

A C REFRESHER

MARK DALRYMPLE

A C Refresher

by Mark Dalrymple

Copyright © 2005 Mark Dalrymple

Table of Contents

1. A C Refresher, Part 1: The Basics	1
The C Compiler Pipeline	1
The Preprocessor	3
Including other files	3
Macro expansion	5
Conditional compilation	6
Constants	8
Data Types	9
Predefined Types	11
Enumerations	11
Variables	12
LValues	14
Operators	14
Bitwise Operators	14
BitMasks	17
Functions	20
Logical Expressions	21
Logical Operators	22
Control Structures	23
if	23
while	23
for	24
do-while	24
switch	24
goto	25
For the More Curious: Multiple Source Files	25
For the More Curious: Common Compiler Errors	29
Challenge	30
2. A C Refresher, Part 2: Pointers, Structures, and Arrays	31
Pointers	31
Pointer syntax	31
An actual use of pointers: pass by reference	33
Another use for pointers: strings	35
Pointer math	37
The NULL pointer	39
Structures	39
Declaring structures	39
Pointers to structs	40
Bitfields	41
Typedef	42
Casts	43
void *	44
Function pointers	44
Unions	45
Arrays	48

Arrays of pointers 50

For the More Curious: Data Structures Using Pointers 51

Challenge: 54

A C Refresher, Part 1: The Basics

To fully understand the day-to-day details of programming with the Unix APIs, a good knowledge of the C programming language is vital. Most of the popular modern languages like Java, Perl, and Ruby have a strong C flavor. If you know any of these languages it should be easy to pick up the details of C. Well, except for pointers, which many many programmers have trouble with. Pointers, along with structures and arrays, will be described in detail in the next chapter.

One of the things that trips up new C programmers is expecting the language to do more than it actually does. At its foundation, C just moves bytes around with little higher-level meaning projected upon those bytes, and with no hidden behaviors. For instance, most languages have a String type and programmers (rightfully) expect the equality operator to compare the strings and indicate whether they contain the same characters in the same order.

C has no string type, and the equality operator just compares the starting address of two strings (which is an identity rather than a value check). For C to support the “expected” kind of string compare the language would have to supply a mechanism to do an implicit loop over the characters of the string. This hidden, potentially expensive behavior goes against the performance-oriented closer-to-the-metal spirit of C. In general, when faced with possible behaviors in a given situation (“does string compare result in an identity or value comparison”), think of the simplest possible behavior. That is most likely what C does.

The C Compiler Pipeline

Here is a simple program:

Example 1.1. first.c

```
// first.c -- a simple first C program

/* compile with:
cc -g -o first first.c
*/

#include <stdio.h>      // for printf()
#include <math.h>       // for cos()

int main (int argc, char *argv[])
{
    printf ("the cosine of 1 is %g\n", cos(1.0));
    printf ("thank you, and have a nice day\n");
}
```

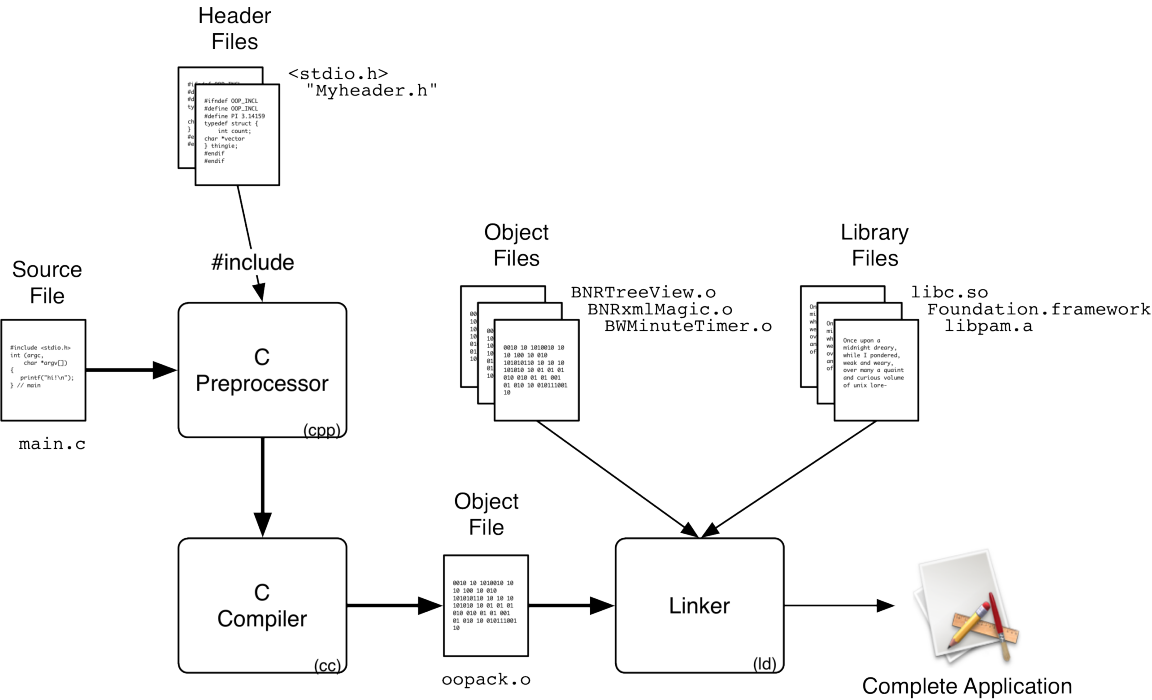
```
    return (0);  
  
} // main  
  
Compile it like this:  
  
$ cc -g -o first first.c -lm  
  
And run it:  
  
$ ./first  
the cosine of 1 is 0.540302  
thank you, and have a nice day
```

In C, the `main()` function has special meaning. It is where your program starts executing once the operating system has finished loading the program into memory.

If you're wanting to use a fancy IDE like Xcode, you'll want to use a project template like “Standard Tool” and put the code in the source file provided by the template.

The compiling and building of a C program is a multi-stage process, as shown in Figure 1.1. The figure shows the compiling of a single file, along with linking that file with other files to create a finished application.

Figure 1.1 The C Pipeline



The source file (`first.c` in this case) gets fed into the C preprocessor which does some textual manipulations like header file inclusion and macro expansion, and it also strips out comments. (A header file is where you declare the public interface for a unit of functionality. We'll talk about

header files in more detail later). The end result gets fed into the C compiler which translates the program text into binary machine code. The compiler will generate an “object file” containing the machine code. If you don't instruct the compiler otherwise, it will just create an executable without generating an intermediate object file (which is what we did for Example 1.1).

After the compiler has done its work, the linker takes over. The linker takes the object file from the compiler, and includes any other object files or libraries (pre-packaged code supplied by your OS or another vendor). The linker crawls through your code and the library code to make sure all of the features used by the program are available. Once the linking process is successful, the completed executable is created and is available for use.

Errors can happen at any stage along the way. The preprocessor might not be able to find a header file. You may have a syntax error in your C code. Your code might reference a function that isn't found in any of the libraries you tell the linker to use.

All of that activity is wrapped up in the single `cc` command. Here is the compilation command used in Example 1.1.

<code>cc</code>	The program name of the compiler. This will typically pick up the vendor's compiler. Mac OS X uses GCC, the GNU Compiler Collection. You can invoke the compiler as <code>cc</code> or <code>gcc</code> , depending on your mood.
<code>-g</code>	Generate debugging symbols (handy if problems develop).
<code>-o first</code>	<code>-o</code> tells the compiler to use the name <code>first</code> for the name of the generated program. Otherwise the compiler uses the non-obvious name “a.out”.
<code>first.c</code>	The name of the source file.
<code>-lm</code>	Link in the math library for the <code>cos()</code> function. This isn't needed in Mac OS X, but is necessary on most other platforms.

The Preprocessor

The C Preprocessor is a textual processor that manipulates your source code before it gets fed to the compiler. Some languages have macro facilities (such as Lisp, although their macros are different from C macros), but C is unique (as far as I know) because its preprocessor is a distinct program rather than being built into the compiler. This has the nice side-effect that you can use it for your own uses outside of C.

The preprocessor is pretty dumb. It just performs blind substitutions with no idea of the semantics of the C language. Thus it is possible to introduce subtle (and not so subtle) errors through misuse of the preprocessor.

The preprocessor brings three main features to the table: including other files, macro expansion, and conditional compilation.

Including other files

Commands to the preprocessor start with a “#” in the left-most column. The “#” character is not a comment character like in most scripting languages. Take a look at the first two lines of `first.c`:

```
#include <stdio.h>      // for printf()
#include <math.h>       // for cos()
```

`#include` looks for a file on disk and then pastes the contents of the file where the `#include` directive is. When the file name is surrounded by angled brackets like these, the preprocessor looks in specific directories hard-coded into the compiler such as the directory `/usr/include`. So, these two lines copy the contents of `/usr/include/stdio.h` and `/usr/include/math.h` into our source file. These are just simple text files, so you are welcome to browse them and see the kind of stuff that is getting included.

The `// for printf()` and `// for cos()` lines are just comments added to show why a particular file has been `#included`. You don't have to include these. I like to notate header files like this so I know why I'm `#including` a header file, in case I don't need it anymore. It takes a non-zero amount of time to include a file, so it's best to not include it if it's not necessary.

If you want to include header files you've written yourself, use double quotes instead of angle brackets, like this:

```
#include "someGroovyHeader.h"
```

This looks for the include file in the same directory as the source file, and you can add additional paths to the compiler by using the `-I` command line flag.

Header files are C's way of importing features that can be used by your program. This is similar to the Java `import` directive or the Perl `use` command. C's `#include` mechanism can be more flexible since the header files can do anything, including defining functions and declaring variables in addition to advertising available features.

It is a good idea to `#include` all of the headers of the features you need. If you try to use a function that the compiler hasn't seen a declaration for, the compiler will try to infer the proper argument types and return type of the function. Sometimes it can guess wrong.

When you look at a man page for a function, you will see which header file you will need. For instance:

```
$ man 2 open
NAME
    open - open or create a file for reading or writing

SYNOPSIS
    #include <fcntl.h>

    int
    open(const char *path, int flags, mode_t mode);
```

The man page tells you that you will need to have `#include <fcntl.h>` before you use the `open()` function.

One common misconception that folks have is that including a header file is all that is necessary to use a feature. Many times that is the case, particularly with functions that live in the standard C library or are standard Unix API calls. Sometimes, in addition to `#including` a header file, you will also need to tell the linker to use a specific library, such as a crypto library or a UI framework. Example 1.1 includes `-lm` in the command line. The man page for a function often tell you if you

need to link in a specific library, and third-party toolkits should have some kind of documentation to let you know what you have to do to use it.

The Objective-C language adds a `#import` directive to the preprocessor that does the same job as `#include`, but with one important difference: if you `#include` the same file twice, it will only actually get included once. Why is this an issue? If you're not careful, multiple inclusions of a header file can lead to compiler or linker errors. The author of the header file has to put in special guards to protect against multiple-inclusion. Because the Objective-C compiler has `#import`, header file authors don't have to worry about such details.

Macro expansion

The C preprocessor does simple textual manipulations. You define a sequence of characters to be replaced by another sequence by using `#define`:

```
#define PI 3.14159
```

will replace the word “PI” with “3.14159” before the file gets shipped off to the compiler. This is handy for making symbolic constants. (In reality you'd really want to `#include <math.h>` and use the value of `M_PI`.)

Macros can also have arguments, and these arguments get textually substituted. Define macros with arguments like this:

```
#define AVERAGE(x,y) ((x) + (y) / 2)
```

and you can use it like this:

Table 1.1. Macro Expansion

Code	Expands to
<code>x = AVERAGE (3, 4);</code>	<code>x = (3 + 4) / 2;</code>
<code>x = AVERAGE (PI, 23);</code>	<code>x = (PI + 23) / 2;</code>
<code>x = (PI + 23) / 2;</code>	<code>x = (3.14159 + 23) / 2;</code>

You can see that macro expansion is recursive, with symbols getting continually expanded until there are no more. Luckily the preprocessor is smart enough to not let you define a self-referential macro, being expanded until memory runs out.

Using all-uppercase for macro names is just a convention. You can make macros using lower case letters, numbers, underscores, etc. Issues involving macros are explored in the compiler chapter of the book “Advanced Mac OS X Programming”.

Example 1.2 shows some macros in action:

Example 1.2. macros.c

```
// macros.c -- look at C Preprocessor macros

/* compile with:
cc -g -Wall -o macros macros.c
*/

#include <stdio.h>          // for printf()
```

```
// make some symbolic constants
#define PI          3.14159
#define Version     "beta5"

// make some macros
#define SQUARE(x)    ((x) * (x))
#define AreaOfACircle(r) (PI * SQUARE(r))

int main (int argc, char *argv[])
{
    printf ("Welcome to version %s of macros\n", Version);
    printf ("The area of a circle with radius 5 is %f\n",
        AreaOfACircle(5));

    return (0);
} // main
```

Note that you can give **printf()** (short for “print formatted”) some format specifiers that tell it to look for values after the first string in the list of arguments you pass it, like “%f” for a floating point value. Many languages and libraries have adopted **printf**’s format specifiers, so they are not going to be explained here.

Conditional compilation

C is a language that has been implemented on a huge variety of different computing platforms, from the Apple II up through Crays and MacPros. When you write code that runs on different platforms, you frequently have code that is peculiar to each specific platform, but you do not want that code to be included on the other platforms.

C uses the preprocessor to help you solve this problem. The preprocessor can conditionally remove chunks of text from your program using the **#ifdef**, **#else**, and **#endif** commands.

For instance:

```
#ifdef SYMBOL_NAME
// first chunk of text
#else
// second chunk of text
#endif
```

If the symbol **SYMBOL_NAME** has been **#defined** previously, let the first chunk of text through and drop the second chunk of text on the floor. If **SYMBOL_NAME** has not been **#defined**, the second chunk remains and the first chunk gets dropped.

If you have a symbol with a numeric value, you can use **#if** instead of **#ifdef**. If the symbol (or constant) given to **#if** is not zero, the first chunk of text survives. If that symbol is zero, the second chunk is what makes it through. **#if** can also be used with undefined symbols. An undefined symbol is interpreted as a false value as far as **#if** is concerned.

Example 1.3. conditional.c

```
// conditional.c -- look at conditional compilation

/* compile with:
cc -g -Wall -o conditional conditional.c
*/
```

```
#include <stdio.h>      // for printf()

int main (int argc, char *argv[])
{
#define THING1

#ifdef THING1
    printf ("thing1 defined\n");
#else
    printf ("thing1 is not defined\n");
#endif

#ifdef THING2
    printf ("thing2 is defined\n");
#else
    printf ("thing2 is not defined\n");
#endif

    return (0);
} // main
```

A sample run:

```
$ ./conditional
thing1 defined
thing2 is not defined
```

One handy use of `#if` with constant values (rather than the name of a symbol) is to do a block “comment out” of a big chunk of stuff. Example 1.4 shows `#ifs` and constants in action.

Example 1.4. leaveout.c

// leaveout.c -- use the preprocessor to comment out a chunk of code

```
/* compile with:
cc -g -Wall -o leaveout leaveout.c
*/

#include <stdio.h>      // for printf()

int main (int argc, char *argv[])
{
#ifdef 0
    printf ("oh happy day\n");
    printf ("bork bork bork\n");
    we_can even have syntax errors in here
    since the compiler will never see this part
#endif

#ifdef 1
    printf ("this is included. wheee.\n");
#endif

    printf ("that is all folks\n");

    return (0);
} // main
```

A sample run:

```
$ ./leaveout
this is included.  wheee.
that is all folks
```

Constants

OK, we're done with the preprocessor for now, and it's time to dig into some features of the C language. Like all other languages, C lets you specify constant values in your code:

integer constant:	1234
floating point constant:	3.14159
single character constant:	'c'
string constant:	"I seem to be a verb"

Notice that different quotes are used to discriminate between single-character constants (single-quote `'`), and string constants (double-quote `"`).

With characters there are some variations that indicate to the compiler a special character should be used. A backslash escapes the next character so that it does not get interpreted. For example, this string constant has an embedded double quote:

```
"hello double quote \" you are so cute"
```

Without the escaping you would get a syntax error since the double quote would terminate the string constant, and “you are so cute” would be interpreted as more C code. Escaping also gives special meaning to some characters inside character constants and string constants:

`\n` Newline

`\r` Return character

`\t` Tab

`\\` Backslash

Specific byte values can be given in octal (base-8) by using `\\###` such as `\\007` causes the terminal to beep, and `\\075` is the equals signs.

`\\007` Beep character

`\\075` Equals sign

`\\u####` Unicode character whose encoding is `####`. So `\\u005c` is the backslash character.

The `ascii` man page has a list of characters and their numerical value.

Integer constants can be expressed in decimal (base-10), octal (base-8), and hexadecimal (base-16). Octal constants have a leading zero, and hex constants have a leading `0x`:

octal constant:	0217
hex constant:	0xF33DF4C3

Example 1.5 shows literal constants in-use:

Example 1.5. constants.c

```
// constants.c -- show various constants

/* compile with:
cc -g -o constants constants.c
*/

#include <stdio.h>      // for printf()

int main (int argc, char *argv[])
{
    printf ("some integer constants: %d %d %d %d\n",
            1, 3, 32767, -521);
    printf ("some floating-point constants: %f %f %f %f\n",
            3.14159, 1.414213, 1.5, 2.0);
    printf ("single character constants: %c%c%c%c%c\n",
            'W', 'P', '\114', '\125', '\107');
    printf ("and finally a character string constant: '%s'\n",
            "this is a string");

    return (0);
} // main
```

A sample run:

```
$ ./constants
some integer constants: 1 3 32767 -521
some floating-point constants: 3.141590 1.414213 1.500000 2.000000
single character constants: WPLUG
and finally a character string constant: 'this is a string'
```

Data Types

C has two fundamental data types: integer and floating point. Everything else is an interpretation of one of these types, such as characters being short integers, or an aggregation of these types, with arrays of characters are strings, a group of three pointers can be a tree node, and so on.

These are the specific data types available:

Table 1.2. C Data Types

Name	“Usual” Storage Size	“Minimum” Storage Size
char	1 byte	1 byte
short	2 bytes	2 bytes
int	4 bytes	2 bytes
long	4 bytes	4 bytes
long long	8 bytes	8 bytes

So, a long being a 4 byte (32 bit) number, can have values between -2,147,483,648 and +2,147,483,647. Which is 31 bits of numerical data and 1 bit to indicate whether the number is positive or negative (called the sign bit). 64-bit Mac systems have longs being 64-bits (8 bytes).

The header file `<limits.h>` includes `#defined` constants for examining the maximum and minimum values for the basic data types. The constants have names like `INT_MIN`, `INT_MAX`, `SHRT_MIN`, `SHRT_MAX`, and so on.

The integer types can also be declared unsigned, meaning that the sign bit is considered to be actual data. An unsigned `long` therefore can have values between 0 and 4,294,967,295. If you want a “byte” data type, use an unsigned `char`.

There are two floating point data types: `float` and `double`, which differ by how many significant bits of storage they have.

You can run this to see what limits your platform has:

Example 1.6. `limits.c`

```
// limits.c -- show some info about various built-in data types

/* compile with:
cc -g -Wall -o limits limits.c
*/

#include <limits.h>          // for limit constants
#include <stdio.h>           // for printf()
#include <stdlib.h>          // for EXIT_SUCCESS

int main (int argc, char *argv[])
{
    printf ("      type:  bytes %14s %14s %14s\n",
            "min value", "max value", "max unsigned");

    printf ("      char:  %5ld %14d %14d %14u\n", sizeof(char),
            CHAR_MIN, CHAR_MAX, UCHAR_MAX);

    printf ("     short:  %5ld %14d %14d %14u\n", sizeof(short),
            SHRT_MIN, SHRT_MAX, USHRT_MAX);

    printf ("        int:   %5ld %14d %14d %14u\n", sizeof(int),
            INT_MIN, INT_MAX, UINT_MAX);

    printf ("        long:  %5ld %14ld %14ld %14lu\n", sizeof(long),
            LONG_MIN, LONG_MAX, ULONG_MAX);

    // not available on all platforms
#ifdef LLONG_MIN
    printf (" long long:  %5ld %20lld %20lld \n"
            "                %20llu\n", sizeof(long long),
            LLONG_MIN, LLONG_MAX, (long long)ULLONG_MAX);
#endif
    printf ("     float:  %5ld\n", sizeof(float));
    printf ("    double:  %5ld\n", sizeof(double));

    return (EXIT_SUCCESS);
} // main
```

This is what it prints on my machine:

```
$ ./limits
      type:  bytes      min value      max value      max unsigned
```

char:	1	-128	127	255
short:	2	-32768	32767	65535
int:	4	-2147483648	2147483647	4294967295
long:	4	-2147483648	2147483647	4294967295
long long:	8	-9223372036854775808	9223372036854775807	18446744073709551615
float:	4			
double:	8			

The `sizeof` operator evaluates to the number of bytes of the indicated type. Later on, when we talk about structures, `sizeof` can be used to figure out how many bytes a structure takes. Note that `sizeof` is a compile time operator, not a run time function. A common mistake new C programmers make is to try to use `sizeof` on a parameter passed to a function to try to figure out how much data you have to manipulate.

Predefined Types

Since the standard C types cannot be depended on to have a specific storage size, the standard C library and the Unix API define some more abstract integral types that should be used, like

`size_t` An unsigned integral value for indicating sizes (in bytes) of data types. Also used as the type of argument for `malloc()`, which dynamically allocates memory. (`malloc()` and memory issues are discussed in the Memory chapter of the book.)

`pid_t` An unsigned integral value for representing the process IDs in a Unix system.

`uid_t` An unsigned integral value for representing the user ID in a Unix system.

There are many more types. The man pages for functions indicate which of these types they use. So why go to all the trouble to make these specific types? For one, it does help to make functions more self-describing. If a parameter is a `pid_t`, you can be reasonably sure that you should pass a Unix process id. Other types, like `uintptr_t` that is large enough to hold both an integer and pointer type, help make your code safer, just in case integers and pointers (which we'll talk about in detail later) need to be stored into the same variable.

Enumerations

Related to the integer types are enums. enums are another way of creating symbolic constants (the first way is using the preprocessor that you saw before). In C enums are just a little syntactic sugar around integer constants. In particular, there is no error checking involved when you pass an enum value around. There is no support to make sure that a function that accepts an enum type will only get the predefined enum values. It's all just integer types under the hood.

The syntax for defining an enum is:

```
enum {
    Hearts,
    Moons,
    Stars,
    Clovers
};
```

The first symbol gets a value of zero, and subsequent symbols increment the previous value by one. So the above snippet would have values of:

```
enum {
    Hearts,      // 0
    Moons,       // 1
    Stars,       // 2
    Clovers      // 3
};
```

You can also assign values:

```
enum {
    Doc,          // 0
    Bashful,      // 1
    Dopey = 23,    // 23
    Cranky,       // 24
    Funky,        // 25
    Shifty = 23,   // 23
    Hungry = 42,   // 42

    Mask1 = 0xFFFF0000,
    Mask2 = 0x00000040
};
```

One nice side effect of using enums (as opposed to just `#defining` everything) is that sometimes the debugger can map integer values back to the enum name and show you data symbolically.

Variables

Variables are where you store values that can change over the lifetime of your program. C requires you to declare all variables before you use them (unlike Python or Javascript). Older versions of C require you to declare all of the variables in a function before any executable lines of code. Newer versions of C, and Objective-C lift this restriction, so you can declare variables immediately before first-use if you wish (like in Java and C++).

The declaration syntax for a variable is:

```
data-type variable-name [ = optional-initializer];
```

Where `data-type` is one of the predefined data types seen above, or a composite data type defined by you or by the system. Variables can be local to functions or global to the whole file (or even the whole program) depending on where they are defined.

A variable declared inside of a function is local to that function and cannot be seen outside of the function. This is like `my` variables in Perl or `var` variables in Javascript. The storage for the variable goes away when the function exits. In C, local variables can have a random, uninitialized value. Another common C programmer mistake is forgetting to initialize a local variable, and then having a garbage value cause strange non-reproducible bugs.

A variable declared outside of any functions is global, and unlike local variables, is initialized to zero.

C also introduces the idea of a “static variable” (one of the many uses of the term `static` in C). A static variable is a variable that syntactically is local to a function, but the value persists from call to call as if it was a global variable. In reality, it really is a global variable with local scope, and is initialized to zero automatically.

Example 1.7 shows some simple variable declarations.

Example 1.7. variables.c

```
// variables.c -- some simple variable declarations

/* compile with:
cc -g -Wall -o variables variables.c
*/

#include <stdio.h>      // for printf()

int aGlobalInt;         // global
float pi = 3.14159;     // global

void someFunction ()
{
    int aLocalVariable = 0;    // local, random value but
                                //      initialized to zero
    unsigned short myShort;    // local, random value
    static unsigned char aByte; // static, initialized to
                                //      zero, persists

    myShort = 500 + aLocalVariable;
    aGlobalInt = 5;

    aByte++;

    printf ("aByte: %d, myShort: %d  aGlobalInt: %d\n",
            aByte, myShort, aGlobalInt);
} // someFunction

int main (int argc, char *argv[])
{
    printf ("aGlobalInt before someFunction: %d\n", aGlobalInt);
    someFunction ();

    printf ("aGlobalInt after someFunction: %d\n", aGlobalInt);
    someFunction ();

    aGlobalInt = 23;
    someFunction ();

    return (0);
} // main
```

And the program in action:

```
$ ./variables
aGlobalInt before someFunction: 0
aByte: 1, myShort: 500  aGlobalInt: 5
aGlobalInt after someFunction: 5
aByte: 2, myShort: 500  aGlobalInt: 5
aByte: 3, myShort: 500  aGlobalInt: 5
```

You should always initialize variables declared in functions before you use them. Otherwise, they will have undefined (random) values, which if used can cause your program to behave in a random fashion. Note that global variables and static variables are initialized to zero for you.

LValues

You will sometimes hear folks (or compiler messages) refer to the term “lvalue”. That just means something can appear on the left side of an assignment operator. For example:

```
int myVar;  
myVar = 5;
```

`myVar` is an lvalue, and this is legal code.

But

```
7 = 23;
```

or

```
(i == 3) = 23;
```

is not. Sometimes you hear the fancy term “rvalue”, and it just means the expression on the right side of an assignment operator.

Operators

The C operators, like `+`, `-`, `*`, `/`, have been adopted by most modern languages and should be familiar. The increment and decrement operators may be new to some:

`x++` Use the value of `x`, then add 1 to `x`

`++x` Add 1 to `x`, then use the value

`x--` Use the value of `x`, then subtract 1 from `x`

`--x` Subtract 1 from `x`, then use the value

Plus there are shorthand assignment operators. `x += y` is the same as `x = x + y`. This idea also generalizes to `+`, `-`, `*`, `/`, and other operators (like the bitwise operators).

Bitwise Operators

Many languages have adopted C's bitwise operators. Those languages tend not to need to do as much bit twiddling as C, so the bitwise operators are mentioned in passing and are relegated to the reference chapter nobody reads. But sometimes you just have to be able to set or test an individual bit. In C, particularly using the Unix APIs, you need to do that on a regular basis.

The bitwise operators let you do bitwise AND, OR, XOR, and complement to integral values. You can also shift the bits left or right within an integer value.

The examples below use these two numbers:

Figure 1.2 Binary

A:	<table border="1"><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	1	1	0	0	1	1	0	0	= 0xCC	= 204
1	1	0	0	1	1	0	0				
B:	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	1	1	0	0	0	0	= 0xF0	= 240
1	1	1	1	0	0	0	0				

A & B has a result that has bits set only if bits are set in both A and B.

Figure 1.3 AND (&)

$$\begin{array}{rcl}
 \text{A:} & \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} & = 0xCC = 204 \\
 \text{B:} & \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} & = 0xF0 = 240 \\
 \hline
 \text{Result:} & \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} \boxed{0} & = 0xC0 = 192
 \end{array}$$

A | B has a result that has bits set if bits are set in either A or B.

Figure 1.4 OR (|)

$$\begin{array}{rcl}
 \text{A:} & \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} & = 0xCC = 204 \\
 \text{B:} & \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} & = 0xF0 = 240 \\
 \hline
 \text{Result:} & \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{0} & = 0xFC = 252
 \end{array}$$

A ^ B has a result that has bits set if bits are set in either A or B, but not in both. The “or” in this case is like when the wait staff asks, “Would you like soup or salad with that?”

Figure 1.5 XOR (^)

$$\begin{array}{rcl}
 \text{A:} & \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} & = 0xCC = 204 \\
 \text{B:} & \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{0} & = 0xF0 = 240 \\
 \hline
 \text{Result:} & \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{0} & = 0x3C = 60
 \end{array}$$

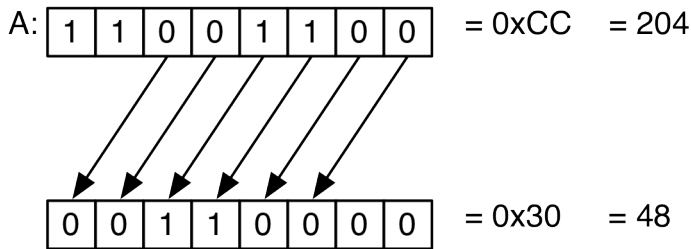
A ~ flips the bits from zero to 1 and 1 to zero.

Figure 1.6 NOT (~)

$$\begin{array}{rcl}
 \text{A:} & \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} & = 0xCC = 204 \\
 \hline
 \text{Result:} & \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} & = 0x33 = 51
 \end{array}$$

$A \ll n$ shifts the bits of A n places to the left, filling in zeros in the low-numbered bits.

Figure 1.7 $A \ll 2$

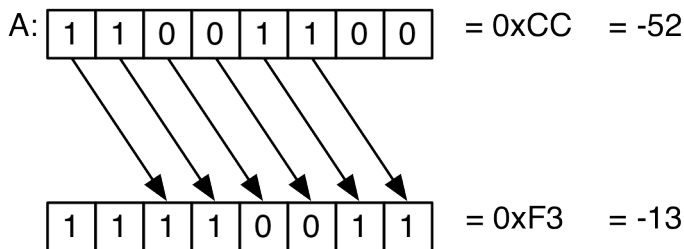


Note that the two high-order 1 bits got shifted out and dropped and zero got shifted in. Note that due to how binary math works, shifting to the left is the same as multiplying by two. Modern processors can shift much faster than they can multiply, which can make for a handy optimization.

$A \gg n$ shifts the bits of A n places to the right.

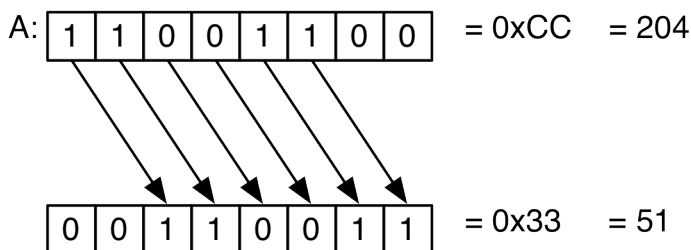
There is one complication with shift-right. The high-order bit of a piece of signed integer data is the sign bit indicating whether the value is negative or positive. When you shift right, do you preserve the sign bit, or fill it in with zeros? The rule C uses is: if the number is signed and negative, the sign bit fills in with 1:

Figure 1.8 $A \gg 2$



Otherwise, for unsigned integers or positive numbers, the sign bit fills in with zero.

Figure 1.9 Bitwise Shift Right



BitMasks

The AND, OR, and NOT operators are handy when dealing with bitmasks. Bitmasking is a technique where information can be compactly stored in small integers by attaching meaning to specific bit positions.

Using bitmasks:

1. Define a constant that matches the bit you wish to deal with.

```
#define MY_BIT_VALUE 8 // 00001000
```

Or use bit shifting.

```
#define MY_BIT_VALUE (1 << 3) // 00001000
```

2. Set the bit with bitwise OR.

```
myFlags |= MY_BIT_VALUE
```

Figure 1.10 Set Flag

myFlags:	0	1	1	0	0	0	1	= 0x61	= 97
MY_BIT_VALUE:	0	0	0	0	1	0	0	= 0x08	= 8
<hr/>									
Result:	0	1	1	0	1	0	0	= 0x69	= 105

3. Test the bit with bitwise AND.

```
if (myFlags & MY_BIT_VALUE) {
    // do something appropriate
}
```

Figure 1.11 Test Flag

myFlags:	0	1	1	0	1	0	0	1	= 0x69	= 105
MY_BIT_VALUE:	0	0	0	0	1	0	0	0	= 0x08	= 8
<hr/>										
Result:	0	0	0	0	1	0	0	0	= 0x08	= 8

4. Clear the bit by using a complement and an AND.

```
myFlags &= ~MY_BIT_VALUE;
```

This is a bit trickier. What is needed is a new mask that will let through all the original values of the bits of the variable you are masking, but leave the masked-out bit behind.

Figure 1.12 Clear Flag

myFlags:	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	1	0	1	0	0	1	= 0x69	= 105
0	1	1	0	1	0	0	1				
~MY_BIT_VALUE:	<table><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1	0	1	1	1	= 0xF7	= 247
1	1	1	1	0	1	1	1				
<hr/>											
Result:	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	1	0	0	0	0	1	= 0x61	= 97
0	1	1	0	0	0	0	1				

When using a mask with multiple bits set, it is safer to compare against the mask rather than doing a simple logical test:

```
#define MULTI_BITS      0xF0    // 11110000
```

and assuming that myFlags is 0x61 (01100001), the statement

```
if (myFlags & MULTI_BITS) {  
    // do something if all bits are set  
}
```

is bad code. The result of the bitwise-AND is:

Figure 1.13 Result of bitwise-AND

myFlags:	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	1	0	1	0	0	1	= 0x69	= 105
0	1	1	0	1	0	0	1				
MULTI_BITS:	<table><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	1	1	0	0	0	0	= 0xF7	= 247
1	1	1	1	0	0	0	0				
<hr/>											
Result:	<table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	0	0	0	0	0	= 0x60	= 96
0	1	1	0	0	0	0	0				

The result is 0x60, which will be treated by the if as a true value (everything except zero is considered a true value), even if only a couple of the bits in MULTI_BITS is set. It is more correct to do

```
if ((myFlags & MULTI_BITS) == MULTI_BITS) {  
    // do something if all bits are set  
}
```

Example 1.8. bitmask.c

```
// bitmask.c -- play with bitmasks  
  
/* compile with:
```

```

cc -g -Wall -o bitmask bitmask.c
*/

#include <stdio.h>      // for printf()

#define THING_1_MASK    1          // 00000001
#define THING_2_MASK    2          // 00000010
#define THING_3_MASK    4          // 00000100
#define ALL_THINGS      (THING_1_MASK | THING_2_MASK | THING_3_MASK)
                        // 00000111

#define ANOTHER_MASK    (1 << 15) // 00100000
#define ANOTHER_MASK_2  (1 << 16) // 01000000

#define ALL_ANOTHERS    (ANOTHER_MASK | ANOTHER_MASK_2) // 01100000
#define ALL_USEFUL_BITS (ALL_THINGS | ALL_ANOTHERS)      // 01100111

void showMaskValue (int value)
{
    printf ("\n"); // space out the output
    printf ("value %x:\n", value);

    if (value & THING_1_MASK) printf (" THING_1\n");
    if (value & THING_2_MASK) printf (" THING_2\n");
    if (value & THING_3_MASK) printf (" THING_3\n");

    if (value & ANOTHER_MASK) printf (" ANOTHER_MASK\n");
    if (value & ANOTHER_MASK_2)
        printf (" ANOTHER_MASK\n");

    if ((value & ALL_ANOTHERS) == ALL_ANOTHERS)
        printf (" ALL ANOTHERS\n");
} // showMaskValue

int setBits (int value, int maskValue)
{
    // set a bit by just OR-ing in a value
    value |= maskValue;
    return (value);
} // setBits

int clearBits (int value, int maskValue)
{
    // to clear a bit, we and it with the complement of the mask.

    value &= ~maskValue;
    return (value);
} // clearBits

int main (int argc, char *argv[])
{
    int intval = 0;

    intval = setBits (intval, THING_1_MASK); // 00000001 = 0x01
    intval = setBits (intval, THING_3_MASK); // 00000101 = 0x05
    showMaskValue (intval);

    intval = setBits (intval, ALL_ANOTHERS); // 01100101 = 0x65

```

```
    intval = clearBits (intval, THING_2_MASK); // 01100101 = 0x65
    intval = clearBits (intval, THING_3_MASK); // 01100001 = 0x61
    showMaskValue (intval);

    return (0);

} // main
```

A sample run:

```
$ ./bitmask
```

```
value 5:
```

```
    THING_1
    THING_3
```

```
value 61:
```

```
    THING_1
    ANOTHER_MASK
    ANOTHER_MASK
    ALL ANOTHERS
```

Functions

Functions are C's way of breaking up code into smaller pieces. A function is composed of a declaration of a return type, the name of the function, and any parameters it takes. Following this declaration is a block of code surrounded by braces. Inside of the braces can be variable declarations and executable code. For example:

```
int someFunction (int a, float b, unsigned char c)
{
    int aLocalVar;

    printf ("the character is %c\n", c);
    aLocalVar = a + floor(b);

    return (aLocalVar * 2); // ok, so this does nothing useful
} // someFunction
```

The return statement causes the function to exit immediately and the expression that follows the return statement is used as the return value of the function. (The parentheses around the return value are optional, and their use is a personal preference.)

You can declare your function for others to use writing just the first declaration line:

```
int someFunction (int a, float b, unsigned char c);
```

This is called a “function prototype”, or just “prototype” for short. This function prototype tells the compiler “if you see someone calling **someFunction** it had better take three arguments, an `int`, a `float`, and an `unsigned char`. If someone is using the return value, it should be assigned to an `int`.” With that information the compiler can help error-check your code. Due to C's history, any functions that are used without including the prototype are assumed to return an `int` and take an arbitrary number of `int`-sized arguments. This can wreak havoc if you pass a value larger than an `int`, like a `double`. The calling-side and the function side get confused. Header files are handy places to put function prototypes.

Even though C has procedures (functions that return no values), they are still called functions. A C function that does not return anything is declared to return void. A function that does not take any parameters has void listed in its argument list:

```
void functionReturnsAndTakesNothing (void);
```

In C, parameters to functions are passed by value. That is, the functions get copies of the arguments. Inside of a function, you can fold, spindle and mutilate the value of the parameters all you want, and the caller will never notice. To achieve pass by reference, where changes to the variable are visible to the caller, you must use pointers, described in the next chapter.

Lastly, functions can be recursive, that is, they can call themselves. The canonical example of a recursive function is calculating factorials. That is, $X!$ is equal to $1 * 2 * 3 \dots * X-1 * X$

Example 1.9. factorial.c

```
// factorial.c -- calculate factorials recursively

/* compile with:
cc -g -Wall -o factorial factorial.c
*/

#include <stdio.h>          // for printf()

long long factorial (long long value)
{
    if (value == 1) {
        return (1);
    } else {
        return (value * factorial (value - 1));
    }
} // factorial

int main (int argc, char *argv[])
{
    printf ("factorial of 16 is %lld\n", factorial(16));

    return (0);
} // main
```

And here's a sample run:

```
$ ./factorial
factorial of 16 is 20922789888000
```

(factorials add up quickly)

Logical Expressions

In C, any expression (even assignment) can be used as a truth value. An expression evaluating to zero is considered a false value. Everything else is considered true. For instance, this is a valid loop:

```
while (result = read(...)) {
    // do stuff with result
    ...
}
```

read() reads a number of bytes from an open file. Upon a successful read it returns the number of bytes read. When the end of file is reached, it returns zero. When this loop is executed, the assignment inside of the parentheses is evaluated first.

When end of file is reached **read()** returns zero, **result** takes the value of zero, and the value of the expression is zero (false), so the loop terminates.

To compare value, use **==**. A common, common error is to use a single **=** (assignment) when you mean to use the **==** operator. The result of an **==** expression is a truth value. For instance:

- **5 == 10** results in zero. **5 == 5** results in a non-zero value (you cannot depend on the actual value, just that it is non-zero).
- **!=** is the “not equal” operator.
- **5 != 10** results in a non-zero value, and **5 != 5** results in zero.

You can also use **<**, **>**, **<=**, and **>=** for inequality tests.

Logical Operators

Expressions can be chained together with the logical operators, **&&** (AND) **||** (OR), **!** (NOT). Unlike their single-character siblings, logical AND and OR deal with the total truth value rather than individual bits. That is:

```
if (a && b) {  
    doSomething ();  
}
```

doSomething() is only invoked if both **a** and **b** are non-zero. Likewise,

```
if (a || b) {  
    doSomething ();  
}
```

doSomething() is only invoked if either **a** or **b** are non-zero.

The “not” operator (**!**), of course, flips the truth meaning of the expression.

```
if (!var) {  
    doSomething ();  
}
```

doSomething() is invoked only if **var** is zero.

Note that **&&** and **||** short circuit if the value of the total expression can be determined after the evaluation of the first part.

```
if ( ( x != 0 ) && ( y = z / x ) ) ...
```

is a handy way to check for zero before doing a divide. If **x** is 0, the first expression will evaluate to false. For a logical AND to be true both sides of the AND need to be true. Since the first part is false, it does not matter what the second part is since there is no way the entire expression can be true, so the second expression is not evaluated.

Likewise, with **||**, if the first part is true, the whole expression is true and there is no need to evaluate the second part.

Control Structures

The C control statements have found their way into many of the languages in common use. This will be a very quick overview of what they do.

Where you see expression below you can use any logical expression. Braces are put around the part where you put in code that gets executed. If you have only one line of code, you can omit the braces. Keeping everything fully braced makes adding code to the blocks easier, and it eliminates some classes of errors. (It is also a good idea in any case.)

if

```
if (expression) {
    // true-branch
} else {
    // false-branch
}
```

Evaluate the expression and execute one of the branches. The else part is optional. When you have nested if statements, the else matches to the closest if:

```
if (expression1) {
    if (expression2) {
        // true-true branch
    } else {
        // true-false branch
    }
}
```

This situation can look ambiguous without braces:

```
if (expression1)
    if (expression2)
        // true-true-branch
else
    // something else
```

This is erroneous indentation, but the behavior will be like that of the braced statements.

Related to the if statement is the conditional expression (also known as the ternary, trinary, or question-mark operator).

This

```
x = (expression) ? a : b;
```

is the same as

```
if (expression) {
    x = a;
} else {
    x = b;
}
```

but does it all in one expression.

while

```
while (expression) {  
    // executes while expression is true  
}
```

while executes its code zero or more times, so long as the expression evaluates to a true value. The expression is evaluated every time through the loop, before the body is executed.

for

```
for (initializer; expression; increment) {  
    // body  
}
```

This is mostly the same as this:

```
initializer;  
  
while (expression) {  
    // body  
    increment;  
}
```

Here is a real for loop:

```
for (i = 0; i < 10; i++) {  
    printf ("%d\n", i);  
}
```

This prints the numbers from zero to 9.

for is mostly the same as the while loop shown above, except in the face of the continue statement. continue can be used in any loop to immediately jump to the beginning of the loop, bypassing any subsequent instructions. With the for loop, the increment part is performed on a continue, while with the while loop given above, the increment would not happen.

The break statement breaks out of the loop entirely, bypassing the increment stage. break can be used to break out of any loop. When nested loops are involved, break only breaks out of the innermost one. To break out of multiple loops you need to use a goto.

do-while

```
do {  
    // body  
} while (expression);
```

The body of the loop is executed one or more times. If expression evaluates to zero, the loop terminates.

switch

switch is C's case statement. Given an integral value, choose a block of code to execute.

For instance:

```
myChar = getchar();  
switch (myChar) {  
    case 'a': printf ("got an a\n"); break;
```

```

    case 'e': printf ("got an e\n"); break;
    case '9': printf ("got a 9\n");
    case '8': printf ("got an 8\n");
    case '7': printf ("got a 7\n"); break;
    default: printf ("got something else");
}

```

There are some properties of C's switch statement that trip up programmers used to other languages. The first is that the value given to switch is an integer value. It cannot be a float or a character string. A single character is OK since a char is really an int type. The second is that the case labels must also be integer, constant values. You cannot use variables or strings there. In other words, this is very illegal:

```

int x = 5;
switch (someVar) {
    case "hello": // can't use a string
    case x: // can't use a variable
    case 'x': // this is ok. it's a constant
}

```

Another bit of odd behavior is that when a case label starts executing, code flow will go straight down until it hits a break. In the example above, if myChar is 9, the program will print out

```

got a 9
got an 8
got a 7

```

Because execution will start at the case '9' label, drop down into the '8' label and the '7' label before hitting a break.

goto

goto is an unconditional branch to a label anywhere in the same function the goto is used. You can even goto the middle of a for or a switch if you wish.

For instance:

```

for (i = 0; i < 10; i++) {
    for (j = 0; j < 10; j++) {
        if ((i == 5) && (j == 9)) {
            goto bailout;
        }
    }
}

bailout:
printf ("done!\n");

```

For the More Curious: Multiple Source Files

Once a program gets beyond a trivial stage it will require using multiple source files. In C, each file is treated as an independent unit.

Within each file there can be variables global to just that file as well as functions that are only visible in that file. When you precede the variable or function declaration with `static` visibility of the variable or function becomes restricted to only that source file. If you do not declare functions or global variables `static`, they will be visible to other files.

“Visibility” means visibility to the linker so that the linker can patch up references between files.

Here is an example program composed of 4 files: two source files and two header files.

Example 1.10. file-1.h

```
extern int g_globalVar;
```

Example 1.11. file-1.c

```
// file-1.c -- half of a program split across two files

/* compile with:
cc -g -Wall -c file-1.c
*/

/* link with
cc -o multifile file-1.o file-2.o
*/

// include our definitions
#include "file-1.h"

// see the stuff file-2 is exporting
#include "file-2.h"

#include <stdio.h>      // for printf()

static int localVar;

int g_globalVar;

static void localFunction ()
{
    printf ("this is file 1's local function. ");
    printf ("No arguments passed.  localVar is %d\n", localVar);
} // localFunction

int main (int argc, char *argv[])
{
    float pi;

    localFunction ();
    pi = file2pi ();

    localVar = 23;
    g_globalVar = 23;

    printf ("g_globalVar before is %d\n", g_globalVar);
    printf ("localVar before is %d\n", localVar);

    file2Function ();

    printf ("g_globalVar was changed to %d\n", g_globalVar);
    printf ("localVar after is still %d\n", localVar);
    return (0);
} // main
```

Example 1.12. file-2.h

```
// returns the value of Pi
float file2pi (void);

// changes the value of g_globalVar
void file2Function (void);
```

Example 1.13. file-2.c

```
// file-2.c -- second half of a program split across two files

/* compile with:
cc -g -Wall -c file-2.c
*/

#include "file-1.h"
#include "file-2.h"

static double localVar;

static float localFunction (char dummy)
{
    return (3.14159);
} // localFunction

float file2pi (void)
{
    return (localFunction('x'));
} // file2pi

// changes the value of g_globalVar
void file2Function (void)
{
    g_globalVar = 42;
    localVar = 1.2345;
} // file2Function
```

This one is a little more complicated to compile and link:

```
$ cc -g -Wall -c file-1.c
$ cc -g -Wall -c file-2.c
$ cc -o multifile file-1.o file-2.o
```

When you have multiple files like this, you would usually use makefiles or an IDE like Xcode to put things together.

A sample run:

```
$ ./multifile
this is file 1's local function. No arguments passed.  localVar is 0
g_globalVar before is 23
localVar before is 23
g_globalVar was changed to 42
localVar after is still 23
```

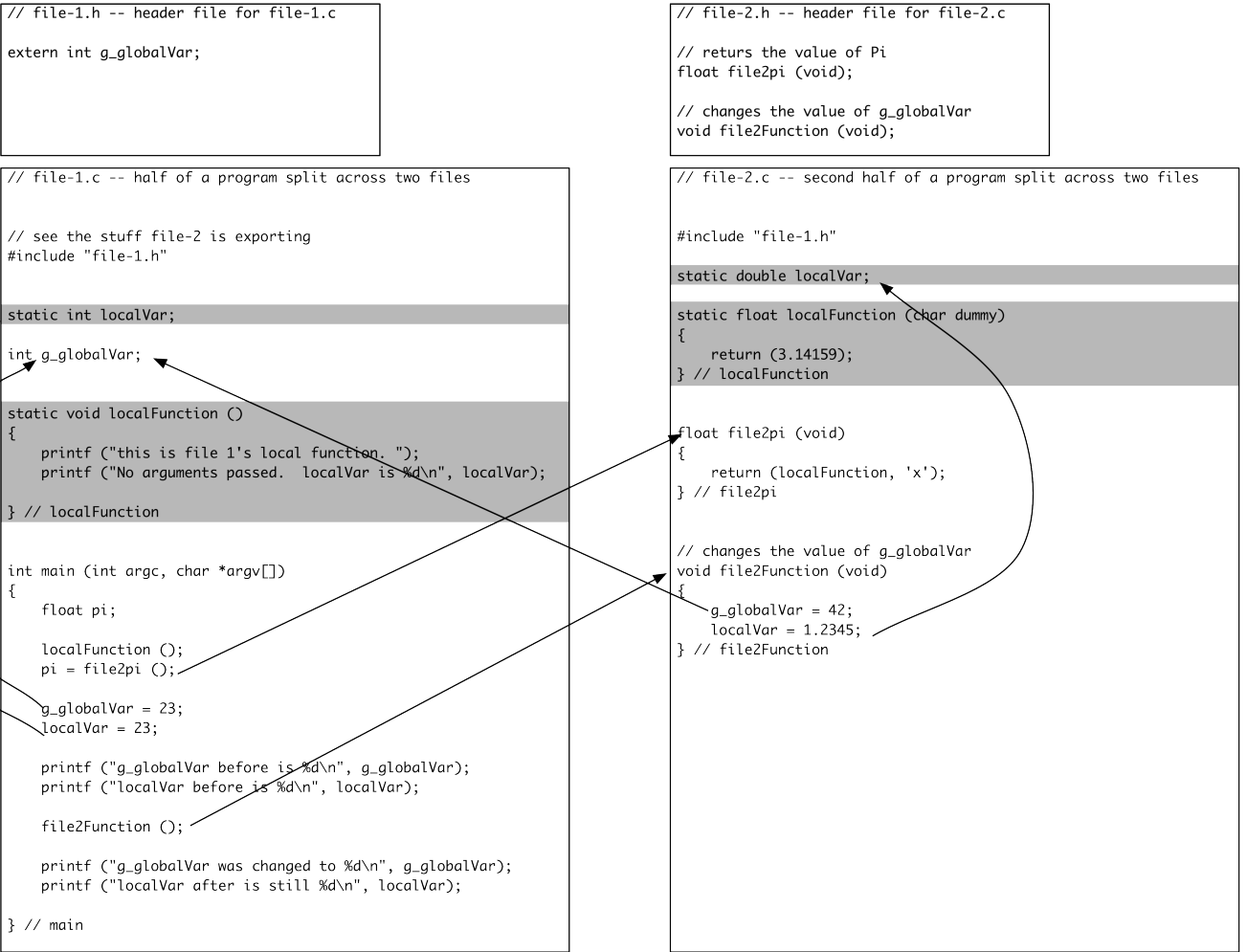
Some things of interest to note:

- The use of `extern` in `file-1.h`. That declaration is what will be in `file-2.c`; that is, `file-2.c` knows about the existence of a global variable called `g_globalVar`, but the compiler does not

- actually allocate any storage for it since the variable was declared extern. When the compiler compiles file-1.c, it does not have g_globalVar declared as extern, and so the compiler will allocate some space for it.
- Static variables and functions are truly private to the file. Both of the source files have their own localVar and localFunction with radically different definitions.
 - The header file is the means of communication between the two files. The headers contain the “published API,” or set of features that the particular file brings to the table. Implementation details are left to the innards of the file.

This shows who can see and call what:

Figure 1.14 Two-File Scope



For the More Curious: Common Compiler Errors

The C compiler can be pretty dumb when it encounters errors. Usually a simple mistake will cascade into a huge flurry of syntax errors. In general, when you get a huge pile of errors, fix the first couple and then try compiling again, especially if the errors start looking weird.

For example, consider the program `variables.c`. If you remove a single semicolon (after the constant 3.14159) you will get these errors:

```
$ cc -g -o variables variables.c
variables.c:13: illegal declaration, missing ';' after '3.14159'
cpp-precomp: warning: errors during smart preprocessing, \
retrying in basic mode
variables.c:16: parse error before "void"
variables.c:22: conflicting types for 'myShort'
variables.c:19: previous declaration of 'myShort'
variables.c:22: warning: data definition has no type or storage class
variables.c:23: warning: data definition has no type or storage class
variables.c:25: parse error before '++' token
variables.c:27: parse error before string constant
variables.c:28: warning: conflicting types for function 'printf'
variables.c:28: warning: data definition has no type or storage class
```

For the first error, it is pretty obvious what went wrong (which makes it easy to fix). The subsequent errors and warnings are bogus. Add in that one semicolon back and life will be happy.

Here are some errors you will probably see, and what they (probably) mean. (Do not worry if some of them are gibberish now. Many of the concepts will be explained in the next chapter.)

warning: ANSI C forbids newline in string constant	Missing double quote from the end of a literal string.
structure has no member named 'indData' union has no member named 'shoeSize' 'aGlobalint' undeclared (first use in this function)	Using a variable or structure member that has not been declared. Double check the case and spelling (both are significant).
warning: control reaches end of non-void function	A function is declared as returning a value, but there is no return statement.
unterminated #if conditional	You have an #if or #ifdef statement, but not the closing #endif. Check to make sure you have end #endifs when nesting them.
illegal function call, found 'blah'	Check to see if there is a missing "?" using the ?: operator. For example, this code: (*ptrvar) "blah" : "ook"; will generate this error.
illegal member declaration, missing name, found '}'	This happens if you forget the semicolon at the end of a union declaration. typedef struct Individual {

```
int type;
union {
    Person person;
    Alien alien;
} indData
} Individual;
```

will cause this error.

`Alien' redeclared as
different kind of symbol

Make sure you do not use the same name for different purposes, like having `typedef struct Alien;` followed by `enum { Human, Alien }.`

undefined type, found
`individual'

The type of a parameter in a function declaration does not exist. For instance, this error was caused by

```
void fillAlien (individual *individual)
```

The proper type name is `Individual` (upper case I).

warning: comparison between
pointer and integer

Make sure the types on either side of the `==` or `!=` sign make sense.

Challenge

The Fibonacci number sequence is a sequence that looks like this: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... That is, it starts with 1, 1, then subsequent numbers are the sum of the previous two.

There are two ways of calculating the n^{th} fibonacci number, either iteratively (run a loop and calculate the numbers), or recursively (finding the n^{th} Fibonacci number by calculating the $n-1$ and $n-2$ Fibonacci number.) For Fibonacci numbers $F(1)$ and $F(2)$, return a value of 1.

Implement both methods of calculating the Fibonacci numbers and compare their performance characteristics.

A C Refresher, Part 2: Pointers, Structures, and Arrays

The single C feature that seems to cause most programmers a lot of problems are pointers. Pointers are the use of memory addresses to indicate what data to operate on, providing a layer of indirection that becomes handy for solving a lot of problems. Unfortunately the syntax is a bit odd, and it is easy to get confused amongst all the &s and *s flying around.

C structures, the aggregation of individual basic data types, are used to package smaller bits of data into a cohesive whole, so you can abstract data to a higher level. It is easier to tell a function, “Here is a network address,” rather than telling it individually, “Here is a machine IP address, here is a port, here is the network protocol to use.” Structures and pointers work closely together, allowing structures to reference other structures to build webs of related variables.

Finally, arrays are linear collections of same-type chunks of data. Arrays and pointers are very closely related, and ultimately understanding both arrays and pointers are necessary to fully grasp how C deals with addresses and data.

Pointers

So, what is a pointer? It is nothing more than the address of a byte in memory. What can you do with a pointer? You can read bytes starting from the address and you can write bytes starting at the address. You can also calculate new addresses based on the original one.

Imagine that you like the book *Zen and the Art of Motorcycle Maintenance*. You come across a particular passage that you think your friend would find interesting. You can either mail the book over (which can be expensive), or you can send a postcard that says, “Zen, page 387, starting at the first letter.” When he gets the postcard, he knows where to look to find the interesting passage.

A C pointer is like this postcard. It is a small amount of storage (there is only so much you can write on a postcard) that references something larger. It is also a lot cheaper to send around.

Pointer syntax

Pointer syntax involves two special characters, the star (*) and the ampersand (&). The star pulls a double duty; in some cases it is a description, and some cases it is an action. This duality is one of the major causes of confusion.

Here are the fundamental pieces of pointer syntax:

1. Pointer declaration

```
char *ptrvar;
```

`ptrvar` is the postcard. On the Mac this variable is 4 bytes long and can point to any address the program can access. The star in this case is for declaration. Reading the declaration backwards it says “`ptrvar *` (is a pointer to) a `char` value.”

Something like

```
int *fooby;
```

says “`fooby *` (is a pointer to) an `int` value.”

Just like other variable declarations, if these are found as variables in a function they will have random values, like picking up a postcard from the recycling stack to erase and send again. In any case you should initialize the variable.

2. Getting an address

The ampersand (&) yields an address in memory when it is applied to a variable. This gives you the value to write on your postcard.

```
int myVar;  
int *pointerToMyVar;  
pointerToMyVar = &myVar;
```

Now `pointerToMyVar` contains the address of the first byte of `myVar`. Equivalently, the postcard `pointerToMyVar` has written on it where to find the beginning of the number that `myVar` stores.

3. Getting data from an address

The star is used again, this time as an action. When a star is used on a pointer variable on the right-hand side of an assignment, it means “fetch.” It is the same as reading the location in the book from the postcard, opening the book, turning to the page, and then reading at the first letter there.

```
int myVar = 12;  
int *pointerToMyVar = &myVar;  
int anotherInt;  
  
anotherInt = *pointerToMyVar;
```

Reading from right to left again, you have:

`pointerToMyVar:`

Look at `pointerToMyVar`. Use the address there.

`*`:

Go fetch an `int`'s worth of bytes from that address.

`=:`

Assign that `int`'s worth of bytes.

`anotherInt:`

To this variable.

Now `anotherInt` has the value 12.

4. Putting data to an address

Once again the star is used. When it is used on a pointer on the left-hand side of an assignment, it means “put”. It is like reading the book location from the postcard, opening the book, turning to the page, then erasing the letter there and writing in a new letter.

Using the declarations above, this code:

```
*pointerToMyVar = 42;
```

reading from right to left means:

```
42:
```

```
==:
```

```
pointerToMyVar:
```

```
*:
```

Take this constant value
and assign it. To figure out where you want to
assign it, look at
for where in memory to start storing the bytes
that comprise the value 42
now that you know where to store the data,
actually put the bytes there.

Without the star on the assignment, you would have `pointerToMyVar = 42;` which would be wrong. That would write “42” onto the postcard, rather than on the pages of the book.

An actual use of pointers: pass by reference

As mentioned in the last chapter, arguments to functions are passed by value, meaning that the values of the arguments are duplicated and then these duplicates are given to the function. The function is free to change them without any impact on the caller.

Sometimes you want to pass arguments by reference and have the function fill in values so that the caller can see the change. This is useful if you are returning more than one result from a function, or you want the function to write into a pre-allocated buffer.

Pointers are used in this case. If you want a function to change an argument, have the function take a pointer argument, and then pass a pointer to your variable. Here is an example:

Example 2.1. pass-reference.c

```
// pass-reference.c -- show pass by reference using pointers

/* compile with:
cc -g -Wall -o pass-reference pass-reference.c
*/

#include <stdio.h>          // for printf()

void addemUp (int a, int b, int *result)
{
    *result = a + b;
} // addemUp

int main (int argc, char *argv[])
{
    int answer;
```

```
    addemUp (1, 2, &answer);

    printf ("1 + 2 = %d\n", answer);
    return (0);

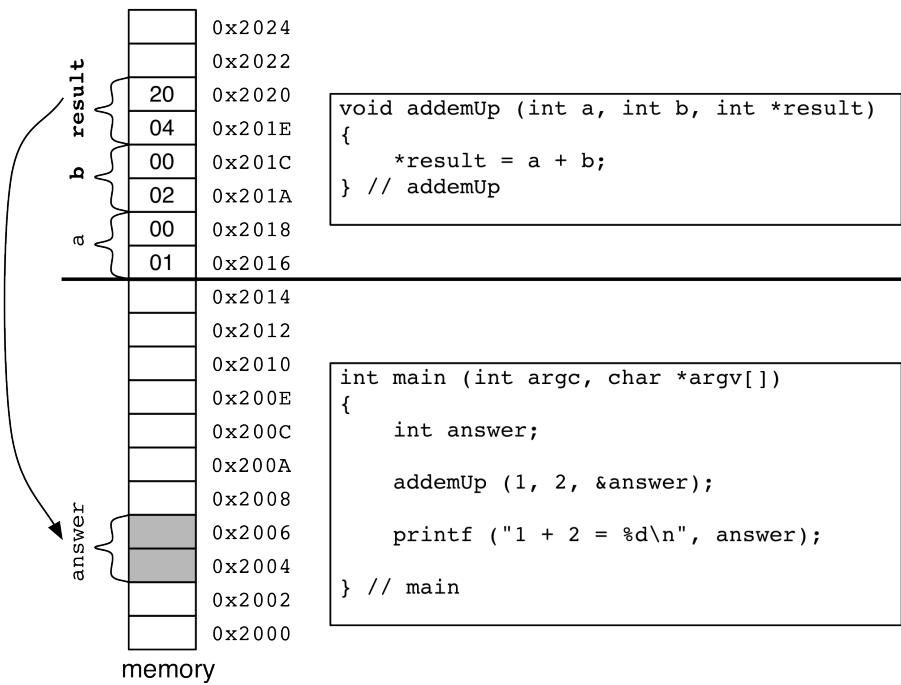
} // main
```

A sample run:

```
$ ./pass-reference
1 + 2 = 3
```

Here is what is happening in memory:

Figure 2.1 Pass By Reference



The storage for the variable `answer` lives at address `0x2004`, and extends for 4 bytes through address `0x2007`. The **addemUp** function gets called. The numeric values get written to memory (pass by value, remember), and the address of the `answer` variable (`0x2004`) gets written to this parameter memory.

When the function executes

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

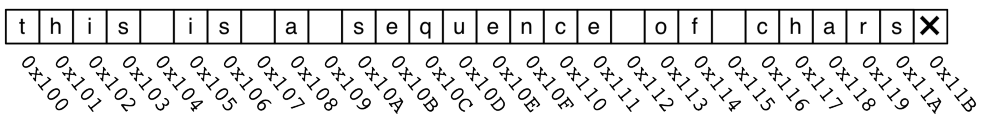
This is what it does:

1. Figures out the values of a (1) and b (2).
2. Adds 1 + 2, and stores the result (3) somewhere, probably a processor register.
3. Looks at the value of the result variable (0x2004).
4. Writes the result (a four-byte int value that happens to be 3) starting at address 0x2004, which happens to be the same place the variable answer is stored.

Another use for pointers: strings

C has no string type. There is a convention, though, where a run of characters ending with a zero byte is considered a “string.” This is what a string looks like in memory (where the big X is a byte with a value of zero):

Figure 2.2 C String



Starting at any address from 0x100 through 0x11B will result in a valid C “string.” You can use such a string like this:

```
char *myString;
myString = "this is a sequence of chars";
```

Pulling this code apart you have:

<code>char *myString</code>	<code>myString</code> is a pointer to a character. This is the postcard where the location of the first letter of a sequence of characters starts.
<code>"this is..."</code>	a sequence of bytes, terminated by a zero byte, that the compiler and linker stick into read-only space. A literal string expression evaluates to the address of the first byte of the sequence.
<code>=</code>	Will assign the address value.
<code>myString</code>	The postcard you are writing on has the location of the first “t” of “this is a sequence of chars”

`myString` now has the value of 0x100, the address of the first character.

Note the continued pedantry of referring to “the first something of a sequence of somethings.” Pointers have no intrinsic knowledge of what they point to. `myString` could point to the beginning of “this is a sequence of chars,” or it could be made to point into the middle.

Chapter 2 A C Refresher, Part 2: Pointers, Structures, and Arrays

OK, now that you have `myString` pointing to the first byte of the literal string, what can we do with it?

You can get the first letter of the string by doing

```
char charVar = *myString;
```

This assigns the value of “t” to `charVar`.

You can increment the pointer:

```
myString++;
```

That makes `myString` point to the next character. Note that since there is no star attached to this expression, nothing is done with the value being pointed to. There is no star operator to initiate a reading or writing of the pointed-to value. This just changes the location value on the postcard.

`myString` now has the value of `0x101`, the address of the second character.

```
charVar = *myString;
```

assigns the value of “h” to `charVar`.

You can calculate the length of a “string” like this by counting characters until you hit the zero byte at the end. For example:

Example 2.2. `strlen1.c`

```
// strlen1.c -- calculate the length of a C string

/* compile with:
cc -g -Wall -o strlen1 strlen1.c
*/

#include <stdio.h>          // for printf()

int main (int argc, char *argv[])
{
    char *myString = "this is a sequence of chars";
    int length = 0;

    while (*myString != '\0') {
        length++;
        myString++;
    }

    printf ("the length of the string is %d\n", length);
    return (0);
} // main
```

See how it runs:

```
$ ./strlen1
the length of the string is 27
```


Things of note: the expression `'\0'` is a way of signifying a zero byte. You could use just a naked zero in the expression, but using `'\0'` makes it clear to whomever is reading the code that a character value is being used.

The above idiom, using a pointer to scan through a sequence of data, is very common in C, and it will frequently be abbreviated. This is the same program, but written more idiomatically:

Example 2.3. `strlen2.c`

```
// strlen2.c -- calculate the length of a C string, idiomatically

/* compile with:
cc -g -Wall -o strlen2 strlen2.c
*/

#include <stdio.h>          // for printf()

int main (int argc, char *argv[])
{
    char *myString = "this is a sequence of chars";
    int length = 0;

    while (*myString++) {
        length++;
    }

    printf ("the length of the string is %d\n", length);
    return (0);
} // main
```

Pull apart the inner loop:

```
while (*myString++) {
    length++;
}
```

`*myString++` is the same as `*myString` followed by `myString++`. What does that mean?

`*myString` is executed, which looks at the postcard (`myString`), follows the address and fetches a value (the star operator). That value is then used to decide whether the body of the while loop should be evaluated. Then `myString++` is evaluated, which updates the location written on the postcard to indicate the next letter to process.

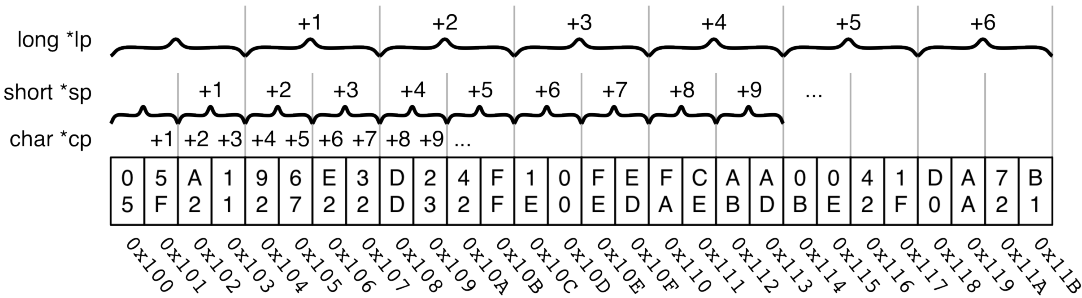
As an aside, this loop is how the standard library call **`strlen()`** is implemented. This is an $O(N)$ operation, so it is not a function you would call if you do not have to (like every time in the controlling expression of a while loop).

Pointer math

Given a pointer to an address in memory, you can calculate other addresses. Big deal. What is kind of nice is that C takes into account the size of the data the pointer refers to and scales the arithmetic accordingly.

In this example, all the pointers are pointing at the same starting address, but the additions refer to different bytes.

Figure 2.3 Pointer Math



The same sequence of (random) bytes are interpreted with different pointers.

Assume that lp, sp, and cp start off with the same value, 0x100. When you do something like `sp + 3`, the address (`sp + (3 * sizeof(pointer-base-type))`) is calculated. In the case of a short pointer (where a short is two bytes), `sp` is 0x100, `sp + 1` is 0x102, `sp + 2` is 0x104, and so on.

If `lp` is a long pointer, `lp` is 0x100, `lp + 1` is 0x104, `lp + 2` is 0x108, and so on.

You dereference (fetch the value of) a pointer that is being added to like this:

```
long longval = *(lp + 5);
```

Which works like this:

Table 2.1. Play-by-play

Code	Behavior
lp + 5	Start with a base address of lp (0x100). Scale 5 by the sizeof(long) so that you get 20 (hex 14). Add them together to get 0x114. This becomes a temporary postcard with 0x114 written on it.
*:	Go to address 0x114, and fetch a long's worth of bytes (in this case, it would be 0x0b0e421f)
longval =	Copy the value 0x0b0e421f into longval

Since ++ is the same as adding 1, it too gets scaled by the size of the type.

If you had a loop like this:

```
long *lp = 0x100;
for (i = 0; i < 5; i++) {
    printf ("%d: %x\n", i, *lp);
    lp++;
}
```

You would get output like this:

```

0: 0x055fa211
1: 0x9267e232
2: 0xdd2342ff
3: 0x1e00feed
4: 0xfaceabad

```

For comparison, here it is with a short (2 bytes) pointer:

```

short *sp = 0x100;
for (i = 0; i < 5; i++) {
    printf ("%d: %x\n", i, *sp);
    sp++;
}

```

```

0: 0x055f
1: 0xa211
2: 0x9267
3: 0xe232
4: 0xdd23

```

And for completeness, with a char (1 byte) pointer:

```

char *cp = 0x100;
for (i = 0; i < 5; i++) {
    printf ("%d: %x\n", i, *cp);
    cp++;
}

```

```

0: 0x05
1: 0x5f
2: 0xa2
3: 0x11
4: 0x92

```

It is the same data in memory, the same starting address, just different pointer types.

The NULL pointer

The NULL pointer, having a zero value, is what you use to say, “This pointer points to nowhere.” It is a handy initialization value, which happens automatically for global and static variables. In general, trying to dereference a NULL pointer will crash your program.

Structures

Declaring structures

Structures are C's way of aggregating a bunch of different pieces of data under one name (like a table definition in SQL). For instance:

```

struct Person {
    char    *name;
    int     age;
    char    gender; // use 'm', 'f'
    double  shoeSize;
}

```

```
};
```

You can then declare instances of `Person` like this:

```
struct Person perfectMate;
```

and set/read the structure field members like this:

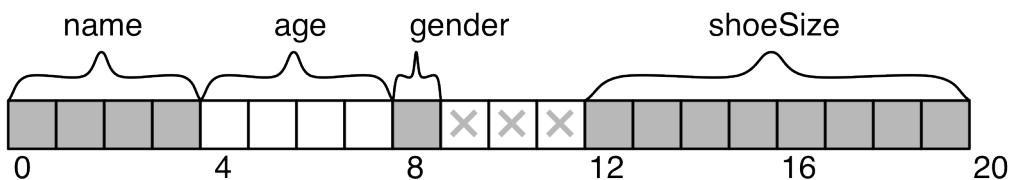
```
perfectMate.name = "Brenda";  
perfectMate.age = 23;  
perfectMate.gender = 'f';  
perfectMate.shoeSize = 8.5;
```

```
printf ("my perfect mate is named %s, and is %d years old\n",  
        perfectMate.name, perfectMate.age);
```

Use a dot between the name of the variable and the name of the field to use.

The layout of the structure in memory looks like this:

Figure 2.4 Structure Layout



The elements in the structure are laid down in the order they are declared. Note the three bytes that are X'd out. The compiler puts in padding bytes so that types are aligned on byte boundaries that maximize accessing performance. `shoeSize`, being a double, is on a 4-byte boundary. If it were closely packed in against the gender byte the CPU would have to do multiple fetches and assemble the data every time it was needed (this is much slower than if it is aligned and the processor can grab it all at once).

One thing to notice is the declaration of `name`. It is `char *name`. This tells the compiler to leave space for a postcard to find the actual name. The name itself will not be stored in the struct, it will just be referenced elsewhere.

Pointers to structs

`sstructs` and pointers are often used together. `structs` can become arbitrarily large, so passing them by value (copying all the bytes) can become expensive. Also, it is pretty common to have a function fill in the members of a structure. Pointers come to the rescue here again.

Example 2.4. `struct-point.c`

```
// struct-point.c -- some structure, pointer stuff  
  
/* compile with:
```

```

cc -g -Wall -o struct-point struct-point.c
*/

#include <stdio.h>          // for printf()

struct Person {
    char    *name;
    int     age;
    char    gender; // use 'm', 'f'
    double  shoeSize;
};

void populatePerson (struct Person *person)
{
    person->name = "Bork";
    person->age = 34;
    person->gender = 'm';
    person->shoeSize = 10.5;
} // populatePerson

void printPerson (struct Person *person)
{
    printf ("name:      %s\n", person->name);
    printf ("age:       %d\n", person->age);
    printf ("gender:    %s\n",
            (person->gender == 'm') ? "male" : "female");
    printf ("shoe size: %f\n", person->shoeSize);
} // printPerson

int main (int argc, char *argv[])
{
    struct Person me;

    populatePerson (&me);
    printPerson (&me);

    return (0);
} // main

```

The output

```

$ ./struct-point
name:      Bork
age:       34
gender:    male
shoe size: 10.500000

```

Rather than using a dot to separate a variable from the field you are referencing, you use the cool-looking arrow operator: `->` (minus + greater than) An expression like `person->name` is a shorthand for `(*person).name`.

One very powerful feature of pointers to structs is that you can employ more interesting data structures like linked lists and trees. The nodes of your data structure can refer to each other via pointers.

Bitfields

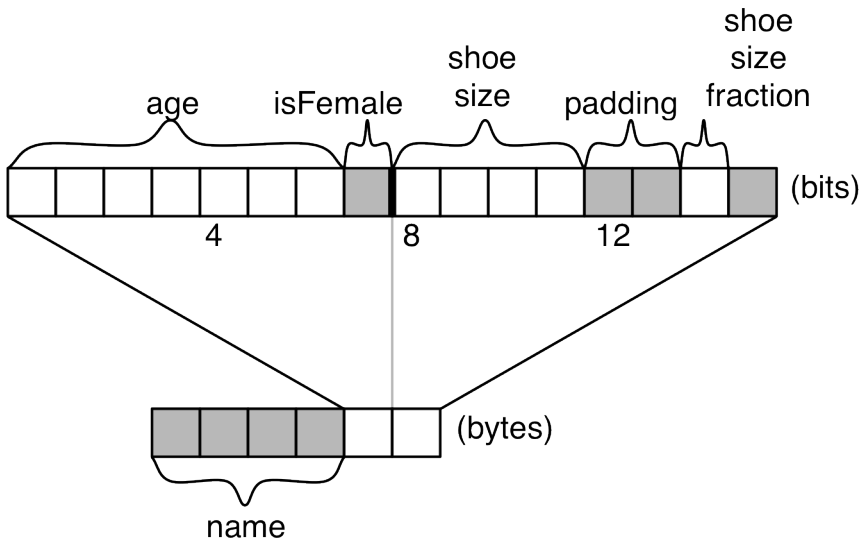
C lets you specify bit layout for individual fields in a struct.

A struct like this:

```
struct PackedPerson {
    char *name;
    unsigned int age : 7;           // enough to hold 127 years
    unsigned int isFemale : 1;      // set if female, clear if male
    unsigned int shoeSize : 4;      // support up to a size 15
    unsigned int padding : 2;       // add breathing room for BigFoot
    unsigned int shoeSizeFraction : 1; // set if this is a half-size
}
```

would have a memory layout like this:

Figure 2.5 Packed Structure Layout



`struct PackedPerson` only takes 6 bytes of storage (vs 20 bytes for just a regular `struct Person`) The access time for the bitfields will be slower since the compiler is generating shifting and masking instructions to pack and unpack the data.

Of course, you can use masks and bitwise operators to do this stuff manually, but sometimes these bitfields can be more convenient.

Typedef

Up in `struct-point.c`, every time you reference the `struct Person` structure in parameter lists and variable declarations you have to include the `struct` keyword. That can get pretty annoying pretty quickly. C allows you to associate an alias name that is easier to deal with, using the `typedef` facility:

```
struct Person {
```

```

    char    *name;
    int      age;
    char     gender; // use 'm', 'f'
    double   shoeSize;
};

```

```
typedef struct Person Person;
```

Now you can use just `Person` everywhere:

```

void populatePerson (Person *person);
void printPerson (Person *person);
...
    Person me;

```

And given C's love for brevity, you can combine those two:

```

typedef struct Person {
    ...
} Person;

```

Casts

A cast is an operation where you tell the compiler to override the default interpretation of a particular variable. You can use casts to tell the compiler to treat an `int` value as if it were floating point:

```

int thing = 5;
float blah = (float) thing + 3.1415;

```

Here the compiler will convert `thing` to a floating point value before adding. (The compiler will do this anyway, but you can use the explicit cast to tell the reader what is going on.) Casts can also be used to go between signed and unsigned values. It is a way of telling the compiler, “Hey, I know what I am doing. Go ahead and reinterpret the bits.”

Casts are also used to convert between pointer types.

You can have something like this:

```
unsigned char *bytePointer = ....;
```

You have a pointer to a sequence of bytes:

```
int blah = *((int *)bytePointer);
```

The `(int *)bytePointer` cast tells the compiler to treat `bytePointer` as if it were a pointer to ints rather than a pointer to chars. So, when you dereference `*((int *)bytePointer)`, you will be picking up an int's worth of bytes. If you just did `*bytePointer`, you would only pick up one byte's worth.

Similarly you can use casts on structure pointers:

```

int *blah = ....; // a pointer somewhere
Person *person = (Person *)blah;

```

now `blah` and `person` both point to the same address. You can now do things like

`printf ("name is %s\n", person->name);` and it will pick up the bytes that started where `blah` pointed. In most cases this would probably crash your program by interpreting the bytes in a nonsensical manner, but the compiler lets you do this since it trusts that you know what you are doing.

void *

There are times when you want to say, “Here is a pointer to something. I do not really care what it is, it is just an address in memory.” The return values from dynamic memory functions are like this. The C type `void *` is what indicates a pointer-to-anything. When you get a `void *`, you typically assign it to a pointer variable of a particular type and then manipulate it, or you cast it to the type you want.

Function pointers

You can have pointers to code as well as pointers to data. This allows a degree of abstraction by allowing functions to act generically and delegate their work to another function. You can do object-oriented programming in C (polymorphism and dynamic dispatch) using function pointers.

The syntax for declaring function pointers is a bit odd:

```
void (*pointer1)(int, int);
```

which means a pointer to a function that takes two `ints` and returns nothing.

```
int (*pointer2)(void);
```

is a pointer to a function that takes no arguments and returns an integer.

You invoke a function by something like this:

```
(pointer1)(5, 5);
```

or

```
int myvar = (pointer2)();
```

In fact, the parentheses can be omitted too. I like leaving them in that it makes it obvious that a function pointer is being used rather than an explicit function.

This little program invokes a function to print out an integer via a function pointer.

Example 2.5. function-pointer.c

```
// function-pointer.c -- play with function pointers

/* compile with:
cc -g -Wall -o function-pointer function-pointer.c
*/

#include <stdio.h>          // for printf()

void printAsChar (int value)
{
```



```

    printf ("%d as a char is '%c'\n", value, value);
} // printAsChar

void printAsInt (int value)
{
    printf ("%d as an int is '%d'\n", value, value);
} // printAsInt

void printAsHex (int value)
{
    printf ("%d as hex is '0x%x'\n", value, value);
} // printAsHex

void printIt (int value, void (*printingFunction)(int))
{
    (printingFunction)(value);
} // printIt

int main (int argc, char *argv[])
{
    int value = 35;

    printIt (value, printAsChar);
    printIt (value, printAsInt);
    printIt (value, printAsHex);
    return (0);

} // main

```

The program in action:

```

$ ./function-pointer
35 as a char is '#'
35 as an int is '35'
35 as hex is '0x23'

```

Unions

Unions are a way of interpreting the bits stored in a structure in two different ways.

Consider this:

```

typedef struct Person {
    char    *name;
    int     age;
    char    gender;    // use 'm', 'f'
    double  shoeSize;
} Person;

typedef struct Alien {
    char    *designation;
    char    bloodType; // use 'Q', 'X', or "@"
    int     hearts;
    short   psiPower;
    short   ducks;     // can be negative
} Alien;

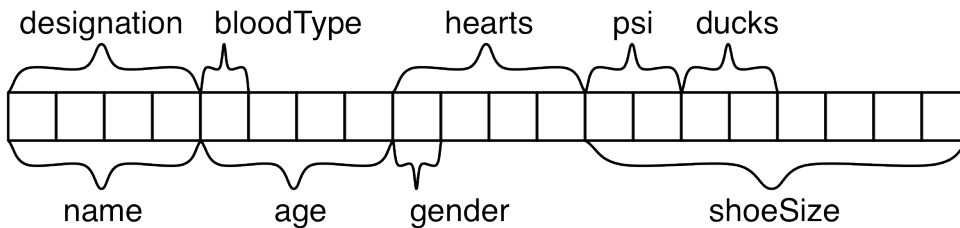
enum { Human, NonHuman };

```

```
typedef struct Individual {
    int type;
    union {
        Person person;
        Alien  alien;
    } indData;
} Individual;
```

The layout of Individual looks like this:

Figure 2.6 Layout of Individual Union



You can store an Alien's worth of data or a Person's worth of data in the same space, but not both at the same time (you will need a second chunk of memory to do that). Unions let you overlay storage like this in order to reduce type proliferation.

Referencing the fields of unions is verbose. You need to specify:

```
variableName.union-name.union-type-name.field;
```

To get the number of hearts for an alien, you need to do:

```
myAlien.indData.alien.hearts;
```

It looks like a lot of extra work to drill down into the union like that. There is not a run-time cost for these, just wear and tear on your keyboard (some `#defines` can come in handy). Internally the compiler calculates the offset from the beginning of the structure and fetches data from there.

Here is a program that uses this union. The type struct member tells folks who receive one of these structures what type it is, so they know which side of the union to use.

Example 2.6. union.c

```
// union.c -- play with unions

/* compile with:
cc -g -Wall -o union union.c
*/

#include <stdio.h>          // for printf()

typedef struct Person {
    char    *name;
    int     age;
```

```

    char    gender;    // use 'm', 'f'
    double  shoeSize;
} Person;

typedef struct Alien {
    char    *designation;
    char    bloodType; // use 'Q', 'X', or "@"
    int     hearts;
    short   psiPower;
    short   ducks;     // can be negative
} Alien;

enum { Human, NonHuman };

typedef struct Individual {
    int type;
    union {
        Person person;
        Alien  alien;
    } indData;
} Individual;

void fillAlien (Individual *individual)
{
    individual->type = NonHuman;
    individual->indData.alien.designation = "qwzzk";
    individual->indData.alien.bloodType = 'X';
    individual->indData.alien.hearts = 7;
    individual->indData.alien.psiPower = 2870;
    individual->indData.alien.ducks = 3;
} // fillAlien

void fillPerson (Individual *individual)
{
    individual->type = Human;
    individual->indData.person.name = "Bork";
    individual->indData.person.age = 34;
    individual->indData.person.gender = 'm';
    individual->indData.person.shoeSize = 10.5;
} // fillPerson

void printIndividual (Individual *individual)
{
    if (individual->type == Human) {
        printf ("Human:\n");
        printf ("  name:      %s\n", individual->indData.person.name);
        printf ("  age:       %d\n", individual->indData.person.age);
        printf ("  gender:    %c\n", individual->indData.person.gender);
        printf ("  shoeSize:  %f\n", individual->indData.person.shoeSize);
    } else if (individual->type == NonHuman) {
        printf ("NonHuman:\n");

        printf ("  designation: %s\n",
            individual->indData.alien.designation);
        printf ("  bloodType:   %c\n",
            individual->indData.alien.bloodType);
        printf ("  hearts:      %d\n",
            individual->indData.alien.hearts);
        printf ("  psiPower:    %d\n",

```

```

        individual->indData.alien.psiPower);
    printf ("   ducks:           %d\n",
        individual->indData.alien.ducks);

    } else {
        printf ("oops, bad union qualifier\n");
    }
} // printIndividual

int main (int argc, char *argv[])
{
    Individual being;

    fillAlien (&being);
    printIndividual (&being);

    fillPerson (&being);
    printIndividual (&being);

    return (0);
} // main

```

Arrays

The last stop on the tour are arrays. Arrays are ordered collections of data of a uniform type, and are similar to arrays in other languages like Java and Pascal.

You can declare an array like this:

```
int intArray[20];
```

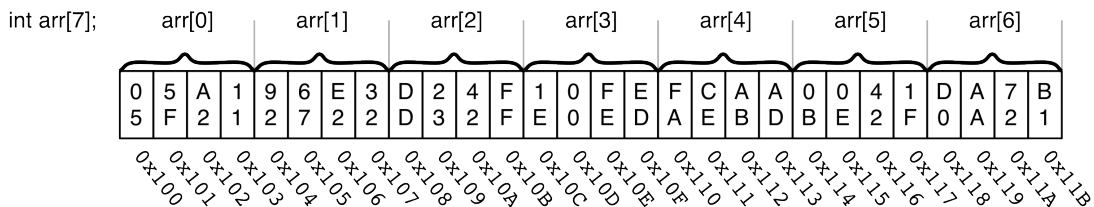
This makes an array of 20 elements. You index the array to read and write values like this:

```
intArray[2] = 20;
int intVal = intArray[6];
```

Like all variables, if an array is declared in a function, it will have random values. Global (non-static and not declared in a function) or static arrays will be initialized to all zeros.

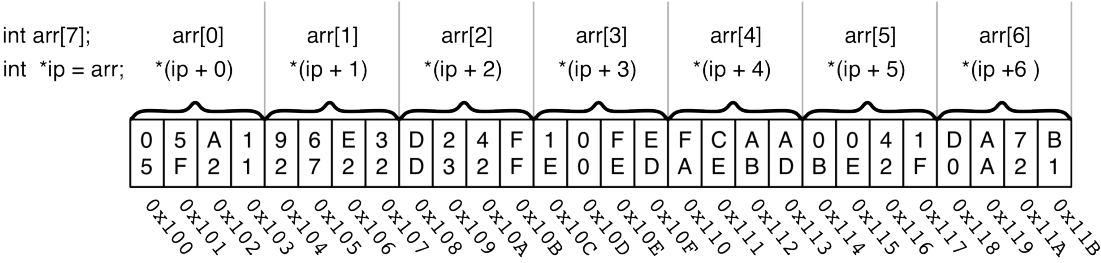
This is how an array looks in memory:

Figure 2.7 Array in Memory



Look familiar? Here is another way to look at it:

Figure 2.8 Array as a Pointer



Arrays have a very close relationship with pointers. In general, you can index a pointer with square brackets (just like an array), and you can do pointer math on an array. In fact, pointers are how arrays get passed to functions.

Consider this program, where arrays get passed to the `printArray()` function as a pointer, and indexed as an array:

Example 2.7. array-parameter.c

```
// array-parameter.c

/* compile with:
cc -g -Wall -o array-parameter array-parameter.c
*/

#include <stdio.h>      // for printf()

void printArray (int *array, int length)
{
    int i;

    for (i = 0; i < length; i++) {
        printf ("%d: %d\n", i, array[i]);
    }
} // printArray

int main (int argc, char *argv[])
{
    int array1[5];
    int array2[] = { 23, 42, 55 };
    int i;

    for (i = 0; i < 5; i++) {
        array1[i] = i;
    }

    printf ("array 1:\n");
    printArray (array1, 5);

    printf ("array 2:\n");
    printArray (array2, sizeof(array2) / sizeof(int));
}
```

```
    return (0);  
  
} // main
```

A sample run:

```
$ ./array-parameter  
array 1:  
0: 0  
1: 1  
2: 2  
3: 3  
4: 4  
array 2:  
0: 23  
1: 42  
2: 55
```

One thing of interest is the bulk initialization of `array2`. Since there are three elements in that initialization list, the length of `array2` is that of three ints, or 12 bytes ($3 * 4 = 12$). Look at the technique used to calculate the number of elements in the array by dividing the total size (via `sizeof`) by the `sizeof` a single element. This comes in handy when you have tables of stuff defined in your code.

Another thing to note is that the `printArray()` function needs to take a length (or you need to stick some sentinel value) so that the loop knows when to stop. There is no length information encoded in the pointer or array. That is why there exist functions like `strlen()` that have to count characters in a string.

A logical question is, “If that is the case, how come you could do that `sizeof(array2) / sizeof(int)` thing?” Recall that `sizeof` is a compile-time directive. The compiler knows that `array2` has three members because of the initialization list. The function `printArray()` cannot know that since it does not know at compile time which array it is going to be given.

Arrays of pointers

If you've been following along, you've probably noticed a couple dozen of these:

```
int main (int argc, char *argv[])
```

Look at the last argument: what is that?

Here it is taken apart:

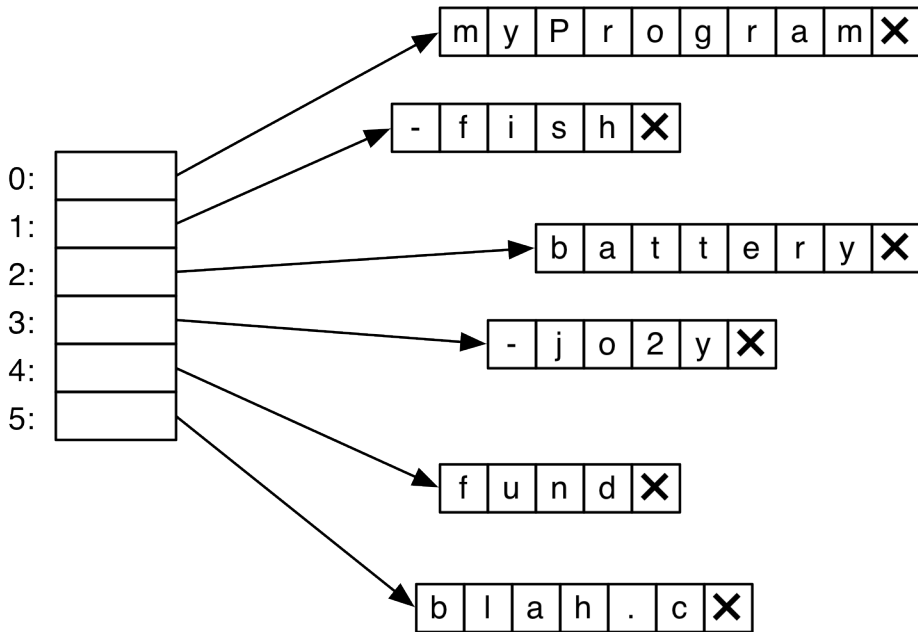
<code>argv[]:</code>	<code>argv</code> is an array of undetermined length
<code>char *:</code>	of character pointers.

In other words, `argv[]` is an array of postcards, and on the postcards are the locations of C strings (zero-terminated sequences of characters). This array (`argv`, for “argument vector”) is the set of program arguments given to the program. `argc` (argument count) is the number of elements in `argv`.

For a program started like

```
$ myProgram -fish battery -jo2y fund blah.c
```

the resulting `argv` array looks like this:

Figure 2.9 `argv`

Because of the equivalence of arrays and pointers (they are not identical, but the differences between them are subtle), the declaration for `main()` can also be written as

```
int main (int argc, char **argv)
```

For the More Curious: Data Structures Using Pointers

structs and pointer to structs can be used to build powerful data structures. One of the simpler ones is a binary search tree. Each node of the tree holds a value and a pair of pointers to other nodes. New values less than the value contained in the node get put into the left fork, otherwise they go onto the right fork. If someone is already living on a particular side of the fork, you look at that node and try to place the new value accordingly.

Here is a program that puts some values into a tree, then looks to see if particular values can be found.

Example 2.8. `tree.c`

```
// tree.c -- use structs and pointers to build a tree of nodes

/* compile with:
cc -g -Wall -o tree tree.c
*/

#include <stdio.h>           // for printf()
```

Chapter 2 A C Refresher, Part 2: Pointers, Structures, and Arrays

```
#include <stdlib.h>          // for malloc()

typedef struct TreeNode {
    int value;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

void addValue (TreeNode *node, int value)
{
    if (value < node->value) {
        // left side
        if (node->left == NULL) {
            TreeNode *newNode = malloc (sizeof(TreeNode));
            newNode->value = value;
            newNode->left = newNode->right = NULL;
            node->left = newNode;
        } else {
            addValue (node->left, value);
        }
    } else {
        // right side
        if (node->right == NULL) {
            TreeNode *newNode = malloc (sizeof(TreeNode));
            newNode->value = value;
            newNode->left = newNode->right = NULL;
            node->right = newNode;
        } else {
            addValue (node->right, value);
        }
    }
} // addValue

#define TRUE 1
#define FALSE 0

int findValue (TreeNode *node, int value)
{
    if (node == NULL) {
        return (FALSE);
    } else if (node->value == value) {
        return (TRUE);
    } else {
        if (value < node->value) {
            return (findValue(node->left, value));
        } else {
            return (findValue(node->right, value));
        }
    }
} // findValue

int main (int argc, char *argv[])
{
    TreeNode root;

    // put 23 in manually to bootstrap the tree
    root.value = 23;
    root.left = root.right = NULL;
```



```
// now add some stuff

addValue (&root, 5);
addValue (&root, 50);
addValue (&root, 8);
addValue (&root, 2);
addValue (&root, 34);

if (findValue(&root, 23)) {
    printf ("23 lives in the tree\n");
} else {
    printf ("23 does not live in the tree\n");
}

if (findValue(&root, 42)) {
    printf ("42 lives in the tree\n");
} else {
    printf ("42 does not live in the tree\n");
}

return (0);

} // main
```

A sample run:

```
$ ./tree
23 lives in the tree
42 does not live in the tree
```

After the tree is built, it would look something like this in memory:

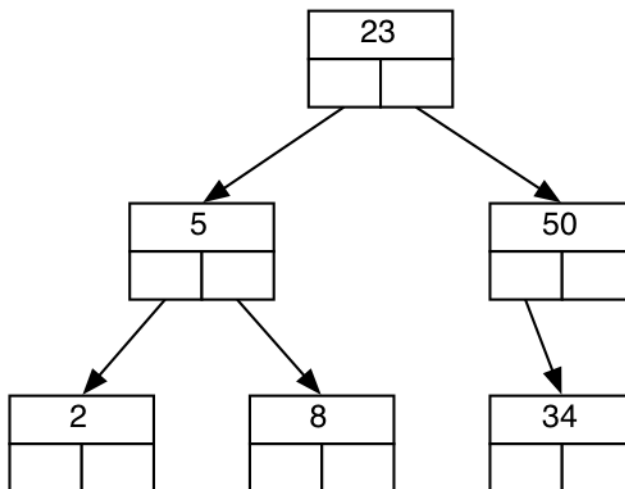


Figure 2.10 Binary Search Tree

The most interesting part of the code is the declaration of the tree node:

```
typedef struct TreeNode {
```

```
int value;  
struct TreeNode *left;  
struct TreeNode *right;  
} TreeNode;
```

You cannot reference the `TreeNode` typedef while you are declaring it, but you can reference `struct TreeNode`.

The other interesting thing is the use of dynamic memory allocation:

```
TreeNode *newNode = malloc (sizeof(TreeNode));
```

This allocates a chunk of memory the size of a `TreeNode` (12 bytes) and gives you a pointer to it. This allows the tree to grow arbitrarily large.

Challenge:

Add a function to `tree.c` to print out the tree.