Introduction to C

David Chisnall

Contents

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. C also 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 is that C does not expose anything that is specific to the underlying CPU, only abstractions common to most modern CPUs. 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 almost 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

In almost all cases, C code is compiled into native machine executables. There are a few places where this is not the case. Intel's Extended Firmware Interface (EFI), for example, compiles a dialect of C into bytecode that can then be executed on x86 or Itanium processors.

In most places where C is used, the end product is native machine code. In this section, we'll look at exactly how this translation works.

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).

In some compilers, step 2 is actually two steps - the first produces assembly code and the second assembles this to produce object code. In modern compilers, the object code contains some intermediate representation and the link step involves running some extra optimisations.

Modern C compilers have a *compiler driver* which hides these stages, but they can be run separately. On a GNU platform you will probably 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.

All UNIX systems (i.e. those conforming to the Single UNIX Specification) are expected to include executables called `cc` and `c99`. These compile the C89 and C99 dialects, respectively. On GNU/Linux, these are typically provided as wrappers around GCC.

There are a few differences between the C89 and C99 dialects. For the remainder of this course, we will only be looking at C99.

When you compile C code with GCC, it is a good example to add some command-line options specifying that you want additional error checks. Adding `-Wall` will turn on all warnings and adding `-Werror` will treat all warnings as errors, so you can't continue until you've fixed them. 99% of compiler warnings indicate a real bug, so fixing them is always a good idea.

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.

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[1] 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:

```
__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.

C1X, the upcoming version of the C standard, provides a standard way of doing this, but at the time of writing it is not supported by any compilers.

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 tot 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.

[1] The Executable and Linking Format used in Linux, *BSD, and most other UNIX-like systems, with Mac OS X being the most notable exception.

The `malloc()` and `free()` functions are not part of C-the-language, they are provided by the standard library. Typically, the operating system can provide memory in 4KB pages and the `malloc()` function is responsible for either returning a pointer to some memory in one of these pages or for requesting another one (or more).

This means that memory that has been freed may not yet have been returned to the operating system. When you try to access it, it may work, or it may not. The most famous example of this kind of bug was in the original Sim City game. This used some memory after freeing it, but still worked in DOS without any problems. When Microsoft developers tried to run the game in Windows 95, it crashed because the memory had already been returned to the operating system by the time that the game tried to access it for the last time. The Windows C library includes a special hack to detect if the current application is Sim City and, if so, not return memory to the operating system until a little while after it's been accessed.

3.3 Stack Memory

The stack is used for lexically-scoped variables - i.e. variables whose lifespan is defined by their visibility in the source code. 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.

Stack variables (also called variables with *automatic storage*) are the easiest to use because the memory that they represent is available for as long as the variable is valid. It is still possible to create memory-related bugs with stack variables, however. Listing 3 shows an example. In this listing, the pointer to a variable persists for longer than the variable itself, so any uses of the pointer will have undefined behaviour[2].

Listing 3: Memory bug with automatic storage.

```
int *a = NULL;
if (condition)
{
    int b = function();
    a = &b;
}
// a contains a pointer to b, but b is no longer valid.
```

Another common error is to return a pointer to something that is allocated on the stack.

4 Scoping

In every Algol-family language, including C, scope is a very important concept. Scope defines *visibility* - every declaration is visible until the end of the scope in which it is declared.

[2]Undefined behaviour in C usually means crashing.

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 4.

Listing 4: Braces are used to create blocks.

```
{ // This is the start of a block.
    ... // Everything here is in the block.
} // This is the end of a block.
```

Any variable in C is visible from the line on which it is declared until the end of the block. Listing 5 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 5: Variable scoping.

```
{
    int a; // a is valid from here.
    {
        int b; //b is valid from here
    } // b is invalid from here.
    int c; // c is valid from here
    {
        int a; // This a hides the other one.
    }
} // 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.

C Pointers	Java References
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 `NULL`.
Point to memory. User must know type of destination.	Point to objects which know their own type.

Table 1: Pointers versus references

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.

You can, for example, declare a static variable inside a function. The variable is then only visible inside that function, but you can pass a pointer to it out of the function without any problems - the memory that the static variable represents is valid for the entire lifespan of the program.

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.

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, and for implementing data structures like linked lists and trees.

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 6.3.1, 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. When you declare a pointer type, you add a star (*) to the type, for example:

```
int *a;      //Pointer to integer
float *b;    //Pointer to float
int *a;      //Pointer to pointer to integer
int **a;     //Pointer to pointer to integer
int ***a;    //Pointer to pointer to pointer to integer
```

All of these are stored in exactly the same way, the only difference is the annotation in the source code which tells the compiler what to interpret the pointee as when you load it from memory.

You can take the address of any variable in C with the *address-of operator*, `&`. For example `&a` means 'the address of the variable called a'.

Pointers are the reason C is so powerful, and the cause of most bugs in C code.

6 The C Type System

Memory in most modern systems does not contain the type of the stored value, but it's important to know what the type of a value is before you can do some computation on it. The number 1 can be represented in both integer and floating point values, but these representations are different. The C type system helps you keep track of what kind of data is in each location in memory.

6.1 Primitive Types

In Java, the type system is divided into two categories: intrinsic types and objects. This is something that Java inherits from Objective-C and so it's not surprising that the intrinsic types look a lot like C types. C defines five categories of type:

- Integers.
- Floating point values.
- Pointers.
- Structures
- Arrays

Note that none of these is directly analogous to a Java object, although you can implement similar behaviour with structures.

6.1.1 Integers

C defines five integer types. These don't have fixed sizes, but each must 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

Name	Minimum Size	Typical Size	Notes
`char`	8	8	Used for characters
`short`	16	16	
`int`	16	32 or 64	Machine word size
`long`	16	32 or 64	
`long long`	64	64	C99 only

Table 2: Integer sizes in C.

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 `int`s are taken as being half the size. LP64 is a common model on new systems, where `int`s are 32-bits and `long` and pointer values are both 64-bits. On older platforms LP32 was common, where `int`s 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).

On almost any platform, `long` is the same size as a pointer. Unfortunately, Win64 is the one exception, so assuming this is quite dangerous. C99 provides a `stdint.h` file that includes a number of extra types. These are implemented using one of the other types, depending on which is most appropriate for the current platform. These include things like `int32_t`, a 32-bit integer, and `uintptr_r`, a pointer-sized unsigned integer.

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()` operator. 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 compile time[3] and so can only be used to find sizes which

[3]Well, almost always. C99 variable length arrays are a special case.

are known to the compiler. It can not be used to find the size of a block of memory indicated by a pointer. This is a common source of bugs for novice C programmers. For example:

```
$ cat sizeof.c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
int *a = malloc(100 * sizeof(int));
printf("%d\n", (int)sizeof(a));
return 0;
}
$ c99 sizeof.c && ./a.out
4
```

This program prints the size of the `a` variable, which is the size of a pointer. The `sizeof()` operator is most useful when allocating memory. In this example, the code is requesting that `a` be a pointer to enough memory for 100 integers.

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 6: 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 6, 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 6 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

Most nontrivial programs will need to make use of some of the aggregate types in C, just as most nontrivial Java programs will need to define some classes.

6.3.1 Arrays

Arrays in C are just blocks 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 7 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 7: 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.

6.3.2 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 8 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 8: Creating a Structure

```
//Points have x and y coordinates
struct Point
{
    int x;
    int y;
}; // Don't forget this semicolon
// Rectangles are defined by two corners
struct Rectangle
{
    // Structures can contain other structures
    struct Point topLeft;
    struct Point bottomRight;
};
```

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 9. 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;
```

The second is much more readable, so it should be preferred. Structures have a similar use in C to objects in Java. Unlike Java classes, a structure does not have any methods associated with it. You can get the same behaviour, however, by defining functions that take a pointer to a structure as an argument.

In C++, C structures are extended to be slightly more like real classes. A method in C++ (also called a *member function*) has a hidden pointer called *this*, which is implicitly passed as an argument when the method is called.

Listing 9: Using Structures

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

6.4 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 10 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 10: 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 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 Operators in C

Each of the basic C types supports a few operations. These are subject to *operator precedence*, so, for example, `a+b*c` means 'multiply `b` by `c` and then add the result to `a`' because the multiply operator has higher precedence than addition. If you are uncertain of precedence, it is a good idea to insert parentheses.

Note that C++ supports operator overloading, meaning that the semantics of any of these operators can change depending on the types in C++. This

Operator	Meaning
`+`	add
`-`	subtract
`*`	multiply
`/`	divide
`%`	modulus
`++`	increment
`--`	decrement

Table 3: Common C arithmetic operators

makes it incredibly hard to understand a piece of C++ code without reading a lot of other code.

Wikipedia has a good summary of the C/C++ operators here: https://secure.wikimedia.org/wikipedia/en/wiki/Operators_in_C_and_C++

7.1 Unusual Operators

In C, array subscripting and function calls are treated as operators. The operator for a function call is a suffix operator: `()`. Placing this at the end of any expression will try to call this expression as a function. The preceding expression must evaluate to some kind of function pointer.

This is a good example of the way C exposes low-level details to the programmer. A function name in C is just a constant identifying the location in memory where a function is stored.

Another operator in this category is the array subscripting operator: `[]`. This is really a shorthand for some pointer arithmetic, and so is also treated as an operator in C.

The array subscripting and function call operators have the highest precedence, so you don't have to worry about the fact that they're operators.

The element selection operators, `.` and `->`, are also regarded as operators.

7.2 Arithmetic Operators

The simplest operators are the set related to arithmetic. The standard arithmetic operations—addition, subtraction, division, and multiplication—all exist as infix operators. Table 3 show the most common arithmetic operators.

It's worth noting that there are actually two increment and decrement operators; the prefix and suffix versions. These both increment a value, but the expression result is different:

```
int a = 12;
int b = a++; // b is 12, a is 13
int c = ++a; // c and a are both 14
```

The prefix version increments the variable and then returns the result. The postfix version increments the variable and then returns the old value.

7.3 Logical Operators

In C, there is no boolean type, so you can treat any integer as a boolean value. The logical operators, listed in Table 4, work this way. For example, the negation

Operator	Meaning
`!`	logical not
——	logical or
`&&`	logical and

Table 4: C logical operators

Operator	Meaning
`~`	bitwise negation
`^`	bitwise exclusive or
—	bitwise or
`&`	bitwise and
`<<`	left shift
`>>`	right shift

Table 5: C bitwise operators

operator is guaranteed to give a zero value if the operand is not zero, and a non-zero value if the operand is zero.

The most common use for logical operators is in constructing conditionals expressions, such as:

```
if ((a || b) && !c)
```

This means 'if a or b, and not c'. The logical or and and operations support *short-circuit evaluation*, so the second operand is only evaluated if the result can't be determined from the first. For example, if `a` is true, then `b` is not evaluated. If `a`— b— is false, then `c` is not evaluated. This doesn't make a difference here, but it does if you replace these single variables with expressions that access memory or have side effects. A very common use for this is:

```
if ((NULL != ptr) && (ptr->field == value))
```

If the pointer is `NULL`, then the second clause in this expression won't be evaluated. This is very useful, because if you tried to dereference a `NULL` pointer, you'd end up with the program crashing. This saves the need to nest two `if` clauses.

7.4 Bitwise Operators

The reason that the logical and and or operators have the symbol twice is that the single-symbol version is the bitwise version. There are important differences between them. The bitwise version applies the logical version to each pair of bits in the operands. For example, `1 2`— evaluates to 3, because in binary this is 01 and 10, so the result of each or operation between bits is 1, giving 11 as the result. In contrast, the result of a logical or would probably be 1 (but could be any non-zero value), because both inputs are true (not zero).

The bitwise operators are listed in Table 5.

Operator	Meaning
==	equal
!=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

Table 6: C comparison operators

7.5 Comparison Operators

The comparison operators are effectively logical operators, since they always evaluate to true or false (not-zero and zero), but they do so by comparing other values. They are listed in Table 6.

Hopefully the meanings of these is quite self explanatory. They compare two values and evaluate to either true or false depending on the result of the comparison.

7.6 Assignment Operators

In C, an assignment operation is an expression, which evaluates to the value that is being assigned and performs the assignment as a side effect. This lets you put assignment in conditionals, for example:

```
if (result = function())
```

This will store the result of the function call in `result` and enter the conditional block if the result is true (i.e. not 0). This is sometimes useful, but it also leads to a lot of errors, since both of the following are valid C, but have very different meanings:

```
if (a = b)
if (a == b)
```

Hopefully, your compiler will warn about the first form. It's a good idea to use one of these versions instead, to make it clean that you really mean to use the assignment in the conditional:

```
if ((a = b))
if (0 != (a = b))
```

Note the constant on the left-hand side of the comparison. This is a good habit to get into. If you are comparing between two values, one of which is a variable and one which is not, put the one that can not be used for assignment on the left. This means that typos become compile-time errors, rather than bugs:

```
if (0 = a) // Compiler error, can't assign to 0
if (a = 0) // Logic error, always false, a is set to 0
```

In addition to simple assignment, C has a group of assignment operators that combine assignment with an operation, for example:

```
a += 12;
// Equivalent to:
a = a + 12;
```

These are exactly equivalent to the longer form, they just involve a bit less typing.

8 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 11 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 11: 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 12. A function taking no arguments is still called like this, but with nothing between the brackets.

Listing 12: 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 13 is being called. The function 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[4].

[4]The order of the last two is implementation-dependent and the return value is often stored

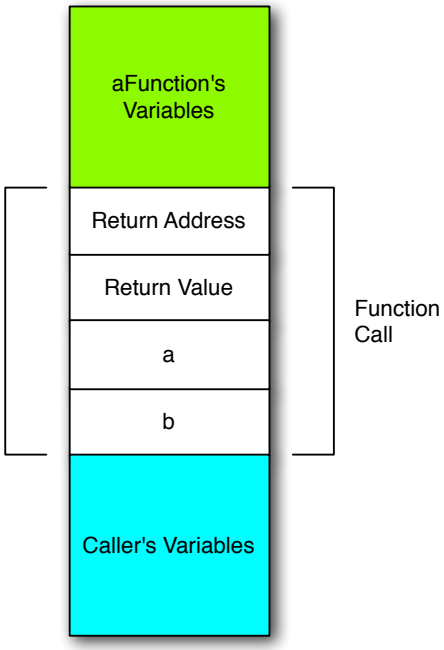

Figure 1: The call stack in C.

Listing 13: 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 }
```

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 looks, 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.

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.

8.1 The main() Function

The `main()` function is the entry point for any C program. There is exactly one per program. Listing 14 shows the skeleton of such a function.

Listing 14: A simple main() function

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

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.

8.2 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 15 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 15: Using function pointers.

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

Function pointers provide a way of implementing Java-like classes. You can use a structure to define a set of function pointers, one for each of the operations on an object, and then another structure to define the fields. Listing 16 shows a simple string 'class' in C. You'd call the 'methods' like this:

```
int length = str->cls->length(str);
```

This is a bit ugly. When you call a C++ member function that is declared *virtual*, it is translated into code that looks exactly like this. The extra indirection is hidden from you, which is one of the reasons that it's harder to judge the performance of a C++ program than a C one.

Listing 16: A simple string 'class' in C

```
struct string;
struct string_class;
{
    int (*length)(struct string*);
    char (*characterAtIndex)(struct string*, int);
};
struct string
{
    struct string_class *cls;
    char *string;
    int length;
};
```

9 Flow Control

9.1 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 17 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 17: An if statement.

```
if (a > 0)
{
    // Do something.
}
else
{
    // Do something else
}
```

9.2 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 18 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.

Listing 18: A switch statement.

```
switch(a)
{
    case 0:
        // Do something if a is 0
    case 1:
        // Do something is a is 0 or 1
        break;
    case 2:
        // Do something if a is 2
        break;
    default:
        // Do something if a is some other value
}
```

9.3 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 be negating the condition. C includes both pre- and post-test forms of a while loop.

Listing 19 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 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 19: A pre-test loop.

```
while (condition)
{
    // loop until condition is 0
}
```

The other type of loop is the post-test loop, known in C as the `do...while` loop, shown in Listing 20. Note the semicolon after the while at the end of this loop. This is very easy to accidentally omit.

Listing 20: A post-test loop in C.

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

9.3.1 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 21 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 21: A for loop.

```
for (int i=0 ; i<100 ; i++)
{
    // Loop 100 times.
}
// The for loop is equivalent to this
{
    int i = 0;
    while(i<100)
    {
        // Loop 100 times
        i++;
    }
}
```

10 The Peprocessor

The C preprocessor runs before the compiler. In modern compilers, preprocessing is just a step that is part of the parsing process, but it can also be run separately.

The preprocessor was a work around for the limited memory on the systems where C was originally intended to run. It handles every line in the source that starts with a hash (#) - these lines are ignored by the compiler.

10.1 Includes

The most important use of the C preprocessor is combining multiple files into a single compilation unit. Most C code users declarations (of functions, constants, and global variables) from elsewhere. These are included in the main source file using the `#include` directive. This has two forms:

```
#include <stdio.h>
#include "MyHeader.h"
```

The first is used for including system headers and instructs the preprocessor to look in the standard header locations, which vary depending on the platform. The second tells it to first look in the current directory. This form is useful when you have some declarations that need to be shared between files in your project.

10.2 Macros

The preprocessor does not know anything about C, so you can use it with files that are not valid C:

```
$ cat preprocess.c
#define MACRO value
MACRO
#define MACRO1(x) x.value(x)
MACRO1(foo)
$ cpp preprocess.c
# 1 "preprocess.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "preprocess.c"

value

foo.value(foo)
```

This example shows two uses of the `#define` preprocessor directive. This defines a macro, which is expanded anywhere that it is used. The first few lines of the output are some metadata that the compiler will use when generating line number information in debugging info, but are otherwise ignored.

The `MACRO` macro takes no arguments, so everywhere that `MACRO` appears in the input file it is replaced with the text `value`. The `MACRO1` macro is slightly more complicated - it takes an argument, which is inserted into the expansion.

This is a very common source of errors. The argument to the macro can be an expression (or any other sequence of tokens; remember that the preprocessor doesn't know about C syntax). This macro uses its argument twice, so the expression will be evaluated twice. The canonical example of a problem caused by this is in a `MIN` macro. The obvious implementation would look something like this:

```
#define MIN(x,y) (x < y ? x : y)
```

You might then call this with two function calls as arguments:

```
int a = MIN(b(), c());
```

This will expand to:

```
int a = (b() < c() ? b() : c());
```

This will cause whichever function has the lower value to be called twice. If the function has side effects, then this will cause them to be evaluated twice, which is not obvious from the code.

Function-like macros look like functions, but they don't behave in quite the same way. This means that you have to be very careful using them. Some C style guides recommend avoiding the entirely, but this can cause other problems. In general, if there are two ways of doing something, and one uses macros, you should use the other one.

Note: Preprocessor directives are not C statements, so they do not end with a semicolon. If you end a macro definition with a semicolon, the semicolon will be inserted at the end of every macro instantiation, which can break things in a very confusing way.

10.3 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.

```
#if 0
// Never compile this bit
#else
// Always compile this bit
#endif
#ifdef UNIX
// Some code for UNIX platforms
#else
// Code for weird platforms
#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

```
#ifndef __MY_HEADER_INCLUDED__
#define __MY_HEADER_INCLUDED__
// Header file contents here
#endif
```

11 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.

On a UNIX-like system, you can get help about most of the C library functions via the system manual, for example:

```
$ man fprintf
```

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 `strlcpy()`, 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.

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

Listing 24: Using printf

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

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 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

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 }
```

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 $<
```