allocate a variable-length array if the size might be very large (more than 100,000), or in a recursive function, because that is likely to cause stack overflow. Allocate the array dynamically instead (see Section 15.3 [Dynamic Memory Allocation], page 76).

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; that gives an error message.

You can also use variable-length arrays as arguments to functions:

```
struct entry
tester (int len, char data[len][len])
{
  ...
}
```

As usual, a function argument declared with an array type is really a pointer to an array that already exists. Calling the function does not allocate the array, so there's no particular danger of stack overflow in using this construct.

To pass the array first and the length afterward, use a forward declaration in the function's parameter list (another GNU extension). For example,

```
struct entry
tester (int len; char data[len][len], int len)
{
  ...
}
```

The `int len` before the semicolon is a *parameter forward declaration*, and it serves the purpose of making the name `len` known when the declaration of `data` is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the “real” parameter declarations. Each forward declaration must match a “real” declaration in parameter name and data type. ISO C11 does not support parameter forward declarations.

17 Enumeration Types

An *enumeration type* represents a limited set of integer values, each with a name. It is effectively equivalent to a primitive integer type.

Suppose we have a list of possible emotional states to store in an integer variable. We can give names to these alternative values with an enumeration:

```
enum emotion_state { neutral, happy, sad, worried,
                     calm, nervous };
```

(Never mind that this is a simplistic way to classify emotional states; it's just a code example.)

The names inside the enumeration are called *enumerators*. The enumeration type defines them as constants, and their values are consecutive integers; `neutral` is 0, `happy` is 1, `sad` is 2, and so on. Alternatively, you can specify values for the enumerators explicitly like this:

```
enum emotion_state { neutral = 2, happy = 5,
                     sad = 20, worried = 10,
                     calm = -5, nervous = -300 };
```

Each enumerator which does not specify a value gets value zero (if it is at the beginning) or the next consecutive integer.

```
/* neutral is 0 by default,
   and worried is 21 by default.  */
enum emotion_state { neutral,
                      happy = 5, sad = 20, worried,
                      calm = -5, nervous = -300 };
```

If an enumerator is obsolete, you can specify that using it should cause a warning, by including an attribute in the enumerator's declaration. Here is how `happy` would look with this attribute:

```
happy __attribute__
      ((deprecated
        ("impossible under plutocratic rule")))
      = 5,
```

See Appendix D [Attributes], page 222.

You can declare variables with the enumeration type:

```
enum emotion_state feelings_now;
```

In the C code itself, this is equivalent to declaring the variable `int`. (If all the enumeration values are positive, it is equivalent to `unsigned int`.) However, declaring it with the enumeration type has an advantage in debugging, because GDB knows it should display the current value of the variable using the corresponding name. If the variable's type is `int`, GDB can only show the value as a number.

The identifier that follows `enum` is called a *type tag* since it distinguishes different enumeration types. Type tags are in a separate name space and belong to scopes like most other names in C. See Section 15.21 [Type Tags], page 88, for explanation.

You can predeclare an `enum` type tag like a structure or union type tag, like this:

```
enum foo;
```

The `enum` type is incomplete until you finish defining it.

You can optionally include a trailing comma at the end of a list of enumeration values:

```
enum emotion_state { neutral, happy, sad, worried,
                     calm, nervous, };
```

This is useful in some macro definitions, since it enables you to assemble the list of enumerators without knowing which one is last. The extra comma does not change the meaning of the enumeration in any way.

18 Defining Typedef Names

You can define a data type keyword as an alias for any type, and then use the alias syntactically like a built-in type keyword such as `int`. You do this using `typedef`, so these aliases are also called *typedef names*.

`typedef` is followed by text that looks just like a variable declaration, but instead of declaring variables it defines data type keywords.

Here's how to define `fooptr` as a typedef alias for the type `struct foo *`, then declare `x` and `y` as variables with that type:

```
typedef struct foo *fooptr;

fooptr x, y;
```

That declaration is equivalent to the following one:

```
struct foo *x, *y;
```

You can define a typedef alias for any type. For instance, this makes `frobcount` an alias for type `int`:

```
typedef int frobcount;
```

This doesn't define a new type distinct from `int`. Rather, `frobcount` is another name for the type `int`. Once the variable is declared, it makes no difference which name the declaration used.

There is a syntactic difference, however, between `frobcount` and `int`: A typedef name cannot be used with `signed`, `unsigned`, `long` or `short`. It has to specify the type all by itself. So you can't write this:

```
unsigned frobcount f1;  /* Error! */
```

But you can write this:

```
typedef unsigned int unsigned_frobcount;

unsigned_frobcount f1;
```

In other words, a typedef name is not an alias for *a keyword* such as `int`. It stands for a *type*, and that could be the type `int`.

Typedef names are in the same namespace as functions and variables, so you can't use the same name for a typedef and a function, or a typedef and a variable. When a typedef is declared inside a code block, it is in scope only in that block.

Warning: Avoid defining typedef names that end in '`_t`', because many of these have standard meanings.

You can redefine a typedef name to the exact same type as its first definition, but you cannot redefine a typedef name to a different type, even if the two types are compatible. For example, this is valid:

```
typedef int frobcount;
typedef int frotzcount;
typedef frotzcount frobcount;
typedef frobcount frotzcount;
```

because each typedef name is always defined with the same type (`int`), but this is not valid:

```
enum foo {f1, f2, f3};
typedef enum foo frobcount;
typedef int frobcount;
```

Even though the type `enum foo` is compatible with `int`, they are not the *same* type.

19 Statements

A *statement* specifies computations to be done for effect; it does not produce a value, as an expression would. In general a statement ends with a semicolon (‘`;`’), but blocks (which are statements, more or less) are an exception to that rule.

The places to use statements are inside a block, and inside a complex statement. A *complex statement* contains one or two components that are nested statements. Each such component must consist of one and only one statement. The way to put multiple statements in such a component is to group them into a *block* (see Section 19.4 [Blocks], page 104), which counts as one statement.

The following sections describe the various kinds of statement.

19.1 Expression Statement

The most common kind of statement in C is an *expression statement*. It consists of an expression followed by a semicolon. The expression’s value is discarded, so the expressions that are useful are those that have side effects: assignment expressions, increment and decrement expressions, and function calls. Here are examples of expression statements:

```
x = 5;              /* Assignment expression. */
p++;                /* Increment expression. */
printf ("Done\n");  /* Function call expression. */
*p;                 /* Cause SIGSEGV signal if p is null. */
x + y;              /* Useless statement without effect. */
```

In very unusual circumstances we use an expression statement whose purpose is to get a fault if an address is invalid:

```
volatile char *p;
...
*p;                 /* Cause signal if p is null. */
```

If the target of `p` is not declared `volatile`, the compiler might optimize away the memory access, since it knows that the value isn’t really used. See Section 21.2 [volatile], page 129.

19.2 `if` Statement

An `if` statement computes an expression to decide whether to execute the following statement or not. It looks like this:

```
if (condition)
  execute-if-true
```

The first thing this does is compute the value of *condition*. If that is true (nonzero), then it executes the statement *execute-if-true*. If the value of *condition* is false (zero), it doesn’t execute *execute-if-true*; instead, it does nothing.

This is a *complex statement* because it contains a component *if-true-substatement* that is a nested statement. It must be one and only one statement. The way to put multiple statements there is to group them into a *block* (see Section 19.4 [Blocks], page 104).

19.3 `if-else` Statement

An `if-else` statement computes an expression to decide which of two nested statements to execute. It looks like this:

```
if (condition)
  if-true-substatement
else
  if-false-substatement
```

The first thing this does is compute the value of *condition*. If that is true (nonzero), then it executes the statement *if-true-substatement*. If the value of *condition* is false (zero), then it executes the statement *if-false-substatement* instead.

This is a *complex statement* because it contains components *if-true-substatement* and *if-else-substatement* that are nested statements. Each must be one and only one statement. The way to put multiple statements in such a component is to group them into a *block* (see Section 19.4 [Blocks], page 104).

19.4 Blocks

A *block* is a construct that contains multiple statements of any kind. It begins with '`{`' and ends with '`}`', and has a series of statements and declarations in between. Another name for blocks is *compound statements*.

Is a block a statement? Yes and no. It doesn't *look* like a normal statement—it does not end with a semicolon. But you can *use* it like a statement; anywhere that a statement is required or allowed, you can write a block and consider that block a statement.

So far it seems that a block is a kind of statement with an unusual syntax. But that is not entirely true: a function body is also a block, and that block is definitely not a statement. The text after a function header is not treated as a statement; only a function body is allowed there, and nothing else would be meaningful there.

In a formal grammar we would have to choose—either a block is a kind of statement or it is not. But this manual is meant for humans, not for parser generators. The clearest answer for humans is, "a block is a statement, in some ways."

A block that isn't a function body is called an *internal block* or a *nested block*. You can put a nested block directly inside another block, but more often the nested block is inside some complex statement, such as a `for` statement or an `if` statement.

There are two uses for nested blocks in C:

- To specify the scope for local declarations. For instance, a local variable's scope is the rest of the innermost containing block.
- To write a series of statements where, syntactically, one statement is called for. For instance, the *execute-if-true* of an `if` statement is one statement. To put multiple statements there, they have to be wrapped in a block, like this:

```
if (x < 0)
  {
    printf ("x was negative\n");
    x = -x;
  }
```

This example (repeated from above) shows a nested block which serves both purposes: it includes two statements (plus a declaration) in the body of a `while` statement, and it provides the scope for the declaration of `q`.

```
void
free_intlist (struct intlistlink *p)
{
  while (p)
    {
      struct intlistlink *q = p;
      p = p->next;
      free (q);
    }
}
```

19.5 `return` Statement

The `return` statement makes the containing function return immediately. It has two forms. This one specifies no value to return:

```
return;
```

That form is meant for functions whose return type is `void` (see Section 11.4 [The Void Type], page 50). You can also use it in a function that returns nonvoid data, but that's a bad idea, since it makes the function return garbage.

The form that specifies a value looks like this:

```
return value;
```

which computes the expression *value* and makes the function return that. If necessary, the value undergoes type conversion to the function's declared return value type, which works like assigning the value to a variable of that type.

19.6 Loop Statements

You can use a loop statement when you need to execute a series of statements repeatedly, making an *iteration*. C provides several different kinds of loop statements, described in the following subsections.

Every kind of loop statement is a complex statement because contains a component, here called *body*, which is a nested statement. Most often the body is a block.

19.6.1 `while` Statement

The `while` statement is the simplest loop construct. It looks like this:

```
while (test)
  body
```

Here, *body* is a statement (often a nested block) to repeat, and *test* is the test expression that controls whether to repeat it again. Each iteration of the loop starts by computing *test* and, if it is true (nonzero), that means the loop should execute *body* again and then start over.

Here's an example of advancing to the last structure in a chain of structures chained through the `next` field:

```
#include <stddef.h> /* Defines NULL. */
...
while (chain->next != NULL)
  chain = chain->next;
```

This code assumes the chain isn't empty to start with; if the chain is empty (that is, if `chain` is a null pointer), the code gets a `SIGSEGV` signal trying to dereference that null pointer (see Appendix E [Signals], page 224).

19.6.2 `do-while` Statement

The `do–while` statement is a simple loop construct that performs the test at the end of the iteration.

```
do
  body
while (test);
```

Here, *body* is a statement (possibly a block) to repeat, and *test* is an expression that controls whether to repeat it again.

Each iteration of the loop starts by executing *body*. Then it computes *test* and, if it is true (nonzero), that means to go back and start over with *body*. If *test* is false (zero), then the loop stops repeating and execution moves on past it.

19.6.3 `break` Statement

The `break` statement looks like '`break;`'. Its effect is to exit immediately from the innermost loop construct or `switch` statement (see Section 19.7 [switch Statement], page 110).

For example, this loop advances `p` until the next null character or newline.

```
while (*p)
  {
    /* End loop if we have reached a newline.  */
    if (*p == '\n')
      break;
    p++
  }
```

When there are nested loops, the `break` statement exits from the innermost loop containing it.

```
struct list_if_tuples
{
  struct list_if_tuples next;
  int length;
  data *contents;
};

void
process_all_elements (struct list_if_tuples *list)
{
    while (list)
      {
        /* Process all the elements in this node's vector,
           stopping when we reach one that is null.  */
        for (i = 0; i < list->length; i++)
          {
            /* Null element terminates this node's vector.  */
            if (list->contents[i] == NULL)
              /* Exit the for loop.  */
              break;
            /* Operate on the next element.  */
            process_element (list->contents[i]);
          }

        list = list->next;
      }
  }
```

The only way in C to exit from an outer loop is with `goto` (see Section 19.12 [goto Statement], page 114).

19.6.4 `for` Statement

A `for` statement uses three expressions written inside a parenthetical group to define the repetition of the loop. The first expression says how to prepare to start the loop. The second says how to test, before each iteration, whether to continue looping. The third says how to advance, at the end of an iteration, for the next iteration. All together, it looks like this:

```
for (start; continue-test; advance)
  body
```

The first thing the `for` statement does is compute *start*. The next thing it does is compute the expression *continue-test*. If that expression is false (zero), the `for` statement finishes immediately, so *body* is executed zero times.

However, if *continue-test* is true (nonzero), the `for` statement executes *body*, then *advance*. Then it loops back to the not-quite-top to test *continue-test* again. But it does not compute *start* again.

19.6.5 Example of `for`

Here is the `for` statement from the iterative Fibonacci function:

```
int i;
for (i = 1; i < n; ++i)
  /* If n is 1 or less, the loop runs zero times,  */
  /* since i < n is false the first time.  */
  {
    /* Now last is fib (i)
       and prev is fib (i − 1).  */
    /* Compute fib (i + 1).  */
    int next = prev + last;
    /* Shift the values down.  */
    prev = last;
    last = next;
    /* Now last is fib (i + 1)
       and prev is fib (i).
       But that won't stay true for long,
       because we are about to increment i.  */
  }
```

In this example, *start* is `i = 1`, meaning set `i` to 1. *continue-test* is `i < n`, meaning keep repeating the loop as long as `i` is less than `n`. *advance* is `i++`, meaning increment `i` by 1. The body is a block that contains a declaration and two statements.

19.6.6 Omitted `for`-Expressions

A fully-fleshed `for` statement contains all these parts,

```
for (start; continue-test; advance)
  body
```

but you can omit any of the three expressions inside the parentheses. The parentheses and the two semicolons are required syntactically, but the expressions between them may be missing. A missing expression means this loop doesn't use that particular feature of the `for` statement.

Instead of using *start*, you can do the loop preparation before the `for` statement: the effect is the same. So we could have written the beginning of the previous example this way:

```
int i = 0;
for (; i < n; ++i)
```

instead of this way:

```
int i;
for (i = 0; i < n; ++i)
```

Omitting *continue-test* means the loop runs forever (or until something else causes exit from it). Statements inside the loop can test conditions for termination and use '`break;`' to exit. This is more flexible since you can put those tests anywhere in the loop, not solely at the beginning.

Putting an expression in *advance* is almost equivalent to writing it at the end of the loop body; it does almost the same thing. The only difference is for the `continue` statement (see Section 19.6.8 [continue Statement], page 109). So we could have written this:

```
for (i = 0; i < n;)
  {
    ...
    ++i;
  }
```

instead of this:

```
for (i = 0; i < n; ++i)
  {
    ...
  }
```

The choice is mainly a matter of what is more readable for programmers. However, there is also a syntactic difference: *advance* is an expression, not a statement. It can't include loops, blocks, declarations, etc.

19.6.7 `for`-Index Declarations

You can declare loop-index variables directly in the *start* portion of the `for`-loop, like this:

```
for (int i = 0; i < n; ++i)
  {
    ...
  }
```

This kind of *start* is limited to a single declaration; it can declare one or more variables, separated by commas, all of which are the same *basetype* (`int`, in this example):

```
for (int i = 0, j = 1, *p = NULL; i < n; ++i, ++j, ++p)
  {
    ...
  }
```

The scope of these variables is the `for` statement as a whole. See Section 20.1 [Variable Declarations], page 120, for a explanation of *basetype*.

Variables declared in `for` statements should have initializers. Omitting the initialization gives the variables unpredictable initial values, so this code is erroneous.

```
for (int i; i < n; ++i)
  {
    ...
  }
```

19.6.8 `continue` Statement

The `continue` statement looks like '`continue;`', and its effect is to jump immediately to the end of the innermost loop construct. If it is a `for`-loop, the next thing that happens is to execute the loop's *advance* expression.

For example, this loop increments `p` until the next null character or newline, and operates (in some way not shown) on all the characters in the line except for spaces. All it does with spaces is skip them.

```
for (;*p; ++p)
  {
    /* End loop if we have reached a newline.  */
    if (*p == '\n')
      break;
    /* Pay no attention to spaces.  */
    if (*p == ' ')
      continue;
    /* Operate on the next character.  */
    ...
  }
```

Executing '`continue;`' skips the loop body but it does not skip the *advance* expression, `p++`.

We could also write it like this:

```
for (;*p; ++p)
  {
    /* Exit if we have reached a newline.  */
    if (*p == '\n')
      break;
    /* Pay no attention to spaces.  */
    if (*p != ' ')
      {
        /* Operate on the next character.  */
        ...
      }
  }
```

The advantage of using `continue` is that it reduces the depth of nesting.

Contrast `continue` with the `break` statement. See Section 19.6.3 [break Statement], page 106.

19.7 `switch` Statement

The `switch` statement selects code to run according to the value of an expression. The expression, in parentheses, follows the keyword `switch`. After that come all the cases to select among, inside braces. It looks like this:

```
switch (selector)
  {
    cases...
  }
```

A case can look like this:

```
case value:
  statements
  break;
```

which means "come here if *selector* happens to have the value *value*," or like this (a GNU C extension):

```
case rangestart ... rangeend:
  statements
  break;
```

which means "come here if *selector* happens to have a value between *rangestart* and *rangeend* (inclusive)." See Section 19.10 [Case Ranges], page 113.

The values in `case` labels must reduce to integer constants. They can use arithmetic, and `enum` constants, but they cannot refer to data in memory, because they have to be computed at compile time. It is an error if two `case` labels specify the same value, or ranges that overlap, or if one is a range and the other is a value in that range.

You can also define a default case to handle "any other value," like this:

```
default:
    statements
    break;
```

If the `switch` statement has no `default:` label, then it does nothing when the value matches none of the cases.

The brace-group inside the `switch` statement is a block, and you can declare variables with that scope just as in any other block (see Section 19.4 [Blocks], page 104). However, initializers in these declarations won't necessarily be executed every time the `switch` statement runs, so it is best to avoid giving them initializers.

`break;` inside a `switch` statement exits immediately from the `switch` statement. See Section 19.6.3 [break Statement], page 106.

If there is no `break;` at the end of the code for a case, execution continues into the code for the following case. This happens more often by mistake than intentionally, but since this feature is used in real code, we cannot eliminate it.

Warning: When one case is intended to fall through to the next, write a comment like '`falls through`' to say it's intentional. That way, other programmers won't assume it was an error and "fix" it erroneously.

Consecutive `case` statements could, pedantically, be considered an instance of falling through, but we don't consider or treat them that way because they won't confuse anyone.

19.8 Example of `switch`

Here's an example of using the `switch` statement to distinguish among characters:

```
struct vp { int vowels, punct; };

struct vp
count_vowels_and_punct (char *string)
{
  int c;
  int vowels = 0;
  int punct = 0;
  /* Don't change the parameter itself.  */
  /* That helps in debugging.  */
  char *p = string;
  struct vp value;

  while (c = *p++)
    switch (c)
      {
        case 'y':
        case 'Y':
          /* We assume y_is_consonant will check surrounding
          letters to determine whether this y is a vowel.  */
          if (y_is_consonant (p - 1))
            break;

          /* Falls through */
```

```
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
            case 'A':
            case 'E':
            case 'I':
            case 'O':
            case 'U':
              vowels++;
              break;

            case '.':
            case ',':
            case ':':
            case ';':
            case '?':
            case '!':
            case '\"':
            case '\'':
              punct++;
              break;
          }

      value.vowels = vowels;
      value.punct = punct;

      return value;
    }
```

19.9 Duff's Device

The cases in a `switch` statement can be inside other control constructs. For instance, we can use a technique known as *Duff's device* to optimize this simple function,

```
void
copy (char *to, char *from, int count)
{
  while (count > 0)
    *to++ = *from++, count--;
}
```

which copies memory starting at *from* to memory starting at *to*.

Duff's device involves unrolling the loop so that it copies several characters each time around, and using a `switch` statement to enter the loop body at the proper point:

```
void
```

```
copy (char *to, char *from, int count)
{
  if (count <= 0)
    return;
  int n = (count + 7) / 8;
  switch (count % 8)
    {
      do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
        } while (--n > 0);
    }
}
```

19.10 Case Ranges

You can specify a range of consecutive values in a single `case` label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual `case` labels, one for each integer value from *low* to *high*, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: with integers, write spaces around the `...` to prevent it from being parsed wrong. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

19.11 Null Statement

A *null statement* is just a semicolon. It does nothing.

A null statement is a placeholder for use where a statement is grammatically required, but there is nothing to be done. For instance, sometimes all the work of a `for`-loop is done in the `for`-header itself, leaving no work for the body. Here is an example that searches for the first newline in `array`:

```
for (p = array; *p != '\n'; p++)
  ;
```

19.12 `goto` Statement and Labels

The `goto` statement looks like this:

```
goto label;
```

Its effect is to transfer control immediately to another part of the current function—where the label named *label* is defined.

An ordinary label definition looks like this:

```
label:
```

and it can appear before any statement. You can't use `default` as a label, since that has a special meaning for `switch` statements.

An ordinary label doesn't need a separate declaration; defining it is enough.

Here's an example of using `goto` to implement a loop equivalent to `do`–`while`:

```
{
 loop_restart:
  body
  if (condition)
    goto loop_restart;
}
```

The name space of labels is separate from that of variables and functions. Thus, there is no error in using a single name in both ways:

```
{
  int foo;     // Variable foo.
 foo:          // Label foo.
  body
  if (foo > 0)  // Variable foo.
    goto foo;   // Label foo.
}
```

Blocks have no effect on ordinary labels; each label name is defined throughout the whole of the function it appears in. It looks strange to jump into a block with `goto`, but it works. For example,

```
if (x < 0)
  goto negative;
if (y < 0)
  {
   negative:
    printf ("Negative\n");
    return;
  }
```

If the goto jumps into the scope of a variable, it does not initialize the variable. For example, if `x` is negative,

```
if (x < 0)
  goto negative;
if (y < 0)
  {
      int i = 5;
     negative:
      printf ("Negative, and i is %d\n", i);
      return;
    }
```

prints junk because `i` was not initialized.

If the block declares a variable-length automatic array, jumping into it gives a compilation error. However, jumping out of the scope of a variable-length array works fine, and deallocates its storage.

A label can't come directly before a declaration, so the code can't jump directly to one. For example, this is not allowed:

```
{
  goto foo;
foo:
  int x = 5;
  bar(&x);
}
```

The workaround is to add a statement, even an empty statement, directly after the label. For example:

```
{
  goto foo;
foo:
  ;
  int x = 5;
  bar(&x);
}
```

Likewise, a label can't be the last thing in a block. The workaround solution is the same: add a semicolon after the label.

These unnecessary restrictions on labels make no sense, and ought in principle to be removed; but they do only a little harm since labels and `goto` are rarely the best way to write a program.

These examples are all artificial; it would be more natural to write them in other ways, without `goto`. For instance, the clean way to write the example that prints '`Negative`' is this:

```
if (x < 0 || y < 0)
  {
    printf ("Negative\n");
    return;
  }
```

It is hard to construct simple examples where `goto` is actually the best way to write a program. Its rare good uses tend to be in complex code, thus not apt for the purpose of explaining the meaning of `goto`.

The only good time to use `goto` is when it makes the code simpler than any alternative. Jumping backward is rarely desirable, because usually the other looping and control constructs give simpler code. Using `goto` to jump forward is more often desirable, for instance when a function needs to do some processing in an error case and errors can occur at various different places within the function.

19.13 Locally Declared Labels

In GNU C you can declare *local labels* in any nested block scope. A local label is used in a `goto` statement just like an ordinary label, but you can only reference it within the block in which it was declared.

A local label declaration looks like this:

```
__label__ label;
```

or

```
__label__ label1, label2, ...;
```

Local label declarations must come at the beginning of the block, before any ordinary declarations or statements.

The label declaration declares the label *name*, but does not define the label itself. That's done in the usual way, with `label:`, before one of the statements in the block.

The local label feature is useful for complex macros. If a macro contains nested loops, a `goto` can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(value, array, target)              \
do {                                              \
  __label__ found;                                \
  __auto_type _SEARCH_target = (target);          \
  __auto_type _SEARCH_array = (array);            \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
        { (value) = i; goto found; }              \
  (value) = -1;                                   \
 found:;                                          \
} while (0)
```

This could also be written using a statement expression (see Section 19.15 [Statement Exprs], page 118):

```
#define SEARCH(array, target)                     \
({                                                \
  __label__ found;                                \
  __auto_type _SEARCH_target = (target);      \
  __auto_type _SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
      for (j = 0; j < max; j++)                         \
        if (_SEARCH_array[i][j] == _SEARCH_target)  \
          { value = i; goto found; }                    \
    value = -1;                                         \
   found:                                               \
    value;                                              \
  })
```

Ordinary labels are visible throughout the function where they are defined, and only in that function. However, explicitly declared local labels of a block are visible in nested function definitions inside that block. See Section 22.7.3 [Nested Functions], page 147, for details.

See Section 19.12 [goto Statement], page 114.

19.14 Labels as Values

In GNU C, you can get the address of a label defined in the current function (or a local label defined in the containing function) with the unary operator '`&&`'. The value has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values requires a way to jump to one. This is done with the computed goto statement[1], `goto *`*`exp`*`;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

See Section 19.12 [goto Statement], page 114.

19.14.1 Label Value Uses

One use for label-valued constants is to initialize a static array to serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds—array indexing in C never checks that.

You can make the table entries offsets instead of addresses by subtracting one label from the others. Here is an example:

```
static const int array[] = { &&foo - &&foo, &&bar - &&foo,
                             &&hack - &&foo };
goto *(&&foo + array[i]);
```

Using offsets is preferable in shared libraries, as it avoids the need for dynamic relocation of the array elements; therefore, the array can be read-only.

---

[1] The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, since you can do more with label addresses than store them in special label variables.

An array of label values or offsets serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner, so use `switch` by preference when feasible.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

19.14.2 Label Value Caveats

Jumping to a label defined in another function does not work. It can cause unpredictable results.

The best way to avoid this is to store label values only in automatic variables, or static variables whose names are declared within the function. Never pass them as arguments.

An optimization known as *cloning* generates multiple simplified variants of a function's code, for use with specific fixed arguments. Using label values in certain ways, such as saving the address in one call to the function and using it again in another call, would make cloning give incorrect results. These functions must disable cloning.

Inlining calls to the function would also result in multiple copies of the code, each with its own value of the same label. Using the label in a computed goto is no problem, because the computed goto inhibits inlining. However, using the label value in some other way, such as an indication of where an error occurred, would be optimized wrong. These functions must disable inlining.

To prevent inlining or cloning of a function, specify `__attribute__((__noinline__,__noclone__))` in its definition. See Appendix D [Attributes], page 222.

When a function uses a label value in a static variable initializer, that automatically prevents inlining or cloning the function.

19.15 Statements and Declarations in Expressions

A block enclosed in parentheses can be used as an expression in GNU C. This provides a way to use local variables, loops and switches within an expression. We call it a *statement expression*.

Recall that a block is a sequence of statements surrounded by braces. In this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

The last statement in the block should be an expression statement; an expression followed by a semicolon, that is. The value of this expression serves as the value of statement expression. If the last statement is anything else, the statement expression's value is `void`.

This feature is mainly useful in making macro definitions compute each operand exactly once. See Section 26.5.10.5 [Macros and Auto Type], page 182.

Statement expressions are not allowed in expressions that must be constant, such as the value for an enumerator, the width of a bit-field, or the initial value of a static variable.

Jumping into a statement expression—with `goto`, or using a `switch` statement outside the statement expression—is an error. With a computed `goto` (see Section 19.14 [Labels as Values], page 117), the compiler can't detect the error, but it still won't work.

Jumping out of a statement expression is permitted, but since subexpressions in C are not computed in a strict order, it is unpredictable which other subexpressions will have been computed by then. For example,

```
foo (), (({ bar1 (); goto a; 0; }) + bar2 ()), baz();
```

calls `foo` and `bar1` before it jumps, and never calls `baz`, but may or may not call `bar2`. If `bar2` does get called, that occurs after `foo` and before `bar1`.

20 Variables

Every variable used in a C program needs to be made known by a *declaration*. It can be used only after it has been declared. It is an error to declare a variable name more than once in the same scope; an exception is that `extern` declarations and tentative definitions can coexist with another declaration of the same variable.

Variables can be declared anywhere within a block or file. (Older versions of C required that all variable declarations within a block occur before any statements.)

Variables declared within a function or block are *local* to it. This means that the variable name is visible only until the end of that function or block, and the memory space is allocated only while control is within it.

Variables declared at the top level in a file are called *file-scope*. They are assigned fixed, distinct memory locations, so they retain their values for the whole execution of the program.

20.1 Variable Declarations

Here's what a variable declaration looks like:

```
keywords basetype decorated-variable [= init];
```

The *keywords* specify how to handle the scope of the variable name and the allocation of its storage. Most declarations have no keywords because the defaults are right for them.

C allows these keywords to come before or after *basetype*, or even in the middle of it as in `unsigned static int`, but don't do that—it would surprise other programmers. Always write the keywords first.

The *basetype* can be any of the predefined types of C, or a type keyword defined with `typedef`. It can also be `struct tag`, `union tag`, or `enum tag`. In addition, it can include type qualifiers such as `const` and `volatile` (see Chapter 21 [Type Qualifiers], page 128).

In the simplest case, *decorated-variable* is just the variable name. That declares the variable with the type specified by *basetype*. For instance,

```
int foo;
```

uses `int` as the *basetype* and `foo` as the *decorated-variable*. It declares `foo` with type `int`.

```
struct tree_node foo;
```

declares `foo` with type `struct tree_node`.

20.1.1 Declaring Arrays and Pointers

To declare a variable that is an array, write `variable[length]` for *decorated-variable*:

```
int foo[5];
```

To declare a variable that has a pointer type, write `*variable` for *decorated-variable*:

```
struct list_elt *foo;
```

These constructs nest. For instance,

```
int foo[3][5];
```

declares `foo` as an array of 3 arrays of 5 integers each,

```
struct list_elt *foo[5];
```

declares `foo` as an array of 5 pointers to structures, and

```
struct list_elt **foo;
```

declares `foo` as a pointer to a pointer to a structure.

```
int **(*foo[30])(int, double);
```

declares `foo` as an array of 30 pointers to functions (see Section 22.5 [Function Pointers], page 140), each of which must accept two arguments (one `int` and one `double`) and return type `int **`.

```
void
bar (int size)
{
  int foo[size];
  ...
}
```

declares `foo` as an array of integers with a size specified at run time when the function `bar` is called.

20.1.2 Combining Variable Declarations

When multiple declarations have the same *keywords* and *basetype*, you can combine them using commas. Thus,

```
keywords basetype
   decorated-variable-1 [= init1],
   decorated-variable-2 [= init2];
```

is equivalent to

```
keywords basetype
   decorated-variable-1 [= init1];
keywords basetype
   decorated-variable-2 [= init2];
```

Here are some simple examples:

```
int a, b;
int a = 1, b = 2;
int a, *p, array[5];
int a = 0, *p = &a, array[5] = {1, 2};
```

In the last two examples, `a` is an `int`, `p` is a pointer to `int`, and `array` is an array of 5 `ints`. Since the initializer for `array` specifies only two elements, the other three elements are initialized to zero.

20.2 Initializers

A variable's declaration, unless it is `extern`, should also specify its initial value. For numeric and pointer-type variables, the initializer is an expression for the value. If necessary, it is converted to the variable's type, just as in an assignment.

You can also initialize a local structure-type (see Chapter 15 [Structures], page 74) or local union-type (see Section 15.14 [Unions], page 83) variable this way, from an expression whose value has the same type. But you can't initialize an array this way (see Chapter 16

[Arrays], page 91), since arrays are not first-class objects in C (see Section 16.6 [Limitations of C Arrays], page 94) and there is no array assignment.

You can initialize arrays and structures componentwise, with a list of the elements or components. You can initialize a union with any one of its alternatives.

- A component-wise initializer for an array consists of element values surrounded by '`{...}`'. If the values in the initializer don't cover all the elements in the array, the remaining elements are initialized to zero.

  You can omit the size of the array when you declare it, and let the initializer specify the size:

  ```
  int array[] = { 3, 9, 12 };
  ```
- A component-wise initializer for a structure consists of field values surrounded by '`{...}`'. Write the field values in the same order as the fields are declared in the structure. If the values in the initializer don't cover all the fields in the structure, the remaining fields are initialized to zero.
- The initializer for a union-type variable has the form `{ value }`, where *value* initializes the *first alternative* in the union definition.

For an array of arrays, a structure containing arrays, an array of structures, etc., you can nest these constructs. For example,

```
struct point { double x, y; };

struct point series[]
  = { {0, 0}, {1.5, 2.8}, {99, 100.0004} };
```

You can omit a pair of inner braces if they contain the right number of elements for the sub-value they initialize, so that no elements or fields need to be filled in with zeros. But don't do that very much, as it gets confusing.

An array of `char` can be initialized using a string constant. Recall that the string constant includes an implicit null character at the end (see Section 12.7 [String Constants], page 56). Using a string constant as initializer means to use its contents as the initial values of the array elements. Here are examples:

```
char text[6] = "text!";     /* Includes the null. */
char text[5] = "text!";     /* Excludes the null. */
char text[] = "text!";      /* Gets length 6. */
char text[]
  = { 't', 'e', 'x', 't', '!', 0 };  /* same as above. */
char text[] = { "text!" };  /* Braces are optional. */
```

and this kind of initializer can be nested inside braces to initialize structures or arrays that contain a `char`-array.

In like manner, you can use a wide string constant to initialize an array of `wchar_t`.

20.3 Designated Initializers

In a complex structure or long array, it's useful to indicate which field or element we are initializing.

To designate specific array elements during initialization, include the array index in brackets, and an assignment operator, for each element:

```
int foo[10] = { [3] = 42, [7] = 58 };
```

This does the same thing as:

```
int foo[10] = { 0, 0, 0, 42, 0, 0, 0, 58, 0, 0 };
```

The array initialization can include non-designated element values alongside designated indices; these follow the expected ordering of the array initialization, so that

```
int foo[10] = { [3] = 42, 43, 44, [7] = 58 };
```

does the same thing as:

```
int foo[10] = { 0, 0, 0, 42, 43, 44, 0, 58, 0, 0 };
```

Note that you can only use constant expressions as array index values, not variables.

If you need to initialize a subsequence of sequential array elements to the same value, you can specify a range:

```
int foo[100] = { [0 ... 19] = 42, [20 ... 99] = 43 };
```

Using a range this way is a GNU C extension.

When subsequence ranges overlap, each element is initialized by the last specification that applies to it. Thus, this initialization is equivalent to the previous one.

```
int foo[100] = { [0 ... 99] = 43, [0 ... 19] = 42 };
```

as the second overrides the first for elements 0 through 19.

The value used to initialize a range of elements is evaluated only once, for the first element in the range. So for example, this code

```
int random_values[100]
  = { [0 ... 99] = get_random_number() };
```

would initialize all 100 elements of the array `random_values` to the same value—probably not what is intended.

Similarly, you can initialize specific fields of a structure variable by specifying the field name prefixed with a dot:

```
struct point { int x; int y; };

struct point foo = { .y = 42; };
```

The same syntax works for union variables as well:

```
union int_double { int i; double d; };

union int_double foo = { .d = 34 };
```

This casts the integer value 34 to a double and stores it in the union variable `foo`.

You can designate both array elements and structure elements in the same initialization; for example, here's an array of point structures:

```
struct point point_array[10] = { [4].y = 32, [6].y = 39 };
```

Along with the capability to specify particular array and structure elements to initialize comes the possibility of initializing the same element more than once:

```
int foo[10] = { [4] = 42, [4] = 98 };
```

In such a case, the last initialization value is retained.

20.4 Referring to a Type with `__auto_type`

You can declare a variable copying the type from the initializer by using `__auto_type` instead of a particular type. Here's an example:

```
#define max(a,b) \
  ({ __auto_type _a = (a); \
      __auto_type _b = (b); \
    _a > _b ? _a : _b })
```

This defines `_a` to be of the same type as `a`, and `_b` to be of the same type as `b`. This is a useful thing to do in a macro that ought to be able to handle any type of data (see Section 26.5.10.5 [Macros and Auto Type], page 182).

The original GNU C method for obtaining the type of a value is to use `typeof`, which takes as an argument either a value or the name of a type. The previous example could also be written as:

```
#define max(a,b) \
  ({ typeof(a) _a = (a); \
      typeof(b) _b = (b); \
    _a > _b ? _a : _b })
```

`typeof` is more flexible than `__auto_type`; however, the principal use case for `typeof` is in variable declarations with initialization, which is exactly what `__auto_type` handles.

20.5 Local Variables

Declaring a variable inside a function definition (see Section 22.1 [Function Definitions], page 132) makes the variable name *local* to the containing block—that is, the containing pair of braces. More precisely, the variable's name is visible starting just after where it appears in the declaration, and its visibility continues until the end of the block.

Local variables in C are generally *automatic* variables: each variable's storage exists only from the declaration to the end of the block. Execution of the declaration allocates the storage, computes the initial value, and stores it in the variable. The end of the block deallocates the storage.[1]

Warning: Two declarations for the same local variable in the same scope are an error.

Warning: Automatic variables are stored in the run-time stack. The total space for the program's stack may be limited; therefore, in using very large arrays, it may be necessary to allocate them in some other way to stop the program from crashing.

Warning: If the declaration of an automatic variable does not specify an initial value, the variable starts out containing garbage. In this example, the value printed could be anything at all:

```
{
  int i;

  printf ("Print junk %d\n", i);
}
```

---

[1] Due to compiler optimizations, allocation and deallocation don't necessarily really happen at those times.

In a simple test program, that statement is likely to print 0, simply because every process starts with memory zeroed. But don't rely on it to be zero—that is erroneous.

Note: Make sure to store a value into each local variable (by assignment, or by initialization) before referring to its value.

20.6 File-Scope Variables

A variable declaration at the top level in a file (not inside a function definition) declares a *file-scope variable*. Loading a program allocates the storage for all the file-scope variables in it, and initializes them too.

Each file-scope variable is either *static* (limited to one compilation module) or *global* (shared with all compilation modules in the program). To make the variable static, write the keyword `static` at the start of the declaration. Omitting `static` makes the variable global.

The initial value for a file-scope variable can't depend on the contents of storage, and can't call any functions.

```
int foo = 5;          /* Valid. */
int bar = foo;        /* Invalid! */
int bar = sin (1.0); /* Invalid! */
```

But it can use the address of another file-scope variable:

```
int foo;
int *bar = &foo;      /* Valid. */
int arr[5];
int *bar3 = &arr[3]; /* Valid. */
int *bar4 = arr + 4; /* Valid. */
```

It is valid for a module to have multiple declarations for a file-scope variable, as long as they are all global or all static, but at most one declaration can specify an initial value for it.

20.7 Static Local Variables

The keyword `static` in a local variable declaration says to allocate the storage for the variable permanently, just like a file-scope variable, even if the declaration is within a function.

Here's an example:

```
int
increment_counter ()
{
  static int counter = 0;
  return ++counter;
}
```

The scope of the name `counter` runs from the declaration to the end of the containing block, just like an automatic local variable, but its storage is permanent, so the value persists from one call to the next. As a result, each call to `increment_counter` returns a different, unique value.

The initial value of a static local variable has the same limitations as for file-scope variables: it can't depend on the contents of storage or call any functions. It can use the address of a file-scope variable or a static local variable, because those addresses are determined before the program runs.

20.8 `extern` Declarations

An `extern` declaration is used to refer to a global variable whose principal declaration comes elsewhere—in the same module, or in another compilation module. It looks like this:

```
extern basetype decorated-variable;
```

Its meaning is that, in the current scope, the variable name refers to the file-scope variable of that name—which needs to be declared in a non-`extern`, non-`static` way somewhere else.

For instance, if one compilation module has this global variable declaration

```
int error_count = 0;
```

then other compilation modules can specify this

```
extern int error_count;
```

to allow reference to the same variable.

The usual place to write an `extern` declaration is at top level in a source file, but you can write an `extern` declaration inside a block to make a global or static file-scope variable accessible in that block.

Since an `extern` declaration does not allocate space for the variable, it can omit the size of an array:

```
extern int array[];
```

You can use `array` normally in all contexts where it is converted automatically to a pointer. However, to use it as the operand of `sizeof` is an error, since the size is unknown.

It is valid to have multiple `extern` declarations for the same variable, even in the same scope, if they give the same type. They do not conflict—they agree. For an array, it is legitimate for some `extern` declarations can specify the size while others omit it. However, if two declarations give different sizes, that is an error.

Likewise, you can use `extern` declarations at file scope (see Section 20.6 [File-Scope Variables], page 125) followed by an ordinary global (non-static) declaration of the same variable. They do not conflict, because they say compatible things about the same meaning of the variable.

20.9 Allocating File-Scope Variables

Some file-scope declarations allocate space for the variable, and some don't.

A file-scope declaration with an initial value *must* allocate space for the variable; if there are two of such declarations for the same variable, even in different compilation modules, they conflict.

An `extern` declaration *never* allocates space for the variable. If all the top-level declarations of a certain variable are `extern`, the variable never gets memory space. If that variable is used anywhere in the program, the use will be reported as an error, saying that the variable is not defined.

A file-scope declaration without an initial value is called a *tentative definition*. This is a strange hybrid: it *can* allocate space for the variable, but does not insist. So it causes no conflict, no error, if the variable has another declaration that allocates space for it, perhaps in another compilation module. But if nothing else allocates space for the variable, the tentative definition will do it. Any number of compilation modules can declare the same variable in this way, and that is sufficient for all of them to use the variable.

In programs that are very large or have many contributors, it may be wise to adopt the convention of never using tentative definitions. You can use the compilation option `-fno-common` to make them an error, or `--warn-common` to warn about them.

If a file-scope variable gets its space through a tentative definition, it starts out containing all zeros.

20.10 `auto` and `register`

For historical reasons, you can write `auto` or `register` before a local variable declaration. `auto` merely emphasizes that the variable isn't static; it changes nothing.

`register` suggests to the compiler storing this variable in a register. However, GNU C ignores this suggestion, since it can choose the best variables to store in registers without any hints.

It is an error to take the address of a variable declared `register`, so you cannot use the unary '`&`' operator on it. If the variable is an array, you can't use it at all (other than as the operand of `sizeof`), which makes it rather useless.

20.11 Omitting Types in Declarations

The syntax of C traditionally allows omitting the data type in a declaration if it specifies a storage class, a type qualifier (see the next chapter), or `auto` or `register`. Then the type defaults to `int`. For example:

```
auto foo = 42;
```

This is bad practice; if you see it, fix it.

21 Type Qualifiers

A declaration can include type qualifiers to advise the compiler about how the variable will be used. There are three different qualifiers, `const`, `volatile` and `restrict`. They pertain to different issues, so you can use more than one together. For instance, `const volatile` describes a value that the program is not allowed to change, but might have a different value each time the program examines it. (This might perhaps be a special hardware register, or part of shared memory.)

If you are just learning C, you can skip this chapter.

21.1 `const` Variables and Fields

You can mark a variable as "constant" by writing `const` in front of the declaration. This says to treat any assignment to that variable as an error. It may also permit some compiler optimizations—for instance, to fetch the value only once to satisfy multiple references to it. The construct looks like this:

```
const double pi = 3.14159;
```

After this definition, the code can use the variable `pi` but cannot assign a different value to it.

```
pi = 3.0; /* Error! */
```

Simple variables that are constant can be used for the same purposes as enumeration constants, and they are not limited to integers. The constantness of the variable propagates into pointers, too.

A pointer type can specify that the *target* is constant. For example, the pointer type `const double *` stands for a pointer to a constant `double`. That's the type that results from taking the address of `pi`. Such a pointer can't be dereferenced in the left side of an assignment.

```
*(&pi) = 3.0; /* Error! */
```

Nonconstant pointers can be converted automatically to constant pointers, but not vice versa. For instance,

```
const double *cptr;
double *ptr;

cptr = &pi;     /* Valid. */
cptr = ptr;     /* Valid. */
ptr = cptr;     /* Error! */
ptr = &pi;      /* Error! */
```

This is not an ironclad protection against modifying the value. You can always cast the constant pointer to a nonconstant pointer type:

```
ptr = (double *)cptr;    /* Valid. */
ptr = (double *)&pi;     /* Valid. */
```

However, `const` provides a way to show that a certain function won't modify the data structure whose address is passed to it. Here's an example:

```
int
```

```
string_length (const char *string)
{
  int count = 0;
  while (*string++)
    count++;
  return count;
}
```

Using `const char *` for the parameter is a way of saying this function never modifies the memory of the string itself.

In calling `string_length`, you can specify an ordinary `char *` since that can be converted automatically to `const char *`.

21.2 `volatile` Variables and Fields

The GNU C compiler often performs optimizations that eliminate the need to write or read a variable. For instance,

```
int foo;
foo = 1;
foo++;
```

might simply store the value 2 into `foo`, without ever storing 1. These optimizations can also apply to structure fields in some cases.

If the memory containing `foo` is shared with another program, or if it is examined asynchronously by hardware, such optimizations could confuse the communication. Using `volatile` is one way to prevent them.

Writing `volatile` with the type in a variable or field declaration says that the value may be examined or changed for reasons outside the control of the program at any moment. Therefore, the program must execute in a careful way to assure correct interaction with those accesses, whenever they may occur.

The simplest use looks like this:

```
volatile int lock;
```

This directs the compiler not to do certain common optimizations on use of the variable `lock`. All the reads and writes for a volatile variable or field are really done, and done in the order specified by the source code. Thus, this code:

```
lock = 1;
list = list->next;
if (lock)
  lock_broken (&lock);
lock = 0;
```

really stores the value 1 in `lock`, even though there is no sign it is really used, and the `if` statement reads and checks the value of `lock`, rather than assuming it is still 1.

A limited amount of optimization can be done, in principle, on `volatile` variables and fields: multiple references between two sequence points (see Section 10.3 [Sequence Points], page 44) can be simplified together.

Use of `volatile` does not eliminate the flexibility in ordering the computation of the operands of most operators. For instance, in `lock + foo ()`, the order of accessing `lock` and calling `foo` is not specified, so they may be done in either order; the fact that `lock` is `volatile` has no effect on that.

21.3 `restrict`-Qualified Pointers

You can declare a pointer as “restricted” using the `restrict` type qualifier, like this:

```
int *restrict p = x;
```

This enables better optimization of code that uses the pointer.

If `p` is declared with `restrict`, and then the code references the object that `p` points to (using `*p` or `p[i]`), the `restrict` declaration promises that the code will not access that object in any other way—only through `p`.

For instance, it means the code must not use another pointer to access the same space, as shown here:

```
int *restrict p = whatever;
int *q = p;
foo (*p, *q);
```

That contradicts the `restrict` promise by accessing the object that `p` points to using `q`, which bypasses `p`. Likewise, it must not do this:

```
int *restrict p = whatever;
struct { int *a, *b; } s;
s.a = p;
foo (*p, *s.a);
```

This example uses a structure field instead of the variable `q` to hold the other pointer, and that contradicts the promise just the same.

The keyword `restrict` also promises that `p` won’t point to the allocated space of any automatic or static variable. So the code must not do this:

```
int a;
int *restrict p = &a;
foo (*p, a);
```

because that does direct access to the object (`a`) that `p` points to, which bypasses `p`.

If the code makes such promises with `restrict` then breaks them, execution is unpredictable.

21.4 `restrict` Pointer Example

Here are examples where `restrict` enables real optimization.

In this example, `restrict` assures GCC that the array `out` points to does not overlap with the array `in` points to.

```
void
process_data (const char *in,
              char * restrict out,
              size_t size)
{
  for (i = 0; i < size; i++)
```

```
    out[i] = in[i] + in[i + 1];
}
```

Here's a simple tree structure, where each tree node holds data of type `PAYLOAD` plus two subtrees.

```
struct foo
  {
    PAYLOAD payload;
    struct foo *left;
    struct foo *right;
  };
```

Now here's a function to null out both pointers in the `left` subtree.

```
void
null_left (struct foo *a)
{
  a->left->left = NULL;
  a->left->right = NULL;
}
```

Since `*a` and `*a->left` have the same data type, they could legitimately alias (see Appendix B [Aliasing], page 218). Therefore, the compiled code for `null_left` must read `a->left` again from memory when executing the second assignment statement.

We can enable optimization, so that it does not need to read `a->left` again, by writing `null_left` in a less obvious way.

```
void
null_left (struct foo *a)
{
  struct foo *b = a->left;
  b->left = NULL;
  b->right = NULL;
}
```

A more elegant way to fix this is with `restrict`.

```
void
null_left (struct foo *restrict a)
{
  a->left->left = NULL;
  a->left->right = NULL;
}
```

Declaring `a` as `restrict` asserts that other pointers such as `a->left` will not point to the same memory space as `a`. Therefore, the memory location `a->left->left` cannot be the same memory as `a->left`. Knowing this, the compiled code may avoid reloading `a->left` for the second statement.

22 Functions

We have already presented many examples of functions, so if you've read this far, you basically understand the concept of a function. It is vital, nonetheless, to have a chapter in the manual that collects all the information about functions.

22.1 Function Definitions

We have already presented many examples of function definitions. To summarize the rules, a function definition looks like this:

```
returntype
functionname (parm_declarations...)
{
  body
}
```

The part before the open-brace is called the *function header*.

Write `void` as the *returntype* if the function does not return a value.

22.1.1 Function Parameter Variables

A function parameter variable is a local variable (see Section 20.5 [Local Variables], page 124) used within the function to store the value passed as an argument in a call to the function. Usually we say "function parameter" or "parameter" for short, not mentioning the fact that it's a variable.

We declare these variables in the beginning of the function definition, in the *parameter list*. For example,

```
fib (int n)
```

has a parameter list with one function parameter `n`, which has type `int`.

Function parameter declarations differ from ordinary variable declarations in several ways:

- Inside the function definition header, commas separate parameter declarations, and each parameter needs a complete declaration including the type. For instance, if a function `foo` has two `int` parameters, write this:

  ```
  foo (int a, int b)
  ```

  You can't share the common `int` between the two declarations:

  ```
  foo (int a, b)       /* Invalid! */
  ```

- A function parameter variable is initialized to whatever value is passed in the function call, so its declaration cannot specify an initial value.
- Writing an array type in a function parameter declaration has the effect of declaring it as a pointer. The size specified for the array has no effect at all, and we normally omit the size. Thus,

  ```
  foo (int a[5])
  foo (int a[])
  foo (int *a)
  ```

  are equivalent.

- The scope of the parameter variables is the entire function body, notwithstanding the fact that they are written in the function header, which is just outside the function body.

If a function has no parameters, it would be most natural for the list of parameters in its definition to be empty. But that, in C, has a special meaning for historical reasons: "Do not check that calls to this function have the right number of arguments." Thus,

```
int
foo ()
{
  return 5;
}

int
bar (int x)
{
  return foo (x);
}
```

would not report a compilation error in passing `x` as an argument to `foo`. By contrast,

```
int
foo (void)
{
  return 5;
}

int
bar (int x)
{
  return foo (x);
}
```

would report an error because `foo` is supposed to receive no arguments.

22.1.2 Forward Function Declarations

The order of the function definitions in the source code makes no difference, except that each function needs to be defined or declared before code uses it.

The definition of a function also declares its name for the rest of the containing scope. But what if you want to call the function before its definition? To permit that, write a compatible declaration of the same function, before the first call. A declaration that prefigures a subsequent definition in this way is called a *forward declaration*. The function declaration can be at top level or within a block, and it applies until the end of the containing scope.

See Section 22.2 [Function Declarations], page 137, for more information about these declarations.

22.1.3 Static Functions

The keyword `static` in a function definition limits the visibility of the name to the current compilation module. (That's the same thing `static` does in variable declarations; see Section 20.6 [File-Scope Variables], page 125.) For instance, if one compilation module contains this code:

```
static int
foo (void)
{
  ...
}
```

then the code of that compilation module can call `foo` anywhere after the definition, but other compilation modules cannot refer to it at all.

To call `foo` before its definition, it needs a forward declaration, which should use `static` since the function definition does. For this function, it looks like this:

```
static int foo (void);
```

It is generally wise to use `static` on the definitions of functions that won't be called from outside the same compilation module. This makes sure that calls are not added in other modules. If programmers decide to change the function's calling convention, or understand all the consequences of its use, they will only have to check for calls in the same compilation module.

22.1.4 Arrays as Parameters

Arrays in C are not first-class objects: it is impossible to copy them. So they cannot be passed as arguments like other values. See Section 16.6 [Limitations of C Arrays], page 94. Rather, array parameters work in a special way.

22.1.4.1 Array parameters are pointers

Declaring a function parameter variable as an array really gives it a pointer type. C does this because an expression with array type, if used as an argument in a function call, is converted automatically to a pointer (to the zeroth element of the array). If you declare the corresponding parameter as an "array", it will work correctly with the pointer value that really gets passed.

This relates to the fact that C does not check array bounds in access to elements of the array (see Section 16.1 [Accessing Array Elements], page 91).

For example, in this function,

```
void
clobber4 (int array[20])
{
  array[4] = 0;
}
```

the parameter `array`'s real type is `int *`; the specified length, 20, has no effect on the program. You can leave out the length and write this:

```
void
clobber4 (int array[])
```

```
{
  array[4] = 0;
}
```

or write the parameter declaration explicitly as a pointer:

```
void
clobber4 (int *array)
{
  array[4] = 0;
}
```

They are all equivalent.

22.1.4.2 Passing array arguments

The function call passes this pointer by value, like all argument values in C. However, the result is paradoxical in that the array itself is passed by reference: its contents are treated as shared memory—shared between the caller and the called function, that is. When `clobber4` assigns to element 4 of `array`, the effect is to alter element 4 of the array specified in the call.

```
#include <stddef.h>  /* Defines NULL. */
#include <stdlib.h>  /* Declares malloc, */
                     /* Defines EXIT_SUCCESS. */

int
main (void)
{
  int data[] = {1, 2, 3, 4, 5, 6};
  int i;

  /* Show the initial value of element 4. */
  for (i = 0; i < 6; i++)
    printf ("data[%d] = %d\n", i, data[i]);

  printf ("\n");

  clobber4 (data);

  /* Show that element 4 has been changed. */
  for (i = 0; i < 6; i++)
    printf ("data[%d] = %d\n", i, data[i]);

  printf ("\n");

  return EXIT_SUCCESS;
}
```

shows that `data[4]` has become zero after the call to `clobber4`.

The array `data` has 6 elements, but passing it to a function whose argument type is written as `int [20]` is not an error, because that really stands for `int *`. The pointer that is the real argument carries no indication of the length of the array it points into. It is not required to point to the beginning of the array, either. For instance,

```
clobber4 (data+1);
```

passes an “array” that starts at element 1 of `data`, and the effect is to zero `data[5]` instead of `data[4]`.

If all calls to the function will provide an array of a particular size, you can specify the size of the array to be `static`:

```
void
clobber4 (int array[static 20])
...
```

This is a promise to the compiler that the function will always be called with an array of 20 elements, so that the compiler can optimize code accordingly. If the code breaks this promise and calls the function with, for example, a shorter array, unpredictable things may happen.

22.1.4.3 Type qualifiers on array parameters

You can use the type qualifiers `const`, `restrict`, and `volatile` with array parameters; for example:

```
void
clobber4 (volatile int array[20])
...
```

denotes that `array` is equivalent to a pointer to a volatile `int`. Alternatively:

```
void
clobber4 (int array[const 20])
...
```

makes the array parameter equivalent to a constant pointer to an `int`. If we want the `clobber4` function to succeed, it would not make sense to write

```
void
clobber4 (const int array[20])
...
```

as this would tell the compiler that the parameter should point to an array of constant `int` values, and then we would not be able to store zeros in them.

In a function with multiple array parameters, you can use `restrict` to tell the compiler that each array parameter passed in will be distinct:

```
void
foo (int array1[restrict 10], int array2[restrict 10])
...
```

Using `restrict` promises the compiler that callers will not pass in the same array for more than one `restrict` array parameter. Knowing this enables the compiler to perform better code optimization. This is the same effect as using `restrict` pointers (see Section 21.3 [restrict Pointers], page 130), but makes it clear when reading the code that an array of a specific size is expected.

22.1.5 Functions That Accept Structure Arguments

Structures in GNU C are first-class objects, so using them as function parameters and arguments works in the natural way. This function `swapfoo` takes a `struct foo` with two fields as argument, and returns a structure of the same type but with the fields exchanged.

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

struct foo x;

struct foo
swapfoo (struct foo inval)
{
  struct foo outval;
  outval.a = inval.b;
  outval.b = inval.a;
  return outval;
}
```

This simpler definition of `swapfoo` avoids using a local variable to hold the result about to be return, by using a structure constructor (see Section 15.17 [Structure Constructors], page 86), like this:

```
struct foo
swapfoo (struct foo inval)
{
  return (struct foo) { inval.b, inval.a };
}
```

It is valid to define a structure type in a function's parameter list, as in

```
int
frob_bar (struct bar { int a, b; } inval)
{
  body
}
```

and *body* can access the fields of *inval* since the structure type `struct bar` is defined for the whole function body. However, there is no way to create a `struct bar` argument to pass to `frob_bar`, except with kludges. As a result, defining a structure type in a parameter list is useless in practice.

22.2 Function Declarations

To call a function, or use its name as a pointer, a *function declaration* for the function name must be in effect at that point in the code. The function's definition serves as a declaration of that function for the rest of the containing scope, but to use the function in code before the definition, or from another compilation module, a separate function declaration must precede the use.

A function declaration looks like the start of a function definition. It begins with the return value type (`void` if none) and the function name, followed by argument declarations in parentheses (though these can sometimes be omitted). But that's as far as the similarity goes: instead of the function body, the declaration uses a semicolon.

A declaration that specifies argument types is called a *function prototype*. You can include the argument names or omit them. The names, if included in the declaration, have no effect, but they may serve as documentation.

This form of prototype specifies fixed argument types:

```
rettype function (argtypes...);
```

This form says the function takes no arguments:

```
rettype function (void);
```

This form declares types for some arguments, and allows additional arguments whose types are not specified:

```
rettype function (argtypes..., ...);
```

For a parameter that's an array of variable length, you can write its declaration with '`*`' where the "length" of the array would normally go; for example, these are all equivalent.

```
double maximum (int n, int m, double a[n][m]);
double maximum (int n, int m, double a[*][*]);
double maximum (int n, int m, double a[ ][*]);
double maximum (int n, int m, double a[ ][m]);
```

The old-fashioned form of declaration, which is not a prototype, says nothing about the types of arguments or how many they should be:

```
rettype function ();
```

Warning: Arguments passed to a function declared without a prototype are converted with the default argument promotions (see Section 24.3 [Argument Promotions], page 156. Likewise for additional arguments whose types are unspecified.

Function declarations are usually written at the top level in a source file, but you can also put them inside code blocks. Then the function name is visible for the rest of the containing scope. For example:

```
void
foo (char *file_name)
{
  void save_file (char *);
  save_file (file_name);
}
```

If another part of the code tries to call the function `save_file`, this declaration won't be in effect there. So the function will get an implicit declaration of the form `extern int save_file ();`. That conflicts with the explicit declaration here, and the discrepancy generates a warning.

The syntax of C traditionally allows omitting the data type in a function declaration if it specifies a storage class or a qualifier. Then the type defaults to `int`. For example:

```
static foo (double x);
```

defaults the return type to `int`. This is bad practice; if you see it, fix it.

Calling a function that is undeclared has the effect of an creating *implicit* declaration in the innermost containing scope, equivalent to this:

```
extern int function ();
```

This declaration says that the function returns `int` but leaves its argument types unspecified. If that does not accurately fit the function, then the program **needs** an explicit declaration of the function with argument types in order to call it correctly.

Implicit declarations are deprecated, and a function call that creates one causes a warning.

22.3 Function Calls

Starting a program automatically calls the function named `main` (see Section 22.6 [The main Function], page 142). Aside from that, a function does nothing except when it is *called*. That occurs during the execution of a function-call expression specifying that function.

A function-call expression looks like this:

```
function (arguments...)
```

Most of the time, *function* is a function name. However, it can also be an expression with a function pointer value; that way, the program can determine at run time which function to call.

The *arguments* are a series of expressions separated by commas. Each expression specifies one argument to pass to the function.

The list of arguments in a function call looks just like use of the comma operator (see Section 8.5 [Comma Operator], page 39), but the fact that it fills the parentheses of a function call gives it a different meaning.

Here's an example of a function call, taken from an example near the beginning (see Chapter 2 [Complete Program], page 9).

```
printf ("Fibonacci series item %d is %d\n",
        19, fib (19));
```

The three arguments given to `printf` are a constant string, the integer 19, and the integer returned by `fib (19)`.

22.4 Function Call Semantics

The meaning of a function call is to compute the specified argument expressions, convert their values according to the function's declaration, then run the function giving it copies of the converted values. (This method of argument passing is known as *call-by-value*.) When the function finishes, the value it returns becomes the value of the function-call expression.

Call-by-value implies that an assignment to the function argument variable has no direct effect on the caller. For instance,

```
#include <stdlib.h>  /* Defines EXIT_SUCCESS. */
#include <stdio.h>   /* Declares printf. */

void
subroutine (int x)
{
  x = 5;
}
```

```
void
main (void)
{
  int y = 20;
  subroutine (y);
  printf ("y is %d\n", y);
  return EXIT_SUCCESS;
}
```

prints '`y is 20`'. Calling `subroutine` initializes `x` from the value of `y`, but this does not establish any other relationship between the two variables. Thus, the assignment to `x`, inside `subroutine`, changes only *that* `x`.

If an argument's type is specified by the function's declaration, the function call converts the argument expression to that type if possible. If the conversion is impossible, that is an error.

If the function's declaration doesn't specify the type of that argument, then the *default argument promotions* apply. See Section 24.3 [Argument Promotions], page 156.

22.5 Function Pointers

A function name refers to a fixed function. Sometimes it is useful to call a function to be determined at run time; to do this, you can use a *function pointer value* that points to the chosen function (see Chapter 14 [Pointers], page 62).

Pointer-to-function types can be used to declare variables and other data, including array elements, structure fields, and union alternatives. They can also be used for function arguments and return values. These types have the peculiarity that they are never converted automatically to `void *` or vice versa. However, you can do that conversion with a cast.

22.5.1 Declaring Function Pointers

The declaration of a function pointer variable (or structure field) looks almost like a function declaration, except it has an additional '`*`' just before the variable name. Proper nesting requires a pair of parentheses around the two of them. For instance, `int (*a) ();` says, "Declare `a` as a pointer such that `*a` is an `int`-returning function."

Contrast these three declarations:

```
/* Declare a function returning char *.  */
char *a (char *);
/* Declare a pointer to a function returning char.  */
char (*a) (char *);
/* Declare a pointer to a function returning char *.  */
char *(*a) (char *);
```

The possible argument types of the function pointed to are the same as in a function declaration. You can write a prototype that specifies all the argument types:

```
rettype (*function) (arguments...);
```

or one that specifies some and leaves the rest unspecified:

```
rettype (*function) (arguments..., ...);
```

or one that says there are no arguments:

```
rettype (*function) (void);
```

You can also write a non-prototype declaration that says nothing about the argument types:

```
rettype (*function) ();
```

For example, here's a declaration for a variable that should point to some arithmetic function that operates on two `double`s:

```
double (*binary_op) (double, double);
```

Structure fields, union alternatives, and array elements can be function pointers; so can parameter variables. The function pointer declaration construct can also be combined with other operators allowed in declarations. For instance,

```
int **(*foo)();
```

declares `foo` as a pointer to a function that returns type `int **`, and

```
int **(*foo[30])();
```

declares `foo` as an array of 30 pointers to functions that return type `int **`.

```
int **(**foo)();
```

declares `foo` as a pointer to a pointer to a function that returns type `int **`.

22.5.2 Assigning Function Pointers

Assuming we have declared the variable `binary_op` as in the previous section, giving it a value requires a suitable function to use. So let's define a function suitable for the variable to point to. Here's one:

```
double
double_add (double a, double b)
{
  return a+b;
}
```

Now we can give it a value:

```
binary_op = double_add;
```

The target type of the function pointer must be upward compatible with the type of the function (see Chapter 23 [Compatible Types], page 154).

There is no need for '`&`' in front of `double_add`. Using a function name such as `double_add` as an expression automatically converts it to the function's address, with the appropriate function pointer type. However, it is ok to use '`&`' if you feel that is clearer:

```
binary_op = &double_add;
```

22.5.3 Calling Function Pointers

To call the function specified by a function pointer, just write the function pointer value in a function call. For instance, here's a call to the function `binary_op` points to:

```
binary_op (x, 5)
```

Since the data type of `binary_op` explicitly specifies type `double` for the arguments, the call converts `x` and 5 to `double`.

The call conceptually dereferences the pointer `binary_op` to “get” the function it points to, and calls that function. If you wish, you can explicitly represent the dereference by writing the `*` operator:

```
(*binary_op) (x, 5)
```

The ‘`*`’ reminds people reading the code that `binary_op` is a function pointer rather than the name of a specific function.

22.6 The `main` Function

Every complete executable program requires at least one function, called `main`, which is where execution begins. You do not have to explicitly declare `main`, though GNU C permits you to do so. Conventionally, `main` should be defined to follow one of these calling conventions:

```
int main (void) {...}
int main (int argc, char *argv[]) {...}
int main (int argc, char *argv[], char *envp[]) {...}
```

Using `void` as the parameter list means that `main` does not use the arguments. You can write `char **argv` instead of `char *argv[]`, and likewise for `envp`, as the two constructs are equivalent.

You can call `main` from C code, as you can call any other function, though that is an unusual thing to do. When you do that, you must write the call to pass arguments that match the parameters in the definition of `main`.

The `main` function is not actually the first code that runs when a program starts. In fact, the first code that runs is system code from the file `crt0.o`. In Unix, this was hand-written assembler code, but in GNU we replaced it with C code. Its job is to find the arguments for `main` and call that.

22.6.1 Returning Values from `main`

When `main` returns, the process terminates. Whatever value `main` returns becomes the exit status which is reported to the parent process. While nominally the return value is of type `int`, in fact the exit status gets truncated to eight bits; if `main` returns the value 256, the exit status is 0.

Normally, programs return only one of two values: 0 for success, and 1 for failure. For maximum portability, use the macro values `EXIT_SUCCESS` and `EXIT_FAILURE` defined in `stdlib.h`. Here’s an example:

```
#include <stdlib.h>  /* Defines EXIT_SUCCESS */
                     /* and EXIT_FAILURE. */

int
main (void)
{
  ...
  if (foo)
    return EXIT_SUCCESS;
  else
```

```
    return EXIT_FAILURE;
}
```

Some types of programs maintain special conventions for various return values; for example, comparison programs including `cmp` and `diff` return 1 to indicate a mismatch, and 2 to indicate that the comparison couldn't be performed.

22.6.2 Accessing Command-line Parameters

If the program was invoked with any command-line arguments, it can access them through the arguments of `main`, `argc` and `argv`. (You can give these arguments any names, but the names `argc` and `argv` are customary.)

The value of `argv` is an array containing all of the command-line arguments as strings, with the name of the command invoked as the first string. `argc` is an integer that says how many strings `argv` contains. Here is an example of accessing the command-line parameters, retrieving the program's name and checking for the standard `--version` and `--help` options:

```
#include <string.h> /* Declare strcmp. */

int
main (int argc, char *argv[])
{
  char *program_name = argv[0];

  for (int i = 1; i < argc; i++)
    {
      if (!strcmp (argv[i], "--version"))
        {
          /* Print version information and exit. */
          ...
        }
      else if (!strcmp (argv[i], "--help"))
        {
          /* Print help information and exit. */
          ...
        }
    }
  ...
}
```

22.6.3 Accessing Environment Variables

You can optionally include a third parameter to `main`, another array of strings, to capture the environment variables available to the program. Unlike what happens with `argv`, there is no additional parameter for the count of environment variables; rather, the array of environment variables concludes with a null pointer.

```
#include <stdio.h>   /* Declares printf. */

int
```

```
main (int argc, char *argv[], char *envp[])
{
  /* Print out all environment variables. */
  int i = 0;
  while (envp[i])
    {
      printf ("%s\n", envp[i]);
      i++;
    }
}
```

Another method of retrieving environment variables is to use the library function `getenv`, which is defined in `stdlib.h`. Using `getenv` does not require defining `main` to accept the `envp` pointer. For example, here is a program that fetches and prints the user's home directory (if defined):

```
#include <stdlib.h>  /* Declares getenv. */
#include <stdio.h>   /* Declares printf. */

int
main (void)
{
  char *home_directory = getenv ("HOME");
  if (home_directory)
    printf ("My home directory is: %s\n", home_directory);
  else
    printf ("My home directory is not defined!\n");
}
```

22.7 Advanced Function Features

This section describes some advanced or obscure features for GNU C function definitions. If you are just learning C, you can skip the rest of this chapter.

22.7.1 Variable-Length Array Parameters

An array parameter can have variable length: simply declare the array type with a size that isn't constant. In a nested function, the length can refer to a variable defined in a containing scope. In any function, it can refer to a previous parameter, like this:

```
struct entry
tester (int len, char data[len][len])
{
  ...
}
```

Alternatively, in function declarations (but not in function definitions), you can use `[*]` to denote that the array parameter is of a variable length, such that these two declarations mean the same thing:

```
struct entry
```

```
tester (int len, char data[len][len]);

struct entry
tester (int len, char data[*][*]);
```

The two forms of input are equivalent in GNU C, but emphasizing that the array parameter is variable-length may be helpful to those studying the code.

You can also omit the length parameter, and instead use some other in-scope variable for the length in the function definition:

```
struct entry
tester (char data[*][*]);
...
int dataLength = 20;
...
struct entry
tester (char data[dataLength][dataLength])
{
  ...
}
```

In GNU C, to pass the array first and the length afterward, you can use a *parameter forward declaration*, like this:

```
struct entry
tester (int len; char data[len][len], int len)
{
  ...
}
```

The '`int len`' before the semicolon is the parameter forward declaration; it serves the purpose of making the name `len` known when the declaration of `data` is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a subsequent "real" declaration in parameter name and data type.

Standard C does not support parameter forward declarations.

22.7.2 Variable-Length Parameter Lists

A function that takes a variable number of arguments is called a *variadic function*. In C, a variadic function must specify at least one fixed argument with an explicitly declared data type. Additional arguments can follow, and can vary in both quantity and data type.

In the function header, declare the fixed parameters in the normal way, then write a comma and an ellipsis: '`, ...`'. Here is an example of a variadic function header:

```
int add_multiple_values (int number, ...)
```

The function body can refer to fixed arguments by their parameter names, but the additional arguments have no names. Accessing them in the function body uses certain standard macros. They are defined in the library header file `stdarg.h`, so the code must `#include` that file.

In the body, write

```
va_list ap;
va_start (ap, last_fixed_parameter);
```

This declares the variable `ap` (you can use any name for it) and then sets it up to point before the first additional argument.

Then, to fetch the next consecutive additional argument, write this:

```
va_arg (ap, type)
```

After fetching all the additional arguments (or as many as need to be used), write this:

```
va_end (ap);
```

Here's an example of a variadic function definition that adds any number of `int` arguments. The first (fixed) argument says how many more arguments follow.

```
#include <stdarg.h> /* Defines va... macros. */
...

int
add_multiple_values (int argcount, ...)
{
  int counter, total = 0;

  /* Declare a variable of type va_list. */
  va_list argptr;

  /* Initialize that variable.. */
  va_start (argptr, argcount);

  for (counter = 0; counter < argcount; counter++)
    {
      /* Get the next additional argument. */
      total += va_arg (argptr, int);
    }

  /* End use of the argptr variable. */
  va_end (argptr);

  return total;
}
```

With GNU C, `va_end` is superfluous, but some other compilers might make `va_start` allocate memory so that calling `va_end` is necessary to avoid a memory leak. Before doing `va_start` again with the same variable, do `va_end` first.

Because of this possible memory allocation, it is risky (in principle) to copy one `va_list` variable to another with assignment. Instead, use `va_copy`, which copies the substance but allocates separate memory in the variable you copy to. The call looks like `va_copy (to, from)`, where both *to* and *from* should be variables of type `va_list`. In principle, do `va_end` on each of these variables before its scope ends.

Since the additional arguments' types are not specified in the function's definition, the default argument promotions (see Section 24.3 [Argument Promotions], page 156) apply to them in function calls. The function definition must take account of this; thus, if an argument was passed as `short`, the function should get it as `int`. If an argument was passed as `float`, the function should get it as `double`.

C has no mechanism to tell the variadic function how many arguments were passed to it, so its calling convention must give it a way to determine this. That's why `add_multiple_values` takes a fixed argument that says how many more arguments follow. Thus, you can call the function like this:

```
sum = add_multiple_values (3, 12, 34, 190);
/* Value is 12+34+190. */
```

In GNU C, there is no actual need to use the `va_end` function. In fact, it does nothing. It's used for compatibility with other compilers, when that matters.

It is a mistake to access variables declared as `va_list` except in the specific ways described here. Just what that type consists of is an implementation detail, which could vary from one platform to another.

22.7.3 Nested Functions

A *nested function* is a function defined inside another function. (The ability to do this is indispensable for automatic translation of certain programming languages into C.) The nested function's name is local to the block where it is defined. For example, here we define a nested function named `square`, then call it twice:

```
foo (double a, double b)
{
  double square (double z) { return z * z; }

  return square (a) + square (b);
}
```

The nested function definition can access all the variables of the containing function that are visible at the point of its definition. This is called *lexical scoping*. For example, here we show a nested function that uses an inherited variable named `offset`:

```
bar (int *array, int offset, int size)
{
  int access (int *array, int index)
    { return array[index + offset]; }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
}
```

Nested function definitions can appear wherever automatic variable declarations are allowed; that is, in any block, interspersed with the other declarations and statements in the block.

The nested function's name is visible only within the parent block; the name's scope starts from its definition and continues to the end of the containing block. If the nested function's name is the same as the parent function's name, there will be no way to refer to the parent function inside the scope of the name of the nested function.

Using `extern` or `static` on a nested function definition is an error.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function. You can do this safely, but you must be careful:

```
hack (int *array, int size, int addition)
{
  void store (int index, int value)
    { array[index] = value + addition; }

  intermediate (store, size);
}
```

Here, the function `intermediate` receives the address of `store` as an argument. If `intermediate` calls `store`, the arguments given to `store` are used to store into `array`. `store` also accesses `hack`'s local variable `addition`.

It is safe for `intermediate` to call `store` because `hack`'s stack frame, with its arguments and local variables, continues to exist during the call to `intermediate`.

Calling the nested function through its address after the containing function has exited is asking for trouble. If it is called after a containing scope level has exited, and if it refers to some of the variables that are no longer in scope, it will refer to memory containing junk or other data. It's not wise to take the risk.

The GNU C Compiler implements taking the address of a nested function using a technique called *trampolines*. This technique was described in *Lexical Closures for C++* (Thomas M. Breuel, USENIX C++ Conference Proceedings, October 17–21, 1988).

A nested function can jump to a label inherited from a containing function, provided the label was explicitly declared in the containing function (see Section 19.13 [Local Labels], page 116). Such a jump returns instantly to the containing function, exiting the nested function that did the `goto` and any intermediate function invocations as well. Here is an example:

```
bar (int *array, int offset, int size)
{
  /* Explicitly declare the label failure. */
  __label__ failure;
  int access (int *array, int index)
    {
      if (index > size)
        /* Exit this function,
           and return to bar. */
        goto failure;
      return array[index + offset];
    }
  int i;
  ...
  for (i = 0; i < size; i++)
    ... access (array, i) ...
  ...
  return 0;

 /* Control comes here from access
    if it does the goto.  */
 failure:
  return -1;
}
```

To declare the nested function before its definition, use `auto` (which is otherwise meaningless for function declarations; see Section 20.10 [auto and register], page 127). For example,

```
bar (int *array, int offset, int size)
{
  auto int access (int *, int);
  ...
  ... access (array, i) ...
  ...
  int access (int *array, int index)
    {
      ...
    }
  ...
}
```

22.7.4 Inline Function Definitions

To declare a function inline, use the `inline` keyword in its definition. Here's a simple function that takes a pointer-to-`int` and increments the integer stored there—declared inline.

```
struct list
{
  struct list *first, *second;
};

inline struct list *
list_first (struct list *p)
{
  return p->first;
}

inline struct list *
list_second (struct list *p)
{
  return p->second;
}
```

optimized compilation can substitute the inline function's body for any call to it. This is called *inlining* the function. It makes the code that contains the call run faster, significantly so if the inline function is small.

Here's a function that uses `list_second`:

```
int
pairlist_length (struct list *l)
{
  int length = 0;
  while (l)
    {
      length++;
      l = list_second (l);
    }
  return length;
}
```

Substituting the code of `list_second` into the definition of `pairlist_length` results in this code, in effect:

```
int
pairlist_length (struct list *l)
{
  int length = 0;
  while (l)
    {
      length++;
      l = l->second;
    }
  return length;
}
```

Since the definition of `list_second` does not say `extern` or `static`, that definition is used only for inlining. It doesn't generate code that can be called at run time. If not all the calls to the function are inlined, there must be a definition of the same function name in another module for them to call.

Adding `static` to an inline function definition means the function definition is limited to this compilation module. Also, it generates run-time code if necessary for the sake of any calls that were not inlined. If all calls are inlined then the function definition does not generate run-time code, but you can force generation of run-time code with the option `-fkeep-inline-functions`.

Specifying `extern` along with `inline` means the function is external and generates run-time code to be called from other separately compiled modules, as well as inlined. You can define the function as `inline` without `extern` in other modules so as to inline calls to the same function in those modules.

Why are some calls not inlined? First of all, inlining is an optimization, so non-optimized compilation does not inline.

Some calls cannot be inlined for technical reasons. Also, certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: variadic functions, use of `alloca`, use of computed goto (see Section 19.14 [Labels as Values], page 117), and use of nonlocal goto. The option `-Winline` requests a warning when a function marked `inline` is unsuitable to be inlined. The warning explains what obstacle makes it unsuitable.

Just because a call *can* be inlined does not mean it *should* be inlined. The GNU C compiler weighs costs and benefits to decide whether inlining a particular call is advantageous.

You can force inlining of all calls to a given function that can be inlined, even in a non-optimized compilation. by specifying the '`always_inline`' attribute for the function, like this:

```
/* Prototype.  */
inline void foo (const char) __attribute__((always_inline));
```

This is a GNU C extension. See Appendix D [Attributes], page 222.

A function call may be inlined even if not declared `inline` in special cases where the compiler can determine this is correct and desirable. For instance, when a static function is called only once, it will very likely be inlined. With `-flto`, link-time optimization, any function might be inlined. To absolutely prevent inlining of a specific function, specify `__attribute__((__noinline__))` in the function's definition.

22.8 Obsolete Function Features

These features of function definitions are still used in old programs, but you shouldn't write code this way today. If you are just learning C, you can skip this section.

22.8.1 Older GNU C Inlining

The GNU C spec for inline functions, before GCC version 5, defined `extern inline` on a function definition to mean to inline calls to it but *not* generate code for the function that could be called at run time. By contrast, `inline` without `extern` specified to generate run-time code for the function. In effect, ISO incompatibly flipped the meanings of these two cases. We changed GCC in version 5 to adopt the ISO specification.

Many programs still use these cases with the previous GNU C meanings. You can specify use of those meanings with the option `-fgnu89-inline`. You can also specify this for a single function with `__attribute__ ((gnu_inline))`. Here's an example:

```
inline __attribute__ ((gnu_inline))
int
inc (int *a)
{
  (*a)++;
}
```

22.8.2 Old-Style Function Definitions

The syntax of C traditionally allows omitting the data type in a function declaration if it specifies a storage class or a qualifier. Then the type defaults to `int`. For example:

```
static foo (double x);
```

defaults the return type to `int`. This is bad practice; if you see it, fix it.

An *old-style* (or “K&R”) function definition is the way function definitions were written in the 1980s. It looks like this:

```
rettype
function (parmnames)
  parm_declarations
{
  body
}
```

In *parmnames*, only the parameter names are listed, separated by commas. Then *parm_declarations* declares their data types; these declarations look just like variable declarations. If a parameter is listed in *parmnames* but has no declaration, it is implicitly declared `int`.

There is no reason to write a definition this way nowadays, but they can still be seen in older GNU programs.

An old-style variadic function definition looks like this:

```
#include <varargs.h>

int
add_multiple_values (va_alist)
    va_dcl
{
  int argcount;
  int counter, total = 0;

  /* Declare a variable of type va_list. */
  va_list argptr;

  /* Initialize that variable. */
  va_start (argptr);

  /* Get the first argument (fixed). */
  argcount = va_arg (int);

  for (counter = 0; counter < argcount; counter++)
    {
      /* Get the next additional argument. */
      total += va_arg (argptr, int);
    }

  /* End use of the argptr variable. */
  va_end (argptr);

  return total;
}
```

Note that the old-style variadic function definition has no fixed parameter variables; all arguments must be obtained with `va_arg`.

23 Compatible Types

Declaring a function or variable twice is valid in C only if the two declarations specify *compatible* types. In addition, some operations on pointers require operands to have compatible target types.

In C, two different primitive types are never compatible. Likewise for the defined types `struct`, `union` and `enum`: two separately defined types are incompatible unless they are defined exactly the same way.

However, there are a few cases where different types can be compatible:

- Every enumeration type is compatible with some integer type. In GNU C, the choice of integer type depends on the largest enumeration value.
- Array types are compatible if the element types are compatible and the sizes (when specified) match.
- Pointer types are compatible if the pointer target types are compatible.
- Function types that specify argument types are compatible if the return types are compatible and the argument types are compatible, argument by argument. In addition, they must all agree in whether they use `...` to allow additional arguments.
- Function types that don't specify argument types are compatible if the return types are.
- Function types that specify the argument types are compatible with function types that omit them, if the return types are compatible and the specified argument types are unaltered by the argument promotions (see Section 24.3 [Argument Promotions], page 156).

In order for types to be compatible, they must agree in their type qualifiers. Thus, `const int` and `int` are incompatible. It follows that `const int *` and `int *` are incompatible too (they are pointers to types that are not compatible).

If two types are compatible ignoring the qualifiers, we call them *nearly compatible*. (If they are array types, we ignore qualifiers on the element types.[1]) Comparison of pointers is valid if the pointers' target types are nearly compatible. Likewise, the two branches of a conditional expression may be pointers to nearly compatible target types.

If two types are compatible ignoring the qualifiers, and the first type has all the qualifiers of the second type, we say the first is *upward compatible* with the second. Assignment of pointers requires the assigned pointer's target type to be upward compatible with the right operand (the new value)'s target type.

---

[1] This is a GNU C extension.

24 Type Conversions

C converts between data types automatically when that seems clearly necessary. In addition, you can convert explicitly with a *cast*.

24.1 Explicit Type Conversion

You can do explicit conversions using the unary *cast* operator, which is written as a type designator (see Section 11.6 [Type Designators], page 51) in parentheses. For example, `(int)` is the operator to cast to type `int`. Here's an example of using it:

```
{
  double d = 5.5;

  printf ("Floating point value: %f\n", d);
  printf ("Rounded to integer: %d\n", (int) d);
}
```

Using `(int) d` passes an `int` value as argument to `printf`, so you can print it with '`%d`'. Using just `d` without the cast would pass the value as `double`. That won't work at all with '`%d`'; the results would be gibberish.

To divide one integer by another without rounding, cast either of the integers to `double` first:

```
(double) dividend / divisor
dividend / (double) divisor
```

It is enough to cast one of them, because that forces the common type to `double` so the other will be converted automatically.

The valid cast conversions are:

- One numerical type to another.
- One pointer type to another. (Converting between pointers that point to functions and pointers that point to data is not standard C.)
- A pointer type to an integer type.
- An integer type to a pointer type.
- To a union type, from the type of any alternative in the union (see Section 15.14 [Unions], page 83). (This is a GNU extension.)
- Anything, to `void`.

24.2 Assignment Type Conversions

Certain type conversions occur automatically in assignments and certain other contexts. These are the conversions assignments can do:

- Converting any numeric type to any other numeric type.
- Converting `void *` to any other pointer type (except pointer-to-function types).
- Converting any other pointer type to `void *`. (except pointer-to-function types).
- Converting 0 (a null pointer constant) to any pointer type.
- Converting any pointer type to `bool`. (The result is 1 if the pointer is not null.)

- Converting between pointer types when the left-hand target type is upward compatible with the right-hand target type. See Chapter 23 [Compatible Types], page 154.

These type conversions occur automatically in certain contexts, which are:

- An assignment converts the type of the right-hand expression to the type wanted by the left-hand expression. For example,

  ```
  double i;
  i = 5;
  ```

  converts 5 to `double`.
- A function call, when the function specifies the type for that argument, converts the argument value to that type. For example,

  ```
  void foo (double);
  foo (5);
  ```

  converts 5 to `double`.
- A `return` statement converts the specified value to the type that the function is declared to return. For example,

  ```
  double
  foo ()
  {
    return 5;
  }
  ```

  also converts 5 to `double`.

In all three contexts, if the conversion is impossible, that constitutes an error.

24.3 Argument Promotions

When a function's definition or declaration does not specify the type of an argument, that argument is passed without conversion in whatever type it has, with these exceptions:

- Some narrow numeric values are *promoted* to a wider type. If the expression is a narrow integer, such as `char` or `short`, the call converts it automatically to `int` (see Section 11.1 [Integer Types], page 46).[1]

  In this example, the expression `c` is passed as an `int`:

  ```
  char c = '$';

  printf ("Character c is '%c'\n", c);
  ```
- If the expression has type `float`, the call converts it automatically to `double`.
- An array as argument is converted to a pointer to its zeroth element.
- A function name as argument is converted to a pointer to that function.

---

[1] On an embedded controller where `char` or `short` is the same width as `int`, `unsigned char` or `unsigned short` promotes to `unsigned int`, but that never occurs in GNU C on real computers.

24.4 Operand Promotions

The operands in arithmetic operations undergo type conversion automatically. These *operand promotions* are the same as the argument promotions except without converting `float` to `double`. In other words, the operand promotions convert

- `char` or `short` (whether signed or not) to `int`.
- an array to a pointer to its zeroth element, and
- a function name to a pointer to that function.

24.5 Common Type

Arithmetic binary operators (except the shift operators) convert their operands to the *common type* before operating on them. Conditional expressions also convert the two possible results to their common type. Here are the rules for determining the common type.

If one of the numbers has a floating-point type and the other is an integer, the common type is that floating-point type. For instance,

```
5.6 * 2    ⇒ 11.2 /* a double value */
```

If both are floating point, the type with the larger range is the common type.

If both are integers but of different widths, the common type is the wider of the two.

If they are integer types of the same width, the common type is unsigned if either operand is unsigned, and it's `long` if either operand is `long`. It's `long long` if either operand is `long long`.

These rules apply to addition, subtraction, multiplication, division, remainder, comparisons, and bitwise operations. They also apply to the two branches of a conditional expression, and to the arithmetic done in a modifying assignment operation.

25 Scope

Each definition or declaration of an identifier is visible in certain parts of the program, which is typically less than the whole of the program. The parts where it is visible are called its *scope*.

Normally, declarations made at the top-level in the source—that is, not within any blocks and function definitions—are visible for the entire contents of the source file after that point. This is called *file scope* (see Section 20.6 [File-Scope Variables], page 125).

Declarations made within blocks of code, including within function definitions, are visible only within those blocks. This is called *block scope*. Here is an example:

```
void
foo (void)
{
  int x = 42;
}
```

In this example, the variable `x` has block scope; it is visible only within the `foo` function definition block. Thus, other blocks could have their own variables, also named `x`, without any conflict between those variables.

A variable declared inside a subblock has a scope limited to that subblock,

```
void
foo (void)
{
  {
    int x = 42;
  }
  // x is out of scope here.
}
```

If a variable declared within a block has the same name as a variable declared outside of that block, the definition within the block takes precedence during its scope:

```
int x = 42;

void
foo (void)
{
  int x = 17;
  printf ("%d\n", x);
}
```

This prints 17, the value of the variable `x` declared in the function body block, rather than the value of the variable `x` at file scope. We say that the inner declaration of `x` *shadows* the outer declaration, for the extent of the inner declaration's scope.

A declaration with block scope can be shadowed by another declaration with the same name in a subblock.

```
void
foo (void)
{
  char *x = "foo";
  {
    int x = 42;
    ...
    exit (x / 6);
  }
}
```

A function parameter's scope is the entire function body, but it can be shadowed. For example:

```
int x = 42;

void
foo (int x)
{
  printf ("%d\n", x);
}
```

This prints the value of `x` the function parameter, rather than the value of the file-scope variable `x`.

Labels (see Section 19.12 [goto Statement], page 114) have *function* scope: each label is visible for the whole of the containing function body, both before and after the label declaration:

```
void
foo (void)
{
  ...
  goto bar;
  ...
  {  // Subblock does not affect labels.
    bar:
    ...
  }
  goto bar;
}
```

Except for labels, a declared identifier is not visible to code before its declaration. For example:

```
int x = 5;
int y = x + 10;
```

will work, but:

```
int x = y + 10;
int y = 5;
```

cannot refer to the variable `y` before its declaration.

This is part of the GNU C Intro and Reference Manual and covered by its license.

26 Preprocessing

As the first stage of compiling a C source module, GCC transforms the text with text substitutions and file inclusions. This is called *preprocessing*.

26.1 Preprocessing Overview

GNU C performs preprocessing on each line of a C program as the first stage of compilation. Preprocessing operates on a line only when it contains a *preprocessing directive* or uses a *macro*—all other lines pass through preprocessing unchanged.

Here are some jobs that preprocessing does. The rest of this chapter gives the details.

- Inclusion of header files. These are files (usually containing declarations and macro definitions) that can be substituted into your program.
- Macro expansion. You can define *macros*, which are abbreviations for arbitrary fragments of C code. Preprocessing replaces the macros with their definitions. Some macros are automatically predefined.
- Conditional compilation. You can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file that is then compiled, you can use line control to inform the compiler where each source line originally came from.
- Compilation control. `#pragma` and `_Pragma` invoke some special compiler features in how to handle certain constructs.
- Diagnostics. You can detect problems at compile time and issue errors or warnings.

Except for expansion of predefined macros, all these operations happen only if you use preprocessing directives to request them.

Preprocessing does not recognize the operators of the C language itself, outside of the operand of the `#if` directive (see Section 26.6.2.2 [if], page 186). However, it supports three *preprocessing operators* of its own: `#` (see Section 26.5.4 [Stringification], page 171), `##` (see Section 26.5.5 [Concatenation], page 173) and `defined` (see Section 26.6.2.2 [if], page 186).

26.2 Directives

Preprocessing directives are lines in the program that start with ‘`#`’. Whitespace is allowed before and after the ‘`#`’. The ‘`#`’ is followed by an identifier, the *directive name*. It specifies the operation to perform. Here are a couple of examples:

```
#define LIMIT 51
  #   undef LIMIT
# error You screwed up!
```

We usually refer to a directive as `#name` where *name* is the directive name. For example, `#define` means the directive that defines a macro.

The ‘`#`’ that begins a directive cannot come from a macro expansion. Also, the directive name is not macro expanded. Thus, if `foo` is defined as a macro expanding to `define`, that does not make `#foo` a valid preprocessing directive.

The set of valid directive names is fixed. Programs cannot define new preprocessing directives.

Some directives require arguments; these make up the rest of the directive line and must be separated from the directive name by whitespace. For example, `#define` must be followed by a macro name and the intended expansion of the macro.

A preprocessing directive cannot cover more than one line. The line can, however, be continued with backslash-newline, or by a '`/*...*/`'-style comment that extends past the end of the line. These will be replaced (by nothing, or by whitespace) before the directive is processed.

26.3 Preprocessing Tokens

Preprocessing divides C code (minus its comments) into *tokens* that are similar to C tokens, but not exactly the same. Here are the quirks of preprocessing tokens.

The main classes of preprocessing tokens are identifiers, preprocessing numbers, string constants, character constants, and punctuators; there are a few others too.

identifier

An *identifier* preprocessing token is syntactically like an identifier in C: any sequence of letters, digits, or underscores, as well as non-ASCII characters represented using '`\U`' or '`\u`', that doesn't begin with a digit.

During preprocessing, the keywords of C have no special significance; at that stage, they are simply identifiers. Thus, you can define a macro whose name is a keyword. The only identifier that is special during preprocessing is `defined` (see Section 26.6.2.3 [defined], page 187).

preprocessing number

A *preprocessing number* is something that preprocessing treats textually as a number, including C numeric constants, and other sequences of characters which resemble numeric constants. Preprocessing does not try to verify that a preprocessing number is a valid number in C, and indeed it need not be one.

More precisely, preprocessing numbers begin with an optional period, a required decimal digit, and then continue with any sequence of letters, digits, underscores, periods, and exponents. Exponents are the two-character sequences '`e+`', '`e-`', '`E+`', '`E-`', '`p+`', '`p-`', '`P+`', and '`P-`'. (The exponents that begin with '`p`' or '`P`' are new to C99. They are used for hexadecimal floating-point constants.)

The reason behind this unusual syntactic class is that the full complexity of numeric constants is irrelevant during preprocessing. The distinction between lexically valid and invalid floating-point numbers, for example, doesn't matter at this stage. The use of preprocessing numbers makes it possible to split an identifier at any position and get exactly two tokens, and reliably paste them together using the `##` preprocessing operator (see Section 26.5.5 [Concatenation], page 173).

punctuator

A *punctuator* is syntactically like an operator. These are the valid punctuators:

```
[  ]   (  )  {  }  .  ->
++ --  &  *  +  -  ~  !
```

```
/  %   << >> <  >  <= >=  ==  !=  ^  |  &&  ||
?  :   ;  ...
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
,  #   ##
<: :>  <% %>  %:  %:%:
```

string constant

A string constant in the source code is recognized by preprocessing as a single preprocessing token.

character constant

A character constant in the source code is recognized by preprocessing as a single preprocessing token.

header name

Within the `#include` directive, preprocessing recognizes a *header name* token. It consists of ‘`"name"`’, where *name* is a sequence of source characters other than newline and ‘`"`’, or ‘`<name>`’, where *name* is a sequence of source characters other than newline and ‘`>`’.

In practice, it is more convenient to think that the `#include` line is exempt from tokenization.

other

Any other character that’s valid in a C source program is treated as a separate preprocessing token.

Once the program is broken into preprocessing tokens, they remain separate until the end of preprocessing. Macros that generate two consecutive tokens insert whitespace to keep them separate, if necessary. For example,

```
#define foo() bar
foo()baz
     ↦ bar baz
not
     ↦ barbaz
```

The only exception is with the `##` preprocessing operator, which pastes tokens together (see Section 26.5.5 [Concatenation], page 173).

Preprocessing treats the null character (code 0) as whitespace, but generates a warning for it because it may be invisible to the user (many terminals do not display it at all) and its presence in the file is probably a mistake.

26.4 Header Files

A header file is a file of C code, typically containing C declarations and macro definitions (see Section 26.5 [Macros], page 167), to be shared between several source files. You request the use of a header file in your program by *including* it, with the C preprocessing directive `#include`.

Header files serve two purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations that you need to invoke system calls and libraries.

- Program-specific header files contain declarations for interfaces between the source files of a particular program. It is a good idea to create a header file for related declarations and macro definitions if all or most of them are needed in several different source files.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, you can change them in one place, and programs that include the header file will then automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to change one copy will result in inconsistencies within a program.

In C, the usual convention is to give header files names that end with `.h`. It is most portable to use only letters, digits, dashes, and underscores in header file names, and at most one dot.

The operation of including another source file isn't actually limited to the sort of code we put into header files. You can put any sort of C code into a separate file, then use `#include` to copy it virtually into other C source files. But that is a strange thing to do.

26.4.1 `#include` Syntax

You can specify inclusion of user and system header files with the preprocessing directive `#include`. It has two variants:

`#include <file>`

This variant is used for system header files. It searches for a file named *file* in a standard list of system directories. You can prepend directories to this list with the `-I` option (see Section “Invoking GCC” in *Using the GNU Compiler Collection*).

`#include "file"`

This variant is used for header files of your own program. It searches for a file named *file* first in the directory containing the current file, then in the quote directories, then the same directories used for `<file>`. You can prepend directories to the list of quote directories with the `-iquote` option.

The argument of `#include`, whether delimited with quote marks or angle brackets, behaves like a string constant in that comments are not recognized, and macro names are not expanded. Thus, `#include <x/*y>` specifies inclusion of a system header file named `x/*y`.

However, if backslashes occur within *file*, they are considered ordinary text characters, not escape characters: character escape sequences such as used in string constants in C are not meaningful here. Thus, `#include "x\n\\y"` specifies a filename containing three backslashes. By the same token, there is no way to escape ‘`"`’ or ‘`>`’ to include it in the header file name if it would instead end the file name.

Some systems interpret ‘`\`’ as a file name component separator. All these systems also interpret ‘`/`’ the same way. It is most portable to use only ‘`/`’.

It is an error to put anything other than comments on the `#include` line after the file name.

26.4.2 `#include` Operation

The `#include` directive works by scanning the specified header file as input before continuing with the rest of the current file. The result of preprocessing consists of the text already generated, followed by the result of preprocessing the included file, followed by whatever results from the text after the `#include` directive. For example, if you have a header file `header.h` as follows,

```
char *test (void);
```

and a main program called `program.c` that uses the header file, like this,

```
int x;
#include "header.h"

int
main (void)
{
  puts (test ());
}
```

the result is equivalent to putting this text in `program.c`:

```
int x;
char *test (void);

int
main (void)
{
  puts (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, an included file must consist of complete tokens. Comments and string literals that have not been closed by the end of an included file are invalid. For error recovery, the compiler terminates them at the end of the file.

To avoid confusion, it is best if header files contain only complete syntactic units—function declarations or definitions, type declarations, etc.

The line following the `#include` directive is always treated as a separate line, even if the included file lacks a final newline. There is no problem putting a preprocessing directive there.

26.4.3 Search Path

GCC looks in several different places for header files to be included. On the GNU system, and Unix systems, the default directories for system header files are:

```
libdir/gcc/target/version/include
/usr/local/include
libdir/gcc/target/version/include-fixed
```

```
libdir/target/include
/usr/include/target
/usr/include
```

The list may be different in some operating systems. Other directories are added for C++.

In the above, *target* is the canonical name of the system GCC was configured to compile code for; often but not always the same as the canonical name of the system it runs on. *version* is the version of GCC in use.

You can add to this list with the `-Idir` command-line option. All the directories named by `-I` are searched, in left-to-right order, *before* the default directories. The only exception is when `dir` is already searched by default. In this case, the option is ignored and the search order for system directories remains unchanged.

Duplicate directories are removed from the quote and bracket search chains before the two chains are merged to make the final search chain. Thus, it is possible for a directory to occur twice in the final search chain if it was specified in both the quote and bracket chains.

You can prevent GCC from searching any of the default directories with the `-nostdinc` option. This is useful when you are compiling an operating system kernel or some other program that does not use the standard C library facilities, or the standard C library itself. `-I` options are not ignored as described above when `-nostdinc` is in effect.

GCC looks for headers requested with `#include "file"` first in the directory containing the current file, then in the *quote directories* specified by `-iquote` options, then in the same places it looks for a system header. For example, if `/usr/include/sys/stat.h` contains `#include "types.h"`, GCC looks for `types.h` first in `/usr/include/sys`, then in the quote directories and then in its usual search path.

`#line` (see Section 26.8 [Line Control], page 189) does not change GCC's idea of the directory containing the current file.

The `-I-` is an old-fashioned, deprecated way to specify the quote directories. To look for headers in a directory named `-`, specify `-I./-`. There are several more ways to adjust the header search path. See Section "Invoking GCC" in *Using the GNU Compiler Collection*.

26.4.4 Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice. This is very likely to cause an error, e.g. when the compiler sees the same structure definition twice.

The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
/* File foo.  */
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

the entire file

#endif /* !FILE_FOO_SEEN */
```

This construct is commonly known as a *wrapper #ifndef*. When the header is included again, the conditional will be false, because `FILE_FOO_SEEN` is defined. Preprocessing skips over the entire contents of the file, so that compilation will never "see" the file contents twice in one module.

GCC optimizes this case even further. It remembers when a header file has a wrapper `#ifndef`. If a subsequent `#include` specifies that header, and the macro in the `#ifndef` is still defined, it does not bother to rescan the file at all.

You can put comments in the header file outside the wrapper. They do not interfere with this optimization.

The macro `FILE_FOO_SEEN` is called the *controlling macro* or *guard macro*. In a user header file, the macro name should not begin with '`_`'. In a system header file, it should begin with '`__`' (or '`_`' followed by an upper-case letter) to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

26.4.5 Computed Includes

Sometimes it is necessary to select one of several different header files to be included into your program. They might specify configuration parameters to be used on different sorts of operating systems, for instance. You could do this with a series of conditionals,

```
#if SYSTEM_1
# include "system_1.h"
#elif SYSTEM_2
# include "system_2.h"
#elif SYSTEM_3
/* ... */
#endif
```

That rapidly becomes tedious. Instead, GNU C offers the ability to use a macro for the header name. This is called a *computed include*. Instead of writing a header name as the direct argument of `#include`, you simply put a macro name there instead:

```
#define SYSTEM_H "system_1.h"
/* ... */
#include SYSTEM_H
```

`SYSTEM_H` is expanded, then `system_1.h` is included as if the `#include` had been written with that name. `SYSTEM_H` could be defined by your Makefile with a `-D` option.

You must be careful when you define such a macro. `#define` saves tokens, not text. GCC has no way of knowing that the macro will be used as the argument of `#include`, so it generates ordinary tokens, not a header name. This is unlikely to cause problems if you use double-quote includes, which are syntactically similar to string constants. If you use angle brackets, however, you may have trouble.

The syntax of a computed include is actually a bit more general than the above. If the first non-whitespace character after `#include` is not '`"`' or '`<`', then the entire line is macro-expanded like running text would be.

If the line expands to a single string constant, the contents of that string constant are the file to be included. Preprocessing does not re-examine the string for embedded quotes, but neither does it process backslash escapes in the string. Therefore

```
#define HEADER "a\"b"
```

```
#include HEADER
```

looks for a file named `a\"b`. Preprocessing searches for the file according to the rules for double-quoted includes.

If the line expands to a token stream beginning with a '`<`' token and including a '`>`' token, then the tokens between the '`<`' and the first '`>`' are combined to form the filename to be included. Any whitespace between tokens is reduced to a single space; then any space after the initial '`<`' is retained, but a trailing space before the closing '`>`' is ignored. Preprocessing searches for the file according to the rules for angle-bracket includes.

In either case, if there are any tokens on the line after the file name, an error occurs and the directive is not processed. It is also an error if the result of expansion does not match either of the two expected forms.

These rules are implementation-defined behavior according to the C standard. To minimize the risk of different compilers interpreting your computed includes differently, we recommend you use only a single object-like macro that expands to a string constant. That also makes it clear to people reading your program.

26.5 Macros

A *macro* is a fragment of code that has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. *Object-like* macros resemble data objects when used, *function-like* macros resemble function calls.

You may define any valid identifier as a macro, even if it is a C keyword. In the preprocessing stage, GCC does not know anything about keywords. This can be useful if you wish to hide a keyword such as `const` from an older compiler that does not understand it. However, the preprocessing operator `defined` (see Section 26.6.2.3 [defined], page 187) can never be defined as a macro.

The preprocessing operator `#` is used in macros for stringification of an argument (see Section 26.5.4 [Stringification], page 171), and `##` is used for concatenation of arguments into larger tokens (see Section 26.5.5 [Concatenation], page 173)

26.5.1 Object-like Macros

An *object-like macro* is a simple identifier that will be replaced by a code fragment. It is called object-like because in most cases the use of the macro looks like reference to a data object in code that uses it. These macros are most commonly used to give symbolic names to numeric constants.

The way to define macros with the `#define` directive. `#define` is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's *body*, *expansion* or *replacement list*. For example,

```
#define BUFFER_SIZE 1024
```

defines a macro named `BUFFER_SIZE` as an abbreviation for the token `1024`. If somewhere after this `#define` directive there comes a C statement of the form

```
foo = (char *) malloc (BUFFER_SIZE);
```

then preprocessing will recognize and *expand* the macro `BUFFER_SIZE`, so that compilation will see the tokens:

```
foo = (char *) malloc (1024);
```

By convention, macro names are written in upper case. Programs are easier to read when it is possible to tell at a glance which names are macros. Macro names that start with ‘__’ are reserved for internal uses, and many of them are defined automatically, so don’t define such macro names unless you really know what you’re doing. Likewise for macro names that start with ‘_’ and an upper-case letter.

The macro’s body ends at the end of the `#define` line. You may continue the definition onto multiple lines, if necessary, using backslash-newline. When the macro is expanded, however, it will all come out on one line. For example,

```
#define NUMBERS 1, \
                2, \
                3
int x[] = { NUMBERS };
     ↦ int x[] = { 1, 2, 3 };
```

The most common visible consequence of this is surprising line numbers in error messages.

There is no restriction on what can go in a macro body provided it decomposes into valid preprocessing tokens. Parentheses need not balance, and the body need not resemble valid C code. (If it does not, you may get error messages from the C compiler when you use the macro.)

Preprocessing scans the program sequentially. A macro definition takes effect right after its appearance. Therefore, the following input

```
foo = X;
#define X 4
bar = X;
```

produces

```
foo = X;
bar = 4;
```

When preprocessing expands a macro name, the macro’s expansion replaces the macro invocation, then the expansion is examined for more macros to expand. For example,

```
#define TABLESIZE BUFSIZE
#define BUFSIZE 1024
TABLESIZE
     ↦ BUFSIZE
     ↦ 1024
```

`TABLESIZE` is expanded first to produce `BUFSIZE`, then that macro is expanded to produce the final result, `1024`.

Notice that `BUFSIZE` was not defined when `TABLESIZE` was defined. The `#define` for `TABLESIZE` uses exactly the expansion you specify—in this case, `BUFSIZE`—and does not check to see whether it too contains macro names. Only when you *use* `TABLESIZE` is the result of its expansion scanned for more macro names.

This makes a difference if you change the definition of `BUFSIZE` at some point in the source file. `TABLESIZE`, defined as shown, will always expand using the definition of `BUFSIZE` that is currently in effect:

```
#define BUFSIZE 1020
```

```
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Now `TABLESIZE` expands (in two stages) to `37`.

If the expansion of a macro contains its own name, either directly or via intermediate macros, it is not expanded again when the expansion is examined for more macros. This prevents infinite recursion. See Section 26.5.10.6 [Self-Referential Macros], page 182, for the precise details.

26.5.2 Function-like Macros

You can also define macros whose use looks like a function call. These are called *function-like macros*. To define one, use the `#define` directive with a pair of parentheses immediately after the macro name. For example,

```
#define lang_init()  c_init ()
lang_init ()
     ↦ c_init ()
lang_init     ()
     ↦ c_init ()
lang_init()
     ↦ c_init ()
```

There must be no space between the macro name and the following open-parenthesis in the the `#define` directive; that's what indicates you're defining a function-like macro. However, you can add unnecessary whitespace around the open-parenthesis (and around the close-parenthesis) when you *call* the macro; they don't change anything.

A function-like macro is expanded only when its name appears with a pair of parentheses after it. If you write just the name, without parentheses, it is left alone. This can be useful when you have a function and a macro of the same name, and you wish to use the function sometimes. Whitespace and line breaks before or between the parentheses are ignored when the macro is called.

```
extern void foo(void);
#define foo() /* optimized inline version */
/* ... */
  foo();
  funcptr = foo;
```

Here the call to `foo()` expands the macro, but the function pointer `funcptr` gets the address of the real function `foo`. If the macro were to be expanded there, it would cause a syntax error.

If you put spaces between the macro name and the parentheses in the macro definition, that does not define a function-like macro, it defines an object-like macro whose expansion happens to begin with a pair of parentheses. Here is an example:

```
#define lang_init ()    c_init()
lang_init()
     ↦ () c_init()()
```

The first two pairs of parentheses in this expansion come from the macro. The third is the pair that was originally after the macro invocation. Since `lang_init` is an object-like macro, it does not consume those parentheses.

Any name can have at most one macro definition at a time. Thus, you can't define the same name as an object-like macro and a function-like macro at once.

26.5.3 Macro Arguments

Function-like macros can take *arguments*, just like true functions. To define a macro that uses arguments, you insert *parameters* between the pair of parentheses in the macro definition that make the macro function-like. The parameters must be valid C identifiers, separated by commas and optionally whitespace.

To invoke a macro that takes arguments, you write the name of the macro followed by a list of *actual arguments* in parentheses, separated by commas. The invocation of the macro need not be restricted to a single logical line—it can cross as many lines in the source file as you wish. The number of arguments you give must match the number of parameters in the macro definition. When the macro is expanded, each use of a parameter in its body is replaced by the tokens of the corresponding argument. (The macro body is not required to use all of the parameters.)

As an example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs, and some uses.

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
  x = min(a, b);        ↦ x = ((a) < (b) ? (a) : (b));
  y = min(1, 2);        ↦ y = ((1) < (2) ? (1) : (2));
  z = min(a+28, *p);    ↦ z = ((a+28) < (*p) ? (a+28) : (*p));
```

In this small example you can already see several of the dangers of macro arguments. See Section 26.5.10 [Macro Pitfalls], page 179, for detailed explanations.

Leading and trailing whitespace in each argument is dropped, and all whitespace between the tokens of an argument is reduced to a single space. Parentheses within each argument must balance; a comma within such parentheses does not end the argument. However, there is no requirement for square brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

```
macro (array[x = y, x + 1])
```

passes two arguments to `macro`: `array[x = y` and `x + 1]`. If you want to supply `array[x = y, x + 1]` as an argument, you can write it as `array[(x = y, x + 1)]`, which is equivalent C code. However, putting an assignment inside an array subscript is to be avoided anyway.

All arguments to a macro are completely macro-expanded before they are substituted into the macro body. After substitution, the complete text is scanned again for macros to expand, including the arguments. This rule may seem strange, but it is carefully designed so you need not worry about whether any function call is actually a macro invocation. You can run into trouble if you try to be too clever, though. See Section 26.5.10.7 [Argument Prescan], page 183, for detailed discussion.

For example, `min (min (a, b), c)` is first expanded to

```
min (((a) < (b) ? (a) : (b)), (c))
```

and then to

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(The line breaks shown here for clarity are not actually generated.)

You can leave macro arguments empty without error, but many macros will then expand to invalid code. You cannot leave out arguments entirely; if a macro takes two arguments, there must be exactly one comma at the top level of its argument list. Here are some silly examples using `min`:

```
min(, b)        ↦ ((   ) < (b) ? (   ) : (b))
min(a, )        ↦ ((a  ) < ( ) ? (a  ) : ( ))
min(,)          ↦ ((   ) < ( ) ? (   ) : ( ))
min((,),)       ↦ (((,)) < ( ) ? ((,)) : ( ))

min()      error  macro "min" requires 2 arguments, but only 1 given
min(,,)    error  macro "min" passed 3 arguments, but takes just 2
```

Whitespace is not a preprocessing token, so if a macro `foo` takes one argument, `foo ()` and `foo ( )` both supply it an empty argument.

Macro parameters appearing inside string literals are not replaced by their corresponding actual arguments.

```
#define foo(x) x, "x"
foo(bar)        ↦ bar, "x"
```

See the next subsection for how to insert macro arguments into a string literal.

The token following the macro call and the last token of the macro expansion do not become one token even if it looks like they could:

```
#define foo()  abc
foo()def        ↦ abc def
```

26.5.4 Stringification

Sometimes you may want to convert a macro argument into a string constant. Parameters are not replaced inside string constants, but you can use the `#` preprocessing operator instead. When a macro parameter is used with a leading `#`, preprocessing replaces it with the literal text of the actual argument, converted to a string constant. Unlike normal parameter replacement, the argument is not macro-expanded first. This is called *stringification*.

There is no way to combine an argument with surrounding text and stringify it all together. But you can write a series of string constants and stringified arguments. After preprocessing replaces the stringified arguments with string constants, the consecutive string constants will be concatenated into one long string constant (see Section 12.7 [String Constants], page 56).

Here is an example that uses stringification and concatenation of string constants:

```
#define WARN_IF(EXP) \
  do { if (EXP) \
          fprintf (stderr, "Warning: " #EXP "\n"); } \
  while (0)

WARN_IF (x == 0);
     ↦
  do { if (x == 0)
          fprintf (stderr, "Warning: " "x == 0" "\n"); }
  while (0);
```

The argument for `EXP` is substituted once, as is, into the `if` statement, and once, stringified, into the argument to `fprintf`. If `x` were a macro, it would be expanded in the `if` statement but not in the string.

The `do` and `while (0)` are a kludge to make it possible to write `WARN_IF (`*arg*`);`. The resemblance of `WARN_IF` to a function makes that a natural way to write it. See Section 26.5.10.3 [Swallowing the Semicolon], page 180.

Stringification in C involves more than putting double-quote characters around the fragment. It also backslash-escapes the quotes surrounding embedded string constants, and all backslashes within string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying `p = "foo\n";` results in `"p = \"foo\\n\";"`. However, backslashes that are not inside string or character constants are not duplicated: '`\n`' by itself stringifies to `"\n"`.

All leading and trailing whitespace in text being stringified is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result. Comments are replaced by whitespace long before stringification happens, so they never appear in stringified text.

There is no way to convert a macro argument into a character constant.

To stringify the result of expansion of a macro argument, you have to use two levels of macros, like this:

```
#define xstr(S) str(S)
#define str(s) #s
#define foo 4
str (foo)
     ↦ "foo"
xstr (foo)
     ↦ xstr (4)
     ↦ str (4)
     ↦ "4"
```

`s` is stringified when it is used in `str`, so it is not macro-expanded first. But `S` is an ordinary argument to `xstr`, so it is completely macro-expanded before `xstr` itself is expanded (see Section 26.5.10.7 [Argument Prescan], page 183). Therefore, by the time `str` gets to its argument text, that text already been macro-expanded.

26.5.5 Concatenation

It is often useful to merge two tokens into one while expanding macros. This is called *token pasting* or *token concatenation*. The `##` preprocessing operator performs token pasting. When a macro is expanded, the two tokens on either side of each `##` operator are combined into a single token, which then replaces the `##` and the two original tokens in the macro expansion. Usually both will be identifiers, or one will be an identifier and the other a preprocessing number. When pasted, they make a longer identifier.

Concatenation into an identifier isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as `1.5` and `e3`) into a number. Also, multi-character operators such as `+=` can be formed by token pasting.

However, two tokens that don't together form a valid token cannot be pasted together. For example, you cannot concatenate `x` with `+`, not in either order. Trying this issues a warning and keeps the two tokens separate. Whether it puts white space between the tokens is undefined. It is common to find unnecessary uses of `##` in complex macros. If you get this warning, it is likely that you can simply remove the `##`.

The tokens combined by `##` could both come from the macro body, but then you could just as well write them as one token in the first place. Token pasting is useful when one or both of the tokens comes from a macro argument. If either of the tokens next to an `##` is a parameter name, it is replaced by its actual argument before `##` executes. As with stringification, the actual argument is not macro-expanded first. If the argument is empty, that `##` has no effect.

Keep in mind that preprocessing converts comments to whitespace before it looks for uses of macros. Therefore, you cannot create a comment by concatenating '`/`' and '`*`'. You can put as much whitespace between `##` and its operands as you like, including comments, and you can put comments in arguments that will be concatenated.

It is an error to use `##` at the beginning or end of a macro body.

Multiple `##` preprocessing operators are handled left-to-right, so that '`1 ## e ## -2`' pastes into '`1e-2`'. (Right-to-left processing would first generate '`e-2`', which is an invalid token.) When `#` and `##` are used together, they are all handled left-to-right.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
  char *name;
  void (*function) (void);
};

struct command commands[] =
{
  { "quit", quit_command },
  { "help", help_command },
  /* ... */
};
```

It would be cleaner not to have to write each command name twice, once in the string constant and once in the function name. A macro that takes the name of a command as an argument can make this unnecessary. It can create the string constant with stringification, and the function name by concatenating the argument with '`_command`'. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  /* ... */
};
```

26.5.6 Variadic Macros

A macro can be declared to accept a variable number of arguments much as a function can. The syntax for defining the macro is similar to that of a function. Here is an example:

```
#define eprintf(...) fprintf (stderr, __VA_ARGS__)
```

This kind of macro is called *variadic*. When the macro is invoked, all the tokens in its argument list after the last named argument (this macro has none), including any commas, become the *variable argument*. This sequence of tokens replaces the identifier `__VA_ARGS__` in the macro body wherever it appears. Thus, we have this expansion:

```
eprintf ("%s:%d: ", input_file, lineno)
     ↦  fprintf (stderr, "%s:%d: ", input_file, lineno)
```

The variable argument is completely macro-expanded before it is inserted into the macro expansion, just like an ordinary argument. You may use the `#` and `##` preprocessing operators to stringify the variable argument or to paste its leading or trailing token with another token. (But see below for an important special case for `##`.)

Warning: don't use the identifier `__VA_ARGS__` for anything other than this.

If your macro is complicated, you may want a more descriptive name for the variable argument than `__VA_ARGS__`. You can write an argument name immediately before the '`...`'; that name is used for the variable argument.[1] The `eprintf` macro above could be written thus:

```
#define eprintf(args...) fprintf (stderr, args)
```

A variadic macro can have named arguments as well as variable arguments, so `eprintf` can be defined like this, instead:

```
#define eprintf(format, ...) \
  fprintf (stderr, format, __VA_ARGS__)
```

This formulation is more descriptive, but what if you want to specify a format string that takes no arguments? In GNU C, you can omit the comma before the variable arguments if they are empty, but that puts an extra comma in the expansion:

```
eprintf ("success!\n")
     ↦ fprintf(stderr, "success!\n", )
```

That's an error in the call to `fprintf`.

---

[1] GNU C extension.

To get rid of that comma, the `##` token paste operator has a special meaning when placed between a comma and a variable argument.[2] If you write

```
#define eprintf(format, ...) \
  fprintf (stderr, format, ##__VA_ARGS__)
```

then use the macro `eprintf` with empty variable arguments, `##` deletes the preceding comma.

```
eprintf ("success!\n")
     ↦ fprintf(stderr, "success!\n")
```

This does *not* happen if you pass an empty argument, nor does it happen if the token preceding `##` is anything other than a comma.

When the only macro parameter is a variable arguments parameter, and the macro call has no argument at all, it is not obvious whether that means an empty argument or a missing argument. Should the comma be kept, or deleted? The C standard says to keep the comma, but the preexisting GNU C extension deleted the comma. Nowadays, GNU C retains the comma when implementing a specific C standard, and deletes it otherwise.

C99 mandates that the only place the identifier `__VA_ARGS__` can appear is in the replacement list of a variadic macro. It may not be used as a macro name, macro parameter name, or within a different type of macro. It may also be forbidden in open text; the standard is ambiguous. We recommend you avoid using that name except for its special purpose.

Variadic macros where you specify the parameter name is a GNU C feature that has been supported for a long time. Standard C, as of C99, supports only the form where the parameter is called `__VA_ARGS__`. For portability to previous versions of GNU C you should use only named variable argument parameters. On the other hand, for portability to other C99 compilers, you should use only `__VA_ARGS__`.

26.5.7 Predefined Macros

Several object-like macros are predefined; you use them without supplying their definitions. Here we explain the ones user programs often need to use. Many other macro names starting with '`__`' are predefined; in general, you should not define such macro names yourself.

`__FILE__` This macro expands to the name of the current input file, in the form of a C string constant. This is the full name by which the GCC opened the file, not the short name specified in `#include` or as the input file name argument. For example, `"/usr/local/include/myheader.h"` is a possible expansion of this macro.

`__LINE__` This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

`__func__`
`__FUNCTION__`
These names are like variables that have as value a string containing the name of the current function definition. They are not really macros, but this is the best place to mention them.

---

[2] GNU C extension.

`__FUNCTION__` is the name that has been defined in GNU C since time immemorial; `__func__` is defined by the C standard. With the following conditionals, you can use whichever one is defined.

```
#if __STDC_VERSION__ < 199901L
# if __GNUC__ >= 2
#  define __func__ __FUNCTION__
# else
#  define __func__ "<unknown>"
# endif
#endif
```

`__PRETTY_FUNCTION__`

This is equivalent to `__FUNCTION__` in C, but in C++ the string includes argument type information as well. It is a GNU C extension.

Those features are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line where the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: "
                 "negative string length "
                 "in function %s "
                 "%d at %s, line %d.",
         __func__, length, __FILE__, __LINE__);
```

A `#line` directive changes `__LINE__`, and may change `__FILE__` as well. See Section 26.8 [Line Control], page 189.

`__DATE__` This macro expands to a string constant that describes the date of compilation. The string constant contains eleven characters and looks like `"Feb 12 1996"`. If the day of the month is just one digit, an extra space precedes it so that the date is always eleven characters.

If the compiler cannot determine the current date, it emits a warning messages (once per compilation) and `__DATE__` expands to `"??? ?? ????"`.

We deprecate the use of `__DATE__` for the sake of reproducible compilation.

`__TIME__` This macro expands to a string constant that describes the time of compilation. The string constant contains eight characters and looks like `"23:59:01"`.

If the compiler cannot determine the current time, it emits a warning message (once per compilation) and `__TIME__` expands to `"??:??:??"`.

We deprecate the use of `__TIME__` for the sake of reproducible compilation.

`__STDC__` In normal operation, this macro expands to the constant 1, to signify that this compiler implements ISO Standard C.

`__STDC_VERSION__`

This macro expands to the C Standard's version number, a long integer constant of the form *yyyymm*L where *yyyy* and *mm* are the year and month of the Standard version. This states which version of the C Standard the compiler implements.

The current default value is `201112L`, which signifies the C 2011 standard.

`__STDC_HOSTED__`

This macro is defined, with value 1, if the compiler's target is a *hosted environment*. A hosted environment provides the full facilities of the standard C library.

The rest of the predefined macros are GNU C extensions.

`__COUNTER__`

This macro expands to sequential integral values starting from 0. In other words, each time the program uses this macro, it generates the next successive integer. This, with the `##` preprocessing operator, provides a convenient means for macros to generate unique identifiers.

`__GNUC__`
`__GNUC_MINOR__`
`__GNUC_PATCHLEVEL__`

These macros expand to the major version, minor version, and patch level of the compiler, as integer constants. For example, GCC 3.2.1 expands `__GNUC__` to 3, `__GNUC_MINOR__` to 2, and `__GNUC_PATCHLEVEL__` to 1.

If all you need to know is whether or not your program is being compiled by GCC, or a non-GCC compiler that claims to accept the GNU C extensions, you can simply test `__GNUC__`. If you need to write code that depends on a specific version, you must check more carefully. Each change in the minor version resets the patch level to zero; each change in the major version (which happens rarely) resets the minor version and the patch level to zero. To use the predefined macros directly in the conditional, write it like this:

```
/* Test for version 3.2.0 or later. */
#if __GNUC__ > 3 || \
    (__GNUC__ == 3 && (__GNUC_MINOR__ > 2 || \
                       (__GNUC_MINOR__ == 2 && \
                        __GNUC_PATCHLEVEL__ > 0))
```

Another approach is to use the predefined macros to calculate a single number, then compare that against a threshold:

```
#define GCC_VERSION (__GNUC__ * 10000 \
                     + __GNUC_MINOR__ * 100 \
                     + __GNUC_PATCHLEVEL__)
/* ... */
/* Test for GCC > 3.2.0 */
#if GCC_VERSION > 30200
```

Many people find this form easier to understand.

`__VERSION__`

This macro expands to a string constant that describes the version of the compiler in use. You should not rely on its contents' having any particular form, but you can count on it to contain at least the release number.

`__TIMESTAMP__`

This macro expands to a string constant that describes the date and time of the last modification of the current source file. The string constant contains abbreviated day of the week, month, day of the month, time in hh:mm:ss form, and the year, in the format `"Sun Sep 16 01:03:52 1973"`. If the day of the month is less than 10, it is padded with a space on the left.

If GCC cannot determine that information date, it emits a warning message (once per compilation) and `__TIMESTAMP__` expands to `"??? ??? ?? ??:??:?? ????"`.

We deprecate the use of this macro for the sake of reproducible compilation.

26.5.8 Undefining and Redefining Macros

You can *undefine* a macro with the `#undef` directive. `#undef` takes a single argument, the name of the macro to undefine. You use the bare macro name, even if the macro is function-like. It is an error if anything appears on the line after the macro name. `#undef` has no effect if the name is not a macro.

```
#define FOO 4
x = FOO;        ↦ x = 4;
#undef FOO
x = FOO;        ↦ x = FOO;
```

Once a macro has been undefined, that identifier may be *redefined* as a macro by a subsequent `#define` directive. The new definition need not have any resemblance to the old definition.

You can define a macro again without first undefining it only if the new definition is *effectively the same* as the old one. Two macro definitions are effectively the same if:

- Both are the same type of macro (object- or function-like).
- All the tokens of the replacement list are the same.
- If there are any parameters, they are the same.
- Whitespace appears in the same places in both. It need not be exactly the same amount of whitespace, though. Remember that comments count as whitespace.

These definitions are effectively the same:

```
#define FOUR (2 + 2)
#define FOUR         (2    +    2)
#define FOUR (2 /* two */ + 2)
```

but these are not:

```
#define FOUR (2 + 2)
#define FOUR ( 2+2 )
#define FOUR (2 * 2)
#define FOUR(score,and,seven,years,ago) (2 + 2)
```

This allows two different header files to define a common macro.

You can redefine an existing macro with #define, but redefining an existing macro name with a different definition results in a warning.

26.5.9 Directives Within Macro Arguments

GNU C permits and handles preprocessing directives in the text provided as arguments for a macro. That case is undefined in the C standard. but in GNU C conditional directives in macro arguments are clear and valid.

A paradoxical case is to redefine a macro within the call to that same macro. What happens is, the new definition takes effect in time for pre-expansion of *all* the arguments, then the original definition is expanded to replace the call. Here is a pathological example:

```
#define f(x) x x
f (first f second
#undef f
#define f 2
f)
```

which expands to

```
first 2 second 2 first 2 second 2
```

with the semantics described above. We suggest you avoid writing code which does this sort of thing.

26.5.10 Macro Pitfalls

In this section we describe some special rules that apply to macros and macro expansion, and point out certain cases in which the rules have counter-intuitive consequences that you must watch out for.

26.5.10.1 Misnesting

When a macro is called with arguments, the arguments are substituted into the macro body and the result is checked, together with the rest of the input file, for more macro calls. It is possible to piece together a macro call coming partially from the macro body and partially from the arguments. For example,

```
#define twice(x) (2*(x))
#define call_with_1(x) x(1)
call_with_1 (twice)
     ↦ twice(1)
     ↦ (2*(1))
```

Macro definitions do not have to have balanced parentheses. By writing an unbalanced open parenthesis in a macro body, it is possible to create a macro call that begins inside the macro body but ends outside of it. For example,

```
#define strange(file) fprintf (file, "%s %d",
/* ... */
strange(stderr) p, 35)
     ↦ fprintf (stderr, "%s %d", p, 35)
```

The ability to piece together a macro call can be useful, but the use of unbalanced open parentheses in a macro body is just confusing, and should be avoided.

26.5.10.2 Operator Precedence Problems

You may have noticed that in most of the macro definition examples shown above, each occurrence of a macro parameter name had parentheses around it. In addition, another pair of parentheses usually surrounds the entire macro definition. Here is why it is best to write macros that way.

Suppose you define a macro as follows,

```
#define ceil_div(x, y) (x + y - 1) / y
```

whose purpose is to divide, rounding up. (One use for this operation is to compute how many `int` objects are needed to hold a certain number of `char` objects.) Then suppose it is used as follows:

```
a = ceil_div (b & c, sizeof (int));
     ↦ a = (b & c + sizeof (int) - 1) / sizeof (int);
```

This does not do what is intended. The operator-precedence rules of C make it equivalent to this:

```
a = (b & (c + sizeof (int) - 1)) / sizeof (int);
```

What we want is this:

```
a = ((b & c) + sizeof (int) - 1)) / sizeof (int);
```

Defining the macro as

```
#define ceil_div(x, y) ((x) + (y) - 1) / (y)
```

provides the desired result.

Unintended grouping can result in another way. Consider `sizeof ceil_div(1, 2)`. That has the appearance of a C expression that would compute the size of the type of `ceil_div (1, 2)`, but in fact it means something very different. Here is what it expands to:

```
sizeof ((1) + (2) - 1) / (2)
```

This would take the size of an integer and divide it by two. The precedence rules have put the division outside the `sizeof` when it was intended to be inside.

Parentheses around the entire macro definition prevent such problems. Here, then, is the recommended way to define `ceil_div`:

```
#define ceil_div(x, y) (((x) + (y) - 1) / (y))
```

26.5.10.3 Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement. Consider, for example, the following macro, that advances a pointer (the parameter `p` says where to find it) across whitespace characters:

```
#define SKIP_SPACES(p, limit)  \
{ char *lim = (limit);         \
  while (p < lim) {            \
    if (*p++ != ' ') {         \
      p--; break; }}}
```

Here backslash-newline is used to split the macro definition, which must be a single logical line, so that it resembles the way such code would be laid out if not part of a macro definition.

A call to this macro might be `SKIP_SPACES (p, lim)`. Strictly speaking, the call expands to a compound statement, which is a complete statement with no need for a semicolon to end it. However, since it looks like a function call, it minimizes confusion if you can use it like a function call, writing a semicolon afterward, as in `SKIP_SPACES (p, lim);`

This can cause trouble before `else` statements, because the semicolon is actually a null statement. Suppose you write

```
if (*p != 0)
  SKIP_SPACES (p, lim);
```

```
else /* ... */
```

The presence of two statements—the compound statement and a null statement—in between the `if` condition and the `else` makes invalid C code.

The definition of the macro `SKIP_SPACES` can be altered to solve this problem, using a `do ... while` statement. Here is how:

```
#define SKIP_SPACES(p, limit)     \
do { char *lim = (limit);         \
     while (p < lim) {            \
       if (*p++ != ' ') {         \
         p--; break; }}}          \
while (0)
```

Now `SKIP_SPACES (p, lim);` expands into

```
do { /* ... */ } while (0);
```

which is one statement. The loop executes exactly once; most compilers generate no extra code for it.

26.5.10.4 Duplication of Side Effects

Many C programs define a macro `min`, for "minimum", like this:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

When you use this macro with an argument containing a side effect, as shown here,

```
next = min (x + y, foo (z));
```

it expands as follows:

```
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

where `x + y` has been substituted for `X` and `foo (z)` for `Y`.

The function `foo` is used only once in the statement as it appears in the program, but the expression `foo (z)` has been substituted twice into the macro expansion. As a result, `foo` might be called twice when the statement is executed. If it has side effects or if it takes a long time to compute, that may be undesirable. We say that `min` is an *unsafe* macro.

The best solution to this problem is to define `min` in a way that computes the value of `foo (z)` only once. In general, that requires using `__auto_type` (see Section 20.4 [Auto Type], page 124). How to use it for this is described in the following section. See Section 26.5.10.5 [Macros and Auto Type], page 182.

Otherwise, you will need to be careful when *using* the macro `min`. For example, you can calculate the value of `foo (z)`, save it in a variable, and use that variable in `min`:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
/* ... */
{
  int tem = foo (z);
  next = min (x + y, tem);
}
```

(where we assume that `foo` returns type `int`).

When the repeated value appears as the condition of the `?:` operator and again as its *iftrue* expression, you can avoid repeated execution by omitting the *iftrue* expression, like this:

```
#define x_or_y(X, Y)  ((X) ? : (Y))
```

In GNU C, this expands to use the first macro argument's value if that isn't zero. If that's zero, it compiles the second argument and uses that value. See Section 8.4 [Conditional Expression], page 37.

26.5.10.5 Using `__auto_type` for Local Variables

The operator `__auto_type` makes it possible to define macros that can work on any data type even though they need to generate local variable declarations. See Section 20.4 [Auto Type], page 124.

For instance, here's how to define a safe "maximum" macro that operates on any arithmetic type and computes each of its arguments exactly once:

```
#define max(a,b) \
  ({ __auto_type _a = (a); \
      __auto_type _b = (b); \
    _a > _b ? _a : _b; })
```

The '`({ ... })`' notation produces *statement expression*—a statement that can be used as an expression (see Section 19.15 [Statement Exprs], page 118). Its value is the value of its last statement. This permits us to define local variables and store each argument value into one.

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Underscore followed by a lower case letter won't be predefined by the system in any way.

26.5.10.6 Self-Referential Macros

A *self-referential* macro is one whose name appears in its definition. Recall that all macro definitions are rescanned for more macros to replace. If the self-reference were considered a use of the macro, it would produce an infinitely large expansion. To prevent this, the self-reference is not considered a macro call: preprocessing leaves it unchanged. Consider an example:

```
#define foo (4 + foo)
```

where `foo` is also a variable in your program.

Following the ordinary rules, each reference to `foo` will expand into `(4 + foo)`; then this will be rescanned and will expand into `(4 + (4 + foo))`; and so on until the computer runs out of memory.

The self-reference rule cuts this process short after one step, at `(4 + foo)`. Therefore, this macro definition has the possibly useful effect of causing the program to add 4 to the value of `foo` wherever `foo` is referred to.

In most cases, it is a bad idea to take advantage of this feature. A person reading the program who sees that `foo` is a variable will not expect that it is a macro as well. The reader will come across the identifier `foo` in the program and think its value should be that of the variable `foo`, whereas in fact the value is four greater.

It is useful to make a macro definition that expands to the macro name itself. If you write

```
#define EPERM EPERM
```

then the macro `EPERM` expands to `EPERM`. Effectively, preprocessing leaves it unchanged in the source code. You can tell that it's a macro with `#ifdef`. You might do this if you want to define numeric constants with an `enum`, but have `#ifdef` be true for each constant.

If a macro `x` expands to use a macro `y`, and the expansion of `y` refers to the macro `x`, that is an *indirect self-reference* of `x`. `x` is not expanded in this case either. Thus, if we have

```
#define x (4 + y)
#define y (2 * x)
```

then `x` and `y` expand as follows:

```
x     ↦ (4 + y)
      ↦ (4 + (2 * x))

y     ↦ (2 * x)
      ↦ (2 * (4 + y))
```

Each macro is expanded when it appears in the definition of the other macro, but not when it indirectly appears in its own definition.

26.5.10.7 Argument Prescan

Macro arguments are completely macro-expanded before they are substituted into a macro body, unless they are stringified or pasted with other tokens. After substitution, the entire macro body, including the substituted arguments, is scanned again for macros to be expanded. The result is that the arguments are scanned *twice* to expand macro calls in them.

Most of the time, this has no effect. If the argument contained any macro calls, they were expanded during the first scan. The result therefore contains no macro calls, so the second scan does not change it. If the argument were substituted as given, with no prescan, the single remaining scan would find the same macro calls and produce the same results.

You might expect the double scan to change the results when a self-referential macro is used in an argument of another macro (see Section 26.5.10.6 [Self-Referential Macros], page 182): the self-referential macro would be expanded once in the first scan, and a second time in the second scan. However, this is not what happens. The self-references that do not expand in the first scan are marked so that they will not expand in the second scan either.

You might wonder, "Why mention the prescan, if it makes no difference? And why not skip it and make preprocessing go faster?" The answer is that the prescan does make a difference in three special cases:

- Nested calls to a macro.

  We say that *nested* calls to a macro occur when a macro's argument contains a call to that very macro. For example, if `f` is a macro that expects one argument, `f (f (1))` is a nested pair of calls to `f`. The desired expansion is made by expanding `f (1)` and substituting that into the definition of `f`. The prescan causes the expected result to happen. Without the prescan, `f (1)` itself would be substituted as an argument, and the inner use of `f` would appear during the main scan as an indirect self-reference and would not be expanded.

- Macros that call other macros that stringify or concatenate.

  If an argument is stringified or concatenated, the prescan does not occur. If you *want* to expand a macro, then stringify or concatenate its expansion, you can do that by causing one macro to call another macro that does the stringification or concatenation. For instance, if you have

  ```
  #define AFTERX(x) X_ ## x
  #define XAFTERX(x) AFTERX(x)
  #define TABLESIZE 1024
  #define BUFSIZE TABLESIZE
  ```

  then `AFTERX(BUFSIZE)` expands to `X_BUFSIZE`, and `XAFTERX(BUFSIZE)` expands to `X_1024`. (Not to `X_TABLESIZE`. Prescan always does a complete expansion.)

- Macros used in arguments, whose expansions contain unshielded commas.

  This can cause a macro expanded on the second scan to be called with the wrong number of arguments. Here is an example:

  ```
  #define foo  a,b
  #define bar(x) lose(x)
  #define lose(x) (1 + (x))
  ```

  We would like `bar(foo)` to turn into `(1 + (foo))`, which would then turn into `(1 + (a,b))`. Instead, `bar(foo)` expands into `lose(a,b)`, which gives an error because `lose` requires a single argument. In this case, the problem is easily solved by the same parentheses that ought to be used to prevent misnesting of arithmetic operations:

  ```
  #define foo (a,b)
  ```

  or

  ```
  #define bar(x) lose((x))
  ```

  The extra pair of parentheses prevents the comma in `foo`'s definition from being interpreted as an argument separator.

26.6 Conditionals

A *conditional* is a preprocessing directive that controls whether or not to include a chunk of code in the final token stream that is compiled. Preprocessing conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both together using the special `defined` preprocessing operator.

A preprocessing conditional in C resembles in some ways an `if` statement in C, but it is important to understand the difference between them. The condition in an `if` statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessing conditional directive is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Sometimes this distinction makes no practical difference. GCC and other modern compilers often do test `if` statements when a program is compiled, if their conditions are known not to vary at run time, and eliminate code that can never be executed. If you can count on your compiler to do this, you may find that your program is more readable if you use `if` statements with constant conditions (perhaps determined by macros). Of course, you can only use this to exclude code, not type definitions or other preprocessing directives, and you can only do it if the file remains syntactically valid when that code is not used.

26.6.1 Uses of Conditional Directives

There are three usual reasons to use a preprocessing conditional.

- A program may need to use different code depending on the machine or operating system it is to run on. In some cases the code for one operating system may be erroneous on another operating system; for example, it might refer to data types or constants that do not exist on the other system. When this happens, it is not enough to avoid executing the invalid code. Its mere presence will cause the compiler to reject the program. With a preprocessing conditional, the offending code can be effectively excised from the program when it is not valid.
- You may want to be able to compile the same source file into two different programs. One version might make frequent time-consuming consistency checks on its intermediate data, or print the values of those data for debugging, and the other not.
- A conditional whose condition is always false is one way to exclude code from the program but keep it as a sort of comment for future reference.

Simple programs that do not need system-specific logic or complex debugging hooks generally will not need to use preprocessing conditionals.

26.6.2 Syntax of Preprocessing Conditionals

A preprocessing conditional begins with a *conditional directive*: `#if`, `#ifdef` or `#ifndef`.

26.6.2.1 The `#ifdef` directive

The simplest sort of conditional is

```
#ifdef MACRO

controlled text

#endif /* MACRO */
```

This block is called a *conditional group*. The body, *controlled text*, will be included in compilation if and only if *MACRO* is defined. We say that the conditional *succeeds* if *MACRO* is defined, *fails* if it is not.

The *controlled text* inside a conditional can include preprocessing directives. They are executed only if the conditional succeeds. You can nest conditional groups inside other conditional groups, but they must be completely nested. In other words, `#endif` always matches the nearest `#ifdef` (or `#ifndef`, or `#if`). Also, you cannot start a conditional group in one file and end it in another.

Even if a conditional fails, the *controlled text* inside it is still run through initial transformations and tokenization. Therefore, it must all be lexically valid C. Normally the only way this matters is that all comments and string literals inside a failing conditional group must still be properly ended.

The comment following the `#endif` is not required, but it is a good practice if there is a lot of *controlled text*, because it helps people match the `#endif` to the corresponding `#ifdef`.

Older programs sometimes put *macro* directly after the `#endif` without enclosing it in a comment. This is invalid code according to the C standard, but it only causes a warning in GNU C. It never affects which `#ifndef` the `#endif` matches.

Sometimes you wish to use some code if a macro is *not* defined. You can do this by writing `#ifndef` instead of `#ifdef`. One common use of `#ifndef` is to include code only the first time a header file is included. See Section 26.4.4 [Once-Only Headers], page 165.

Macro definitions can vary between compilations for several reasons. Here are some samples.

- Some macros are predefined on each kind of machine (see Section "System-specific Predefined Macros" in *The C Preprocessor*). This allows you to provide code specially tuned for a particular machine.
- System header files define more macros, associated with the features they implement. You can test these macros with conditionals to avoid using a system feature on a machine where it is not implemented.
- Macros can be defined or undefined with the `-D` and `-U` command-line options when you compile the program. You can arrange to compile the same source file into two different programs by choosing a macro name to specify which program you want, writing conditionals to test whether or how this macro is defined, and then controlling the state of the macro with command-line options, perhaps set in the file `Makefile`. See Section "Invoking GCC" in *Using the GNU Compiler Collection*.
- Your program might have a special header file (often called `config.h`) that is adjusted when the program is compiled. It can define or not define macros depending on the features of the system and the desired capabilities of the program. The adjustment can be automated by a tool such as `autoconf`, or done by hand.

26.6.2.2 The `#if` directive

The `#if` directive allows you to test the value of an integer arithmetic expression, rather than the mere existence of one macro. Its syntax is

```
#if expression

controlled text

#endif /* expression */
```

expression is a C expression of integer type, subject to stringent restrictions so its value can be computed at compile time. It may contain

- Integer constants.
- Character constants, which are interpreted as they would be in normal code.
- Arithmetic operators for addition, subtraction, multiplication, division, bitwise operations, shifts, comparisons, and logical operations (`&&` and `||`). The latter two obey the usual short-circuiting rules of standard C.

- Macros. All macros in the expression are expanded before actual computation of the expression's value begins.
- Uses of the `defined` preprocessing operator, which lets you check whether macros are defined in the middle of an `#if`.
- Identifiers that are not macros, which are all considered to be the number zero. This allows you to write `#if MACRO` instead of `#ifdef MACRO`, if you know that MACRO, when defined, will always have a nonzero value. Function-like macros used without their function call parentheses are also treated as zero.

  In some contexts this shortcut is undesirable. The `-Wundef` requests warnings for any identifier in an `#if` that is not defined as a macro.

Preprocessing does not know anything about the data types of C. Therefore, `sizeof` operators are not recognized in `#if`; `sizeof` is simply an identifier, and if it is not a macro, it stands for zero. This is likely to make the expression invalid. Preprocessing does not recognize `enum` constants; they too are simply identifiers, so if they are not macros, they stand for zero.

Preprocessing calculates the value of *expression*, and carries out all calculations in the widest integer type known to the compiler; on most machines supported by GNU C this is 64 bits. This is not the same rule as the compiler uses to calculate the value of a constant expression, and may give different results in some cases. If the value comes out to be nonzero, the `#if` succeeds and the *controlled text* is compiled; otherwise it is skipped.

26.6.2.3 The `defined` test

The preprocessing operator `defined` is used in `#if` and `#elif` expressions to test whether a certain name is defined as a macro. `defined` *`name`* and `defined (`*`name`*`)` are both expressions whose value is 1 if *name* is defined as a macro at the current point in the program, and 0 otherwise. Thus, `#if defined MACRO` is precisely equivalent to `#ifdef MACRO`.

`defined` is useful when you wish to test more than one macro for existence at once. For example,

```
#if defined (__arm__) || defined (__PPC__)
```

would succeed if either of the names `__arm__` or `__PPC__` is defined as a macro—in other words, when compiling for ARM processors or PowerPC processors.

Conditionals written like this:

```
#if defined BUFSIZE && BUFSIZE >= 1024
```

can generally be simplified to just `#if BUFSIZE >= 1024`, since if `BUFSIZE` is not defined, it will be interpreted as having the value zero.

In GCC, you can include `defined` as part of another macro definition, like this:

```
#define MACRO_DEFINED(X) defined X

#if MACRO_DEFINED(BUFSIZE)
```

which would expand the `#if` expression to:

```
#if defined BUFSIZE
```

Generating `defined` in this way is a GNU C extension.

26.6.2.4 The `#else` directive

The `#else` directive can be added to a conditional to provide alternative text to be used if the condition fails. This is what it looks like:

```
#if expression
text-if-true
#else /* Not expression */
text-if-false
#endif /* Not expression */
```

If *expression* is nonzero, the *text-if-true* is included and the *text-if-false* is skipped. If *expression* is zero, the opposite happens.

You can use `#else` with `#ifdef` and `#ifndef`, too.

26.6.2.5 The `#elif` directive

One common case of nested conditionals is used to check for more than two possible alternatives. For example, you might have

```
#if X == 1
/* ... */
#else /* X != 1 */
#if X == 2
/* ... */
#else /* X != 2 */
/* ... */
#endif /* X != 2 */
#endif /* X != 1 */
```

Another conditional directive, `#elif`, allows this to be abbreviated as follows:

```
#if X == 1
/* ... */
#elif X == 2
/* ... */
#else /* X != 2 and X != 1*/
/* ... */
#endif /* X != 2 and X != 1*/
```

`#elif` stands for “else if”. Like `#else`, it goes in the middle of a conditional group and subdivides it; it does not require a matching `#endif` of its own. Like `#if`, the `#elif` directive includes an expression to be tested. The text following the `#elif` is processed only if the original `#if`-condition failed and the `#elif` condition succeeds.

More than one `#elif` can go in the same conditional group. Then the text after each `#elif` is processed only if the `#elif` condition succeeds after the original `#if` and all previous `#elif` directives within it have failed.

`#else` is allowed after any number of `#elif` directives, but `#elif` may not follow `#else`.

26.6.3 Deleted Code

If you replace or delete a part of the program but want to keep the old code in the file for future reference, commenting it out is not so straightforward in C. Block comments do not nest, so the first comment inside the old code will end the commenting-out. The probable result is a flood of syntax errors.

One way to avoid this problem is to use an always-false conditional instead. For instance, put `#if 0` before the deleted code and `#endif` after it. This works even if the code being turned off contains conditionals, but they must be entire conditionals (balanced `#if` and `#endif`).

Some people use `#ifdef notdef` instead. This is risky, because `notdef` might be accidentally defined as a macro, and then the conditional would succeed. `#if 0` can be counted on to fail.

Do not use `#if 0` around text that is not C code. Use a real comment, instead. The interior of `#if 0` must consist of complete tokens; in particular, single-quote characters must balance. Comments often contain unbalanced single-quote characters (known in English as apostrophes). These confuse `#if 0`. They don't confuse '`/*`'.

26.7 Diagnostics

The directive `#error` reports a fatal error. The tokens forming the rest of the line following `#error` are used as the error message.

The usual place to use `#error` is inside a conditional that detects a combination of parameters that you know the program does not properly support. For example,

```
#if !defined(UNALIGNED_INT_ASM_OP) && defined(DWARF2_DEBUGGING_INFO)
#error "DWARF2_DEBUGGING_INFO requires UNALIGNED_INT_ASM_OP."
#endif
```

The directive `#warning` is like `#error`, but it reports a warning instead of an error. The tokens following `#warning` are used as the warning message.

You might use `#warning` in obsolete header files, with a message saying which header file to use instead.

Neither `#error` nor `#warning` macro-expands its argument. Internal whitespace sequences are each replaced with a single space. The line must consist of complete tokens. It is wisest to make the argument of these directives be a single string constant; this avoids problems with apostrophes and the like.

26.8 Line Control

Due to C's widespread availability and low-level nature, it is often used as the target language for translation of other languages, or for the output of lexical analyzers and parsers (e.g., lex/flex and yacc/bison). Line control enables the user to track diagnostics back to the location in the original language.

The C compiler knows the location in the source file where each token came from: file name, starting line and column, and final line and column. (Column numbers are used only for error messages.)

When a program generates C source code, as the Bison parser generator does, often it copies some of that C code from another file. For instance parts of the output from Bison are generated from scratch or come from a standard parser file, but Bison copies the rest from Bison's input file. Errors in that code, at compile time or run time, should refer to that file, which is the real source code. To make that happen, Bison generates line-control directives that the C compiler understands.

`#line` is a directive that specifies the original line number and source file name for subsequent code. `#line` has three variants:

`#line` *`linenum`*

linenum is a non-negative decimal integer constant. It specifies the line number that should be reported for the following line of input. Subsequent lines are counted from *linenum*.

`#line` *`linenum filename`*

linenum is the same as for the first form, and has the same effect. In addition, *filename* is a string constant that specifies the source file name. Subsequent source lines are recorded as coming from that file, until something else happens to change that. *filename* is interpreted according to the normal rules for a string constant. Backslash escapes are interpreted, in contrast to `#include`.

`#line` *`anything else`*

anything else is checked for macro calls, which are expanded. The result should match one of the above two forms.

`#line` directives alter the results of the `__FILE__` and `__LINE__` symbols from that point on. See Section 26.5.7 [Predefined Macros], page 175.

26.9 Null Directive

The *null directive* consists of a `#` followed by a newline, with only whitespace and comments in between. It has no effect on the output of the compiler.

27 Integers in Depth

This chapter explains the machine-level details of integer types: how they are represented as bits in memory, and the range of possible values for each integer type.

27.1 Integer Representations

Modern computers store integer values as binary (base-2) numbers that occupy a single unit of storage, typically either as an 8-bit `char`, a 16-bit `short int`, a 32-bit `int`, or possibly, a 64-bit `long long int`. Whether a `long int` is a 32-bit or a 64-bit value is system dependent.[1]

The macro `CHAR_BIT`, defined in `limits.h`, gives the number of bits in type `char`. On any real operating system, the value is 8.

The fixed sizes of numeric types necessarily limits their *range of values*, and the particular encoding of integers decides what that range is.

For unsigned integers, the entire space is used to represent a nonnegative value. Signed integers are stored using *two's-complement representation*: a signed integer with n bits has a range from $-2^{(n-1)}$ to -1 to 0 to 1 to $+2^{(n-1)} - 1$, inclusive. The leftmost, or high-order, bit is called the *sign bit*.

In two's-complement representation, there is only one value that means zero, and the most negative number lacks a positive counterpart. As a result, negating that number causes overflow; in practice, its result is that number back again. We will revisit that peculiarity shortly.

For example, a two's-complement signed 8-bit integer can represent all decimal numbers from -128 to +127. Negating -128 ought to give +128, but that value won't fit in 8 bits, so the operation yields -128.

Decades ago, there were computers that used other representations for signed integers, but they are long gone and not worth any effort to support. The GNU C language does not support them.

When an arithmetic operation produces a value that is too big to represent, the operation is said to *overflow*. In C, integer overflow does not interrupt the control flow or signal an error. What it does depends on signedness.

For unsigned arithmetic, the result of an operation that overflows is the n low-order bits of the correct value. If the correct value is representable in n bits, that is always the result; thus we often say that "integer arithmetic is exact," omitting the crucial qualifying phrase "as long as the exact result is representable."

In principle, a C program should be written so that overflow never occurs for signed integers, but in GNU C you can specify various ways of handling such overflow (see Section 6.3 [Integer Overflow], page 23).

Integer representations are best understood by looking at a table for a tiny integer size; here are the possible values for an integer with three bits:

---

[1] In theory, any of these types could have some other size, bit it's not worth even a minute to cater to that possibility. It never happens on GNU/Linux.

Unsigned	Signed	Bits	2s Complement
0	0	000	000 (0)
1	1	001	111 (-1)
2	2	010	110 (-2)
3	3	011	101 (-3)
4	-4	100	100 (-4)
5	-3	101	011 (3)
6	-2	110	010 (2)
7	-1	111	001 (1)

The parenthesized decimal numbers in the last column represent the signed meanings of the two's-complement of the line's value. Recall that, in two's-complement encoding, the high-order bit is 0 when the number is nonnegative.

We can now understand the peculiar behavior of negation of the most negative two's-complement integer: start with 0b100, invert the bits to get 0b011, and add 1: we get 0b100, the value we started with.

We can also see overflow behavior in two's-complement:

```
3 + 1 = 0b011 + 0b001 = 0b100 = (-4)
3 + 2 = 0b011 + 0b010 = 0b101 = (-3)
3 + 3 = 0b011 + 0b011 = 0b110 = (-2)
```

A sum of two nonnegative signed values that overflows has a 1 in the sign bit, so the exact positive result is truncated to a negative value.

27.2 Maximum and Minimum Values

For each primitive integer type, there is a standard macro defined in `limits.h` that gives the largest value that type can hold. For instance, for type `int`, the maximum value is `INT_MAX`. On a 32-bit computer, that is equal to 2,147,483,647. The maximum value for `unsigned int` is `UINT_MAX`, which on a 32-bit computer is equal to 4,294,967,295. Likewise, there are `SHRT_MAX`, `LONG_MAX`, and `LLONG_MAX`, and corresponding unsigned limits `USHRT_MAX`, `ULONG_MAX`, and `ULLONG_MAX`.

Since there are three ways to specify a `char` type, there are also three limits: `CHAR_MAX`, `SCHAR_MAX`, and `UCHAR_MAX`.

For each type that is or might be signed, there is another symbol that gives the minimum value it can hold. (Just replace `MAX` with `MIN` in the names listed above.) There is no minimum limit symbol for types specified with `unsigned` because the minimum for them is universally zero.

`INT_MIN` is not the negative of `INT_MAX`. In two's-complement representation, the most negative number is 1 less than the negative of the most positive number. Thus, `INT_MIN` on a 32-bit computer has the value −2,147,483,648. You can't actually write the value that way in C, since it would overflow. That's a good reason to use `INT_MIN` to specify that value. Its definition is written to avoid overflow.

This is part of the GNU C Intro and Reference Manual and covered by its license.

28 Floating Point in Depth

28.1 Floating-Point Representations

Storing numbers as *floating point* allows representation of numbers with fractional values, in a range larger than that of hardware integers. A floating-point number consists of a sign bit, a *significand* (also called the *mantissa*), and a power of a fixed base. GNU C uses the floating-point representations specified by the *IEEE 754-2008 Standard for Floating-Point Arithmetic*.

The IEEE 754-2008 specification defines basic binary floating-point formats of five different sizes: 16-bit, 32-bit, 64-bit, 128-bit, and 256-bit. The formats of 32, 64, and 128 bits are used for the standard C types `float`, `double`, and `long double`. GNU C supports the 16-bit floating point type `_Float16` on some platforms, but does not support the 256-bit floating point type.

Each of the formats encodes the floating-point number as a sign bit. After this comes an exponent that specifies a power of 2 (with a fixed offset). Then comes the significand.

The first bit of the significand, before the binary point, is always 1, so there is no need to store it in memory. It is called the *hidden bit* because it doesn't appear in the floating-point number as used in the computer itself.

All of those floating-point formats are sign-magnitude representations, so +0 and −0 are different values.

Besides the IEEE 754 format 128-bit float, GNU C also offers a format consisting of a pair of 64-bit floating point numbers. This lacks the full exponent range of the IEEE 128-bit format, but is useful when the underlying hardware platform does not support that.

28.2 Floating-Point Type Specifications

The standard library header file `float.h` defines a number of constants that describe the platform's implementation of floating-point types `float`, `double` and `long double`. They include:

`FLT_MIN`
`DBL_MIN`
`LDBL_MIN`
Defines the minimum normalized positive floating-point values that can be represented with the type.

`FLT_HAS_SUBNORM`
`DBL_HAS_SUBNORM`
`LDBL_HAS_SUBNORM`
Defines if the floating-point type supports subnormal (or "denormalized") numbers or not (see [subnormal numbers], page 194).

`FLT_TRUE_MIN`
`DBL_TRUE_MIN`
`LDBL_TRUE_MIN`
Defines the minimum positive values (including subnormal values) that can be represented with the type.

`FLT_MAX`
`DBL_MAX`
`LDBL_MAX` Defines the largest values that can be represented with the type.

`FLT_DECIMAL_DIG`
`DBL_DECIMAL_DIG`
`LDBL_DECIMAL_DIG`
Defines the number of decimal digits `n` such that any floating-point number that can be represented in the type can be rounded to a floating-point number with `n` decimal digits, and back again, without losing any precision of the value.

28.3 Special Floating-Point Values

IEEE floating point provides for special values that are not ordinary numbers.

infinities `+Infinity` and `-Infinity` are two different infinite values, one positive and one negative. These result from operations such as `1 / 0`, `Infinity + Infinity`, `Infinity * Infinity`, and `Infinity +` ***`finite`***, and also from a result that is finite, but larger than the most positive possible value or smaller than the most negative possible value.

See Section 28.13 [Handling Infinity], page 201, for more about working with infinities.

NaNs (not a number)
There are two special values, called Not-a-Number (NaN): a quiet NaN (QNaN), and a signaling NaN (SNaN).

A QNaN is produced by operations for which the value is undefined in real arithmetic, such as `0 / 0`, `sqrt (-1)`, `Infinity - Infinity`, and any basic operation in which an operand is a QNaN.

The signaling NaN is intended for initializing otherwise-unassigned storage, and the goal is that unlike a QNaN, an SNaN *does* cause an interrupt that can be caught by a software handler, diagnosed, and reported. In practice, little use has been made of signaling NaNs, because the most common CPUs in desktop and portable computers fail to implement the full IEEE 754 Standard, and supply only one kind of NaN, the quiet one. Also, programming-language standards have taken decades to catch up to the IEEE 754 standard, and implementations of those language standards make an additional delay before programmers become willing to use these features.

To enable support for signaling NaNs, use the GCC command-line option `-fsignaling-nans`, but this is an experimental feature and may not work as expected in every situation.

A NaN has a sign bit, but its value means nothing.

See Section 28.14 [Handling NaN], page 202, for more about working with NaNs.

subnormal numbers
It can happen that a computed floating-point value is too small to represent, such as when two tiny numbers are multiplied. The result is then said to *underflow*. The traditional behavior before the IEEE 754 Standard was to use

zero as the result, and possibly to report the underflow in some sort of program output.

The IEEE 754 Standard is vague about whether rounding happens before detection of floating underflow and overflow, or after, and CPU designers may choose either.

However, the Standard does something unusual compared to earlier designs, and that is that when the result is smaller than the smallest *normalized* representable value (i.e., one in which the leading significand bit is `1`), the normalization requirement is relaxed, leading zero bits are permitted, and precision is gradually lost until there are no more bits in the significand. That phenomenon is called *gradual underflow*, and it serves important numerical purposes, although it does reduce the precision of the final result. Some floating-point designs allow you to choose at compile time, or even at run time, whether underflows are gradual, or are flushed abruptly to zero. Numbers that have entered the region of gradual underflow are called *subnormal*.

You can use the library functions `fesetround` and `fegetround` to set and get the rounding mode. Rounding modes are defined (if supported by the platform) in `fenv.h` as: `FE_UPWARD` to round toward positive infinity; `FE_DOWNWARD` to round toward negative infinity; `FE_TOWARDZERO` to round toward zero; and `FE_TONEAREST` to round to the nearest representable value, the default mode. It is best to use `FE_TONEAREST` except when there is a special need for some other mode.

28.4 Invalid Optimizations

Signed zeros, Infinity, and NaN invalidate some optimizations by programmers and compilers that might otherwise have seemed obvious:

- `x + 0` and `x - 0` are not the same as `x` when `x` is zero, because the result depends on the rounding rule. See Section 28.7 [Rounding], page 196, for more about rounding rules.
- `x * 0.0` is not the same as `0.0` when `x` is Infinity, a NaN, or negative zero.
- `x / x` is not the same as `1.0` when `x` is Infinity, a NaN, or zero.
- `(x - y)` is not the same as `-(y - x)` because when the operands are finite and equal, one evaluates to `+0` and the other to `-0`.
- `x - x` is not the same as `0.0` when x is Infinity or a NaN.
- `x == x` and `x != x` are not equivalent to `1` and `0` when x is a NaN.
- `x < y` and `isless (x, y)` are not equivalent, because the first sets a sticky exception flag (see Section 28.5 [Exception Flags], page 195) when an operand is a NaN, whereas the second does not affect that flag. The same holds for the other `isxxx` functions that are companions to relational operators. See Section "FP Comparison Functions" in *The GNU C Library Reference Manual*.

The `-funsafe-math-optimizations` option enables these optimizations.

28.5 Floating Arithmetic Exception Flags

Sticky exception flags record the occurrence of particular conditions: once set, they remain set until the program explicitly clears them.

The conditions include *invalid operand*, *division-by_zero*, *inexact result* (i.e., one that required rounding), *underflow*, and *overflow*. Some extended floating-point designs offer several additional exception flags. The functions `feclearexcept`, `feraiseexcept`, `fetestexcept`, `fegetexceptflags`, and `fesetexceptflags` provide a standardized interface to those flags. See Section "Status bit operations" in *The GNU C Library Reference Manual*.

One important use of those flags is to do a computation that is normally expected to be exact in floating-point arithmetic, but occasionally might not be, in which case, corrective action is needed. You can clear the *inexact result* flag with a call to `feclearexcept (FE_INEXACT)`, do the computation, and then test the flag with `fetestexcept (FE_INEXACT)`; the result of that call is 0 if the flag is not set (there was no rounding), and 1 when there was rounding (which, we presume, implies the program has to correct for that).

28.6 Exact Floating-Point Arithmetic

As long as the numbers are exactly representable (fractions whose denominator is a power of 2), and intermediate results do not require rounding, then floating-point arithmetic is *exact*. It is easy to predict how many digits are needed for the results of arithmetic operations:

- addition and subtraction of two *n*-digit values with the *same* exponent require at most `n + 1` digits, but when the exponents differ, many more digits may be needed;
- multiplication of two *n*-digit values requires exactly 2 *n* digits;
- although integer division produces a quotient and a remainder of no more than *n*-digits, floating-point remainder and square root may require an unbounded number of digits, and the quotient can need many more digits than can be stored.

Whenever a result requires more than *n* digits, rounding is needed.

28.7 Rounding

When floating-point arithmetic produces a result that can't fit exactly in the significand of the type that's in use, it has to *round* the value. The basic arithmetic operations—addition, subtraction, multiplication, division, and square root—always produce a result that is equivalent to the exact, possibly infinite-precision result rounded to storage precision according to the current rounding rule.

Rounding sets the `FE_INEXACT` exception flag (see Section 28.5 [Exception Flags], page 195). This enables programs to determine that rounding has occurred.

Rounding consists of adjusting the exponent to bring the significand back to the required base-point alignment, then applying the current *rounding rule* to squeeze the significand into the fixed available size.

The current rule is selected at run time from four options. Here they are:

* *round-to-nearest*, with ties rounded to an even integer;
* *round-up*, towards `+Infinity`;
* *round-down*, towards `-Infinity`;
* *round-towards-zero*.

Under those four rounding rules, a decimal value `-1.2345` that is to be rounded to a four-digit result would become `-1.234`, `-1.234`, `-1.235`, and `-1.234`, respectively.

The default rounding rule is *round-to-nearest*, because that has the least bias, and produces the lowest average error. When the true result lies exactly halfway between two representable machine numbers, the result is rounded to the one that ends with an even digit.

The *round-towards-zero* rule was common on many early computer designs, because it is the easiest to implement: it just requires silent truncation of all extra bits.

The two other rules, *round-up* and *round-down*, are essential for implementing *interval arithmetic*, whereby each arithmetic operation produces lower and upper bounds that are guaranteed to enclose the exact result.

See Section 28.17 [Rounding Control], page 204, for details on getting and setting the current rounding mode.

28.8 Rounding Issues

The default IEEE 754 rounding mode minimizes errors, and most normal computations should not suffer any serious accumulation of errors from rounding.

Of course, you can contrive examples where that is not so. Here is one: iterate a square root, then attempt to recover the original value by repeated squaring.

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

int main (void)
{
  double x = 100.0;
  double y;
  int n, k;
  for (n = 10; n <= 100; n += 10)
    {
      y = x;
      for (k = 0; k < n; ++k) y = sqrt (y);
      for (k = 0; k < n; ++k) y *= y;
      printf ("n = %3d; x = %.0f\ty = %.6f\n", n, x, y);
    }
  return 0;
}
```

Here is the output:

```
n =  10; x = 100        y = 100.000000
n =  20; x = 100        y = 100.000000
n =  30; x = 100        y = 99.999977
n =  40; x = 100        y = 99.981025
n =  50; x = 100        y = 90.017127
n =  60; x = 100        y = 1.000000
n =  70; x = 100        y = 1.000000
n =  80; x = 100        y = 1.000000
n =  90; x = 100        y = 1.000000
n = 100; x = 100        y = 1.000000
```

After 50 iterations, `y` has barely one correct digit, and soon after, there are no correct digits.

28.9 Significance Loss

A much more serious source of error in floating-point computation is *significance loss* from subtraction of nearly equal values. This means that the number of bits in the significand of the result is fewer than the size of the value would permit. If the values being subtracted are close enough, but still not equal, a *single subtraction* can wipe out all correct digits, possibly contaminating all future computations.

Floating-point calculations can sometimes be carefully designed so that significance loss is not possible, such as summing a series where all terms have the same sign. For example, the Taylor series expansions of the trigonometric and hyperbolic sines have terms of identical magnitude, of the general form `x**(2*n + 1) / (2*n + 1)!`. However, those in the trigonometric sine series alternate in sign, while those in the hyperbolic sine series are all positive. Here is the output of two small programs that sum *k* terms of the series for `sin (x)`, and compare the computed sums with known-to-be-accurate library functions:

```
x = 10      k = 51
s (x)   = -0.544_021_110_889_270
sin (x) = -0.544_021_110_889_370

x = 20      k = 81
s (x)   = 0.912_945_250_749_573
sin (x) = 0.912_945_250_727_628

x = 30      k = 109
s (x)   = -0.987_813_746_058_855
sin (x) = -0.988_031_624_092_862

x = 40      k = 137
s (x)   = 0.617_400_430_980_474
sin (x) = 0.745_113_160_479_349

x = 50      k = 159
s (x)   = 57_105.187_673_745_720_532
sin (x) = -0.262_374_853_703_929

// sinh(x) series summation with positive signs
// with k terms needed to converge to machine precision

x = 10      k = 47
t (x)    = 1.101_323_287_470_340e+04
sinh (x) = 1.101_323_287_470_339e+04

x = 20      k = 69
t (x)    = 2.425_825_977_048_951e+08
sinh (x) = 2.425_825_977_048_951e+08
```

```
x = 30      k = 87
t (x)    = 5.343_237_290_762_229e+12
sinh (x) = 5.343_237_290_762_231e+12

x = 40      k = 105
t (x)    = 1.176_926_334_185_100e+17
sinh (x) = 1.176_926_334_185_100e+17

x = 50      k = 121
t (x)    = 2.592_352_764_293_534e+21
sinh (x) = 2.592_352_764_293_536e+21
```

We have added underscores to the numbers to enhance readability.

The `sinh (x)` series with positive terms can be summed to high accuracy. By contrast, the series for `sin (x)` suffers increasing significance loss, so that when $x = 30$ only two correct digits remain. Soon after, all digits are wrong, and the answers are complete nonsense.

An important skill in numerical programming is to recognize when significance loss is likely to contaminate a computation, and revise the algorithm to reduce this problem. Sometimes, the only practical way to do so is to compute in higher intermediate precision, which is why the extended types like `long double` are important.

28.10 Fused Multiply-Add

In 1990, when IBM introduced the POWER architecture, the CPU provided a previously unknown instruction, the *fused multiply-add* (FMA). It computes the value `x * y + z` with an **exact** double-length product, followed by an addition with a *single* rounding. Numerical computation often needs pairs of multiply and add operations, for which the FMA is well-suited.

On the POWER architecture, there are two dedicated registers that hold permanent values of `0.0` and `1.0`, and the normal *multiply* and *add* instructions are just wrappers around the FMA that compute `x * y + 0.0` and `x * 1.0 + z`, respectively.

In the early days, it appeared that the main benefit of the FMA was getting two floating-point operations for the price of one, almost doubling the performance of some algorithms. However, numerical analysts have since shown numerous uses of the FMA for significantly enhancing accuracy. We discuss one of the most important ones in the next section.

A few other architectures have since included the FMA, and most provide variants for the related operations `x * y - z` (FMS), `-x * y + z` (FNMA), and `-x * y - z` (FNMS).

The functions `fmaf`, `fma`, and `fmal` implement fused multiply-add for the `float`, `double`, and `long double` data types. Correct implementation of the FMA in software is difficult, and some systems that appear to provide those functions do not satisfy the single-rounding requirement. That situation should change as more programmers use the FMA operation, and more CPUs provide FMA in hardware.

Use the `-ffp-contract=fast` option to allow generation of FMA instructions, or `-ffp-contract=off` to disallow it.

28.11 Error Recovery

When two numbers are combined by one of the four basic operations, the result often requires rounding to storage precision. For accurate computation, one would like to be able to recover that rounding error. With historical floating-point designs, it was difficult to do so portably, but now that IEEE 754 arithmetic is almost universal, the job is much easier.

For addition with the default *round-to-nearest* rounding mode, we can determine the error in a sum like this:

```
volatile double err, sum, tmp, x, y;

if (fabs (x) >= fabs (y))
  {
    sum = x + y;
    tmp = sum - x;
    err = y - tmp;
  }
else /* fabs (x) < fabs (y) */
  {
    sum = x + y;
    tmp = sum - y;
    err = x - tmp;
  }
```

Now, `x + y` is *exactly* represented by `sum + err`. This basic operation, which has come to be called *twosum* in the numerical-analysis literature, is the first key to tracking, and accounting for, rounding error.

To determine the error in subtraction, just swap the `+` and `-` operators.

We used the `volatile` qualifier (see Section 21.2 [volatile], page 129) in the declaration of the variables, which forces the compiler to store and retrieve them from memory, and prevents the compiler from optimizing `err = y - ((x + y) - x)` into `err = 0`.

For multiplication, we can compute the rounding error without magnitude tests with the FMA operation (see Section 28.10 [Fused Multiply-Add], page 199), like this:

```
volatile double err, prod, x, y;
prod = x * y;                /* rounded product */
err  = fma (x, y, -prod);    /* exact product = prod + err */
```

For addition, subtraction, and multiplication, we can represent the exact result with the notional sum of two values. However, the exact result of division, remainder, or square root potentially requires an infinite number of digits, so we can at best approximate it. Nevertheless, we can compute an error term that is close to the true error: it is just that error value, rounded to machine precision.

For division, you can approximate `x / y` with `quo + err` like this:

```
volatile double err, quo, x, y;
quo = x / y;
err = fma (-quo, y, x) / y;
```

For square root, we can approximate `sqrt (x)` with `root + err` like this:

```
volatile double err, root, x;
```

```
root = sqrt (x);
err = fma (-root, root, x) / (root + root);
```

With the reliable and predictable floating-point design provided by IEEE 754 arithmetic, we now have the tools we need to track errors in the five basic floating-point operations, and we can effectively simulate computing in twice working precision, which is sometimes sufficient to remove almost all traces of arithmetic errors.

28.12 Exact Floating-Point Constants

One of the frustrations that numerical programmers have suffered with since the dawn of digital computers is the inability to precisely specify numbers in their programs. On the early decimal machines, that was not an issue: you could write a constant `1e-30` and be confident of that exact value being used in floating-point operations. However, when the hardware works in a base other than 10, then human-specified numbers have to be converted to that base, and then converted back again at output time. The two base conversions are rarely exact, and unwanted rounding errors are introduced.

As computers usually represent numbers in a base other than 10, numbers often must be converted to and from different bases, and rounding errors can occur during conversion. This problem is solved in C using hexadecimal floating-point constants. For example, `+0x1.fffffcp-1` is the number that is the IEEE 754 32-bit value closest to, but below, `1.0`. The significand is represented as a hexadecimal fraction, and the *power of two* is written in decimal following the exponent letter `p` (the traditional exponent letter `e` is not possible, because it is a hexadecimal digit).

In `printf` and `scanf` and related functions, you can use the '`%a`' and '`%A`' format specifiers for writing and reading hexadecimal floating-point values. '`%a`' writes them with lower case letters and '`%A`' writes them with upper case letters. For instance, this code reproduces our sample number:

```
printf ("%a\n", 1.0 - pow (2.0, -23));
    ⊣ 0x1.fffffcp-1
```

The `strtod` family was similarly extended to recognize numbers in that new format.

If you want to ensure exact data representation for transfer of floating-point numbers between C programs on different computers, then hexadecimal constants are an optimum choice.

28.13 Handling Infinity

As we noted earlier, the IEEE 754 model of computing is not to stop the program when exceptional conditions occur. It takes note of exceptional values or conditions by setting sticky *exception flags*, or by producing results with the special values Infinity and QNaN. In this section, we discuss Infinity; see Section 28.14 [Handling NaN], page 202, for the other.

In GNU C, you can create a value of negative Infinity in software like this:

```
double x;

x = -1.0 / 0.0;
```

GNU C supplies the `__builtin_inf`, `__builtin_inff`, and `__builtin_infl` macros, and the GNU C Library provides the `INFINITY` macro, all of which are compile-time constants for positive infinity.

GNU C also provides a standard function to test for an Infinity: `isinf (x)` returns `1` if the argument is a signed infinity, and `0` if not.

Infinities can be compared, and all Infinities of the same sign are equal: there is no notion in IEEE 754 arithmetic of different kinds of Infinities, as there are in some areas of mathematics. Positive Infinity is larger than any finite value, and negative Infinity is smaller than any finite value.

Infinities propagate in addition, subtraction, multiplication, and square root, but in division, they disappear, because of the rule that `finite / Infinity` is `0.0`. Thus, an overflow in an intermediate computation that produces an Infinity is likely to be noticed later in the final results. The programmer can then decide whether the overflow is expected, and acceptable, or whether the code possibly has a bug, or needs to be run in higher precision, or redesigned to avoid the production of the Infinity.

28.14 Handling NaN

NaNs are not numbers: they represent values from computations that produce undefined results. They have a distinctive property that makes them unlike any other floating-point value: they are *unequal to everything, including themselves*! Thus, you can write a test for a NaN like this:

```
if (x != x)
  printf ("x is a NaN\n");
```

This test works in GNU C, but some compilers might evaluate that test expression as false without properly checking for the NaN value. A more portable way to test for NaN is to use the `isnan` function declared in `math.h`:

```
if (isnan (x))
  printf ("x is a NaN\n");
```

See Section “Floating Point Classes” in *The GNU C Library Reference Manual*.

One important use of NaNs is marking of missing data. For example, in statistics, such data must be omitted from computations. Use of any particular finite value for missing data would eventually collide with real data, whereas such data could never be a NaN, so it is an ideal marker. Functions that deal with collections of data that may have holes can be written to test for, and ignore, NaN values.

It is easy to generate a NaN in computations: evaluating `0.0 / 0.0` is the commonest way, but `Infinity - Infinity`, `Infinity / Infinity`, and `sqrt (-1.0)` also work. Functions that receive out-of-bounds arguments can choose to return a stored NaN value, such as with the `NAN` macro defined in `math.h`, but that does not set the *invalid operand* exception flag, and that can fool some programs.

Like Infinity, NaNs propagate in computations, but they are even stickier, because they never disappear in division. Thus, once a NaN appears in a chain of numerical operations, it is almost certain to pop out into the final results. The programmer has to decide whether that is expected, or whether there is a coding or algorithmic error that needs repair.

In general, when function gets a NaN argument, it usually returns a NaN. However, there are some exceptions in the math-library functions that you need to be aware of, because they violate the *NaNs-always-propagate* rule:

- `pow (x, 0.0)` always returns `1.0`, even if `x` is 0.0, Infinity, or a NaN.
- `pow (1, y)` always returns `1`, even if `y` is a NaN.
- `hypot (INFINITY, y)` and `hypot (-INFINITY, y)` both always return `INFINITY`, even if `y` is a Nan.
- If just one of the arguments to `fmax (x, y)` or `fmin (x, y)` is a NaN, it returns the other argument. If both arguments are NaNs, it returns a NaN, but there is no requirement about where it comes from: it could be `x`, or `y`, or some other quiet NaN.

NaNs are also used for the return values of math-library functions where the result is not representable in real arithmetic, or is mathematically undefined or uncertain, such as `sqrt (-1.0)` and `sin (Infinity)`. However, note that a result that is merely too big to represent should always produce an Infinity, such as with `exp (1000.0)` (too big) and `exp (Infinity)` (truly infinite).

28.15 Signed Zeros

The sign of zero is significant, and important, because it records the creation of a value that is too small to represent, but came from either the negative axis, or from the positive axis. Such fine distinctions are essential for proper handling of *branch cuts* in complex arithmetic (see Section 28.19 [Complex Arithmetic], page 206).

The key point about signed zeros is that in comparisons, their sign does not matter: `0.0 == -0.0` must *always* evaluate to `1` (true). However, they are not *the same number*, and `-0.0` in C code stands for a negative zero.

28.16 Scaling by Powers of the Base

We have discussed rounding errors several times in this chapter, but it is important to remember that when results require no more bits than the exponent and significand bits can represent, those results are *exact*.

One particularly useful exact operation is scaling by a power of the base. While one, in principle, could do that with code like this:

```
y = x * pow (2.0, (double)k);    /* Undesirable scaling: avoid! */
```

that is not advisable, because it relies on the quality of the math-library power function, and that happens to be one of the most difficult functions in the C math library to make accurate. What is likely to happen on many systems is that the returned value from `pow` will be close to a power of two, but slightly different, so the subsequent multiplication introduces rounding error.

The correct, and fastest, way to do the scaling is either via the traditional C library function, or with its C99 equivalent:

```
y = ldexp (x, k);               /* Traditional pre-C99 style. */
y = scalbn (x, k);              /* C99 style. */
```

Both functions return `x * 2**k`. See Section "Normalization Functions" in *The GNU C Library Reference Manual*.

28.17 Rounding Control

Here we describe how to specify the rounding mode at run time. System header file `fenv.h` provides the prototypes for these functions. See Section "Rounding" in *The GNU C Library Reference Manual*.

That header file also provides constant names for the four rounding modes: `FE_DOWNWARD`, `FE_TONEAREST`, `FE_TOWARDZERO`, and `FE_UPWARD`.

The function `fegetround` examines and returns the current rounding mode. On a platform with IEEE 754 floating point, the value will always equal one of those four constants. On other platforms, it may return a negative value. The function `fesetround` sets the current rounding mode.

Changing the rounding mode can be slow, so it is useful to minimize the number of changes. For interval arithmetic, we seem to need three changes for each operation, but we really only need two, because we can write code like this example for interval addition of two reals:

```
{
  struct interval_double
    {
      double hi, lo;
    } v;
  extern volatile double x, y;
  int rule;

  rule = fegetround ();

  if (fesetround (FE_UPWARD) == 0)
    {
      v.hi = x + y;
      v.lo = -(-x - y);
    }
  else
    fatal ("ERROR: failed to change rounding rule");

  if (fesetround (rule) != 0)
    fatal ("ERROR: failed to restore rounding rule");
}
```

The `volatile` qualifier (see Section 21.2 [volatile], page 129) is essential on x86 platforms to prevent an optimizing compiler from producing the same value for both bounds.

28.18 Machine Epsilon

In any floating-point system, three attributes are particularly important to know: *base* (the number that the exponent specifies a power of), *precision* (number of digits in the significand), and *range* (difference between most positive and most negative values). The allocation of bits between exponent and significand decides the answers to those questions.

A measure of the precision is the answer to the question: what is the smallest number that can be added to `1.0` such that the sum differs from `1.0`? That number is called the *machine epsilon*.

We could define the needed machine-epsilon constants for `float`, `double`, and `long double` like this:

```
static const float  epsf = 0x1p-23;  /* about 1.192e-07 */
static const double eps  = 0x1p-52;  /* about 2.220e-16 */
static const long double epsl = 0x1p-63;  /* about 1.084e-19 */
```

Instead of the hexadecimal constants, we could also have used the Standard C macros, `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON`.

It is useful to be able to compute the machine epsilons at run time, and we can easily generalize the operation by replacing the constant `1.0` with a user-supplied value:

```
double
macheps (double x)
{ /* Return machine epsilon for x,  */
   /* such that x + macheps (x) > x.  */
  static const double base = 2.0;
  double eps;

  if (isnan (x))
      eps = x;
  else
    {
      eps = (x == 0.0) ? 1.0 : x;

      while ((x + eps / base) != x)
          eps /= base;          /* Always exact!  */
    }

  return (eps);
}
```

If we call that function with arguments from `0` to `10`, as well as Infinity and NaN, and print the returned values in hexadecimal, we get output like this:

```
macheps (  0) = 0x1.0000000000000p-1074
macheps (  1) = 0x1.0000000000000p-52
macheps (  2) = 0x1.0000000000000p-51
macheps (  3) = 0x1.8000000000000p-52
macheps (  4) = 0x1.0000000000000p-50
macheps (  5) = 0x1.4000000000000p-51
macheps (  6) = 0x1.8000000000000p-51
macheps (  7) = 0x1.c000000000000p-51
macheps (  8) = 0x1.0000000000000p-49
macheps (  9) = 0x1.2000000000000p-50
macheps ( 10) = 0x1.4000000000000p-50
macheps (Inf) = infinity
macheps (NaN) = nan
```

Notice that `macheps` has a special test for a NaN to prevent an infinite loop.

Our code made another test for a zero argument to avoid getting a zero return. The returned value in that case is the smallest representable floating-point number, here the subnormal value `2**(-1074)`, which is about `4.941e-324`.

No special test is needed for an Infinity, because the `eps`-reduction loop then terminates at the first iteration.

Our `macheps` function here assumes binary floating point; some architectures may differ.

The C library includes some related functions that can also be used to determine machine epsilons at run time:

```
#include <math.h>           /* Include for these prototypes. */

double      nextafter  (double x, double y);
float       nextafterf (float x, float y);
long double nextafterl (long double x, long double y);
```

These return the machine number nearest x in the direction of y. For example, `nextafter (1.0, 2.0)` produces the same result as `1.0 + macheps (1.0)` and `1.0 + DBL_EPSILON`. See Section "FP Bit Twiddling" in *The GNU C Library Reference Manual*.

It is important to know that the machine epsilon is not symmetric about all numbers. At the boundaries where normalization changes the exponent, the epsilon below x is smaller than that just above x by a factor `1 / base`. For example, `macheps (1.0)` returns `+0x1p-52`, whereas `macheps (-1.0)` returns `+0x1p-53`. Some authors distinguish those cases by calling them the *positive* and *negative*, or *big* and *small*, machine epsilons. You can produce their values like this:

```
eps_neg = 1.0 - nextafter (1.0, -1.0);
eps_pos = nextafter (1.0, +2.0) - 1.0;
```

If x is a variable, such that you do not know its value at compile time, then you can substitute literal y values with either `-inf()` or `+inf()`, like this:

```
eps_neg = x - nextafter (x, -inf ());
eps_pos = nextafter (x, +inf() - x);
```

In such cases, if x is Infinity, then *the* `nextafter` *functions return* y *if* x *equals* y. Our two assignments then produce `+0x1.fffffffffffffp+1023` (that is a hexadecimal floating point constant and its value is around 1.798e+308; see Section 12.3 [Floating Constants], page 53) for *eps_neg*, and Infinity for *eps_pos*. Thus, the call `nextafter (INFINITY, -INFINITY)` can be used to find the largest representable finite number, and with the call `nextafter (0.0, 1.0)`, the smallest representable number (here, `0x1p-1074` (about 4.491e-324), a number that we saw before as the output from `macheps (0.0)`).

28.19 Complex Arithmetic

We've already looked at defining and referring to complex numbers (see Section 11.3 [Complex Data Types], page 49). What is important to discuss here are some issues that are unlikely to be obvious to programmers without extensive experience in both numerical computing, and in complex arithmetic in mathematics.

The first important point is that, unlike real arithmetic, in complex arithmetic, the danger of significance loss is *pervasive*, and affects *every one* of the basic operations, and *almost all* of the math-library functions. To understand why, recall the rules for complex multiplication and division:

```
a = u + I*v                /* First operand. */
b = x + I*y                /* Second operand. */

prod = a * b
     = (u + I*v) * (x + I*y)
     = (u * x - v * y) + I*(v * x + u * y)

quo  = a / b
     = (u + I*v) / (x + I*y)
     = [(u + I*v) * (x - I*y)] / [(x + I*y) * (x - I*y)]
     = [(u * x + v * y) + I*(v * x - u * y)] / (x**2 + y**2)
```

There are four critical observations about those formulas:

- the multiplications on the right-hand side introduce the possibility of premature underflow or overflow;
- the products must be accurate to twice working precision;
- there is *always* one subtraction on the right-hand sides that is subject to catastrophic significance loss; and
- complex multiplication has up to *six* rounding errors, and complex division has *ten* rounding errors.

Another point that needs careful study is the fact that many functions in complex arithmetic have *branch cuts*. You can view a function with a complex argument, `f (z)`, as `f (x + I*y)`, and thus, it defines a relation between a point `(x, y)` on the complex plane with an elevation value on a surface. A branch cut looks like a tear in that surface, so approaching the cut from one side produces a particular value, and from the other side, a quite different value. Great care is needed to handle branch cuts properly, and even small numerical errors can push a result from one side to the other, radically changing the returned value. As we reported earlier, correct handling of the sign of zero is critically important for computing near branch cuts.

The best advice that we can give to programmers who need complex arithmetic is to always use the *highest precision available*, and then to carefully check the results of test calculations to gauge the likely accuracy of the computed results. It is easy to supply test values of real and imaginary parts where all five basic operations in complex arithmetic, and almost all of the complex math functions, lose *all* significance, and fail to produce even a single correct digit.

Even though complex arithmetic makes some programming tasks easier, it may be numerically preferable to rework the algorithm so that it can be carried out in real arithmetic. That is commonly possible in matrix algebra.

GNU C can perform code optimization on complex number multiplication and division if certain boundary checks will not be needed. The command-line option `-fcx-limited-range` tells the compiler that a range reduction step is not needed when performing complex division, and that there is no need to check if a complex multiplication or division results in the value `Nan + I*NaN`. By default these checks are enabled. You can explicitly enable them with the `-fno-cx-limited-range` option.

28.20 Round-Trip Base Conversion

Most numeric programs involve converting between base-2 floating-point numbers, as represented by the computer, and base-10 floating-point numbers, as entered and handled by the programmer. What might not be obvious is the number of base-2 bits vs. base-10 digits required for each representation. Consider the following tables showing the number of decimal digits representable in a given number of bits, and vice versa:

binary in	24	53	64	113	237
decimal out	9	17	21	36	73

decimal in	7	16	34	70
binary out	25	55	114	234

We can compute the table numbers with these two functions:

```
int
matula(int nbits)
{   /* Return output decimal digits needed for nbits-bits input. */
    return ((int)ceil((double)nbits / log2(10.0) + 1.0));
}

int
goldberg(int ndec)
{   /* Return output bits needed for ndec-digits input. */
    return ((int)ceil((double)ndec / log10(2.0) + 1.0));
}
```

One significant observation from those numbers is that we cannot achieve correct round-trip conversion between the decimal and binary formats in the same storage size! For example, we need 25 bits to represent a 7-digit value from the 32-bit decimal format, but the binary format only has 24 available. Similar observations hold for each of the other conversion pairs.

The general input/output base-conversion problem is astonishingly complicated, and solutions were not generally known until the publication of two papers in 1990 that are listed later near the end of this chapter. For the 128-bit formats, the worst case needs more than 11,500 decimal digits of precision to guarantee correct rounding in a binary-to-decimal conversion!

For further details see the references for Bennett Goldberg and David Matula.

28.21 Further Reading

The subject of floating-point arithmetic is much more complex than many programmers seem to think, and few books on programming languages spend much time in that area. In this chapter, we have tried to expose the reader to some of the key ideas, and to warn of easily overlooked pitfalls that can soon lead to nonsensical results. There are a few good references that we recommend for further reading, and for finding other important material about computer arithmetic:

We include URLs for these references when we were given them, when they are morally legitimate to recommend; we have omitted the URLs that are paywalled or that require running nonfree JavaScript code in order to function. We hope you can find morally legitimate sites where you can access these works.

- Paul H. Abbott and 15 others, *Architecture and software support in IBM S/390 Parallel Enterprise Servers for IEEE Floating-Point arithmetic*, IBM Journal of Research and Development **43**(5/6) 723–760 (1999), This article gives a good description of IBM's algorithm for exact decimal-to-binary conversion, complementing earlier ones by Clinger and others.
- Nelson H. F. Beebe, *The Mathematical-Function Computation Handbook: Programming Using the MathCW Portable Software Library*, Springer (2017). This book describes portable implementations of a large superset of the mathematical functions available in many programming languages, extended to a future 256-bit format (70 decimal digits), for both binary and decimal floating point. It includes a substantial portion of the functions described in the famous *NIST Handbook of Mathematical Functions*, Cambridge (2018), ISBN 0-521-19225-0. See `https://www.math.utah.edu/pub/mathcw/` for compilers and libraries.
- William D. Clinger, *How to Read Floating Point Numbers Accurately*, ACM SIGPLAN Notices **25**(6) 92–101 (June 1990), `https://doi.org/10.1145/93548.93557`. See also the papers by Steele & White.
- I. Bennett Goldberg, *27 Bits Are Not Enough For 8-Digit Accuracy*, Communications of the ACM **10**(2) 105–106 (February 1967), `https://doi.acm.org/10.1145/363067.363112`. This paper, and its companions by David Matula, address the base-conversion problem, and show that the naive formulas are wrong by one or two digits.
- David Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys **23**(1) 5–58 (March 1991), corrigendum **23**(3) 413 (September 1991), `https://doi.org/10.1145/103162.103163`. This paper has been widely distributed, and reissued in vendor programming-language documentation. It is well worth reading, and then rereading from time to time.
- Norbert Juffa and Nelson H. F. Beebe, *A Bibliography of Publications on Floating-Point Arithmetic*, `https://www.math.utah.edu/pub/tex/bib/fparith.bib`. This is the largest known bibliography of publications about floating-point, and also integer, arithmetic. It is actively maintained, and in mid 2019, contains more than 6400 references to original research papers, reports, theses, books, and Web sites on the subject matter. It can be used to locate the latest research in the field, and the historical coverage dates back to a 1726 paper on signed-digit arithmetic, and an 1837 paper by Charles Babbage, the intellectual father of mechanical computers. The entries for the Abbott, Clinger, and Steele & White papers cited earlier contain pointers to several other important related papers on the base-conversion problem.
- William Kahan, *Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit*, (1987), `https://people.freebsd.org/~das/kahan86branch.pdf`. This Web document about the fine points of complex arithmetic also appears in the volume edited by A. Iserles and M. J. D. Powell, *The State of the Art in*

Numerical Analysis: Proceedings of the Joint IMA/SIAM Conference on the State of the Art in Numerical Analysis held at the University of Birmingham, 14–18 April 1986, Oxford University Press (1987), ISBN 0-19-853614-3 (xiv + 719 pages). Its author is the famous chief architect of the IEEE 754 arithmetic system, and one of the world's greatest experts in the field of floating-point arithmetic. An entire generation of his students at the University of California, Berkeley, have gone on to careers in academic and industry, spreading the knowledge of how to do floating-point arithmetic right.

- Donald E. Knuth, *A Simple Program Whose Proof Isn't*, in *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra (eds.), Springer (1990), ISBN 1-4612-8792-8, This book chapter supplies a correctness proof of the decimal to binary, and binary to decimal, conversions in fixed-point arithmetic in the TeX typesetting system. The proof evaded its author for a dozen years.
- David W. Matula, *In-and-out conversions*, Communications of the ACM **11**(1) 57–50 (January 1968), `https://doi.org/10.1145/362851.362887`.
- David W. Matula, *The Base Conversion Theorem*, Proceedings of the American Mathematical Society **19**(3) 716–723 (June 1968). See also other papers here by this author, and by I. Bennett Goldberg.
- David W. Matula, *A Formalization of Floating-Point Numeric Base Conversion*, IEEE Transactions on Computers **C-19**(8) 681–692 (August 1970),
- Jean-Michel Muller and eight others, *Handbook of Floating-Point Arithmetic*, Birkhäuser-Boston (2010), ISBN 0-8176-4704-X (xxiii + 572 pages). This is a comprehensive treatise from a French team who are among the world's greatest experts in floating-point arithmetic, and among the most prolific writers of research papers in that field. They have much to teach, and their book deserves a place on the shelves of every serious numerical programmer.
- Jean-Michel Muller and eight others, *Handbook of Floating-Point Arithmetic*, Second edition, Birkhäuser-Boston (2018), ISBN 3-319-76525-6 (xxv + 627 pages). This is a new edition of the preceding entry.
- Michael Overton, *Numerical Computing with IEEE Floating Point Arithmetic, Including One Theorem, One Rule of Thumb, and One Hundred and One Exercises*, SIAM (2001), ISBN 0-89871-482-6 (xiv + 104 pages), This is a small volume that can be covered in a few hours.
- Guy L. Steele Jr. and Jon L. White, *How to Print Floating-Point Numbers Accurately*, ACM SIGPLAN Notices **25**(6) 112–126 (June 1990), `https://doi.org/10.1145/93548.93559`. See also the papers by Clinger.
- Guy L. Steele Jr. and Jon L. White, *Retrospective: How to Print Floating-Point Numbers Accurately*, ACM SIGPLAN Notices **39**(4) 372–389 (April 2004), Reprint of 1990 paper, with additional commentary.
- Pat H. Sterbenz, *Floating Point Computation*, Prentice-Hall (1974), ISBN 0-13-322495-3 (xiv + 316 pages). This often-cited book provides solid coverage of what floating-point arithmetic was like *before* the introduction of IEEE 754 arithmetic.

29 Compilation

Early in the manual we explained how to compile a simple C program that consists of a single source file (see Section 2.4 [Compile Example], page 11). However, we handle only short programs that way. A typical C program consists of many source files, each of which is usually a separate *compilation module*—meaning that it has to be compiled separately. (The source files that are not separate compilation modules are those that are used via `#include`; see Section 26.4 [Header Files], page 162.)

To compile a multi-module program, you compile each of the program's compilation modules, making an *object file* for that module. The last step is to *link* the many object files together into a single executable for the whole program.

The full details of how to compile C programs (and other programs) with GCC are documented in xxxx. Here we give only a simple introduction.

These commands compile two compilation modules, `foo.c` and `bar.c`, running the compiler for each module:

```
gcc -c -O -g foo.c
gcc -c -O -g bar.c
```

In these commands, `-g` says to generate debugging information, `-O` says to do some optimization, and `-c` says to put the compiled code for that module into a corresponding object file and go no further. The object file for `foo.c` is automatically called `foo.o`, and so on.

If you wish, you can specify the additional compilation options. For instance, `-Wformat -Wparenthesis -Wstrict-prototypes` request additional warnings.

After you compile all the program's modules, you link the object files into a combined executable, like this:

```
gcc -o foo foo.o bar.o
```

In this command, `-o foo` species the file name for the executable file, and the other arguments are the object files to link. Always specify the executable file name in a command that generates one.

One reason to divide a large program into multiple compilation modules is to control how each module can access the internals of the others. When a module declares a function or variable `extern`, other modules can access it. The other functions and variables defined in a module can't be accessed from outside that module.

The other reason for using multiple modules is so that changing one source file does not require recompiling all of them in order to try the modified program. It is sufficient to recompile the source file that you changed, then link them all again. Dividing a large program into many substantial modules in this way typically makes recompilation much faster.

Normally we don't run any of these commands directly. Instead we write a set of *make rules* for the program, then use the `make` program to recompile only the source files that need to be recompiled, by following those rules. See *The GNU Make Manual*.

30 Directing Compilation

This chapter describes C constructs that don't alter the program's meaning *as such*, but rather direct the compiler how to treat some aspects of the program.

30.1 Pragmas

A *pragma* is an annotation in a program that gives direction to the compiler.

30.1.1 Pragma Basics

C defines two syntactical forms for pragmas, the line form and the token form. You can write any pragma in either form, with the same meaning.

The line form is a line in the source code, like this:

```
#pragma line
```

The line pragma has no effect on the parsing of the lines around it. This form has the drawback that it can't be generated by a macro expansion.

The token form is a series of tokens; it can appear anywhere in the program between the other tokens.

```
_Pragma (stringconstant)
```

The pragma has no effect on the syntax of the tokens that surround it; thus, here's a pragma in the middle of an `if` statement:

```
if _Pragma ("hello") (x > 1)
```

However, that's an unclear thing to do; for the sake of understandability, it is better to put a pragma on a line by itself and not embedded in the middle of another construct.

Both forms of pragma have a textual argument. In a line pragma, the text is the rest of the line. The textual argument to `_Pragma` uses the same syntax as a C string constant: surround the text with two '`"`' characters, and add a backslash before each '`"`' or '\' character in it.

With either syntax, the textual argument specifies what to do. It begins with one or several words that specify the operation. If the compiler does not recognize them, it ignores the pragma.

Here are the pragma operations supported in GNU C.

`#pragma GCC dependency "file" [message]`
`_Pragma ("GCC dependency \"file\" [message]")`

Declares that the current source file depends on *file*, so GNU C compares the file times and gives a warning if *file* is newer than the current source file.

This directive searches for *file* the way `#include` searches for a non-system header file.

If *message* is given, the warning message includes that text.

Examples:

```
#pragma GCC dependency "parse.y"
_pragma ("GCC dependency \"/usr/include/time.h\" \
rerun fixincludes")
```

`#pragma GCC poison` *`identifiers`*
`_Pragma ("GCC poison` *`identifiers`*`")`

Poisons the identifiers listed in *identifiers*.

This is useful to make sure all mention of *identifiers* has been deleted from the program and that no reference to them creeps back in. If any of those identifiers appears anywhere in the source after the directive, it causes a compilation error. For example,

```
#pragma GCC poison printf sprintf fprintf
sprintf(some_string, "hello");
```

generates an error.

If a poisoned identifier appears as part of the expansion of a macro that was defined before the identifier was poisoned, it will *not* cause an error. Thus, system headers that define macros that use the identifier will not cause errors.

For example,

```
#define strrchr rindex
_Pragma ("GCC poison rindex")
strrchr(some_string, 'h');
```

does not cause a compilation error.

`#pragma GCC system_header`
`_Pragma ("GCC system_header")`

Specify treating the rest of the current source file as if it came from a system header file. See Section “System Headers” in *Using the GNU Compiler Collection*.

`#pragma GCC warning` *`message`*
`_Pragma ("GCC warning` *`message`*`")`

Equivalent to `#warning`. Its advantage is that the `_Pragma` form can be included in a macro definition.

`#pragma GCC error` *`message`*
`_Pragma ("GCC error` *`message`*`")`

Equivalent to `#error`. Its advantage is that the `_Pragma` form can be included in a macro definition.

`#pragma GCC message` *`message`*
`_Pragma ("GCC message` *`message`*`")`

Similar to ‘`GCC warning`’ and ‘`GCC error`’, this simply prints an informational message, and could be used to include additional warning or error text without triggering more warnings or errors. (Note that unlike ‘`warning`’ and ‘`error`’, ‘`message`’ does not include ‘`GCC`’ as part of the pragma.)

30.1.2 Severity Pragmas

These pragmas control the severity of classes of diagnostics. You can specify the class of diagnostic with the GCC option that causes those diagnostics to be generated.

`#pragma GCC diagnostic error` *`option`*
`_Pragma ("GCC diagnostic error` *`option`*`")`

For code following this pragma, treat diagnostics of the variety specified by *option* as errors. For example:

```
_Pragma ("GCC diagnostic error -Wformat")
```

specifies to treat diagnostics enabled by the *-Wformat* option as errors rather than warnings.

`#pragma GCC diagnostic warning` *`option`*
`_Pragma ("GCC diagnostic warning` *`option`*`")`

For code following this pragma, treat diagnostics of the variety specified by *option* as warnings. This overrides the *-Werror* option which says to treat warnings as errors.

`#pragma GCC diagnostic ignore` *`option`*
`_Pragma ("GCC diagnostic ignore` *`option`*`")`

For code following this pragma, refrain from reporting any diagnostics of the variety specified by *option*.

`#pragma GCC diagnostic push`
`_Pragma ("GCC diagnostic push")`
`#pragma GCC diagnostic pop`
`_Pragma ("GCC diagnostic pop")`

These pragmas maintain a stack of states for severity settings. ‘`GCC diagnostic push`’ saves the current settings on the stack, and ‘`GCC diagnostic pop`’ pops the last stack item and restores the current settings from that.

‘`GCC diagnostic pop`’ when the severity setting stack is empty restores the settings to what they were at the start of compilation.

Here is an example:

```
_Pragma ("GCC diagnostic error -Wformat")

/* -Wformat messages treated as errors.  */

_Pragma ("GCC diagnostic push")
_Pragma ("GCC diagnostic warning -Wformat")

/* -Wformat messages treated as warnings.  */

_Pragma ("GCC diagnostic push")
_Pragma ("GCC diagnostic ignored -Wformat")

/* -Wformat messages suppressed.  */

_Pragma ("GCC diagnostic pop")

/* -Wformat messages treated as warnings again.  */

_Pragma ("GCC diagnostic pop")
```

```
/* -Wformat messages treated as errors again.  */

/* This is an excess ‘pop’ that matches no ‘push’.  */
_Pragma ("GCC diagnostic pop")

/* -Wformat messages treated once again
   as specified by the GCC command-line options.  */
```

30.1.3 Optimization Pragmas

These pragmas enable a particular optimization for specific function definitions. The settings take effect at the end of a function definition, so the clean place to use these pragmas is between function definitions.

`#pragma GCC optimize` *`optimization`*
`_Pragma ("GCC optimize` *`optimization`*`")`

These pragmas enable the optimization *optimization* for the following functions. For example,

```
_Pragma ("GCC optimize -fforward-propagate")
```

says to apply the ‘`forward-propagate`’ optimization to all following function definitions. Specifying optimizations for individual functions, rather than for the entire program, is rare but can be useful for getting around a bug in the compiler.

If *optimization* does not correspond to a defined optimization option, the pragma is erroneous. To turn off an optimization, use the corresponding ‘`-fno-`’ option, such as ‘`-fno-forward-propagate`’.

`#pragma GCC target` *`optimizations`*
`_Pragma ("GCC target` *`optimizations`*`")`

The pragma ‘`GCC target`’ is similar to ‘`GCC optimize`’ but is used for platform-specific optimizations. Thus,

```
_Pragma ("GCC target popcnt")
```

activates the optimization ‘`popcnt`’ for all following function definitions. This optimization is supported on a few common targets but not on others.

`#pragma GCC push_options`
`_Pragma ("GCC push_options")`

The ‘`push_options`’ pragma saves on a stack the current settings specified with the ‘`target`’ and ‘`optimize`’ pragmas.

`#pragma GCC pop_options`
`_Pragma ("GCC pop_options")`

The ‘`pop_options`’ pragma pops saved settings from that stack.

Here’s an example of using this stack.

```
_Pragma ("GCC push_options")
_Pragma ("GCC optimize forward-propagate")
```

```
                /* Functions to compile
                   with the forward-propagate optimization. */

                _Pragma ("GCC pop_options")
                /* Ends enablement of forward-propagate. */
```

`#pragma GCC reset_options`
`_Pragma ("GCC reset_options")`
Clears all pragma-defined '`target`' and '`optimize`' optimization settings.

30.2 Static Assertions

You can add compiler-time tests for necessary conditions into your code using `_Static_assert`. This can be useful, for example, to check that the compilation target platform supports the type sizes that the code expects. For example,

```
_Static_assert ((sizeof (long int) >= 8),
    "long int needs to be at least 8 bytes");
```

reports a compile-time error if compiled on a system with long integers smaller than 8 bytes, with '`long int needs to be at least 8 bytes`' as the error message.

Since calls `_Static_assert` are processed at compile time, the expression must be computable at compile time and the error message must be a literal string. The expression can refer to the sizes of variables, but can't refer to their values. For example, the following static assertion is invalid for two reasons:

```
char *error_message
  = "long int needs to be at least 8 bytes";
int size_of_long_int = sizeof (long int);

_Static_assert (size_of_long_int == 8, error_message);
```

The expression `size_of_long_int == 8` isn't computable at compile time, and the error message isn't a literal string.

You can, though, use preprocessor definition values with `_Static_assert`:

```
#define LONG_INT_ERROR_MESSAGE "long int needs to be \
at least 8 bytes"

_Static_assert ((sizeof (long int) == 8),
  LONG_INT_ERROR_MESSAGE);
```

Static assertions are permitted wherever a statement or declaration is permitted, including at top level in the file, and also inside the definition of a type.

```
union y
{
  int i;
  int *ptr;
  _Static_assert (sizeof (int *) == sizeof (int),
  "Pointer and int not same size");
};
```

Appendix A Type Alignment

Code for device drivers and other communication with low-level hardware sometimes needs to be concerned with the alignment of data objects in memory.

Each data type has a required *alignment*, always a power of 2, that says at which memory addresses an object of that type can validly start. A valid address for the type must be a multiple of its alignment. If a type's alignment is 1, that means it can validly start at any address. If a type's alignment is 2, that means it can only start at an even address. If a type's alignment is 4, that means it can only start at an address that is a multiple of 4.

The alignment of a type (except `char`) can vary depending on the kind of computer in use. To refer to the alignment of a type in a C program, use `_Alignof`, whose syntax parallels that of `sizeof`. Like `sizeof`, `_Alignof` is a compile-time operation, and it doesn't compute the value of the expression used as its argument.

Nominally, each integer and floating-point type has an alignment equal to the largest power of 2 that divides its size. Thus, `int` with size 4 has a nominal alignment of 4, and `long long int` with size 8 has a nominal alignment of 8.

However, each kind of computer generally has a maximum alignment, and no type needs more alignment than that. If the computer's maximum alignment is 4 (which is common), then no type's alignment is more than 4.

The size of any type is always a multiple of its alignment; that way, in an array whose elements have that type, all the elements are properly aligned if the first one is.

These rules apply to all real computers today, but some embedded controllers have odd exceptions. We don't have references to cite for them.

Ordinary C code guarantees that every object of a given type is in fact aligned as that type requires.

If the operand of `_Alignof` is a structure field, the value is the alignment it requires. It may have a greater alignment by coincidence, due to the other fields, but `_Alignof` is not concerned about that. See Chapter 15 [Structures], page 74.

Older versions of GNU C used the keyword `__alignof__` for this, but now that the feature has been standardized, it is better to use the standard keyword `_Alignof`.

You can explicitly specify an alignment requirement for a particular variable or structure field by adding `_Alignas (alignment)` to the declaration, where *alignment* is a power of 2 or a type name. For instance:

```
char _Alignas (8) x;
```

or

```
char _Alignas (double) x;
```

specifies that `x` must start on an address that is a multiple of 8. However, if *alignment* exceeds the maximum alignment for the machine, that maximum is how much alignment `x` will get.

The older GNU C syntax for this feature looked like `__attribute__ ((__aligned__ (alignment)))` to the declaration, and was added after the variable. For instance:

```
char x __attribute__ ((__aligned__ 8));
```

See Appendix D [Attributes], page 222.

Appendix B Aliasing

We have already presented examples of casting a `void *` pointer to another pointer type, and casting another pointer type to `void *`.

One common kind of pointer cast is guaranteed safe: casting the value returned by `malloc` and related functions (see Section 15.3 [Dynamic Memory Allocation], page 76). It is safe because these functions do not save the pointer anywhere else; the only way the program will access the newly allocated memory is via the pointer just returned.

In fact, C allows casting any pointer type to any other pointer type. Using this to access the same place in memory using two different data types is called *aliasing*.

Aliasing is necessary in some programs that do sophisticated memory management, such as GNU Emacs, but most C programs don't need to do aliasing. When it isn't needed, **stay away from it!** To do aliasing correctly requires following the rules stated below. Otherwise, the aliasing may result in malfunctions when the program runs.

The rest of this appendix explains the pitfalls and rules of aliasing.

B.1 Aliasing and Alignment

In order for a type-converted pointer to be valid, it must have the alignment that the new pointer type requires. For instance, on most computers, `int` has alignment 4; the address of an `int` must be a multiple of 4. However, `char` has alignment 1, so the address of a `char` is usually not a multiple of 4. Taking the address of such a `char` and casting it to `int *` probably results in an invalid pointer. Trying to dereference it may cause a `SIGBUS` signal, depending on the platform in use (see Appendix E [Signals], page 224).

```
foo ()
{
  char i[4];
  int *p = (int *) &i[1]; /* Misaligned pointer! */
  return *p;              /* Crash! */
}
```

This requirement is never a problem when casting the return value of `malloc` because that function always returns a pointer with as much alignment as any type can require.

B.2 Aliasing and Length

When converting a pointer to a different pointer type, make sure the object it really points to is at least as long as the target of the converted pointer. For instance, suppose `p` has type `int *` and it's cast as follows:

```
int *p;

struct
  {
    double d, e, f;
  } foo;

struct foo *q = (struct foo *)p;
```

```
q->f = 5.14159;
```

the value `q->f` will run past the end of the `int` that `p` points to. If `p` was initialized to the start of an array of type `int[6]`, the object is long enough for three `doubles`. But if `p` points to something shorter, `q->f` will run on beyond the end of that, overlaying some other data. Storing that will garble that other data. Or it could extend past the end of memory space and cause a `SIGSEGV` signal (see Appendix E [Signals], page 224).

B.3 Type Rules for Aliasing

C code that converts a pointer to a different pointer type can use the pointers to access the same memory locations with two different data types. If the same address is accessed with different types in a single control thread, optimization can make the code do surprising things (in effect, make it malfunction).

Here's a concrete example where aliasing that can change the code's behavior when it is optimized. We assume that `float` is 4 bytes long, like `int`, and so is every pointer. Thus, the structures `struct a` and `struct b` are both 8 bytes.

```
#include <stdio.h>
struct a { int size; char *data; };
struct b { float size; char *data; };

void sub (struct a *p, struct b *q)
{
  int x;
  p->size = 0;
  q->size = 1;
  x = p->size;
  printf("x       =%d\n", x);
  printf("p->size =%d\n", (int)p->size);
  printf("q->size =%d\n", (int)q->size);
}

int main(void)
{
  struct a foo;
  struct a *p = &foo;
  struct b *q = (struct b *) &foo;

  sub (p, q);
}
```

This code works as intended when compiled without optimization. All the operations are carried out sequentially as written. The code sets `x` to `p->size`, but what it actually gets is the bits of the floating point number 1, as type `int`.

However, when optimizing, the compiler is allowed to assume (mistakenly, here) that `q` does not point to the same storage as `p`, because their data types are not allowed to alias.

From this assumption, the compiler can deduce (falsely, here) that the assignment into `q->size` has no effect on the value of `p->size`, which must therefore still be 0. Thus, `x` will be set to 0.

GNU C, following the C standard, *defines* this optimization as legitimate. Code that misbehaves when optimized following these rules is, by definition, incorrect C code.

The rules for storage aliasing in C are based on the two data types: the type of the object, and the type it is accessed through. The rules permit accessing part of a storage object of type *t* using only these types:

- *t*.
- A type compatible with *t*. See Chapter 23 [Compatible Types], page 154.
- A signed or unsigned version of one of the above.
- A qualified version of one of the above. See Chapter 21 [Type Qualifiers], page 128.
- An array, structure (see Chapter 15 [Structures], page 74), or union type (`Unions`) that contains one of the above, either directly as a field or through multiple levels of fields. If *t* is `double`, this would include `struct s { union { double d[2]; int i[4]; } u; int i; };` because there's a `double` inside it somewhere.
- A character type.

What do these rules say about the example in this subsection?

For `foo.size` (equivalently, `a->size`), *t* is `int`. The type `float` is not allowed as an aliasing type by those rules, so `b->size` is not supposed to alias with elements of `j`. Based on that assumption, GNU C makes a permitted optimization that was not, in this case, consistent with what the programmer intended the program to do.

Whether GCC actually performs type-based aliasing analysis depends on the details of the code. GCC has other ways to determine (in some cases) whether objects alias, and if it gets a reliable answer that way, it won't fall back on type-based heuristics.

The importance of knowing the type-based aliasing rules is not so as to ensure that the optimization is done where it would be safe, but so as to ensure it is *not* done in a way that would break the program. You can turn off type-based aliasing analysis by giving GCC the option `-fno-strict-aliasing`.

Appendix C Digraphs

C accepts aliases for certain characters. Apparently in the 1990s some computer systems had trouble inputting these characters, or trouble displaying them. These digraphs almost never appear in C programs nowadays, but we mention them for completeness.

‘`<:`’	An alias for ‘`[`’.
‘`:>`’	An alias for ‘`]`’.
‘`<%`’	An alias for ‘`{`’.
‘`%>`’	An alias for ‘`}`’.
‘`%:`’	An alias for ‘`#`’, used for preprocessing directives (see Section 26.2 [Directives], page 160) and macros (see Section 26.5 [Macros], page 167).

Appendix D Attributes in Declarations

You can specify certain additional requirements in a declaration, to get fine-grained control over code generation, and helpful informational messages during compilation. We use a few attributes in code examples throughout this manual, including

`aligned` The `aligned` attribute specifies a minimum alignment for a variable or structure field, measured in bytes:

```
int foo __attribute__ ((aligned (8))) = 0;
```

This directs GNU C to allocate `foo` at an address that is a multiple of 8 bytes. However, you can't force an alignment bigger than the computer's maximum meaningful alignment.

`packed` The `packed` attribute specifies to compact the fields of a structure by not leaving gaps between fields. For example,

```
struct __attribute__ ((packed)) bar
{
  char a;
  int b;
};
```

allocates the integer field `b` at byte 1 in the structure, immediately after the character field `a`. The packed structure is just 5 bytes long (assuming `int` is 4 bytes) and its alignment is 1, that of `char`.

`deprecated`

Applicable to both variables and functions, the `deprecated` attribute tells the compiler to issue a warning if the variable or function is ever used in the source file.

```
int old_foo __attribute__ ((deprecated));

int old_quux () __attribute__ ((deprecated));
```

`__noinline__`

The `__noinline__` attribute, in a function's declaration or definition, specifies never to inline calls to that function. All calls to that function, in a compilation unit where it has this attribute, will be compiled to invoke the separately compiled function. See Section 22.7.4 [Inline Function Definitions], page 149.

`__noclone__`

The `__noclone__` attribute, in a function's declaration or definition, specifies never to clone that function. Thus, there will be only one compiled version of the function. See Section 19.14.2 [Label Value Caveats], page 118, for more information about cloning.

`always_inline`

The `always_inline` attribute, in a function's declaration or definition, specifies to inline all calls to that function (unless something about the function makes inlining impossible). This applies to all calls to that function in a compilation unit where it has this attribute. See Section 22.7.4 [Inline Function Definitions], page 149.

`gnu_inline`
: The `gnu_inline` attribute, in a function's declaration or definition, specifies to handle the `inline` keyword the way GNU C originally implemented it, many years before ISO C said anything about inlining. See Section 22.7.4 [Inline Function Definitions], page 149.

For full documentation of attributes, see the GCC manual. See Section "System Headers" in *Using the GNU Compiler Collection*.

Appendix E Signals

Some program operations bring about an error condition called a *signal*. These signals terminate the program, by default.

There are various different kinds of signals, each with a name. We have seen several such error conditions through this manual:

`SIGSEGV` This signal is generated when a program tries to read or write outside the memory that is allocated for it, or to write memory that can only be read. The name is an abbreviation for "segmentation violation".

`SIGFPE` This signal indicates a fatal arithmetic error. The name is an abbreviation for "floating-point exception", but covers all types of arithmetic errors, including division by zero and overflow.

`SIGBUS` This signal is generated when an invalid pointer is dereferenced, typically the result of dereferencing an uninitialized pointer. It is similar to `SIGSEGV`, except that `SIGSEGV` indicates invalid access to valid memory, while `SIGBUS` indicates an attempt to access an invalid address.

These kinds of signal allow the program to specify a function as a *signal handler*. When a signal has a handler, it doesn't terminate the program; instead it calls the handler.

There are many other kinds of signal; here we list only those that come from run-time errors in C operations. The rest have to do with the functioning of the operating system. The GNU C Library Reference Manual gives more explanation about signals (see Section "Program Signal Handling" in *The GNU C Library Reference Manual*).

Appendix F GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
`http://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
  Copyright (C)  year  your name.
  Permission is granted to copy, distribute and/or modify this document
  under the terms of the GNU Free Documentation License, Version 1.3
  or any later version published by the Free Software Foundation;
  with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
  Texts.  A copy of the license is included in the section entitled ``GNU
  Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
    with the Invariant Sections being list their titles, with
    the Front-Cover Texts being list, and with the Back-Cover Texts
    being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix G GNU General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. `https://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

   d. Limiting the use for publicity purposes of names of licensors or authors of the material; or
   e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
   f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

   All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

   If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

   Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others’ Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see https://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.

Index of Symbols and Keywords

#

`#define` 167
`#elif` 188
`#else` 188
`#endif` 185
`#error` 189
`#if` 185
`#ifdef` 185
`#ifndef` 186
`#include` 163
`#line` 190
`#undef` 178
`#warning` 189

_

`__aligned__` 217
`__alignof__` 217
`__attribute__` 222
`__auto_type` 124
`__complex__` 49
`__label__` 116
`_Alignas` 217
`_Alignof` 217
`_Complex` 49
`_Static_assert` 216

A

`auto` 127

B

`bool` 48
`break` 106

C

`case` 110
`char` 46
`CHAR_MAX` 192
`const` 128
`continue` 109

D

`DBL_DECIMAL_DIG` 193
`DBL_HAS_SUBNORM` 193
`DBL_MAX` 193
`DBL_MIN` 193
`DBL_TRUE_MIN` 193
`default` 110
`do` 106
`double` 48

E

`else` 104
`extern` 126

F

`float` 48
`FLT_DECIMAL_DIG` 193
`FLT_HAS_SUBNORM` 193
`FLT_MAX` 193
`FLT_MIN` 193
`FLT_TRUE_MIN` 193
`for` 107

G

`goto` 114

I

`if` 103
`inline` 149
`int` 46
`INT_MAX` 192

L

`LDBL_DECIMAL_DIG` 193
`LDBL_HAS_SUBNORM` 193
`LDBL_MAX` 193
`LDBL_MIN` 193
`LDBL_TRUE_MIN` 193
`LLONG_MAX` 192
`long double` 48
`long int` 46
`long long int` 46
`LONG_MAX` 192

M

`main` 142

R

`register` 127
`restrict` 130
`return` 105

S

`SCHAR_MAX` 192
`short int` 46
`SHRT_MAX` 192
`signed` 46
`sizeof` 60
`static` 125, 134
`struct` 74
`switch` 110

T

`typedef` 101
`typeof` 124

U

`UCHAR_MAX` 192
`UINT_MAX` 192
`ULLONG_MAX` 192
`ULONG_MAX` 192
`union` 83
`unsigned` 46
`USHRT_MAX` 192

V

`void` 50
`volatile` 129

W

`while` 105

Concept Index

#

`#` preprocessing operator . . . 171
`##` preprocessing operator . . . 173

?

?: side effect . . . 38

_

'`_`' in variables in macros . . . 182
`__attribute__((packed))` . . . 78
`__complex__` keyword . . . 49
`_Complex` keyword . . . 49
_Complex_I . . . 54

\

'`\a`' . . . 55
'`\b`' . . . 55
'`\e`' . . . 55
'`\f`' . . . 55
'`\n`' . . . 55
'`\r`' . . . 55
'`\t`' . . . 55
'`\v`' . . . 55

A

accessing array elements . . . 91
addition operator . . . 22
address of a label . . . 117
address-of operator . . . 62
aliasing (of storage) . . . 218
alignment of type . . . 217
allocating memory dynamically . . . 76
allocation file-scope variables . . . 126
argument promotions . . . 156
arguments . . . 170
arguments in macro definitions . . . 170
arithmetic operators . . . 22
arithmetic, pointer . . . 67
array . . . 91
array as parameters . . . 134
array elements, accessing . . . 91
array example . . . 14
array fields, flexible . . . 81
array of length zero . . . 81
array of variable length . . . 97
array parameters, variable-length . . . 144
array types, incomplete . . . 93
array values, constructing . . . 96
array, declaring . . . 91, 120
array, layout in memory . . . 95
array, multidimensional . . . 95
arrays and pointers . . . 69
assigning function pointers . . . 141
assigning structures . . . 82
assignment expressions . . . 30
assignment in subexpressions . . . 34
assignment type conversions . . . 155
assignment, modifying . . . 31
assignment, simple . . . 30
associativity and ordering . . . 43
attributes . . . 222
`auto` declarations . . . 127

B

backspace . . . 55
base conversion (floating point) . . . 208
bell character . . . 55
binary integer constants . . . 52
binary operator grammar . . . 41
bit fields . . . 79
bitwise operators . . . 28
block . . . 104
block scope . . . 158
boolean type . . . 48
branch cuts . . . 207
branches of conditional expression . . . 38
`break` statement . . . 106
bytes . . . 12

C

call-by-value . . . 139
calling function pointers . . . 141
calling functions . . . 139
carriage return in source . . . 18
case labels in initializers . . . 122
case of letters in identifiers . . . 19
case ranges . . . 113
cast . . . 155
cast to a union . . . 85
`CHAR_BIT` . . . 191
character constants . . . 55
character set . . . 17
cloning . . . 118
code point . . . 57
combining variable declarations . . . 121
comma operator . . . 39
command-line parameters . . . 143
commenting out code . . . 188
comments . . . 18
common type . . . 157
comparison, pointer . . . 66
comparisons . . . 26

compatible types 154
compilation module 211
compiler options for integer overflow 24
compiling 11
complete example program 9
complex arithmetic in floating-point calculations 206
complex conjugation 50
complex constants 54
complex numbers 49
compound statement 104
compound type 74
computed gotos 117
computed includes 166
concatenation 173
conditional expression 37
conditional group 185
conditionals 184
conjunction operator 36
conjunction, bitwise 28
const fields 80
`const` variables and fields 128
constant data types, integer 52
constants 52
constants, character 55
constants, floating-point 53
constants, imaginary 54
constants, integer 52
constants, string 56
constants, wide character 58
constants, wide string 59
constructing array values 96
constructors, structure 86
continuation of lines 21
`continue` statement 109
controlling macro 166
conversion between pointers and integers 72
conversions, type 155
counting vowels and punctuation 111
crash 5

D

declararing functions 137
declaration of variables 120
declarations inside expressions 118
declarations, combining 121
declarations, `extern` 126
declaring an array 91
declaring arrays and pointers 120
declaring function pointers 140
decrement operator 32
decrementing pointers 70
`defined` 187
defining functions 132
dereferencing pointers 63
designated initializers 122
diagnostic 189
digraphs 221
directive line 160
directive name 160
directives 160
disjunction operator 36
disjunction, bitwise 28
division by zero 26
division operator 22
`do–while` statement 106
downward funargs 147
drawbacks of pointer arithmetic 72
Duff's device 112
dynamic memory allocation 76

E

elements of arrays 91
empty macro arguments 171
enumeration types 99
enumerator 99
environment variables 143
equal operator 26
error recovery (floating point) 200
escape (ASCII character) 55
escape sequence 55
exact floating-point arithmetic 196
exact specification of floating-point constants 201
example program, complete 9
exception flags (floating point) 195
executable file 11
execution control expressions 36
exit status 142
`EXIT_FAILURE` 142
`EXIT_SUCCESS` 142
expansion of arguments 183
explicit type conversion 155
expression statement 103
expression, conditional 37
expressions containing statements 118
expressions, execution control 36
`extern` declarations 126
extern inline function 150

F

failure 142
Fibonacci function, iterative 6
Fibonacci function, recursive 3
field offset 77
fields in structures 74
file-scope variables 125
file-scope variables, allocating 126
first-class object 94
flexible array fields 81
floating arithmetic exception flags 195
floating overflow 195
floating point example 13

floating underflow 194, 195
floating-point arithmetic invalid optimizations 195
floating-point arithmetic with complex numbers 206
floating-point arithmetic, exact 196
floating-point constants 53
floating-point constants, exact specification of 201
floating-point error recovery 200
floating-point fused multiply-add 199
floating-point infinity 201
floating-point machine epsilon 204
floating-point NaN 202
floating-point representations 193
floating-point round-trip base conversion 208
floating-point rounding control 204
floating-point rounding issues 197
floating-point scaling by powers of the base 203
floating-point signed zeros 203
floating-point significance loss 198
floating-point types 48
floating-point values, special 194
`for` statement 107
formfeed 55
formfeed in source 18
forward declaration 134
forward function declarations 133
full expression 44
function body 4
function call semantics 139
function calls 139
function declarations 137
function declarations, forward 133
function definitions 132
function definitions, inline 149
function definitions, old-style 151
function header 4
function parameter lists, variable length 145
function parameter variables 132
function pointers 140
function pointers, assigning 141
function pointers, calling 141
function pointers, declaring 140
function prototype 137
function prototype scope 158
function scope 158
function-like macros 169
functions 132
functions that accept variable-length arrays ... 144
functions with array parameters 134
functions, nested 147
functions, static 134
fused multiply-add in floating-point computations 199

G

global variables 125
`goto` statement 114
goto with computed label 117
grammar, binary operator 41
greater-or-equal operator 26
greater-than operator 26
guard macro 166

H

handler (for signal) 224
header file 162
hexadecimal floating constants 54
hexadecimal floating-point constants 201

I

identifiers 19, 161
IEEE 754-2008 Standard 193
`if` statement 103
`if...else` statement 104
imaginary constants 54
including just once 165
incomplete array types 93
incomplete types 87
increment operator 32
incrementing pointers 70
infinity in floating-point arithmetic 201
initializers 121
initializers with labeled elements 122
inline function definitions 149
inline functions, omission of 150
integer arithmetic 22
integer constant data types 52
integer constants 52
integer overflow 23
integer overflow, compiler options 24
integer ranges 192
integer representations 191
integer types 46
internal block 104
`intptr_t` 70
invalid optimizations in floating-point arithmetic 195
iteration 105
iterative Fibonacci function 6

K

K&R-style function definitions 151
keyword 19

L

label 114
labeled elements in initializers 122
labels as values 117
layout of structures 77
left-associative 41
length-zero arrays 81
less-or-equal operator 26
less-than operator 26
lexical syntax 17
limitations of C arrays 94
line continuation 21
line control 189
linefeed in source 18
link 211
linking object files 211
local labels 116
local variables 124
local variables in macros 182
logical operators 36
loop statements 105
low level pointer arithmetic 69
lvalues 31

M

machine epsilon (floating point) 204
macro argument expansion 183
macro arguments and directives 178
macros 167
macros in include 166
macros with arguments 170
macros with variable arguments 174
macros, local labels 116
macros, local variables in 182
macros, types of arguments 124
`main` function 142
make rules 211
manifest constants 167
maximum integer values 192
memory allocation, dynamic 76
memory organization 12
minimum integer values 192
modifying assignment 31
modulus 25
multidimensional arrays 95
multiplication operator 22

N

NaN in floating-point arithmetic 202
NaNs-always-propagate rule 202
negation operator 22
negation operator, logical 36
negation, bitwise 28
nested block 104
nested functions 147
newline 55
newline in source 18
not a number 202
not-equal operator 26
null directive 190
null pointers 64
null statement 113
numbers, preprocessing 161
numeric comparisons 26

O

object file 211
object-like macro 167
offset of structure fields 77
old-style function definitions 151
omitting types in declarations 127
operand execution ordering 43
operand ordering 45
operand promotions 157
operator precedence 41
operator, addition 22
operator, comma 39
operator, decrement 32
operator, division 22
operator, equal 26
operator, greater-or-equal 26
operator, greater-than 26
operator, increment 32
operator, less-or-equal 26
operator, less-than 26
operator, multiplication 22
operator, negation 22
operator, not-equal 26
operator, postdecrement 33
operator, postincrement 33
operator, remainder 25
operator, subtraction 22
operators 20
operators, arithmetic 22
operators, assignment 30
operators, bitwise 28
operators, comparison 26
operators, logical 36
operators, shift 26
optimization and ordering 45
order of execution 43
ordering and optimization 45
ordering and postincrement 44
ordering of operands 43, 45

overflow, compiler options 24
overflow, floating 195
overflow, integer 23
overlaying structures 82

P

packed structures 78
parameter forward declaration 145
parameter list 132
parameter variables in functions 132
parameters lists, variable length 145
parameters, command-line 143
parentheses in macro bodies 179
pitfalls of macros 179
pointer arithmetic 67
pointer arithmetic, drawbacks 72
pointer arithmetic, low-level 69
pointer comparison 66
pointer dereferencing 63
pointer increment and decrement 70
pointer type conversion 218
pointer-integer conversion 72
pointers 62
pointers and arrays 69
pointers to functions 140
pointers, declaring 120
pointers, null 64
pointers, `restrict`-qualified 130
pointers, void 65
postdecrement expression 33
postincrement and ordering 44
postincrement expression 33
precedence, operator 41
predecrement expression 32
predefined macros 175
preincrement expression 32
preprocessing 160
preprocessing directives 160
preprocessing numbers 161
preprocessing operators 160
preprocessing tokens 161
prescan of macro arguments 183
primitive types 46
`printf` 10
problems with macros 179
promotion of arguments 156
prototype of a function 137
punctuation 20

Q

QNaN 194
quote directories 165

R

ranges in case statements 113
ranges of integer types 192
recursion 4
recursion, drawbacks of 5
recursive Fibonacci function 3
redefining macros 178
referencing structure fields 75
`register` declarations 127
remainder operator 25
reordering of operands 43
repeated inclusion 165
reporting errors 189
reporting warnings 189
representation of floating-point numbers 193
representation of integers 191
reserved words 19
`restrict` pointers 130
return (ASCII character) 55
`return` statement 105
returning values from `main` 142
round-trip base conversion 208
rounding 196
rounding control (floating point) 204
rounding issues (floating point) 197

S

scaling floating point by powers of the base 203
scope 158
segmentation fault 5
self-reference 182
semantics of function calls 139
semicolons (after macro calls) 180
sequence points 44
shift count 26
shift operators 26
side effect in ?: 38
side effects (in macro arguments) 181
`SIGBUS` 224
`SIGFPE` 224
signal 224
signed types 46
signed zeros in floating-point arithmetic 203
significance loss (floating point) 198
`SIGSEGV` 224
simple assignment 30
size of type 60
SNaN 194
space character in source 18
special floating-point values 194
stack 5
stack frame 5
stack overflow 5
standard output 10
statement, `break` 106
statement, `continue` 109
statement, `do–while` 106

statement, expression 103
statement, `for` 107
statement, `goto` 114
statement, `if` 103
statement, `if...else` 104
statement, null 113
statement, `return` 105
statement, `switch` 110
statement, `while` 105
statements 103
statements inside expressions 118
statements, loop 105
static assertions 216
static function, declaration 134
static functions 134
static local variables 125
sticky exception flags (floating point) 195
storage organization 12
string 92
string constants 56
stringification 171
structure assignment 82
structure constructors 86
structure field offset 77
structure fields, constant 80
structure fields, referencing 75
structure layout 77
structures 74
structures, overlaying 82
structures, unnamed 86
subexpressions, assignment in 34
subnormal numbers 194
subtraction operator 22
success 142
`switch` statement 110
symbolic constants 167
system header files 162

T

tab (ASCII character) 55
tab character in source 18
tentative definition 127
thunks 147
token 17
token concatenation 173
token pasting 173
truncation 47
truth value 26
two's-complement representation 191
twosum 200
type alignment 217
type conversion, pointer 218
type conversions 155
type designator 51
type size 60
type tags 88
type, boolean 48
type, void 50
typedef names 101
types of integer constants 52
types, compatible 154
types, complex 49
types, enumeration 99
types, floating-point 48
types, incomplete 87
types, integer 46
types, primitive 46
types, signed 46
types, unsigned 46

U

`uintptr_t` 72
undefining macros 178
underflow, floating 194, 195
underscores in variables in macros 182
Unicode 17
Unicode character codes 57
union, casting to a 85
unions 83
unions, unnamed 86
universal character names 57
unnamed structures 86
unnamed unions 86
unsafe macros 181
unsigned types 46
UTF-8 String Constants 57

V

`va_copy` 146
`va_end` 145
`va_list` 145
`va_start` 145
variable declarations 120
variable declarations, combining 121
variable number of arguments 174
variable-length array parameters 144
variable-length arrays 97
variable-length parameter lists 145
variables 120
variables, `const` 128
variables, file-scope 125
variables, global 125
variables, local 124
variables, local, in macros 182
variables, static local 125
variables, `volatile` 129
variadic function 145
variadic macros 174
vertical tab 55
vertical tab in source 18
void pointers 65
void type 50
`volatile` variables and fields 129

W

`while` statement 105
whitespace characters in source files 18
wide character constants 58
wide string constants........................... 59
wrapper `#ifndef` 165

Z

zero, division by 26
zero-length arrays.............................. 81
zero-origin indexing 14