

# Introduction to C

David Chisnall

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Brief History of C . . . . .	2
1.2	Key Features . . . . .	2
1.3	Why Learn C? . . . . .	3
<b>2</b>	<b>Building</b>	<b>3</b>
2.1	The C Build Process . . . . .	3
2.2	Key Points . . . . .	4
<b>3</b>	<b>C Memory Model</b>	<b>4</b>
3.1	Static Memory . . . . .	4
3.2	Heap Memory . . . . .	5
3.3	Stack Memory . . . . .	5
<b>4</b>	<b>Scoping</b>	<b>5</b>
4.1	Blocks . . . . .	5
4.2	Globals . . . . .	6
<b>5</b>	<b>Pointers</b>	<b>6</b>
5.1	Pointers in C . . . . .	7
<b>6</b>	<b>The C Type System</b>	<b>7</b>
6.1	Primitive Types . . . . .	7
6.1.1	Integers . . . . .	8
6.1.2	Floating Point . . . . .	8
6.2	Pointers . . . . .	9
6.3	Aggregate Types . . . . .	9
6.4	Arrays . . . . .	9
6.4.1	Structures . . . . .	10
6.5	Named Types . . . . .	11
<b>7</b>	<b>Functions</b>	<b>12</b>
7.1	The main() Function . . . . .	14
7.2	Function Pointers . . . . .	14
<b>8</b>	<b>Flow Control</b>	<b>15</b>
8.1	Conditionals . . . . .	15
8.2	Switch Statements . . . . .	15
8.3	Loops . . . . .	15

8.3.1 For Loops . . . . .	16
<b>9 The Preprocessor</b>	<b>17</b>
9.1 Includes . . . . .	17
9.2 Macros . . . . .	17
9.3 Conditional Compilation . . . . .	18
<b>10 The C Standard Library</b>	<b>19</b>
<b>11 The C Standard Library</b>	<b>19</b>
11.1 Strings . . . . .	19
11.2 C I/O . . . . .	20
11.3 Read The Documentation . . . . .	21
<b>12 Make</b>	<b>22</b>

# 1 Introduction

## 1.1 Brief History of C

C was created in 1972 as a cut-down version of BCPL. It was used to port the UNICS operating system (renamed UNIX) from the PDP-7 to the PDP-11. At the time, it was common to write operating systems in the native assembly language for the target platform. UNIX was intended to be portable. Only a small amount of the code was dependent on the underlying architecture, and everything else was written in C.

The first version of the C standard was published in 1989, often referred to as ANSI C, later as ISO C or C89 to distinguish it from later versions of the specification. A set of clarifications and amendments were published in 1990. These only made minor changes to the language, and so C89 and C90 are rarely distinguished. The most recent version of the standard was published in 1999. This course covers C99.

C has a close relation to UNIX. As well as being the language used to write the operating system, it was the languages used for most userspace code as well. All of the system calls in a UNIX system are defined in terms of the C standard library function that issues them. The Single UNIX Specification—the specification that an implementation must conform to in order to use the UNIX trademark—requires a `c99` utility to exist as a C compiler.

## 1.2 Key Features

C was designed as a portable assembly language. Semantically, it is very similar to PDP-11 assembly language. For example, it defines `shift`, but not `rotate`, operations, since the PDP-11 had no rotate instruction. C shares a lot of characteristics with assembly languages. It exposes the capabilities of the hardware directly to the programmer and does not provide many convenient mechanisms for writing safe code. It does, however, provide everything you need to write these yourself.

In spite of this similarity, C is not a assembly language. Listings 1 and 2 illustrate the difference. In the x86 assembly language version, there is code for allocating space on the stack by modifying the stack pointer register, load

operations to get values into memory and then an add operation on values in registers.

Listing 1: x86 Assembly

```
1 movl    %esp, %ebp
2 subl    $24, %esp
3 movl    $12, -12(%ebp)
4 movl    8(%ebp), %edx
5 leal    -12(%ebp), %eax
6 addl    %edx, (%eax)
```

Listing 2: C version

```
1 int a = 12;
2 a += b;
```

C hides much of this detail from the programmer. It provides named, scoped, variables, where assembly languages require space to be manually allocated and tracked (although this can often be done via macros) and provides some type checking. Most importantly, it hides the details of register allocation and provides high-level flow control. In the CISC architectures common around the time of C's birth, complex flow control primitives were not uncommon, however modern CPUs rarely have more than a conditional jump and sometimes a subroutine call instruction. The biggest difference between C and any assembly language, however, is that C does not expose anything that is specific to the underlying CPU. It allows code to be recompiled across different architectures.

### 1.3 Why Learn C?

C is still widely used in industry. It has most of the advantages of an assembly language without the biggest problem with writing code in an assembly language: being tied to one platform. It is well supported on pretty much all operating systems, and is often used when implementing other languages.

A lot of modern languages, including C# and Java, inherit their syntax from C. Knowing C, therefore, makes it easier to learn these languages. Understanding C also helps programming in other languages. Seeing how constructs from high-level languages can be implemented in C is a good way of understanding their performance characteristics and how they really work.

## 2 Building

### 2.1 The C Build Process

Building a C program is a three-stage process:

1. Preprocess source files to produce a single file for the *compilation unit*.
2. Compile the preprocessed source to produce *object code*.
3. Link the object code to produce an executable (or library).

Modern C compilers have a *compiler driver* which hides these stages, but they can be run separately. On a GNU platform you will be using `gcc` as the compiler driver. The basic syntax looks something like this:

```
$ gcc -std=c99 source.c -o program
```

This compiles `source.c` as a C99 program and emits `program`, a binary executable. Be careful if your source and executable files have the same name and you use tab-completion. It is very easy to write:

```
$ gcc -std=c99 program.c -o program.c
```

This will compile `program.c` and output the resulting executable to `program.c`. You will then have lost your source file, and have to restore from an earlier backup.

## 2.2 Key Points

- The preprocessor only understands tokens.
- Files and compilation units are not the same. Programmers see files, but the compiler only sees compilation units.
- Traditional C compilers could not optimise across compilation units (LLVM can, GCC can soon).

## 3 C Memory Model

C divides memory into three regions, with different purposes.

**Static:** fixed allocations.

**Heap:** manual allocations.

**Stack:** Lexical scoping.

### 3.1 Static Memory

When a C program is run, the binary is mapped in to memory (as are any libraries it needs). A binary file in a format like ELF<sup>1</sup> contains a number of different sections. The `.text` section contains the program code, the machine instructions generated from the source code. Any static variables have space for them allocated in the `.data` section. This is mapped in to memory by the loader and typically marked as writable but not executable. Constants may be in a separate section, depending on the system.

All variables in the static data section have a fixed size. Their space is allocated by the loader when it maps the program in to the process' memory. Constants, globals, and static local variables are all in this section.

Some C compilers and loaders provide an extension to the static memory region to allow data to be allocated on a per-thread basis. The GCC extension is `__thread`, and is used in the following way:

---

<sup>1</sup>The Executable and Linking Format used in Linux, \*BSD, and most other UNIX-like systems, with Mac OS X being the most notable exception.

```
__thread int i;
```

This will put `i` in a special section in the binary. Every time a new thread is created, this section will be mapped in to memory and a machine register will be used to indicate the start of the section. Although this extension is common, it is not supported everywhere. The Darwin loader, for example, does not support thread-local storage in this way.

## 3.2 Heap Memory

The heap is the pile of memory that can be manually allocated. A heap is a tree data structure which was traditionally used for managing free memory. A modern implementation may not use this data structure, but the convention is still to refer to manually allocated memory as coming from the heap.

All memory coming from the heap is manually allocated and released, using the `malloc()` and `free()` functions. It is important to pair calls to these two functions. Manual memory management is a common source of errors in C. If you allocate memory without freeing it, you have a *memory leak*—your process will keep consuming memory until the operating system decides it has used too much and kills it. Alternatively, if you free memory before you’ve finished using it, you can get invalid data being referenced. Depending on a number of other factors, this will either cause data corruption or crashing.

## 3.3 Stack Memory

The stack is used for lexically-scoped variables. Any local variables declared in a function are allocated on the stack. Typically these are small and short-lived, while heap data is larger or longer-lived. The amount of space available on the stack can not easily be checked, so it’s typically a good idea to assume it is small.

# 4 Scoping

## 4.1 Blocks

Braces are the most instantly-recognisable feature of any C-like language. A C program is split into nested lexical scopes by braces. An open brace indicates the start of a new scope and a close brace indicates the end, as shown in Listing 3.

Listing 3: Braces are used to create blocks.

```
1 { // This is the start of a block.  
2   ... // Everything here is in the block.  
3 }
```

Any variable in C is visible from the line on which it is declared until the end of the block. Listing 4 shows an example of how the scope of variables works with blocks. Each line starting `int` creates a new variable, and each of these is valid until the closing brace matching the open brace before the declaration.

With C89, variables could only be declared at the start of blocks. This was relaxed in C99. A good rule of thumb is to observe the *principle of least scope*: a variable should be valid for no longer than it's needed.

Listing 4: Variable scoping.

```
1 {
2     int a; // a is valid from here.
3     {
4         int b; //b is valid from here
5     } // b is invalid from here.
6     int c; // c is valid from here
7     {
8         int a; // This a hides the other one.
9     }
10 } // a is invalid from here
```

Be careful about hiding variables. In the example, there are two variables called `a`. When you reference a variable, the compiler will look for the one declared the most recently. This means that the inner one will be referenced, not the outer one. A variable in C exists in the same namespace as a function name, so a variable with the same name as a function can cause problems of this nature, especially if you try to call the function in a scope where the variable is valid. Functions are always at the global scope, so any local variable will hide them.

## 4.2 Globals

Anything declared outside a block is global. If you want to access it from two compilation units, you must declare it as `extern` in one, like this:

```
extern int i;
```

This causes the compiler to assume that `i` is a valid integer variable and is declared in another compilation unit. If it is not then the linker will produce an error complaining about unknown symbols.

Scoping still applies to globals - they are only visible below where they are declared. It is important to distinguish between when a variable is *visible* and when it is *valid*. Variables declared `static` can not be accessed from another scope. They are valid for the duration of the program, but only visible in whichever lexical scope they are declared. `static` variables outside a block can only be accessed from the current compilation unit, and can not be referenced with a `extern` declaration from elsewhere. In contrast, a local (stack) variable is valid only for as long as it is visible.

## 5 Pointers

C is ‘close to the metal’ and as such each variable is a name for a block of memory. When you say `int i;` in a function you are saying ‘allocate enough space for an integer on the stack’ and any subsequent reference to `i` is simply translated in to an offset from the start of the current stack frame. Once a variable has become valid its location in memory does not change until it becomes invalid.

<b>C Pointers</b>	<b>Java References</b>
Numbers indicating an address in memory	Opaque type with only assign (copy) operation defined.
Can be created as the result of arithmetic	Can only be created from valid objects.
Can point to invalid locations	Can only point to valid objects or <code>NULL</code> .
Point to memory. User must know type of destination.	Point to objects which know their own type.

Table 1: Pointers versus references

A *pointer* is the (numerical) address of a region of memory. Each variable can have its address taken to give a pointer. Pointers are similar to references in a language like Smalltalk or Java, but with some important differences, highlighted in Table 1

Java implements a variant of Smalltalk's *object memory model*. C exposes a *flat memory model*.

## 5.1 Pointers in C

C uses pointers **a lot**. When you call `malloc()` to get memory from the heap, you are given a pointer as the return value. Therefore, all data on the heap is accessed via pointers. They are also used for aliasing, for example when implementing things like linked lists.

Because it is the responsibility of the pointer variable to know the type, they can be used to bypass the type system entirely. You can access any blob of memory as any type (although doing so may have bizarre or undefined behaviour).

Arrays, discussed in more detail in Section ??, are a very thin layer of syntactic sugar around pointers in C. In most places, pointers and arrays can be used interchangeably.

Pointers are just a kind of variable, so they can point to other pointers, which can point to other pointers, and so on. Pointers are the reason C is so powerful, and the cause of most bugs in C code.

# 6 The C Type System

## 6.1 Primitive Types

C defines four categories of primitive types:

- Integers.
- Floating point values.
- Pointers.
- Compound types.

Name	Minimum Size	Typical Size	Notes
<code>char</code>	8	8	Used for characters
<code>short</code>	16	16	
<code>int</code>	16	32 or 64	Machine word size
<code>long</code>	16	32 or 64	At least pointer size
<code>long long</code>	64	64	

Table 2: Integer sizes in C.

### 6.1.1 Integers

C defines five integer types. These don't have fixed sizes, but each must be at least as big as the smaller size. Each type, described in Table 2 must be at least as big as the preceding one. For example an `int` can not be smaller than a `short` although they can be the same size. The size of `int`, `long` and pointers vary a lot between different systems. It is common to come across shorthand definitions like ILP64 or LP64. The first of these specifies that all three are 64-bits. When only LP is specified then `ints` are taken as being half the size. LP64 is a common model on new systems, where `ints` are 32-bits and `long` and pointer values are both 64-bits. On older platforms LP32 was common, where `ints` were 16-bit but `long` and pointer values were 32-bits. On embedded systems it is common for all three to be 16-bits.

Integers can be `signed` or `unsigned`. The default is `signed` for all except `char` (which has no portable default).

### 6.1.2 Floating Point

The C standard defines two floating point types, with one extra type being a common extension:

- `float` is single-precision (32 bit).
- `double` is double-precision (64-bit).
- `long double` is nonstandard (x86 80-bit, PowerPC 128-bit).

All floating point types are signed. Floating point code for the x87 is always done in 80-bit operations and so using the `long double` means that the same format is used for storage as for calculation. Modern compilers for x86 chips can emit SSE code instead of x87 code, and in this case 32-bit or 64-bit arithmetic will be used.

You can find the size of any type using the `sizeof()` pseudofunction. This will be replaced by the compiler (note: not the preprocessor—you can not use `sizeof()` in preprocessor conditionals) with the size of the argument. This can be done in two ways:

```
int a = sizeof(int);
int b = sizeof(a);
```

Both of these will be set to the size of an integer. Note that `sizeof()` is always evaluated at runtime<sup>2</sup> and so can only be used to find sizes which are known to the compiler. It can not be used to find the size of a block of memory indicated by a pointer.

<sup>2</sup>Well, almost always. C99 variable length arrays are a special case.

## 6.2 Pointers

Pointers are very similar to integers. In early versions of C, pointers and integers were used interchangeably. A pointer is basically an integer which stores an address in memory, rather than a unitless value. There are two important operations used with pointers; `&` and `*`.

The address-of operator, `&`, is a prefix operator which gives the address of a variable. Writing `&foo` gives a pointer value representing the location of the `foo` variable in memory.

The dereferencing operator, `*`, is the inverse of this. It is a prefix operator taking a pointer as the argument and returning the variable that it represents. Recall that variables in C are just regions of memory—this operator returns the region of memory, not the name of the variable. Writing `*(&foo)` is equivalent to just writing `foo`.

Listing 5: Use of pointers

```
1 int AnInt = 12;
2 int *PointerToAnInt = &AnInt;
3 // Sets AnInt to 42
4 *PointerToAnInt = 42;
```

Be careful with regard to scope and pointers. In Listing 5, the pointer has the same scope as the pointee. If you return `PointerToAnInt`, or assign it to something outside this scope then it will become invalid when `AnInt` goes out of scope. This will leave you with a pointer to some location on the stack that is now invalid.

Pointers are declared to point to a specific type. In Listing 5 the pointer `PointerToAnInt` was declared as a pointer to an `int` by declaring the variable's type as `int*`. There is one extra type available for pointers; `void*`. A pointer with this type is a pointer to anonymous memory. The compiler has no knowledge of the memory's type and so dereferencing it is not allowed. The return type of `malloc()` is a `void*`, which must then be cast (typically implicitly) to another type.

Adding 1 to a pointer adds the size of the pointee to the numerical value.

## 6.3 Aggregate Types

### 6.4 Arrays

Arrays in C are just a block of memory. Declaring an array on the stack or the static data region is just a way of getting a pointer to some space on that region. Arrays can not be resized. Once they are declared, their size is fixed for the duration of their lifespan. C99 allows variable-length arrays to be created on the stack. These have their size determined when they enter scope, like this:

```
// An array of 12 integers
int a[12];
// A variable length array
int b[c + 42];
```

Note: The size of `b` is set when the array is declared. Changing `c` afterwards does not resize it.

Array names are pointers. Elements in an array can be accessed either via the subscript notation or by pointer arithmetic. Listing 6 shows this equivalence. The second element in the array is set twice (to the same value, both times). The first time uses the subscript notation, while the second uses pointer arithmetic.

Listing 6: Arrays and Pointers

```
1 // Array of 42 integers
2 int a[42];
3 // Pointer to first element in the array
4 int *b = a;
5 // Set the first element of the array
6 a[0] = 12;
7 // Set the second element of the array
8 a[1] = 13;
9 // Set the second element in the array a different way
10 *(b+1) = 13;
```

`a[n]` and `*(a+n)` are equivalent. Both give the `n`th element in the array. `&a[n]` and `(a+n)` are also equivalent, both giving the address of the `n`th element.

#### 6.4.1 Structures

Structures are records with a fixed layout. Each structure has one or more fields, with a fixed type, identified by a name. When a structure is compiled, this name is replaced with an offset from the start of the structure. Listing 7 shows how structures are created. The first defines a `Point` as being two integers, representing `x` and `y` coordinates. The second is a compound structure. Structures can contain any other type as their elements, including new structures or fixed-size arrays.

Listing 7: Creating a Structure

```
1 //Points have x and y coordinates
2 struct Point
3 {
4     int x;
5     int y;
6 }; // Don't forget this semicolon
7 // Rectangles are defined by two corners
8 struct Rectangle
9 {
10    // Structures can contain other structures
11    struct Point topLeft;
12    struct Point bottomRight;
13};
```

When using a structure, you specify the name of the structure variable, then a dot and then the name of the field, as shown in Listing 8. If the variable is a pointer to a structure, rather than a structure, you can use an arrow (`->`) instead of dereferencing the pointer and using a dot. The following are equivalent:

```
(*aStructPointer).x;
aStructPointer->x;
```

Listing 8: Using Structures

```

1 // Structures can be initialised by listing the elements in
  order.
2 struct Point origin = { 0, 0 };
3 // This is a GCC extension, but it makes code more readable.
4 struct Point anotherPoint = { .y = 12, .x = 42 };
5 // You reference structure elements using a dot.
6 int anX = anotherPoint.x;
7 // Or with an arrow if it's a pointer
8 struct Point aPointer = &anotherPoint;
9 // This won't change the value of anX
10 anX = aPointer->x;

```

## 6.5 Named Types

It is often convenient to give a name to a compound type, or to a primitive type being used in a certain way. The `typedef` directive allows this. It associates a new name with an existing type. Listing 9 is an example of this. The `struct Point` type is given the name `point_t`. This new name can then be used anywhere where `struct Point` would have been valid.

Listing 9: Using typedef

```

1 // Structures are often typedef'd so you don't have to keep
  typing struct
2 typedef struct Point point_t;
3 point_t aPoint = {1,2};
4 // Pointers can be typedef'd too
5 typedef struct Point* pointpointer;
6 typedef struct AnOpqaueType* opaque;

```

Typedefs are also commonly used for creating opaque types. These rely on the fact that C only requires the type of a pointer to be known when the pointer is dereferenced, not when it is declared. The following, for example, is always valid:

```
struct wibble *foo;
```

Whether or not `struct wibble` has been defined anywhere does not matter. If you tried to access one of the elements of this structure, however, then you would require the structure definition to be visible to the compiler.

It is common to use header files in C for describing interfaces. If you put a typedef like this in a header file, and then the definition of the structure in the implementation file then other source files can include the header file and use the functions defined in it which use this type, but can not easily access the data in the structure directly. This is very commonly used by libraries, since it allows the implementation of the structure to be changed without needing to modify or recompile anything outside the library which uses it.

Note that typedefs are primarily for documentation. The compiler does not check typedefs. The following is perfectly valid C code:

```
typedef int feet;
```

```
typedef int meters;
meters length = 1;
feet width = length;
```

This will not produce an error, even though it's clear to a human that `feet` and `meters` should be different.

## 7 Functions

Functions are the basic flow control primitive in C. They are not functions in the pure mathematical sense, since they can have side effects and their results can depend on things other than their arguments. They can take an arbitrary number of parameters and can return a value. Functions declared with the return type `void` do not return anything.

Listing 10 shows the declaration of two functions. The first simply adds the two arguments together and the second adds a value to an internal counter. Note that you only use a `return` statement in a function which returns a value.

The `static` variable in the `count()` function is only visible within this function. It can not be accessed from outside, but it persists for the entire duration of the program.

Listing 10: Creating functions

```
1 // Function returning an int, takes two ints as parameters
2 int add(int a, int b)
3 {
4     return a + b;
5 }
6 // No return value
7 void count(int a)
8 {
9     // Only one instance of i for the whole program.
10    static i = 0;
11    i = i + a;
12 }
```

Functions are called by writing their name then their arguments in brackets, as shown in Listing 11. A function taking no arguments is still called like this, but with nothing between the brackets.

Listing 11: Calling Functions

```
1 int a = add(12, 42);
2 count(a);
3 // You don't have to use the result of a function
4 add(a, 12);
```

Functions can be called with more parameters than they declare—extras are ignored, and the compiler may issue a warning. This is due to the way in which C handles parameter passing. Figure 1 shows a simplified call stack from a C program where the function in Listing 12 is being called. The function

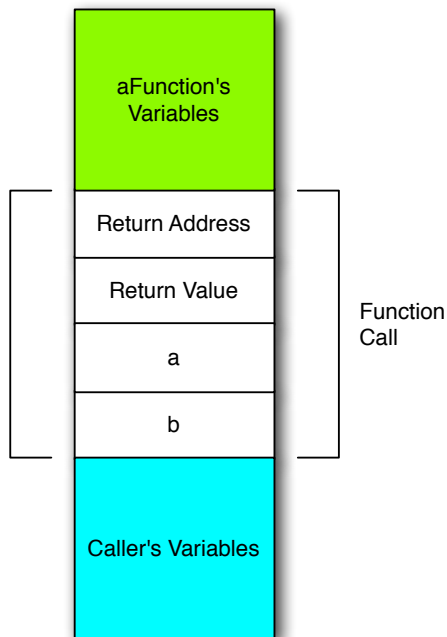


Figure 1: The call stack in C.

Listing 12: Call stack example.

```

1 int aFunction(int a, int b)
2 {
3     int c = a + b;
4     // Do other stuff with c
5     return c;
6 }

```

parameters are pushed from right to left on to the stack, then some space is left for the return value and finally the return address is pushed<sup>3</sup>.

When control enters `aFunction()`, the return address, space for the return value, and first argument are all fixed offsets from the top of the stack. The function will then allocate space for its local variables by modifying the register which points to the top of the stack. If the caller had passed more parameters to the function than were required then these will be further down the stack than the called function knows to look, so they will be ignored.

This is used when implementing *variadic functions*, described later, where the function takes a variable number of arguments. Typically, these take a NULL pointer as their last argument, and keep looking down the stack until they encounter a 0 value.

<sup>3</sup>The order of the last two is implementation-dependent and the return value is often stored in a register, not on the stack. On some architectures, such as SPARC, the return address is also stored in a register, and the first few parameters can also be passed in registers.

## 7.1 The `main()` Function

The `main()` function is the entry point for any C program. There is exactly one per program. Listing 13 shows the skeleton of such a function.

Listing 13: A simple `main()` function

```
1 int main(int argc, char *argv[], char *envp[])
2 {
3     // Do stuff
4     return EXIT_SUCCESS;
5 }
```

The parameters are as follows:

- `argc` is the number of command-line arguments.
- `argv` is an array of strings representing arguments.
- `envp` is a `NULL`-terminated array of environment variables.

The `envp` argument is not standard C, but is in the POSIX standard. Most operating systems allow it, although there are other, more convenient, ways of getting at argument variables, so it is rarely used.

The return value of the `main()` function is program exit code. It should be either `EXIT_SUCCESS` or `EXIT_FAILURE`, conventionally set to 0 and 1 respectively. By convention, many C functions return 0 (false) for success or a non-zero (true) value for success.

## 7.2 Function Pointers

The name of any function pointer can be used as a pointer to that function. A function definition can be thought of as creating a constant function pointer variable and defining it. You can also create other function pointer variables and set them to point to existing functions.

Listing 14 shows how function pointers can be declared and used. Calling a function via a pointer uses exactly the same syntax as calling it directly.

Listing 14: Using function pointers.

```
1 // Declare a type for functions mapping two ints to one int
2 typedef int (*IntFunction)(int, int);
3 // Set this pointer to add()
4 IntFunction addfn = add;
5 // Call the function via the pointer
6 int a = addfn(12, 42);
7 // Another function pointer without using the typedef
8 int (*fnptr)(int, int) = addfn;
```

## 8 Flow Control

### 8.1 Conditionals

The most basic flow control primitive in C is the `if` statement. This executes a statement if a condition holds. The statement is typically a block. Listing 15 shows an `if` statement with an `else` clause. The first block will be executed if `a` is greater than zero, otherwise the second block will run.

Note that C has no boolean type. Any integer or pointer type can be used as if it were a boolean, with zero representing false and every other value representing true.

Listing 15: An if statement.

```
1  if (a > 0)
2  {
3      // Do something.
4  }
5  else
6  {
7      // Do something else
8  }
```

### 8.2 Switch Statements

A slightly more complex conditional is the `switch` statement. This switches between a list of options. In implementation it is typically a jump table or similar. Listing 16 shows an example `switch` statement. The argument, `a`, must be an enumerated type (an integer, a pointer, or an `enum`). Each of the `case` labels must be a constant.

There is automatic *fall-through* in `case` statements. After executing the body of each one, execution will continue to the next one. In this example, the comment “Do something if `a` is 0 or 1” will be reached if `a` is 0 because the case for 0 is above the case for 1. The case for 1 ends with a `break` statement, which causes execution to jump to the end of the `switch` statement. Remember to end all case statements with a `break` statement unless you want fall-through.

### 8.3 Loops

In general, there are two attributes used to describe loops in a programming language. A loop is either a pre- or post-test loop and either a while or until loop. A loop repeats either while or until a condition holds, and it either tests this condition at the start or the end of each loop iteration. C only has while loops, however an until loop can trivially be implemented in terms of a while loop by negating the condition. C includes both pre- and post-test forms of a while loop.

Listing 17 shows a pre-test loop in C. This continues to loop as long as condition is non-zero. As with the `switch` statement, you can jump out of an executing loop with the `break` statement. Another alternative to normal flow control is the `continue` statement. This jumps to the end of the current loop

Listing 16: A switch statement.

```
1 switch(a)
2 {
3     case 0:
4         // Do something if a is 0
5     case 1:
6         // Do something is a is 0 or 1
7         break;
8     case 2:
9         // Do something if a is 2
10        break;
11    default:
12        // Do something if a is some other value
13 }
```

iteration, skipping all of the statements before the end. If the condition still holds, the loop will then begin again, otherwise the loop will exit.

Listing 17: A pre-test loop.

```
1 while (condition)
2 {
3     // loop until condition is 0
4 }
```

The other type of loop is the post-test loop, known in C as the `do...while` loop, shown in Listing 18. Note the semicolon after the while at the end of this loop. This is very easy to accidentally omit.

Listing 18: A post-test loop in C.

```
1 do
2 {
3     // loop at least once.
4     // Keep looping until condition is 0
5 } while (condition); // Note this semicolon!
```

### 8.3.1 For Loops

As well as the two flavours of while loop, C includes a `for` loop. In other languages, a for loop is an iterative loop covering a range of integers. In C, it is a simple bit of syntactic sugar on top of a while loop. Listing 19 shows a `for` loop and the equivalent `while` loop. Although they are no more expressive than `while` loops, `for` loops are often simpler to use. C99 introduced the ability to declare loop variables in the head of the loop (`i` is declared in this way in the example). Variables declared here are only in scope for the body of the loop. This is often a good way of abiding by the principle of least scope.

Listing 19: A for loop.

```
1 for (int i=0 ; i<100 ; i++)
2 {
3     // Loop 100 times.
4 }
5 {
6     int i = 0;
7     while(i<100)
8     {
9         // Loop 100 times
10        i++;
11    }
12 }
```

## 9 The Preprocessor

The C preprocessor is the first step in building a C program. It treats the source file as a stream of tokens and performs substitutions. It can also be used to import the contents of other files. The last thing the preprocessor does is remove comments.

### 9.1 Includes

The `#include` directive inserts the contents of the specified file. This has two forms, one for *local* and one for *system* includes. If the file name is specified in speech marks, it is treated as a local include and the file is searched for in the current directory. If it is in angle brackets then it is a system include and is searched for in the system include path. The system include path depends on the operating system, but on UNIX-like systems it typically includes `/usr/include` and `/usr/local/include`. Listing 20 shows both uses.

Listing 20: Including headers.

```
1 // Include a header from this project.
2 #include "myheader.h"
3 // Include the standard I/O header
4 #include <stdio.h>
```

### 9.2 Macros

There are two kinds of macros understood by the preprocessor. Variable-like macros and function-like macros. A variable-like macro is a single token which is replaced by one or more other tokens. A function-like macro takes some arguments and includes them in the substitution. Listing 21 shows two examples of macros and their use.

Note that macros, even function-like macros, are simple substitutions. The `ADD()` macro could reference variables not visible when the macro is declared and this would only be an error if they were not visible when the macro was used.

Listing 21: Preprocessor macros.

```

1 // Macro used for symbolic constant.
2 #define BUFFER_SIZE 16
3 // Macro used instead of a function
4 #define ADD(x, y) (x + y)
5 int i[BUFFER_SIZE]; // becomes: int i[16];
6 int j = ADD(1, m); // becomes: int j = (1 + m);

```

### 9.3 Conditional Compilation

One common use for the preprocessor is conditional compilation. The preprocessor removes parts of the source file if a condition is not met. The `#if` preprocessor directive is analogous to the C `if` statement, but executed at compile time rather than run time. When the code is compiled, it will only include the body of the `#if` statement if the condition is true. Note that since this is a preprocessor directive, the condition can only be in terms of preprocessor tokens. You can not reference C variables, for example. Listing 22 shows two examples of this. The first uses 0 as the condition. This means that the body of the code will never be compiled. This is sometimes useful for debugging, where wrapping some potentially-broken code in a block like this removes it from being compiled.

The second case compiles things differently if the `UNIX` preprocessor directive is defined. You can either define this in an include file or with the `-D` option to the compiler. This allows you to have two code paths, one for `UNIX` and one for other platforms, in the same source file.

Listing 22: Conditional compilation with the preprocessor.

```

1 #if 0
2 // Never compile this bit
3 #else
4 // Always compile this bit
5 #endif
6 #ifdef UNIX
7 // Some code for UNIX platforms
8 #else
9 // Code for weird platforms
10 #endif

```

Conditional compilation is often used for protecting headers. Including a header more than once can cause problems, and since headers can include other headers it's often difficult to tell whether this has happened. The preprocessor pattern shown in Listing 23 solves this problem by ensuring that the header will only be included once. The first time the header is included, the `__MY_HEADER_INCLUDED__` macro will be defined. The second time, since this macro is already defined, the `#ifndef` (if not defined) directive will prevent it from being included.

Listing 23: Protecting a header

```
1 #ifndef __MY_HEADER_INCLUDED__
2 #define __MY_HEADER_INCLUDED__
3 // Header file contents here
4 #endif
```

## 10 The C Standard Library

The vast majority of the C standard defines the C standard library, not the language. The Single UNIX Specification defines even more functions which must be present. You can find a complete copy of the latest version of the SUS here:

<http://opengroup.org/onlinepubs/000095399/>

This lists a number of header files that must exist and what they must contain. You can find similar information using the `man` utility on any \*NIX system.

Header files typically contain *function prototypes*. These are function declarations with no body. An example would be the `read()` function, declared in `<unistd.h>` like this:

```
size_t read(int fildes, void *buf, size_t nbyte);
```

Note the semicolon at the end of the line, rather than a function body. This tells the compiler that a function with this name exists, and that it takes these arguments, but it does not provide the definition. The definition will be provided when the program is linked against the C standard library (`libc`). Other header files may contain similar prototypes for functions declared in other libraries.

## 11 The C Standard Library

Although C is a small language, much of its power comes from libraries. The standard C library contains a lot of useful routines and, since C is the de-facto ABI standard for UNIX-like systems there are a huge number of third-party libraries available.

### 11.1 Strings

C has no notion of strings in the language. The `char*` type is used for strings; they are just pointers to blocks of memory containing characters and ending with the NULL character. The `<strings.h>` header defines a lot of functions for dealing with strings. See the SUS for a full description:

<http://opengroup.org/onlinepubs/000095399/basedefs/string.h.html>

The most common operations on strings are copying, comparing, and finding the length of them. The `strlen()` function returns the length of a string. You can use this with `malloc()` to create enough space for a copy of the string:

```
const char *aString = "This is a string";
char *copy = malloc(strlen(aString) + 1);
```

Note the `+ 1` in this. A C string needs one byte more than the length of the string in order to store the zero byte at the end which indicates where it

finishes. The `strlen()` function simply scans along the string until it finds a 0 byte.

Copying a string can be done in one of three ways:

```
char *copy1 = strdup(aString);
char *copy2 = malloc(strlen(aString) + 1);
strcpy(copy2, aString);
char *copy3 = malloc(strlen(aString) + 1);
memcpy(copy3, aString, strlen(aString) + 1);
```

The first allocates enough space for the string and duplicates it in a single operation. The second two both allocate space and perform the copy in two operations. This particular use of the `strcpy()` function is safe, however use of this function is discouraged in general, since it is very hard to use safely. There is a portable alternative, `strncpy()` which is slightly safer. This takes the length of the destination as a third argument and won't copy more characters than there is space for. This still has some security problems, however. The safe version is `strncpy()`, which makes accidental truncation harder, however this is not found in GNU libc.

Comparing C strings is done with the `strcmp()` function. This returns a value which is less than, equal to, or greater than zero. This is intended to be used with an `if` statement, like this:

```
if (strcmp(a, b) == 0)
{
    //Two strings are identical
}
```

By using `<` or `>` instead of `==` you can test for ordering of the two strings. All of the functions declared in this header respect the locale settings for the platform, and so can return different values depending on this. For example, strings containing characters with accents will be ordered differently in French, Spanish and English locales.

## 11.2 C I/O

C regards everything outside the program as a file. When you communicate with the outside world in any way, it is via things that look like files. The terminal in which a C program runs is treated as a file with three file descriptors; the standard input, output and error handles, named `stdin`, `stdout` and `stderr` respectively. These can all be redirected elsewhere, so may actually be files in some cases.

Input/output operations in C are mostly defined in two header files, `<stdio.h>` and `<fcntl.h>`. Two of the most flexible functions in C are defined in the standard I/O header; `printf()` and `scanf()`. These are both examples of variadic functions. The first argument for each is a *format string*, which tells the function what the other arguments are.

Listing 24 shows a simple example use of `printf`. The function will scan along the string in the first argument and output each character to the standard output. When it encounters a `%` character it reads the next character to determine the format of the next argument and outputs it. In this example, `%d` indicates that the second argument is an `int` which should be printed as a decimal number. The `%f` string indicates that the third argument is a `double` which should

be written as a decimal and the final argument, indicated by `%x` is an `int` which should be printed in hexadecimal.

Listing 24: Using `printf`

```
1 printf("An integer: %d\nA float: %f\nhexadecimal integer: %x\n", 12, 64.4, 42);
```

The `scanf()` function is the inverse of `printf`. It takes a similar format string, but treats each argument as a pointer to a variable with the specified type and writes the scanned value there. You can write simple parsers in C just by using `scanf`, for example Listing 25 parses a date using this function. It will read from the input expecting three numbers separated by slashes and fill in the three variables, `day`, `month` and `year` with the parsed values.

Listing 25: Parsing a date with `scanf`

```
1 scanf("%d/%d/%d", &day, &month, &year);
```

For reading from and writing to files, there are variants of these, `fprintf()` and `fscanf()` respectively. Look up the definitions of these, and the `fopen()` and `fclose()` functions for accessing files. Files must be opened before they can be accessed and closed afterwards. The ‘f’ suffix on the two functions in this section is short for ‘formatted’. For unformatted input and output take a look at the `fread()` and `fwrite()` functions.

### 11.3 Read The Documentation

There are lots of other functions that are useful in a C standard library. In general, it is better to use standard functions than write your own. The standard version will have been tested and optimised by lots of people, while your version will be a small part of your project and receive less attention.

You can use the `apropos` command on a \*NIX machine to look for C standard library functions or search The Open Group’s web site. You will find a lot of things here, for example the `qsort()` function which implements the Quicksort algorithm. This is particularly notable, since it takes a function pointer as an argument. Listing 26 gives an example of how this is used. This short program sorts the arguments passed to it (as strings) and prints them in order.

This example includes a number of important features. The first three lines include the header files that are needed; `stdio.h` gives `printf()`, `stdlib.h` gives `qsort()` and `strings.h` gives `strcmp()`. On line 7 there is an example of casting a pointer to a real value. The compare function is passed two pointers to elements in the array as arguments. Because the array that is used is an array of strings, these are already pointers (`char*`s). Pointers to these are therefore pointers to pointers to characters. They are first cast to the correct pointer type and are then dereferenced once since the `strcmp()` function expects pointers to characters, not pointers to pointers to characters, as arguments.

Line 12 shows the `qsort()` function in use. The first argument is uses a little bit of pointer arithmetic. This gives a pointer to the list of arguments excluding the first one (which is the name of the executable). The second argument is the number of elements in the array. This is a common pattern in C—taking a

Listing 26: Example use of `qsort()`.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <strings.h>
4
5 int compare(const void* a, const void* b)
6 {
7     return strcmp(*(char**)a, *(char**)b);
8 }
9
10 int main(int argc, char *argv[])
11 {
12     qsort(argv+1, argc-1, sizeof(char*), compare);
13     for (unsigned i=1 ; i<argc ; i++)
14     {
15         printf("%d: %s\n", i, argv[i]);
16     }
17     return 0;
18 }

```

pointer and a size as arguments to a function—since there is no way of getting the size directly from a pointer. The next argument is the size of each element in this array, in this case the size of a pointer. This parameter allows the `qsort()` function to be used on arrays of any fixed-sized data, including structures. The final argument is a pointer to the function used to perform comparisons. Note again that the brackets are omitted when referencing a function as a pointer rather than calling it.

Line 13 is a simple `for` loop, iterating over every element in `argv` except the first one (hence starting the counter at 1 instead of 0). Line 15 is another use of `printf()` to output each sorted argument preceded by the number.

```

$ c99 compare.c
$ ./a.out a string of text passed as arguments
1: a
2: arguments
3: as
4: of
5: passed
6: string
7: text

```

## 12 Make

Although not directly connected to C, an important facility when building code on UNIX-like platforms is the `make` utility. This is an interpreter for a simple declarative language which builds dependency trees and performs operations to get from the leaf nodes to the root. By default, the `make` utility reads a file called `Makefile` from the current directory and attempts to reach the first target specified.

The Single UNIX Specification defines the basic structure of a `Makefile` although implementations (e.g. BSD `make` or GNU `make`) may provide extensions. This definition can be found here:

<http://opengroup.org/onlinepubs/009695399/utilities/make.html>

Listing 27 shows a simple `Makefile` which covers most of the important aspects of the format. The first two lines define macros. These are referenced later, and are simply replaced with their values at each occurrence.

The rest of the file defines *rules*. The first is a specific rule which is used to create the file `executable` from the list of files specified in the `OBJECTS` macro. This is done by passing them to `gcc` for linking. The lines under this are each indented with a single tab character and indicate the commands to be used to create the target from the source files. The first character of each of these is `@` which tells `make` not to print the commands to the terminal. Instead, both rules use `echo` to provide a friendly message indicating what is being done.

Line 6 provides instructions for linking. This uses the `LDFLAGS` macro, which is a standard macro used to specify flags for the linker. The `CFLAGS` macro is similar and is used to provide flags for the C compiler. Note that on line 2 we add things to the `CFLAGS` macro rather than redefining it. This allows extra flags to be passed from outside the file. Another macro referenced in this line is `$$`, a special macro which means ‘the target for this rule’.

The last rule is a *suffix rule*. This is a special kind of rule used for general build rules. This tells `make` how to make `.o` files from `.c` files. Again, this uses `gcc` but this time with the `-c` flag, meaning compile only. The `$$` macro is another special one which means ‘the out of date targets for this rule’. When `make` attempts to build `executable`, it will see that it depends on the listed `.o` files, and then that these can be created from `.c` files. If any of these are older than their source files, or don’t exist, then it will create them using the provided rules.

Listing 27: A simple Makefile.

```
1 OBJECTS = file1.o file2.o file3.o
2 CFLAGS += -std=c99 -Werror -Wall
3
4 executable: $(OBJECTS)
5     @echo Linking...
6     @gcc -o $$ $(LDFLAGS) $(OBJECTS)
7
8 .c.o:
9     @echo Compiling $$<
10    @gcc $(CFLAGS) -c $$<
```