An Introduction to C Programming for Java Programmers

Mark Handley

M.Handley@cs.ucl.ac.uk

CONTENTS

1 INTRODUCTION

This short tutorial is an introduction to programming in C. It is intended for students who already have some programming experience in Java, so know the basics of how to program and are familiar with the basic Java syntax which is shared with C. This is not a reference manual, nor a complete guide to the C language.

To learn more about C than is covered here, I recommend "The C Programming Language" by Kernighan and Ritchie (http://cm.bell-labs.com/cm/cs/cbook/).

In this tutorial I discuss many common mistakes made by C programmers, with particular emphasis on the sort of issues of which a Java programmer might be unaware. The intent is not to scare the reader away from C, but to highlight specific differences with Java, and avoid making the most common errors.

C is a fairly low-level language. Features and limitations of the underlying hardware are often exposed in C, whereas in Java they are largely hidden. The result is that good C programs will usually run quickly, with a small memory footprint. For operating system programming, C allows direct access to the hardware, which might simply not be possible in Java.

The downside of C compared to Java is that C does not protect the programmer from his or her own errors. It is not a strongly typed language, so the compiler provides little warning of many errors that would simply fail to compile in Java. This makes debugging harder. Memory management in C is manual, and manipulating pointers to memory is essential in any non-trivial C program. Getting this right is difficult.

But comparing C against Java on a good/bad scale is not productive; each serves a useful role for a certain set of tasks. For example, these days I would not recommend writing simple non-performance-critical network servers in C (although many are), simply because it is too hard to write really secure code. But writing a general-purpose operating system entirely in Java might not even be possible, and would certainly be rather inefficient on today's common processors. A competent programmer should be aware of the advantages and limitations of a range of languages, and use the right tool for the job.

2 Basics of C

We will start by walking through the basic features of the C language, looking at the control structures it provides, the basic data types and how to build more complex data structures, and how to read data into and out of your programs. But before even that, we need to look at the different files that make up a C program.

2.1 C Source Files

Sometimes a C program can be included in a single source file, but this is infeasible with non-trivial sized programs. There are three types of file that you need to be concerned with:

- C program files. The convention is that these have a `.c` extension such as `foo.c`. C program files contain the actual executable code of your program.
- C header files. The convention is that these have a `.h` extension such as `foo.h`. C header files contain interface definitions, preprocessor macros, and so on. When you need to call a function that is defined in one C program file from a function that is defined in another C program file, you will need to create a header file with a *prototype* for the external function, so that the compiler knows how to compile your function.
- library files. There are several different types of library file, depending on how they are linked to your program, but they all contain pre-compiled functions that you can use in your program.

In addition, it is common to automate the process of figuring out which source files need to be compiled together, using the utility `make`. Thus you'll often see a file called `Makefile` in a directory containing C source files. The `Makefile` contains information about which programs should be built, and how to compile and link them.

We'll discuss separate compilation and the use of libraries later.

2.2 main()

Every C program must contain a procedure called `main()`. This is the first procedure in your program to be executed. The simple *hello world* program in C then looks like:

```
#include <stdio.h>

main()
{
    printf("hello world!\n");
}
```

To compile this using gcc on a Unix or MacOS X system, if the source file is called `helloworld.c`, you'd type (in bold):

```
eve.cs.ucl.ac.uk: gcc -o helloworld helloworld.c
eve.cs.ucl.ac.uk:
```

The program `gcc` is the GNU C compiler, which is the most common C compiler found on most Unix and Linux systems. On some systems the C compiler might be called `cc`.

The "`-o helloworld`" argument indicates that the compiled program should be called "`helloworld`". If no filename is given for the compiled program, the default on Unix is usually `a.out`. The last argument "`helloworld.c`" tells the compiler which source file or files to compile.

On Unix, see the gcc manual page for details of other command line options to gcc (run `man gcc`). A full gcc book can be found online at `http://www.network-theory.co.uk/gcc/intro/`.

Then to run your program, you'd type:

```
eve.cs.ucl.ac.uk: ./helloworld
hello world!
eve.cs.ucl.ac.uk:
```

In addition to showing how to compile code, this trivial example illustrates several things:

- The use of the preprocessor macro `#include` to include the `stdio` header file. Without this, the compiler wouldn't know how to compile the `printf` command which is in the standard C library.
- The use of `main()` as the first procedure to be executed in any C program.
- The use of `printf` to print to the standard output channel (in this case your terminal window). On Unix, see the printf manual page for more details (run `man printf`)
- The use of `\n` to indicate a newline in a string.

We'll see examples of separate compilation and linking of multiple program files in Sections 2.13 and 3.2.

2.3 Built-in Commands

C supports the usual commands to control execution flow that you are used to from Java:

if statements

```
if ( pi == 3 ) {
    location = "indiana";
}
```

The braces ({ and }) are only needed if more than one statement is conditional. In the example below, the `printf` statement will be executed irrespective of the value of `pi`.

```
location = "unknown";
if (pi == 3)
    location = "indiana";
printf("Location = %s\n", location);
```

if/else statements

```
if (pi == 3) {
    location = "indiana";
} else {
    location = "unknown";
}
```

C also supports a shorthand if/else notation. The single statement below has exactly the same effect as the if/else example above.

```
location = (pi == 3 ? "indiana" : "unknown");
```

This notation is very useful in some circumstances such as preprocessor macro definitions, but it should be used sparingly as it is not always as readable as a regular if/else construction.

while statements

```
while (money > 0) {
    money--;
    drink_another_beer();
}
```

do/while statements

```
do {
    sleep++;
} while (sleep < 8)
```

A `do/while` loop is similar to a `while` loop, except that the contents of the loop are always executed at least once, even if the condition is false. The regular `while` loop turns out to be more commonly used in practice.

for loops

```
int i;
for (i = 1; i <= 10; i++) {
    printf("%d squared is %d\n", i, i*i);
}
```

The `for` command consists of three control statements and a body that may get executed multiple times. The three control statements in the above example are:

- `i = 1`. The first control statement is executed only once before the loop is entered for the first time.
- `i <= 10`. The second control statement is the test for the loop. It is executed before the loop is entered each time, and if the value is true, the body of the loop is executed. If the test is false on the first try, the body of the loop will never be executed.
- `i++`. The third control statement is executed after each pass through the loop body, before the test is run. The notation `i++` increments the value of `i` by one.

Thus in the above example, the order of execution is:

1. `i = 1`
2. `i <= 10`. Value is true, so loop body is executed.
3. `printf()`.
4. `i++`. Value of `i` is now 2.
5. `i <= 10`. Value is true, so loop body is executed.
6. `printf()`.
7. `i++`. Value of `i` is now 3.

   ...

   ...
8. `printf()`.
9. `i++`. Value of `i` is now 11.
10. `i <= 10`. Value is false, so execution continues with the next statement after the loop.

Note the use of `%d` to indicate that an integer argument is to be printed as a decimal value.

switch statements

```
char c;
c = getchar();
switch (c) {
    case 'c':
        cont = 1;
        break;
    case 'd':
        done = 1;
        break;
    case 'Q':
    case 'q':
        exit();
    default:
        printf("valid inputs and 'q', 'c' and 'd'\n");
}
```

The `case` keyword is used to indicate each of the possible values that the variable in the `switch` statement might take.

The `default` keyword is used to indicate a catch-all case that matches everything not matched by a prior `case` statement.

The `break` keyword is used to leave the `switch` statement. If this is omitted, execution drops through to the body of the `case` statement below. This is shown in the `case 'Q'` line, where the inputs '`q`' and '`Q`' both cause the program to exit.

goto statements

```
char c;
int characters = 0;
int lines = 0;
while (1) {
    c = getchar();
    if ( c == EOF ) {
        goto Finished;
    }
    characters++;
    if ( c == '\n' ) {
        lines++;
    }
}

Finished:
printf( "lines: %d, characters: %d\n", lines, characters);
```

This simple and rather inefficient program counts the number of lines and characters in an input file. At the end of the file, it uses a `goto` statement to jump out of the while loop, continuing execution after the label `Finished`. In this particular case, a `break` statement could have been used instead, but if there are many nested loops, `break` cannot trivially be used.

`goto` should be used sparingly. It's best used for handling exceptions and error cases. Unlike java, C has no built-in exception handling; goto can sometimes be used as an effective alternative. Too much use of goto can make your code unreadable.

2.4 Variables and Basic Types

The basic types available to you in C are:

- characters
- integer numbers
- floating point numbers

Various modifiers are available to change the size of these, and to specify whether they can hold signed or unsigned values. Unlike with Java, the precise amount of storage (and hence range of values available) for a variable depends on the particular system the program is running on.

On an Intel 32 bit x86 machine, the basic variable types are:

- `char` - 8 bit signed integer, also used to store characters for strings. Numeric value range: -128 to 127
- `unsigned char` - 8 bit unsigned integer. Value range: 0 to 255
- `short` - 16 bit signed integer. Value range: -32768 to 32767
- `unsigned short` - 16 bit signed integer. Value range: 0 to 65535
- `int` - 32 bit signed integer. Value range: -2,147,483,648 to +2,147,483,647
- `long` - same as `int`.
  On some other systems, `int` might be 16 bit and `long` might be 32 bit.
- `unsigned int` - 32 bit unsigned integer. Value range: 0 to 4,294,967,295
- `unsigned long` - same as `unsigned int`.
  On some other systems, `unsigned int` might be 16 bit and `unsigned long` might be 32 bit.
- `long long` - 64 bit signed integer. Value range: -2^{63} to $2^{63} - 1$.
- `float` - 32 bit floating point number.
- `double` - 64 bit floating point number.
- `long double` - 96 bit floating point number.

You might be surprised to learn that a `char` is an 8-bit signed integer. The reality is that if we look at the actual CPU's Arithmetic and Logic Unit (ALU), there are really only two basic types: floating point numbers and integers. C is not strongly typed, so assigning an integer to a `char` is perfectly OK, so long as the value fits in the appropriate data range for a `char`. Thus the following is perfectly legal C:

```
char c;
int i;
i = 67;
c = i;
printf("The ASCII character code for '%c' is %d\n", c, c);
```

And the output from this code fragment is:

```
The ASCII character code for 'C' is 67
```

Note that the `printf` statement prints the value of `c` twice, once as a character (`%c`) and once as an integer (`%d`).

The `sizeof()` command can be used to find out how many bytes a particular type or variable uses on the local system. This can be useful when trying to write code that is portable between 16-bit, 32-bit and 64-bit systems. For example:

```
short s;
int i;
long l;
long long ll;
float f;
double d;
long double ld;
printf("short: %d, int: %d, long: %d long long: %d\n",
      sizeof(s), sizeof(i), sizeof(l), sizeof(ll));
printf("float: %d, double: %d long double: %d\n",
      sizeof(f), sizeof(d), sizeof(ld));
```

On a 32-bit x86 machine, this prints out:

```
short: 2, int: 4, long: 4 long long: 8
float: 4, double: 8 long double: 12
```

On a 64-bit Solaris/Ultrasparc machine, this prints out:

```
short: 2, int: 4, long: 4 long long: 8
float: 4, double: 8 long double: 16
```

The only difference here is in the size of a `long double`. However, on embedded processors or some other 64 bit machines, the some of the other values may vary.

You really get a sense of the low-level nature of C here. Java provides a nice OS-independent set of types. C gives you what the raw hardware provides, and as a result is usually significantly faster. The downside is that it's harder to write portable code.

In general, you want to be careful to use variables that have enough space for the range of values you want to hold, and you want to check untrusted input to be sure it's in the right range before you store it. **Overflows will not be automatically detected by the compiler or at runtime!**

For floating point numbers, you almost always want to use `double`. The type `float` usually doesn't have enough precision. Also be careful about exact comparisons with floating point numbers - hardware rounding errors can often result in very slight variations from the value you're comparing against.

2.5 Declaring Variables

Variables must be declared at the start of a block, and they remain in scope until the end of the block. A block can be a procedure, if statement, while loop, or practically anything enclosed by { and }. For example:

```
#include <stdio.h>
main()
{
    int i = 1, j = 1;
    while (i == 1) {
        int j, k;
        j = 2;
        printf("In here, i is %d, j is %d\n", i, j);
        i--;
    }
    printf("Out here, i is %d, j is %d\n", i, j);
}
```

In this example `i` and `j` are two integer variables declared at the start of `main`.

Two more variables (`j` and `k`) are then declared inside the `while` loop.

But `j` was already declared outside the while loop. What happened here is that a new variable is created, overriding (or "shadowing") the first version of `j` for the duration of the while loop, and then disappearing after termination of the loop.

When you run this program you get:

```
eve.cs.ucl.ac.uk: gcc -o foo foo.c
eve.cs.ucl.ac.uk: ./foo
In here, i is 1, j is 2
Out here, i is 0, j is 1
```

Thus the assignment `j = 2` does not change the value of the original `j`, but instead sets the value of the new version of `j` in the while loop.

Shadowing of this form is usually a mistake - it's too confusing to do intentionally on a regular basis. You can get the compiler to help you spot such errors:

```
eve.cs.ucl.ac.uk: gcc -Wshadow -o foo foo.c
foo.c: In function `main':
foo.c:6: warning: declaration of `j' shadows previous local
```

In general, getting the compiler to tell you about possible errors is a good thing. Often it will spot problems you might have missed. See the gcc man page for lots of other `-W` flags you can set.

2.6 Operators

C provides all the usual operators you would expect.

2.6.1 Arithmetic Operators

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulus (remainder). For example the value `10 % 3` is `1`.

2.6.2 Bitwise Operators

`~`	one's complement
`|`	bitwise or
`&`	bitwise and
`^`	bitwise exclusive or (xor)
`>>`*n*	shift right n bits
`<<`*n*	shift left n bits

Bitwise operators perform a logical operation on all the bits in a value.

For example, the value of `(1 | 4)` is `5`, and the value of `(3 & 2)` is `2`.

2.6.3 Relational and Logical Operators

`==`	equal to
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`!=`	not equal to
`!`	logical not
`||`	logical or
`&&`	logical and

In C, there is no *boolean* type - the *int* type is simply overloaded. Where a boolean type is expected, an int with a value of zero is treated as *false*, and an int with a non-zero value is treated as *true*.

For example, the value of `(0 || 4)` is `1` (true), and the value of `(3 && 2)` is also `1` (true).

Confusing bitwise and logical operators is a common error.

2.6.4 Assignment

C assigns variables in the usual way:

`a = 2;` sets the value of a to be 2.

C also supports many shorthand operator/assignments to change the value of a variable:

`a += 2`	same as `a = a + 2`
`a -= 2`	same as `a = a - 2`
`a *= 2`	same as `a = a * 2`
`a /= 2`	same as `a = a / 2`
`a %= 2`	same as `a = a % 2`
`a &= b`	same as `a = a & b`
`a |= b`	same as `a = a | b`
`a ^= b`	same as `a = a ^ b`
`a <<= 2`	same as `a = a << 2`
`a >>= 2`	same as `a = a >> 2`

A common error is to use = instead of == in a comparison.

For example:

WRONG:
```
    if (a = 2) {
        /*do something*/
    }
```

In the above example, `a` will be *assigned* the value of 2. The assignment itself returns the value of 2, which is interpreted as *true*, and the conditional branch will always be executed. **The compiler will not notice this error, because it's valid C**, but it's rarely what you intended.

CORRECT:
```
    if (a == 2) {
        /*do something*/
    }
```

In the version above, the correct comparison operator has been used.

CORRECT AND SAFER:
```
    if (2 == a) {
        /*do something*/
    }
```

In this version, the correct comparison operator has been used, but also the terms in the comparison have been reversed. This is good practice, because if you accidentally type = instead of ==, the compiler will report an error because you can't assign a value to the constant `2`.

2.6.5 Increment and Decrement

`++x`	increment x, then use new value
`x++`	use old value of x, then increment x
`--x`	decrement x, then use new value
`x--`	use old value of x, then decrement x

For example:

```
int x, y;

x = 1;
y = ++x;
/* value of x is 2, value of y is 2 */

x = 1;
y = x++;
/* value of x is 2, value of y is 1 */

x = 1;
y = x++ * 10;
/* value of x is 2, value of y is 10 */
```

2.7 Overflows, Assignments, and Other Trouble

C allows you to assign pretty much anything to pretty much any type. It expects you know what you're doing. This can allow you to write very efficient and fast code, but it will also get you into trouble unless you're very careful.

Some examples of unexpected consequences:

```
unsigned char c;
c = 255;
c += 2;
```

Adding 2 to the unsigned char `c`, which holds a value of 255 as shown above, results in the value of `c` wrapping. The end result is that `c` has a value of 1.

```
int i;
short s;
i = 32769;
s = i;
```

The least significant 16 bits of the 32-bit `int i` are copied into the 16-bit `short s`. Unfortunately as a 16-bit signed number, 32769 becomes -32767 which is then the new value of `s`.

```
int i;
float f;
i = 1234567890;
f = i;
```

The integer i is copied to the float f. The compiler does the right thing, and adds code to convert from integer representation to floating point representation, but this value is too large to be held in a 32-bit float without losing precision. `f` ends up holding the value `1234567936.000000`. Using a `double` here would have avoided this particular rounding error.

```
int i = 1;
int j = 0;
int k;
k = i/j;
```

This is an arithmetic error. In Java, this would throw an exception. In C, it causes the process to crash, and to dump a core file:

```
eve.cs.ucl.ac.uk: gcc -g -o foo foo.c
eve.cs.ucl.ac.uk: ./foo
Floating exception (core dumped)
eve.cs.ucl.ac.uk:
```

The CPU raised a divide-by-zero exception, which the operating system trapped, and aborted execution of your program. There's no indication as to what exactly you did wrong, or where in your program the error is. But be grateful - if the OS hadn't handled this, the system would have crashed - it did the best it could, and preserved the evidence. Your best hope now is to use a debugger such as *gdb* to examine that core file. Note the use of the `-g` flag above - without this the executable will not include debugging symbols that gdb needs.

2.8 Arrays

An array in C is declared and accessed as follows:

```
int a[10];
int i;
for (i = 0; i < 10; i++) {
    a[i] = 0;
}
```

The statement `int a[10];` declares an array of ten integers, and allocates memory on the stack for this array. As it's on the stack, the memory will be automatically freed when `a` goes out of scope.

The `for` loop here simply sets the values of the elements of `a` to be zero. You cannot expect C to zero memory for you. Note that the first element of `a` is `a[0]` and the last element is `a[9]`.

C does not perform any array bounds checking - this makes array use in C very fast, but it also makes it a little dangerous. A common error is to index beyond the end of an array. For example:

```
int x=0;
int a[10];
int i;
for (i = 0; i <= 10; i++) {
    a[i]=27;
}
printf("The value of x is %d\n", x);
```

This code fragment produces the following output:

```
aardvark.cs.ucl.ac.uk: ./foo
The value of x is 27
```

But the value of `x` was set to 0, so what happened here? The `for` loop set the values of `a[0]` to `a[10]` to all be 27. But the declaration of `a` only allocated enough memory for `a[0]` to `a[9]`. When we wrote the value of 27 into `a[10]`, this overwrote the next four bytes of memory. As the memory on the stack on this system is allocated top-down, the next four bytes of memory happen to correspond to the memory used by variable `x`, so the value of x is overwritten.

The moral here is to be very careful with array subscripting - the compiler won't help you out. Out-by-one errors are notoriously common, and in C then can have subtle adverse side effects without necessarily causing the program to immediately crash.

2.9 Pointers and Addresses

In Java you have no direct access to memory - all accesses to memory have to go through the API for one variable or another, and all accesses are checked. C gives you direct access to memory, as you can see from the array example above.

C also gives you access to *memory addresses*. For example, if you want to find out the address in memory where a variable is stored, you can do the following:

```
int x=0;
printf("The address of x is %x\n", &x);
```

The notation `&x` means "the address of x". This code fragment gives the following output (the address is printed in hex):

```
The address of x is bfbff3ec
```

In C, an address is itself a data type, known as a *pointer*. Pointers can be assigned and copied just like any other variable, and they can be *dereferenced* so that the memory being pointed to can be read or written. For example:

```
int x=0;

int *p;
p = &x;

printf("The address of x is %x\n", p);
printf("The value of x is %d\n\n", *p);

*p = 27;
printf("The value of x is %d\n", x);
```

In the example, a regular variable `x` is declared. A pointer variable is also declared: `int *p;`
In a declaration, the "`*`" used like this declares a pointer to an type, rather than the type itself, so this declares that the variable called `p` is a variable that holds a pointer to an integer.

The line `p = &x;` takes the address of `x` and stores this in pointer `p`.

The code prints out the value of the pointer `p` (ie the address of `x`), and then it prints out the value of the memory pointed to by `p`. The notation `*p`, used like this outside of a declaration, means to dereference the pointer, and hence in this context it gives the value of `x` because `p` points at `x`.

Then we have the line: `*p = 27;`
This dereferences `p` and sets the value of the memory pointed to by `p` to be 27. So in this case it sets the value of `x` to be 27 because `p` points at `x`.

Thus the output of the code fragment above is:

```
eve.cs.ucl.ac.uk: ./foo
The address of x is bfbff3ec
The value of x is 0

The value of x is 27
```

Pointers are fundamental to the use of C for any non-trivial program. They're used to pass around references to data, they're used for all dyanmic memory allocation, and they're a fundamental part of handling text strings in C.

2.10 Dynamic Memory Allocation

So far, we have seen variables and arrays allocated on the *stack* - that is the storage for the variable will be allocated when the variable is declared, and will be removed when the program leaves the block where the variable is in scope. Often we need to store data in one part of a program, leave that part of the program, and without copying it, pass the stored data to another part of the program. To do this in C we need to dynamically allocate memory on the *heap*.

The main way to dynamically allocate memory in C is to use the `malloc()` function, or one of a family of closely related functions. For example:

```
#include <stdlib.h>

main()
{
    int *buffer;
    int i;

    buffer = malloc( 10 * sizeof(int) );

    for (i = 0; i < 10; i++) {
        buffer[i] = i;
    }

    for (i = 0; i < 10; i++) {
        printf("p[%d] is %d\n", i, buffer[i]);
    }

    free(buffer);
}
```

Running this program gives:

```
aardvark.cs.ucl.ac.uk: ./foo p[0] is 0
p[1] is 1
p[2] is 2
p[3] is 3
p[4] is 4
p[5] is 5
p[6] is 6
p[7] is 7
p[8] is 8
p[9] is 9
```

In the example above, we declared a pointer called `buffer`, and then used `malloc()` to allocate enough memory to store ten integers.

After this we then accessed this memory just as if it were an array. This might seem surprising at first, but in C, an array is essentially just a block of memory and a pointer to the base of that memory. Thus array indexing operations can be used to access any block of memory, even one created using malloc.

However, if the memory is allocated using `malloc`, it will not automatically be freed when the pointer to it is removed. Thus when you have finished using the memory, you need to explicitly free it using a call to `free()`, as shown above.

Failure to free memory will result in a memory leak - your program will get bigger and bigger until it crashes because it can't get any more memory.

Equally bad, but more subtle, is when you do free the memory, but leave behind a pointer to that memory, and later access the memory using that pointer. It is likely that before long the memory you freed will be reallocated to your program when you call `malloc` again. Your old pointer will now point at memory being used elsewhere, and accessing this memory will have seemingly random results.

In general, dynamically allocating memory, freeing it correctly, and making sure it is never referenced again after being freed, are among the hardest things to get consistently right in a C program. But dynamic memory allocation is also essential in any non-trivial program, so great care must be taken to make it very clear who is responsible for freeing memory.

2.11 Text Strings

Most programming languages have support for text strings as first class types. However, most CPUs have no special support for strings, so all this entails quite a bit of work by the compiler. C generally closely reflects the hardware capabilities, and so C doesn't have much in the way special support for text strings in the base language. Thus strings in C are implemented using arrays and pointers, as shown below:

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

main()
{
    char *s1;
    char s2[80];
    char *s3;

    s1 = "Hello";
    s3 = malloc(80);

    printf("Enter your first name:\n");
    fgets(s2, 80, stdin);

    printf("Enter your last name:\n");
    fgets(s3, 80, stdin);

    printf("%s %s %s\n", s1, s2, s3);
    free(s3);
}
```

Some sample output is shown below:

```
eve.cs.ucl.ac.uk: ./foo
Enter your first name:
Donald
Enter your last name:
Duck
Hello Donald Duck
```

In this example, three variables are declared, and all three can be regarded as being strings.

First, `s1` is declared to be of type "`char *`". Then `s1` is set to point at the literal string `"Hello"`. This literal string is part of the program code; it can't be changed. The assignment simply sets `s1` to point to this constant string. You can read from this memory, hence print out the string, but not write to it.

Second, `s2` is declared to be an array of 80 chars.

Third, `s3` is declared to be of type "`char *`", but then we `malloc()` 80 bytes of memory, and set `s3` to point to it.

The only practical difference between `s2` and `s3` is than `s2`'s memory is on the stack, whereas `s3`'s memory is on the heap. If we wanted to, we could subsequently re-assign pointer `s3` to point to some other memory, but we can't do this with `s2` because of the way it's declared.

In any event, at this stage both `s2` and `s3` are effectively strings. We can read text into them - in this case we read it in using a standard library call `fgets()` which reads from standard input (usually the keyboard). And we can print out the values stored, in this case using the `%s` formating code to `printf`.

In C, strings are *null-terminated*. For example, `s2` above has 80 bytes of storage allocated, but we only read in 6 characters ("`Donald`"). A 7th character known as a NULL (it has character code 0) is then appended to the end of the string. When `printf` prints the string, it prints each character until it reaches the NULL, and then it stops. All the string handling functions in C expect a string to be null-terminated in ths way.

It is important to understand that a variable of type `char *` is only ever a pointer from the compiler's point of view. It's easy to forget this. For example, if you want to compare two strings, you might incorrectly type:

WRONG:
```
    char s1[80];
    char s2[80];

    printf("Enter first word:\n");
    fgets(s1, 80, stdin);

    printf("Enter second word:\n");
    fgets(s2, 80, stdin);

    if (s1 == s2) {
        printf("Words are the same\n");
    } else {
        printf("Words are different\n");
    }
```

No matter what you enter, this will always say that the strings are different. This is because `s1` and `s2` are really pointers, and in this case they point to different memory. The comparison `if (s1 == s2)` merely checks if the pointers point to the same memory - it does not compare the contents of that memory.

To actually compare the contents of strings, you need to use the `strcmp()` function from the standard library:

CORRECT:
```
#include <string.h>

...

    char s1[80];
    char s2[80];

    printf("Enter first word:\n");
    fgets(s1, 80, stdin);

    printf("Enter second word:\n");
    fgets(s2, 80, stdin);

    if ( strcmp(s1,s2) == 0 ) {
        printf("Words are the same\n");
    } else {
        printf("Words are different\n");
    }
```

To use string functions, you need to include the `string.h` system header file, as shown above. We'll discuss the C preprocessor more in Section 3.

Other useful string functions include:

- `strcmp()` - compare two strings.
- `strncmp()` - compare the first n characters of two strings.
- `strdup()` - save a copy of a string.
- `strncpy()` - copy a string.
- `strncat()` - concatenate strings.
- `snprintf()` - formatted print into a string.

More details of these functions can be found in the man pages.

You may also discover that variants of these functions such as `strcpy()` (as opposed to `strncpy()`) exist. The difference is that you specify the amount of memory available with `strncpy()` whereas you don't with `strcpy()`. Good code **always** uses `strncpy()`, `strncat()`, and `snprintf()` rather than `strcpy()`, `strcat()` and `sprintf()`. The versions that don't require a size to be specified are just too easy to accidentally overflow the available memory. In networked programs, such simple buffer overflows frequently lead to serious security problems.

2.12 Data Structures

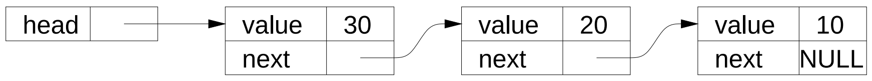
It is a very common requirement to need to group a set of information together and handle it as one entity. In Java, this is done using classes, but C is not object-oriented, and so has no concept of a class. What C does have is `struct`, which has the same ability as a class to group data, but without the accompanying class access methods.

For example, suppose you want to read in a series of integers from the keyboard, and store them in a linked list, and print them out in reverse order. Each element in the list needs to hold an integer and a pointer to the next element. You can use a `struct` for each list element, as shown in the program on the next page.

The output from running this program looks like:

```
aardvark.cs.ucl.ac.uk: ./foo
Enter an integer: 10
Enter an integer: 20
Enter an integer: 30
Enter an integer: Finished
Value: 30
Value: 20
Value: 10
```

In this program we declared a `struct` called `element`, which contains two fields: an integer called `value` and a pointer to another `element` called `next`. We then use this struct to build our linked list like this:



The variable `head` is used to keep track of the start of the list.

Each time through the while loop, we read a line of text from standard input, and if the end of file has not been reached, then we convert this text into an integer using `atoi()`. We create a new list element to store the integer in, using `malloc()` to allocate the memory.

Next we set then `value` field of the new element to be the integer we just read in. This is done in the line:

```
    new_element->value = number;
```

The notation `new_element->value` means the field called `value` pointed to by the pointer called `new_element`.

We then need to link the element into the existing list. As we want to print out the numbers in reverse order, we'll put the new element at the start of the list. To do this, we set the `next` field of the new element to point to what used to be the first element of the list. Then we change the `head` pointer to point to the new element we just added.

Finally, when there's no more input, we loop through the list printing out the values, until we find an element whose next pointer is `NULL`. The value `NULL` is used to denote a pointer that doesn't point to anything; it has the numeric value of zero.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXLEN 80

/* struct to hold an element of the linked list */
struct element {
    int value;
    struct element *next;
};

main()
{
    /* variable to hold the head of the list*/
    struct element *head = NULL;

    /* temporary variable for printing out the list */
    struct element *p;

    while (1) {
        int number;
        char buffer[MAXLEN];
        struct element *new_element;

        /* read in an integer from the keyboard */
        printf("Enter an integer: ");
        fgets(buffer, MAXLEN, stdin);
        if ( feof(stdin) ) {
            printf("Finished \n");
            break;
        }
        number = atoi(buffer);

        /* create a new list element */
        new_element = malloc( sizeof(struct element) );
        new_element->value = number;

        /* add element to the list */
        new_element->next = head;
        head = new_element;
    }

    /* print out the numbers we stored */
    p = head;
    while ( p != NULL ) {
        printf("Value: %d\n", p->value);
        p = p->next;
    }
}
```

Declaring Structs on the Stack

In the linked list example, we allocated heap memory for a `struct` using malloc, and accessed its fields using "`->`" to follow a pointer. Structs can also be allocated on the stack, as you might with a regular variable. For example, in the system header file `sys/time.h`, a struct called `timeval` is defined:

```
struct timeval {
    long tv_sec;     /* seconds */
    long tv_usec;    /* and microseconds */
};
```

To find out the current time, we might use this as follows:

```
#include <stdio.h>
#include <sys/time.h>

main()
{
    /* a struct timeval to hold the current time. */
    /* timeval is defined in sys/time.h */
    struct timeval time;

    double fractionaltime;

    /* get the current time and store it in "time" */
    gettimeofday(&time, NULL);

    /* convert the time to a fraction */
    fractionaltime = time.tv_sec + (time.tv_usec/1000000.0);

    printf("%f seconds since 1st January 1970\n", fractionaltime);
}
```

Which might produce output such as:

```
eve.cs.ucl.ac.uk: ./foo
1096801178.236713 seconds since 1st January 1970
```

In this example we declared the variable `time` to be a `struct timeval` just like we might declare any other local variable on the stack.

We then called the standard system call `gettimeofday()` to get the current time, passing in a pointer to `time` (we get a pointer by using "`&time`"). The call to `gettimeofday()` reads the system clock to find out the current time, and fills in the fields of the struct `time` using the pointer we gave it.

In this example we want to print out the time as a decimal number, so we convert it from the `struct timeval` representation by reading the fields of `time` individually. The notation `time.tv_sec` means the `tv_sec` field of the struct called `time`. Finally we print out the value.

The notation from this example of `time.tv_sec` is functionally similar to the `new_element->value` notation in the previous example. The difference in notation is because `new_element` was a pointer to a struct, whereas `time` actually *is* a struct.

2.13 Functions and Procedures

C is a procedural language, which means that the main code-structuring principle is the *procedure* or *function*. In contrast, Java is object-oriented, and its main code-structuring principle is that of the object or class. In this document, we mostly use the terms *procedure* and *function* interchangably.

A function is simply a named section of code that you can define. Parameters can be passed into the function, and a single result is returned at the end. Functions allow the same code to be called from multiple places in your program without copying the code. But just as importantly, used right they make your code much more readable and easier to maintain than it would otherwise be.

Java methods within classes are very similar in concept to C functions, but the details differ somewhat.

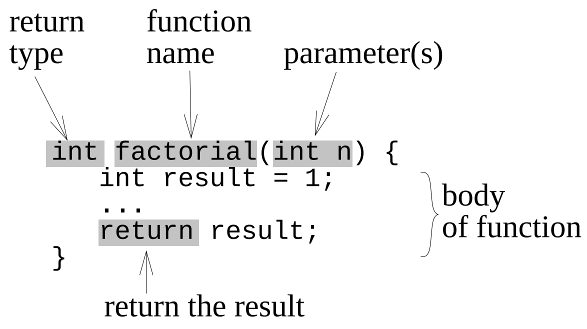
A simple example of a function is shown in the example below.

```
#include <stdio.h>

int factorial(int n) {
    int result = 1;
    int ctr;
    for ( ctr = 2; ctr <= n; ctr++) {
        result *= ctr;
    }
    return result;
}

main()
{
    int i;
    int fact_i;
    for (i = 1; i <= 10 ; i++) {
        fact_i = factorial(i);
        printf("factorial(%d) is %d\n", i, fact_i);
    }
}
```

The output of this program is:

```
eve.cs.ucl.ac.uk: ./foo
factorial(1) is 1
factorial(2) is 2
factorial(3) is 6
factorial(4) is 24
factorial(5) is 120
factorial(6) is 720
factorial(7) is 5040
factorial(8) is 40320
factorial(9) is 362880
factorial(10) is 3628800
```

In this example, we define the function `factorial()`, which takes a single parameter "int n", and also returns a value of type `int`:



```
int factorial(int n) {
   int result = 1;
   ...
   return result;
}
```

We then call this function multiple times from `main()` to calculate the factorials of the integers between one and ten.

It is important to realize that parameters in C are passed by *value*, whereas in Java they are passed by *reference*. When a parameter is passed by value, a *copy* is made, and is passed into the function. Here is a simple example to illustrate this:

```
#include <stdio.h>

void silly_example(int x)
{
    x = 2;
    printf("During call, x is changed to %d\n", x);
}

main()
{
    int i = 1;
    printf("Before call, i is %d\n", i);
    silly_example(i);
    printf("After call, i is %d\n", i);
}
```

```
eve.cs.ucl.ac.uk: ./foo
Before call, i is 1
During call, x is changed to 2
After call, i is 1
```

As you can see, within a function you can change the value of parameters just like you can with any other variable. Such changes have no effect on the original value of the variable passed into the function. By the way, the use of `void` here as the return type indicates that the function doesn't return any value.

What if you *did* want to change the *original* value of the variable passed into the function? Then you need to pass in a pointer, and the function needs to change the variable using that pointer:

```
#include <stdio.h>

void another_silly_example(int *x, int *y)
{
    *x = 2; *y = 3;
    printf("During call, *x is changed to %d, *y is changed to %d\n", *x, *y);
}

main()
{
    int a = 1, b = 1;
    printf("Before call, a is %d, b is %d\n", a, b);
    another_silly_example(&a, &b);
    printf("After call, a is %d, b is %d\n", a, b);
}
```

```
eve.cs.ucl.ac.uk: ./foo
Before call, a is 1, b is 1
During call, *x is changed to 2, *y is changed to 3
After call, a is 2, b is 3
```

In this case we passed in pointers to `a` and `b`. Within the function, because of the ordering of parameters, the pointer to `a` was called `x` and the pointer to `b` was called `y`.

We then changed the values of the memory pointed to by these pointers (`*x = 2; *y = 3;`). Thus when we return from the function, the values of `a` and `b` are seen to have changed.

String Parameters

As we saw in section 2.11, C does not have a first class string datatype. Thus when strings are passed as parameters, they're usually passed as type `char*`:

```
#include <stdio.h>

int string_length(char *s) {
    int i = 0;
    char *p = s;
    while (*p++ != '\0')
        i++;
    return i;
}

main()
{
    char *hw = "Hello World!";
    int len;

    len = string_length(hw);
    printf("The length of '%s' is %d\n", hw, len);
}
```

```
eve.cs.ucl.ac.uk: ./foo
The length of 'Hello World!' is 12
```

As a string was already really just a pointer to a region of memory, passing it into a function doesn't really change anything - it's still a pointer to the same memory. But of course if you change the contents of that memory from within the function, that memory will remain changed after the function.

Function Prototypes

C programs can be compiled in stages if you split your code into multiple files. If you do this though, the compiler will be confused if you try to reference a function in one file from within another file, because it won't know what types your functions return. To avoid this problem, you can declare a *function prototype* to help the compiler out. For example, suppose we wanted to spilt the string example above into two files called `strlen.c` and `main.c`, then we could do the following:

strlen.c:

```
int string_length(char *s) {
    int i = 0;
    char *p = s;
    while (*p++ != '\0')
        i++;
    return i;
}
```

main.c:

```
#include <stdio.h>

int string_length(char *s);

main()
{
    char *hw = "Hello World!";
    int len;

    len = string_length(hw);
    printf("The length of '%s' is %d\n", hw, len);
}
```

The actual `string_length()` function is defined in `strlen.c`, but before we use this function in main.c we declare a prototype:

```
int string_length(char *s);
```

This tells the compiler what the parameters and return value are for the `string_length()` function when it's separately compiling `main.c`.

To actually compile the code, you could use the `-c` flag to gcc, which indicates to compile the module specified to a `.o` object file, but not to link it. For example `strlen.c` gets compiled to `strlen.o`. Then the different object files are all linked together using gcc to produce the final executable program. Thus:

```
eve.cs.ucl.ac.uk: gcc -c strlen.c
eve.cs.ucl.ac.uk: gcc -c main.c
eve.cs.ucl.ac.uk: gcc -o foo strlen.o main.o
```

2.14 Input, Output and File Handling

We have already seen some examples of input and output, including `printf` for formatted printing and `fgets` to read a string. In this section we'll cover input, output, and file handling in a little more detail.

By default, a C program on Unix has access to three open *file descriptors* or *streams*:

- `stdin` - the standard input, which is read-only.
- `stdout` - the standard output, which is write-only.
- `stderr` - the standard error, which is write-only.

If the program was started from the command line, then by default, `stdin` will receive input from the keyboard, and both `stdout` and `stderr` will print to the terminal or terminal window.

The `printf` command sends formatted output to `stdout`. It takes a variable number of parameters. The first parameter is always a formatting string which includes formatting flags dictatinghow the remaining parameters should be interpreted. The formatting flags begin with a "`%`" and take forms like "`%d`" or "`%5.2f`". Each flag consists of a basic code, and various modifiers indicating how that value is to be formatted or otherwise modified. The most common basic codes are:

- `d` - print an integer as a signed decimal number.
- `u` - print an integer as a unsigned decimal number.
- `o` - print an integer as a unsigned octal number.
- `x` - print an integer as a unsigned hexadecimal number.
- `f` - print a double as a decimal number.
- `c` - print a single character.
- `s` - print a `char*` argument as a null-terminated string.

Additionally a field width and a precision may be specified before the basic code. For example:

- `%3d` - print an integer, padding the left with spaces if necessary to make at least three characters.
- `%5.2f` - print a double, padding the left with spaces to make five characters if necessary, and printing two digits to the right of the decimal point.

Many more flags and modifiers than these are available - see the printf man page for more details.

The `fprintf` command is similar to `printf`, but takes an additional parameter specifying which file descriptor to print to. For example:

```
fprintf(stderr, "Something bad happened!\n");
```

would print to the `stderr` stream.

To write a raw block of bytes to a stream, `fwrite` can be used. For example, to write 80 lots of 4 bytes to stdout from memory pointed to by `int *buffer`, you could write:

```
fwrite(buffer, 4, 80, stdout);
```

There are many different ways to read from a stream, including:

- `fread` - read a raw block of bytes from a stream.
- `fgets` - read a string until a newline is found.
- `fgetc` - read a character from a stream.

More details can be found in the relevant man pages.

The standard I/O streams are not the only places that you can read input and write output. You can open new streams using `fopen()` to open a stream to or from a file.

For example, to write to a file:

```
FILE *file;
char *filename = "/tmp/foo";

/* open the file for writing */
file = fopen(filename, "w");
if (file == NULL) {
    fprintf(stderr, "File %s could not be opened\n", filename);
    exit(1);
}

/* write to the file */
fprintf(file, "Hello World!\n");

/* close the file */
fclose(file);
```

In this example, we `fopen` the file "/tmp/foo" for writing. This should return us a pointer to a `FILE` data structure, which we can use as a file handle in subsequent calls. If we are unsuccessful in opening the file, `file` will be NULL, and then we print an error message on stderr.

If we opened the file correctly, we then use `fprintf` to write to the file, and finally close the file using `fclose`.

It is important not to forget to `fclose()` a file you've written to. File operations on files you've opened with `fopen` are *buffered*, meaning that the bytes may not immediately reach the file. This improves performance, but it also means that if your program exits without calling `fclose`, the file may be empty or truncated.

We can use `fopen` to open a file for reading in a similar manner. In the example below, we open a file, and then loop through reading a line at a time from the file and printing them out.

```
    FILE *file;
    char *filename = "/tmp/foo";

    /* open the file for writing */
    file = fopen(filename, "r");
    if (file == NULL) {
        fprintf(stderr, "File %s could not be opened\n", filename);
        exit(1);
    }

    /* loop while reading a line at a time from the file and printing */
    while (1) {
        char buffer[80];
        fgets(buffer, 80, file);

        /* if it's the end of file, break out of this loop */
        if (feof(file))
            break;

        printf("%s", buffer);
    }

    /* close the file */
    fclose(file);
```

In addition to buffered I/O, where the file handle is typically a pointer to a `FILE` data structure, C also supports unbuffered I/O. The file handle for unbuffered I/O is an integer *file descriptor*. Such a file descriptor is returned by the `open()` system call (for file operations) or by the `socket()` system call (for networking operations). File descriptors are closed using the `close()` system call, and reading and writing to file desciptors makes use of the `read()` and `write()` system calls. The basic semantics of these calls are the same as for buffered I/O, but the order and types of the parameters differ. Again more details can be found in the relevant man pages.

These unbuffered operations are a slightly lower level interface than the buffered interface. Most often you will use buffered I/O for file operations and unbuffered I/O for networking operations.

3 The C Preprocessor

The process of C compilation has three main passes - the preprocessor pass, the compilation pass, and program linking.

The preprocessor is practically a language in its own right. It allows you to include header files into your code, define constants and macros, and conditionally compile different parts of your code.

3.1 Including Header Files

In many of the examples so far, you have already seen:

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

This includes the contents of the system header file `stdio.h` into your source code file before starting the compilation pass. The file `stdio.h` contains definitions for things like the `FILE` file handle seen in section 2.14, and function prototypes for standard library functions like `printf`. If you don't include `stdio.h` then the compiler won't know how to compile these functions.

You can define your own header files if you have definitions you want to share across multiple C source files. For example, if you have the header file `constants.h`, then you might include this file at the top of each of your source files using:

```
#include "constants.h"
```

The only difference between this and the `stdio.h` example is the use of quotes instead of angle-brackets - this affects where the compiler searches for the file to include. Typically you use quotes for your own header files and angle-brackets for the system header files.

3.2 Preprocessor Constants and Macros

The preprocessor also supports the definition of constants and macros. Constants are very simple:

```
#define PI 3.141592
#define UCL "University College London"
```

The preprocessor will simply do a text subsitution, replacing every occurrence of the characters `PI` with the characters `3.141592`. These preprocessor statemenst are not regular C commands, so unless you want the substituted text to include a semicolon, do not put a semicolon at the end of a `#define` statement.

The preprocessor command `#define` can also be used to define macros to make your code easier to read. For example, if you wanted to define a macro to return the maximum of two numbers, you could define the macro:

```
#define max(X, Y) ((X) > (Y) ? (X) : (Y))
```

This defines a macro that takes two arguments. In this case it uses C's `?/:` notation to represent a simple if/else statement (see section 2.3). For example, you might use it like:

```
#define max(X, Y) ((X) > (Y) ? (X) : (Y))

a = max(c + d, e + f);
```

After preprocessing, the C compiler will see:

```
    a = (( c + d ) > ( e + f ) ? ( c + d ) : ( e + f )) ;
```

3.3 Conditional Compilation

Sometimes you need to have two different fragments of code in the same source code file, and compile one or the other depending on circumstances. Common uses are to conditionally enable debugging, or to use different code depending on the operating system on which the code is being compiled. The C prepocessor can do this for you using the `#ifdef` and `#if` commands. For example:

```
#include <stdio.h>

/* #define DEBUG */

#ifdef DEBUG
#define debug(s) printf("%s\n", s);
#else
#define debug(s)
#endif


main()
{
    int a = 1;
#ifdef DEBUG
    printf("a is %d\n", a);
#endif

    debug("reached here!\n");
}
```

This program will print out no output. But all you need to do is uncomment the line `#define DEBUG` and all the debugging output will be enabled.

Note that the macro `debug(s)` has two possible definitions. In the debugging case, the program gets compiled including the relevant `printf()` statements. In the no-debugging case, all the debugging information is removed by the preprocessor before the C compilation pass, so all this debugging information doesn't slow down the final program.

In this tutorial we have only touched on a few common uses for the preprocessor. Some C programmers make very extensive use of the preprocessor to balance readability with performance. However, excessive use of the preprocessor can make C programs hard to understand, because it's not so obvious where to look for where something is defined. For an extreme example, take a look at this winning example from the International Obfuscated C Competition by Vern Paxson:

```
    http://www.icir.org/vern/ioccc92.c
```

4 Linking and C Libraries

In Section 2.13 we saw an example of how to compile multiple program files together. Lets examine this a little further.

There are normally four stages to compiling a C program:

1. **Preprocessing.** As discussed in Section 3, this involves the C preprocessor expanding macros, including header files inline, and so forth. The output of this stage is the source code of a slightly longer C program than the one you started with.
2. **Compilation.** The C compiler will attempt to compile your C code. If it encounters no errors, the output of this stage is an assembly language file in the source code of the assemply language for your particular CPU.
3. **Assembly.** The Assembler for your particular CPU architecture will assemble the program into an object file (basically containing machine code for your CPU).
4. **Linking.** Multiple object files are linked together by the linker to form a complete executable program.

The C compiler can be directed to stop after any of these stages, rather than complete the compilation. The most common place to stop is after assembly, but before linking. This is useful when you are compiling a very large piece of code.

For example, if you were to examine the source code for the Unix `ls` utility, you would find the following files:

```
Makefile
cmp.c
extern.h
ls.c
ls.h
print.c
util.c
```

`Makefile` contains directives to the Unix `make` utility detailing how precisely to compile the program. The actual source code is contained in `cmp.c`, `ls.c`, `print.c` and `util.c`, while `ls.h` contains data structure definitions used by all the others and `extern.h` contains function prototypes for functions defined within the source code files that need to be accessed by code in the other source code files.

In principle, all four source code files could have been written as a single large file, but this makes it much harder to navigate and understand your program. Instead the author of the `ls` utility put all the functions related to the pretty-printing of directory listings into `print.c` and all the functions related to comparisons needed for sorting directory listings into `cmp.c`. The actual main body of the `ls` utility is in `ls.c`, and some remaining utility functions are in `util.c`.

Especially with larger programs, such separation by functionality helps the developer structure code and navigate it. Java enforces such a structure much more forcefully, but it's still good practice with C.

To manually compile the `ls` utility you'd do the following:

```
gcc -c cmp.c
gcc -c print.c
gcc -c util.c
gcc -c ls.c
gcc -o ls ls.o print.o util.o cmp.o
```

The first four of these steps involve preprocessing, compiling and assembling each of the four source files separately. The last stage takes the four object files (ending in `.o`) and links them together to produce the final `ls` executable program.

Another advantage with separately compiling the four source files is that when you change your program, you only need to recompile those pieces that have changed. Suppose that after compiling `ls` as shown above, you make a small change to `print.c`. So long as this doesn't change how anything in the other files calls anything in `print.c`, then all you need to do to recompile the program is:

```
gcc -c print.c
gcc -o ls ls.o print.o util.o cmp.o
```

Of course if you did something like change the order of parameters in a function that the other source files use, then you'd need to change the function prototype in `extern.h`, and also recompile all the object files. If you didn't do this, you'd get unpredictable results (well, they'd be predictably wrong, but what they'd actually do is not obvious).

4.1 Linking with Libraries

Suppose you need to write code that performs some trigonometry. The C maths library contains many useful maths functions that are not part of the basic C language, but are common to just about every C compiler suite. So here's a trivial program called `trig.c` to print of the value of $\sin(\pi/4)$.

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

int main() {
    double angle = M_PI/4;
    printf("sin(%f) = %f\n", angle, sin(angle));
}
```

To use the function `sin()`, you need to include `math.h` so that the C compiler knows `sin()` returns a double.

Now, to separately compile your program, in addition to the usual steps you also need to tell the C compiler to include the object code for the `sin()` function, which is typically stored in `/usr/lib/libm.a` or some similar location. So you could enter:

```
gcc -c trig.c
gcc -o trig trig.o /usr/lib/libm.a
```

But that's somewhat cumbersome to remember and not terribly portable if someone happens to have decided that the maths code should reside in `/usr/lib/i386-redhat-linux7/lib/libm.a` instead. So there's a shortcut which says to link with the maths library, wherever it is installed:

```
gcc -c trig.c
gcc -o trig trig.o -lm
```

If you forget the `-lm` flag, the linker will fail to find the code for `sin()` and you'll get a warning about "`undefined reference to sin()`" or some similar error.

Of course the maths library isn't the only library available. Over the years many people have written a huge number of useful functions for you to use. A typical system today comes with many hundreds of libraries pre-installed, and you can install additional ones as needed.

5 Debugging C Code

So your code compiles but doesn't do what you wanted. The first thing is to read your code through, and see that you understand what it should do. Read it carefully, slowly, line by line, and often errors will become obvious. But what if they don't? You're sure, reading your code, that it should work, but it doesn't.

Either you don't understand C or you don't understand your code. Perhaps both. We'll come to debugging strategy later, but for now, lets look at some common ways new (and sometimes experienced) C programmers shoot themselves in the foot - code that looks like it should work but does something completely different.

5.1 Common Errors

We've seen a few errors in the earlier sections, but it's worth repeating them here, since you're probably reading this section while cursing as some recalcitrant piece of C, and don't really want to re-read all the earlier sections.

Assignment instead of comparison

```
if (a = 2) /*do something*/
```

This assigns `a` to be 2. You probably meant:

```
if (2 == a) /*do something*/
```

This compares `a` to 2, and has the added benefit that if you typed = instead of ==, the compiler will complain since you can't assign 2.

Spurious Semicolon

```
int x = 0;
while (x < 10); {
    printf("x=%d\n", x);
    x++;
}
```

This code compiles just fine, but goes into an infinite loop and doesn't print anything at all. The culprit is that semicolon after `(x < 10)`. That terminates a statement; in this case a very very short rather empty statement, which comprises the body of the loop. It's exactly as if you'd typed:

```
int x = 0;
while (x < 10) { /*do nothing forever*/ };

/* completely unrelated block follows */
{
    printf("x=%d\n", x);
    x++;
}
```

So be careful about those semi-colons.

Missing Parameters and Function Prototypes

It's all too easy to miss parameters out when you're calling functions. For example:

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

In this case, there are two cases of `%d` in `printf`'s format string, but only one parameter, `x`. This isn't technically a compile-time error, because it's only at runtime that C really knows what the format string is, and then it just takes the next values off the stack (even if the values aren't there!). Modern C compilers probably warn you about this in common cases, but don't count on it.

With printf, the problem was that printf can take a variable number of arguments, and can't tell at compile time how many there should be. The same problem can happen with functions you declare in separate files.

For example, suppose you take two files that you separately compile (using the `-c` option), and then link the whole program together:

myfunc.c:

```
    double myfunc(double a, double b) {
        return a + b;
    }
```

main.c:

```
    #include<stdio.h>

    main(){
        double x = 2, y = 2, z;
        z = myfunc(x);
        printf("z=%f\n", z);
    }
```

You compile your code, and the compiler produces no errors. You link it: no errors. You run it, and it prints out:

```
z=0.000000
```

Ok, we all know that 2 + 2 is not zero. So what went wrong?

It's the same problem we saw with `printf`, but this time the C compiler never has enough information to see that you missed out the second parameter to `myfunc()`.

When you compiled `main.c`, C didn't know to look in `myfunc.c` to see how many parameters `myfunc` actually took, so it assumed you know what you were doing and put the value of `x` on the stack before inserting the stub call to `myfunc()`.

When you compiled `myfunc.c`, C just assumed you knew what you were doing and assumed the top eight bytes on the stack held the value of `a` and the second eight bytes held the value of `b`, even though you never put the second eight bytes there. Ooops.

OK, lets fix the code:

main.c:
```
#include<stdio.h>

main(){
    double x = 2, y = 2, z;
    z = myfunc(x, y);
    printf("z=%f\n", z);
}
```

You compile it (no errors), link it (still no errors), and run it:

```
z=1073741824.000000
```

Around this time, you're probably cursing and reaching for the next can of Red Bull.

Now, even after 5 cans of "energy drink", you probably remember that two plus two does not normally equal 1073741824. What's wrong now?

The problem is that when C compiled `myfunc.c`, it knows that `myfunc()` returns a `double`, so it put eight floating point bytes on the stack when it returned.

But when C separately compiled `main.c`, it didn't know what type `myfunc()` returned, so it assumed you knew what you were doing. Specifically, it assumed that if you didn't tell it otherwise, `myfunc()` returned an `int`. So it blindly took four bytes off the top of the stack, converted them (unnecessarily) from an `int` to a `double`, and assigned them to `z`.

C doesn't hold your hand.

So what should you do, so C can do the right thing, and so it can warn you when your fingers don't type what your brain as thinking? You just need to add a function prototype to `main.c`, to tell C what you really mean:

main.c:
```
#include<stdio.h>

double myfunc(double a, double b);

main(){
    double x = 2, y = 2, z;
    z = myfunc(x, y);
    printf("z=%f\n", z);
}
```

This provides enough information for C to warn you if you miss out a parameter, to convert parameters to the right types, and to interpret return values correctly. Finally 2 plus 2 does equal 4.

Integer Arithmetic

```
        double half = 1/2;
        printf("half=%f\n", half);
```

You might expect this to print out 0.5, but no:

```
half=0.000000
```

The problem is simply that 1 and 2 are integers, and so C assumes you want integer division. `1/2` rounds down to 0 as an `int`, and only then does C convert the result to a `double`.

If, instead, you had typed:

```
    double half = 1.0/2;
    printf("half=%f\n", half);
```

Then you'd have got the result you expected. In this case, C sees that `1.0` is a `double`, and so it converts `2` from an `int` to a `double` too, then does double precision floating point arithmetic, and finally assigns the result to `z`.

But what if you had code like the following:

```
    int a = 1;
    int b = 2;
    double half = a/b;
    printf("half=%f\n", half);
```

Now you've got the original problem back, and you need to be more explicit about what you really mean. To do this you need to tell C to convert (or "cast") `a` and `b` to doubles before doing the arithmetic. The correct incantation for this is:

```
    int a = 1;
    int b = 2;
    double half = (double)a/(double)b;
    printf("half=%f\n", half);
```

As you had hoped, this now prints out:

```
half=0.500000
```

6 DEBUGGING CRASHES

Sometimes C code will crash. For example, lets say you have been writing a simple program similar to that in Section 2.12 that reads in numbers from the keyboard and prints them out in reverse order:

```
eve.cs.ucl.ac.uk: gcc -o foo foo.c
eve.cs.ucl.ac.uk: ./foo
Enter an integer: 10
Enter an integer: 20
Enter an integer: Ctrl-D Finished
Value: 20
Value: 10
Bus error (core dumped)
```

After a few expletives, you're probably wondering what `Bus error (core dumped)` is trying to tell you.

There are three of these errors you'll probably see more often that you would like:

- `Segmentation fault: core dumped.`
  This tells you that the program tried to access memory that did not belong to it. The error was detected by the OS.
- `Bus error: core dumped.`
  This also tells you that the program tried to access memory that did not belong to it. This time the error was detected by the CPU.
- `Floating point exception: core dumped.`
  This informs you that the CPU's floating point unit trapped an error.

OK, so the program tried to access memory *that did not belong to it*. That's a bit vague. It doesn't even tell you which line of your program tried to access this memory.

To understand completely what's happening, we need to understand a lot more about how the OS virtual memory system works, and that's a whole topic in its own right. Roughly speaking though, the virtual memory layout as seen by your program looks something like Figure 1

When your program first starts, only the program itself (the "Text") is loaded into memory, and the operating system makes it Read-Only, meaning the program is protected from modifying itself.

As your program calls functions, local variables from those functions are stored on the Stack. This means that the stack grows downwards through memory as you call functions, and shrinks back again as those functions terminate.

Every time your program calls `malloc()`, memory is allocated from the Heap. If there's not sufficient spare memory in the Heap to satisfy malloc, the memory allocator asks the OS for more memory, and the Heap grows upwards through memory.

And right at the top of virtual memory sits the operating system kernel. It's protected from your code, so this part of the address space is not accessible.

With this basic memory layout in mind, we can see that if your program tries to write to an address in the Text or in the Kernel, this is a protection error. The write will fail, and your code will crash with `Segmentation`

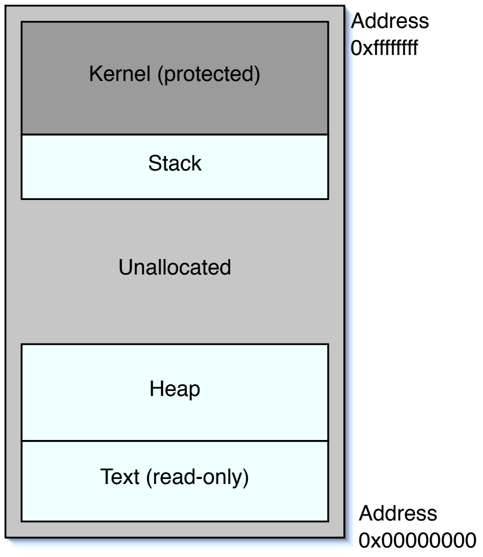


Figure 1: Typical Virtual Memory Layout of a Running Program

`Fault`. If your program tries to read or write from the Unallocated region, the program will crash with `Bus Error`. The very first page of memory starting at address 0x00000000 is also left unallocated to catch common bugs. And if it ties to read from Kernel memory it will also crash. A write will only succeed in Stack or Heap memory, and read will only succeed in Stack, Heap, or Text.

Generally then, a Bus Error or a Segmentation Fault is cause by a read or a write to an unintended piece of memory. Almost always this means you did something wrong with a pointer. But which pointer?

The problem you face is that when your program tried to read or write to protected memory, the error was trapped (either by the hardware or the OS) and the program was terminated. Unlike in Java, there's no virtual machine to unwind what was happening and tell you where the error was. The program simply isn't running anymore, and that's that.

At this point you should reach for the debugger. On Linux, the debugger is called `gdb`, and it will help you understand what went wrong.

6.1 Using gdb

In the example above, the message the Operating System gave you in response to the crash was `Bus error (core dumped)`. In this case the OS has been kind enough to save a complete memory image of your program as it was at the point when it crashed. You can use the debugger to examine this core file to find out what went wrong.

Sometimes though, the OS has been told not to dump core files - this is commonly done to avoid lots of huge files clogging up the filesystem. On Linux and MacOS, you can see what the limits are set to using the `limit` command:

```
eve.cs.ucl.ac.uk: limit
cputime unlimited
filesize unlimited
datasize unlimited
stacksize 10240 kbytes
coredumpsize 0 kbytes
memoryuse unlimited
vmemoryuse unlimited
descriptors 1024
memorylocked 4 kbytes
maxproc 7168
```

You can see here that core dumps are disabled. You can reenable them by setting a large limit as follows:

```
eve.cs.ucl.ac.uk: limit core 100000
```

Now you can run gdb and see what your code was doing at the moment it crashed, but first we need to recompile your code with the `-g` flag to tell gcc to include the symbols gdb needs to work properly:

```
eve.cs.ucl.ac.uk: gcc -g -o foo foo.c
eve.cs.ucl.ac.uk: ./foo
Enter an integer: 10
Enter an integer: 20
Enter an integer: Ctrl-D Finished
Value: 20
Value: 10
Bus error (core dumped)
eve.cs.ucl.ac.uk: gdb foo core
GNU gdb 5.3-20030128 (Apple version gdb-309)
...
Core was generated by `./foo'.
#0 0x00001d20 in main () at foo.c:44
44 printf("Value: %d\n", p->value);
(gdb) print p
$1 = (struct element *) 0x0
```

After the program crashed we loaded the program (“foo”) core file (“core”) into gdb. Right away, gdb can tell you precisely which line of code was being executed at the moment the crash happened. In this case we can see that it is line 44, and all the code was trying to do was print something.

Now, crashes like these are memory access problems, and the way to make a mistake with memory access is to do something wrong with a pointer. In this case there's only one pointer in sight: p. So that's the likely culprit. The code is trying to dereference p, so if p has somehow got an illegal memory address, then that would be a problem. We can print the value of p, and we can see that it has the value 0x0 (zero in hexadecimal). This is not a legal memory location, so that's the problem.

Lets take a look at the source code around line 44:

```
/* print out the numbers we stored */
    p = head;
    for (i=0; i<10; i++) {
        printf("Value: %d\n", p->value);
        p = p->next;
    }
```

We can see that the print statement is embedded in a for loop. Initially p is given the value of head, then on subsequent passes through the loop it is given the value of p->next as the code traverses the linked list. How then did the value 0 get into p?

gdb can help with this too. You can run your code in gdb, stop it, print out variables, and restart it. Our code is already in gdb, so lets re-run it and watch how p changes. To do this we will set a *breakpoint*, which will cause the program to pause each time the breakpoint is reached.

```
(gdb) break foo.c:44
Breakpoint 1 at 0x1d14: file foo.c, line 44.
(gdb) run
Enter an integer: 10
Enter an integer: Ctrl-D Finished

Breakpoint 1, main () at foo.c:44
44 printf("Value: %d\n", p->value);
(gdb) print p
$2 = (struct element *) 0x100140
```

The command break foo.c:44 sets a breakpoint at line 44 (the printf statement) in foo.c. Then we restart the program with the run command. This time we only enter one number before terminating input with CTRL-D. The code runs until it reaches line 44, and then it pauses, handing control back to gdb.

When we print the value of p we can see it's 0x100140. On our system, this is a valid pointer, and it's certainly not zero. So looking back at the code, we can determine that the first assignment of p to the value of head is probably not the culprit.

We can then tell gdb to continue running the program:

```
(gdb) cont
Continuing.
'Value: 10

Breakpoint 1, main () at foo.c:44
44 printf("Value: %d\n", p->value);
(gdb) print p
$3 = (struct element *) 0x0
```

We can see it printed out the first (and only) value correctly, then came back to line 44 again and tried to print out the next value. But there was no next value - we've reached the end of the linked list. This is the culprit. The code is missing a check to see if we've reached the end of the list.

Another very useful feature is to be able to print out a stack trace, which allows you to see which function is calling which other function.

Suppose you had a different version of the linked list code that used a recursive solution to print out the linked list instead of a for loop:

```
struct element {
    int value;
    struct element* next;
};

void print_element(struct element *p) {
    printf("Value: %d\n", p->value);
}

void print_list(struct element *p) {
    print_element(p);
    print_list(p->next);
}

main() {
    struct element *head = NULL;
/* read in some numbers */
...
    print_list(head);
}
```

This code has the same bug and will crash in the same way. We can fire up gdb and see what went wrong:

```
eve.cs.ucl.ac.uk: gdb foo core
GNU gdb 5.3-20030128 (Apple version gdb-309) (Thu Dec 4 15:41:30 GMT 2003)
...
Core was generated by `./foo'.
#0 0x00001c04 in print_element (p=0x0) at foo.c:11
11 printf("Value: %d\n", p->value);
(gdb) backtrace
#0 0x00001c04 in print_element (p=0x0) at foo.c:11
#1 0x00001c40 in print_list (p=0x0) at foo.c:15
#2 0x00001c4c in print_list (p=0x300140) at foo.c:16
#3 0x00001c4c in print_list (p=0x300150) at foo.c:16
#4 0x00001d20 in main () at foo.c:52
(gdb) up 2
#2 0x00001c4c in print_list (p=0x300140) at foo.c:16
16 print_list(p->next);
(gdb) print p->next
$1 = (struct element *) 0x0
```

This time when the code crashed, we used the `backtrace` command to print out a stack trace of which function was calling which. We can see that `main()` called `print_list`, which called `print_list` which called `print_list` which was calling `print_element` when the code crashed. It's clear than on the third iteration, `p` became zero. We can move back up the stack - `up 2` moves up two stack frames - and use print to examine variables. In this case we can see that `p` became zero because `p->next` was zero in the previous element of the linked list. The code needs a check for this added.

Without the debugger, these sort of crashes can be very hard to debug. Using the debugger though, simple crashes like this one are quite easy to understand.

Unfortunately not all crashes are so simple. When using pointers to write memory, the OS can only detect an error and terminate the program if the program writes to an illegal address. If it writes to the wrong address, but it happens to be a legal address on the stack or heap, it won't be detected but it will corrupt memory and make your program behave eratically. Under some circumstances it can corrupt the stack sufficiently that gdb cannot even figure out which function the program was executing. The moral of the story is to be careful with those pointers.