A C tutorial/refresher

2018-2

Joel Fenwick (editor)
Justin Goldizen
Mitchell Krome
Will Toohey

Contents

Chapter 1

Commandline Linux environment — gcc

The following assumes that you have access to a Linux/Posix/BSD commandline. This could either be a local terminal window on operating systems which support it or via an SSH program such as Putty. We will assume you are using the `bash` shell and the `gcc` C compiler.

1.1 Creating source files

C source files can be created and edited using any plain text editor[1] (eg: `nano` or `vim`). Be aware that different operating systems have different ideas about what the end of a line looks like. For example, compiling a file on Linux which has Windows line endings in it will lead to compile errors. With this in mind, try to edit on the same platform you will be compiling on (or failing that, use a tool which respects the target platform's line endings[2].

Some systems have case-sensitive filenames and some are case-insensitive (eg: on Windows, `bob.c` is considered to be the same file as `Bob.C`). Regardless of your home system, you should try to use consistent cases. For `gcc`, C source files must end in `.c` (lower case) or later on `.h`.

Put the following in a file called `test.c`:

```c
int main()
{
    return 5
}
```

Then compile it with the command:

```
gcc test.c
```

which will produce a message like:

[1]Some tools (such as MacOS's TextEdit) can save formatting or not. Make sure you are using text only.

[2]For example, Notepad++ can be set to act in this way.

```
test.c: In function ‘main’:
test.c:4:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

Now type:

```
echo $?
```

This asks the shell to tell you what happened with the previous command (its "exit status"). By tradition, anything other than 0 indicates some form of error. In this case, the compiler failed and `$?` reflects that. Now edit `test.c` to add a semicolon after the 5, then recompile. You should not see any messages.

Look in the current directory, and you should see a new file called `a.out`. This is the default name for programs made with `gcc`. The name can be changed by using the "-o" option:

```
    gcc test.c -o test
```

This would produce an output file named `test`.
Execute your program with:

```
    ./a.out
```

and check that the exit status was 5 as expected. When a program returns from main its "exit status" is the value returned.

Depending on your requirements, you may need to comply with some particular version of the C language standard. Adding one of the following to your compile line will do this.

-ansi	C89/C90
-std=c99	C99
-std=gnu99	C99 with some gnu extras

1.2 Linking and other compile options

`gcc test.c` actually performs a number of phases including compiling and linking. The distinction between these is more important when a program's source consists of multiple files.

```
    gcc -c file1.c
    gcc -c file2.c
    gcc -c file3.c
```

Would compile each of the files (and check them for correctness) to a corresponding `.o` file. Following that:

```
    gcc file1.o file2.o file3.o
```

would "link" i.e. combine the pieces together and check for consistency and completeness.

Adding `-Wall` and/or `-pedantic` will cause `gcc` to report more potential problems. *This is a good thing.*

1.3 Other notes

It may seem obvious, but do not ignore error or warning messages. While warnings for other languages often mean “your code could be written more nicely”, warnings in C often mean “your code will not do what you think it will”.

You can save error messages by adding `2>errfilename` to the end of a command. For example:

```
gcc myfile.c 2>errors.log
```

Chapter 2

First program

This part assumes that you are familiar with strings and arrays from some other programming language.

Compile and run the following:

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     printf("It begins\n");
5     return 0;
6 }
```

1 `#include` is a “preprocessor directive”[1] which tells `gcc` to consult another file before going any further. If you remove this line, you will see that `printf` becomes a problem. This is because the `stdio.h` file is what tells `gcc` about the `printf` function.

  The `<>` around `stdio.h` indicate that it is a system include rather than a header we created ourselves[2]. For our own headers, put the filename in double quotes.

3 begins a function definition[3]. All standard C programs must have a function called `main`. Unless your system has special requirements, you should use the signature[4] shown here. We have not discussed types yet (we’ll look at types in Section 4.1.2) but for now, `int` is an integer[5] while `char` on its own represents a character. The first `int` means that our `main` function returns an int. The parts in parentheses are the parameters for the function. The first one is an integer and the second one has something to do with characters. Without explaining why now, the second parameter

[1]as opposed to a C statement or declaration

[2] More accurately, `<>` indicate which paths the compiler should look for the file in.

[3]A definition tells the compiler both that the function exists(called declaration), and what the function does.

[4]A function’s signature gives the function’s name, its return type and the type and order of parameters.

[5]A whole number, either positive or negative

is an array of strings. Note that this program doesn't actually make use of the parameters it declares.

4 This is a function call. It has the name of a function `printf` followed by arguments in parentheses. Text in double quotes represents a string. So this line passes a string to the `printf` function.

5 indicates which value the function returns/sends back to its caller. For `main`, the return value is sent back to the operating system as the program's exit status.

Chapter 3

Simple output and Command Line arguments

Compile and run the following:

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     printf("Going to print some things %s%s%s\n");
5     return 0;
6 }
```

Although you *can* just print strings directly with printf, it isn't necessarily a good idea (as shown above)[1]. To understand why, we need to look at how `printf` deals with variables. Replace Line 4 with

```
printf("%s\n", "Going to print some things %s%s%s ");
```

The first parameter of `printf` describes what is to be printed: any sequence starting with % is called a placeholder and will be replaced with something else when the output is printed. Characters after the % indicate what type of thing is to be printed (and possibly how it is to be formatted). So %s in the first argument tells `printf` that it should look at the next argument along, interpret it as a string and print it. If there aren't enough arguments, or the argument is the wrong type, unexpected things may occur.

To print an integer instead of a string, use the %d placeholder. For example: `printf("%d + %d = %d\n", 5, 7, 12);` You can use any mixture of placeholders and normal characters in the *format string* that you like. To print a % symbol, use %%.

Compile the following:

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4     printf("There were %d arguments given to the program %s\n",
5       argc, argv[0]);
6     return 0;
```

[1]`puts()` is probably a better option

```
7 }
```

Lines 4 and 5 show that you can break lines to improve layout and readability. Try running the program with a variety of command line arguments. You will see that `argv[0]` contains the name of the program (`argv` stands for "argument values") and that `argc` gives the "argument count" — how many there were. Since `argv` always has at least one thing in it, `argc` ≥ 1 and accessing `argv[0]` should always be safe. But change the program to print `argv[200]` and see what happens.

By using a conditional statement, we can safely print more if it is available.

```c
#include <stdio.h>

int main(int argc, char** argv) {
    printf("0: %s\n", argv[0]);
    if (argc > 1) {
      printf("1: %s\n", argv[1]);
    }
    if (argc > 2) {
      printf("2: %s\n", argv[2]);
    }
    return 0;
}
```

Some type of loop would be a better way to deal with this, so let's do that now.

3.1 Loops

A loop consists of the following parts (some of which may be empty):

Setup — happens once (before the loop starts).

Test — should the loop repeat or stop now?

Body — what the loop is supposed to do each time.

Update — happens after the body runs each time.

So to display command line arguments:

Setup — set a counter variable (`i`) to 0.

Test — is `i` less than `argc`

Body — print the argument

Update — increase `i`

C has three types of loop structure:
The *for* loop:

```
for (Setup ; Test ; Update) {
    Body
}
```

The *while* loop:

```
Setup
while (Test) {
    Body
    Update
}
```

and the *do-while* loop:

```
Setup
do {
    Body
    Update
} while(Test);
```

The *do-while* loop differs from the *while* loop by always running the Body and Update once before the Test is performed and the loop repeats.

So we could do this:

```c
#include <stdio.h>

int main(int argc, char** argv) {
    for (int i = 0; i < argc; i = i + 1) {
        printf("%d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Exercise:

Rewrite the above to use `while` and then to use `do-while`.

Chapter 4

Operators and expressions

In most cases in C, wherever you write literal values, you can write expressions[1]. An expression is a fragment of code which can be "evaluated" — that is, they can be processed to get a value. For example: `3+4` is an expression which gives the value 7 when evaluated. In fact, literal values are also expressions (they evaluate to the value as written).

4.1 Arithmetic operators

The operators $+$ and $-$ work as you would expect. Division is written with $/$ and multiplication as $*$.

The % operator (known as "modulo") returns the remainder of its arguments; `a%b` returns the remainder when `a` is divided by `b`. For example: `7%3` is 1 and `23%5` is 3.

4.1.1 Precedence

When an expression involves more than one type of operator, the issue of precedence comes up. That is, some operators are evaluated before others — this is independent of the order the operators are written in. The rules for $+, -, *, /$ are the same as for normal arithmetic.

You can look up the precedence for operators, but you can also test them. For example, $+$ and $*$ could be compared like this: (Enter and run the following)

```c
#include <stdio.h>

int main(int argc, char** argv) {
   printf("%d %d %d\n", (2 + 3 * 7), ((2 + 3) * 7),
           (2 + (3 * 7))); // Continues from prev line.
   if ((2 + 3 * 7) == ((2 + 3) * 7)) {
      printf("+ is evaluated first\n");
   } else if ((2 + 3 * 7)==(2 + (3 * 7))) {
      printf("* is evaluated first\n");
```

[1]More information: there are some things which need to be known at *compile time* rather than *run time*. The most obvious cases of this are *case labels* and array sizes in ANSI-C.

```
    } else {
        printf("Something has gone horribly wrong\n");
    }
    return 0;
}
```

Exercise:
Modify the program above to test relative precedence of `+` and `%`.

4.1.2 Simple Types

All of C's simple types are numeric in some way. This includes types which have other purposes (`char` and `bool`).

Numeric types can be classified in a number of ways:

- Integer vs Floating point — `bool`[2], `char`, `int` are integer types while `float` and `double` are floating point types. Arithmetic operators which act on integers *will evaluate to an integer result* so for example, `5/2` is `2`, not `2.5`. Do not make this mistake: `float f=2/3; f` will contain 0 not 0.66. This is because storing a value in a variable does not influence the way the expression is evaluated.

  If an expression contains a mixture of types, its arguments will be converted to the most general type. For example: `2/3.0` will give a floating point result because one of its arguments is floating point[3]. You could also achieve this with a *cast*: `2/(float)3` or multiplication `(1.0*2)/3`.

- Signed vs Unsigned — integer types can be restricted to store only positive values[4]. For example: `unsigned int`, `unsigned char`[5].

- size — types can be modified to indicate how much memory they take[6]. Types which take more memory can store a larger range of values. For example: `short int` and `long int` are smaller and larger than normal `int`s respectively. These can be combined with `signed` and `unsigned`.

There are other type modifiers but we'll keep this simple and standard[7].

To output other types with `printf`, different placeholders are required:

`%e %f %g` — all of these will output a `float` *or* `double` but they format it in different ways.

`%c` — prints a char (as a character rather than as a number).

`%hd` — prints a `short int`.

`%ld` — prints a `long int`.

`%u` — prints an `unsigned int`.

Consult documentation for other combinations.

[2]`stdbool.h` must be included to use `bool` (and `true` and `false`).

[3]Somewhat confusingly, the default floating point type in C is `double` not `float`.

[4]and therefore store higher maximum values using the same number of bits because the sign bit can be used for value storage.

[5]`int` defaults to `signed int` but if you care whether your `char`s are signed or unsigned, it is better to be explicit.

[6]In C, these modifications are not absolutely defined in the standard.

[7]For example: `long long int` and `quad double` are non-standard types.

4.2 Bit operators

These operators act on the individual bits in the value.

- `a&b` — bitwise *and* of the two values.
- `a|b` — bitwise *or* of the two values.
- `a^b` — bitwise *exclusive or* of the two values.
- `a<<b` — left shift `a` by `b` bits.
- `a>>b` — right shift `a` by `b` bits.
- `~a` — one's complement of `a`.

4.3 Relational operators

These return a boolean value `true` or `false`. Remember that in C, `bool`s are also integer values. A 0 indicates false, and any non-zero value is interpreted as true.

`a < b, a <= b, a > b, a >= b, a != b, a == b`. Be careful comparing floating point values, since numerical errors can creep in[8].

4.4 Logical operators

These look like their bitwise counterparts so it is important not to mix them up.

- `a && b` — logical *and.* eg: `true && true == true`
- `a || b` — logical *or.* eg: `false || true == true`
- `!a` — logical *not.* eg: `!(a || b) == (!a && !b)` (deMorgan's law)

`&&` and `||` are "short-circuiting" operators: the second argument will not be evaluated unless required (ie: in `false && EXP` and `true || EXP`, `EXP` will not be evaluated).

4.5 Other operators

The following are also operators in C but we won't talk about them here.

- unary `*` for pointers (as opposed to arithmetic)
- `->` for pointers
- `+ -` for pointers
- `[ ]` for array subscripts
- ` ? : ` — the ternary operator.
- `,` — the comma operator.

[8]A better option for `a==b` is to check if the difference is small enough. $|a - b| < tolerance$ $\Rightarrow$ `fabs(a-b)<0.0002`

Chapter 5

Functions and Passing Variables

A function declaration in C consists of a return type, a function name and a parameter list. For example:

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

A function definition has that information followed by a function body. If no body is given, the declaration should be followed by a semi-colon.

So `void show_greeting(void);` is just a declaration. It tells the compiler that the `show_greeting` function exists but not how to do it.

In contrast,

```
void show_greeting(void) {
    printf("Hello\\n");
}
```

is a definition (which also counts as a declaration).

Unlike many other languages, to tell the compiler that a function has no parameters, you need to write `(void)` for the parameter list instead of just `()`. A return type of `void` means that the function does not return anything.

If a function has parameters, then the type of each parameter needs to be given explictly (as opposed to declaring normal variables, where you can do this: `int i,j,k`).

For example, the following function returns the maximum of two integers.

```
int maximum(int a, int b) {
    if (a < b) {
        return b;
    } else {
        return a;
    }
}
```

Exercise:

Write a function which *returns* the maximum of three integers and a program to test it.

Enter and compile the following:

```c
#include <stdio.h>

void swap(int p, int q) {
    int temp;
    printf("p=%d q=%d\n", p, q);
    temp = p;
    p = q;
    q = temp;
    printf("p=%d q=%d\n", p, q);
}

int main(int argc, char** argv) {
    /* variables can be initialised as they are declared */
    int a = 3, b = 4;
    swap(a, b);
    printf("a=%d b=%d\n", a, b);
    return 0;
}
```

Note that the parameters listed for the swap function are called `p` and `q` but the arguments passed to the function when it is called are from variables `a` and `b`. This does not matter. There is no relationship between parameter names (used inside a function) and the names used to pass arguments into the function. If we had named the parameters `a` and `b`, that would not make a link between the variables inside `swap` and inside `main`.

Bear in mind that even though the compiler doesn't pay any notice if the variables are the same, people might. Try to avoid switching variable names around for no reason.

When you have run the above program, you will notice that it doesn't seem to work. The variables in `main` still have their original values. However, `p` and `q` seemed to have swapped over just fine.

The reason for this is that arguments to C functions are passed *by value*. That is, the *value* of each argument is copied into the parameter variables inside the function. So, the function swaps values in variables but that doesn't change anything outside of the function.

To demonstrate this further, try changing the swap call to `swap(2, (5*7)-6)`.

To allow functions to modify their arguments, then you need to use *pass by reference*, which C doesn't really have. Instead we make use of *pointers*. A pointer is a way to refer to memory by its address rather than by variable name. You can get the address of any variable using the `&` operator. So `&length` gives a pointer to the `length` variable.

Now for the fun part. Pointers are values as well and can be stored in variables. `int pl=&length` won't work though because `&length` isn't an `int`[1], it is a *pointer-to-`int`* (or an *`int`-pointer*, the names are equivalent).

In C, pointers are declared by writing `*` between the type-being-pointed-to and the name.

```c
int* lp; // ok but a meaningful name would be better
```

[1]Yes, technically underneath it is an unsigned integer value of some type (probably `unsigned long`). But it is more helpful for programmers when the compiler can notice that this wasn't supposed to be a normal number.

```
2  int* lpointer; // longer but not necessarily any more meaningful
3  int* i, j; // Warning: this does not mean what you think
4  int *k; // the * can also go against the variable name
5  int*p; // also legal. C doesn't care about spaces here
6  int * q; // Now you're just being silly. (Still legal though)
7  int** m; // Pointers to pointers are okay too
```

Lines 5 & 6 are examples of legal code which is unnecessarily hard to read. Remember that people (including you) will need to read your code later and you don't want that to be difficult.

Line 3 illustrates a tricky part of pointer declarations. Regardless of where you put the `*`, only the next variable name will be a pointer. So, Line 3 declares `i` to be a pointer to an `int` and `j` to be an ordinary (non-pointer) `int`.

Important note: there is no compelling argument as to which `*` position is more correct (`int* x` or `int *x`). Many people will try to tell you that there is though. It is more important that you use a form which you are comfortable with and (perhaps more importantly) is consistent with the rest of the code you are writing. Most importantly, you need to conform with whatever style rules are in force for your organisation.

Line 7 is an example of a pointer to a pointer. Since a pointer is a regular value, it must reside in memory, and so can be referred to with another pointer.

Modify the swap declaration to be: `void swap(int* p, int* q)` and try to compile.

The problem is that the function now expects pointers and we are passing in `int`s. So change the swap *call* to `swap(&a, &b);`. This means that the function expects pointers and we are giving it pointers. The code inside the function is not right yet though. Line 3 tries to print out the parameters but we don't care what the addresses of the variables are, we want to know what is at the end of the pointers. Remember that these pointers are pointing at `a` and `b` and we want to swap what is in `a` and `b`. So we need the *dereference* (follow-the-pointer) operator. This is a `*` before the pointer. Line 3 would become: `printf("p=%d q=%d\n", *p, *q);`

In fact, all the other references to `p` and `q` need to be changed to `*p` and `*q`. Compile and run to see if it worked.

So what happened here? Suppose `a` and `b` are stored in locations `xFFFF0000` and `xFFFF0008` respectively.

- `swap(&a, &b)` will become `swap(xFFFF0000, xFFFF0008);`.
- Those pointer *values* will be *copied* into the function variables `p` and `q`.
- `temp = *p` would become `temp = *(xFFFF0000)` ie `temp =` the value stored at location `xFFFF0000`. That is, `temp = 3;`

So, the pointers were passed by value (copied) but a copy of a pointer is just as good at accessing the original thing.

Main points to remember:

1. If you want to change something outside a function from inside the function, you may need to use a pointer.
2. Pointers have two ends. eg `int* fred`: `fred` is the pointer and `*fred` is the thing being pointed at.

3. Pointers to different types of things are different. `int* c` and `char* i` are not the same type of thing.

4. There is a special pointer written as `NULL` or `0` which points to nothing. Trying to follow a *null pointer* leads to sadness.

5. Just like other variables in C, pointer variables are not guaranteed to be initialised. That is, they do not point anywhere sensible unless you tell them to.

Exercise:

Suppose you have 3 variables `a, b, c`. Write a function which you can use to switch the values around so that `a` takes the value of `b`, `b` takes `c` and `c` takes the original value of `a`.

So, if `a=0, b=1, c=2` before the call, then after the call: `a=1, b=2, c=0`. Write a program to test your function.

Chapter 6

Arrays and Memory Allocation

So far we have only talked about single-value types, such as `int` and `char`. The simplest multi-value type is an array, which stores multiple values of another type. For example, you could have an array of 8 `int`s.

6.1 Declaring an array

Given a type `T`, you declare an array of `T` using square brackets as follows:

`T arrayVariable[numElements];`

where `numElements` is the number of elements you want in the array. Note that `numElements` must be a compile-time constant in C90, but can be a runtime value in C99[1]. This means that you generally need to know the size of an array before you declare it, which is not always possible. This issue is dealt with later in this chapter.

Below are some example array declarations. These also show how you can make arrays of arrays. Arrays can be nested arbitrarily to create multidimensional arrays of any dimension.

```c
int nums[6]; // Array of 6 ints
int twoDimenionsOfNums[3][4]; // Array of 3 arrays of 4 ints
```

6.2 Initializing an array

When you declare a variable (including arrays) in a function, C allocates memory for it but does not initialize the memory to any particular value. This means that if we just declare an array like we did above, it will contain gibberish elements. We could loop through the array and assign each element to something sensible, or we could use C's array initialization syntax:

[1]Even in C99, do not attempt to create massive arrays in this way. Stack space is more limited than heap

```c
int a[4] = { 0, 1, 5, 6 }; // All elements are initialized
int b[4] = { 4, 3 }; // Remaining elements are initialized to 0
int d[] = { 8, 7, 9, 2 }; // We can leave off the size if
                          // we initialize every element
```

6.3 Indexing an array

In C, arrays use 0-based indexing, which means that the first element in the array is at index 0. Indexing is also done using square brackets. The arrays declared above could be indexed as follows:

```c
nums[3] = 0; // 4th element of nums = 0
twoDimensionsOfNums[1][2] = 0; // 2nd row, 3rd column = 0
```

There is nothing special about the first index of `twoDimensionsOfNums` being the row and the second being the column. It is just as valid to have the first being the column and the second the row, as long as all of your code treats the array the same way.

6.4 Getting the size of an array

The C language provides no direct way to query the size of an array. It is up to you as the programmer to keep track of the array size, whether by using an additional variable or other means.

6.5 Memory layout of an array

An array in C is just a block of memory large enough to hold all of the array's values. The values are positioned sequentially in memory, from index 0 at the start of the array onwards. The array `int triplets[3]` is a block of memory the size of three `int`s. The first `int`-worth of memory is `triplets[0]`, the second is `triplets[1]`, and the third and last is `triplets[2]`.

This is all the data that the program stores for the array, which explains why C cannot provide us with the size of the array at runtime — it isn't recorded at all.

6.6 Arrays and pointers

We have just seen that items in an array are simply locations in a region of memory. We already have a tool for working directly with memory locations — pointers! It would be nice if we could use pointers to manipulate arrays and their elements.

We can find the location of the first element in an array with `&array[0]`. Indexing into `array` gives us the value of the first element, and the address-of operator `&` gives us its memory address. If we wanted to find the address of the second element we would use `&array[1]`. Unfortunately, this doesn't help us if we don't have access to the `array` variable. Given that the elements of an array are laid out in a contiguous line, and are all the same size[2], there should be a way to calculate the address of the second element from the address of the first.

```c
int array[5];
int* firstElement = &array[0];
int* secondElement = firstElement + 1; // equivalent to &array[1]
```

This is known as *pointer arithmetic*. The value of a pointer is a numerical location, so by adding to it we get a location further along in memory. Notice that we wrote `firstElement + 1` above; while memory addresses are normally thought of in bytes, C realises that `firstElement` is a pointer to an `int` and gives us a pointer 1 `int` along from it, rather than 1 byte along. When using pointer arithmetic, C will always figure out how many bytes it should add to or subtract from the pointer to move the correct number of elements along.

This is a very convenient pattern, and it turns out to not be a mere coincidence. Indexing an array and performing pointer arithmetic are the exact same thing[3]! Remember that an array is really just a block of memory — there is no better way to refer to it than by the memory address it starts at, and this is what C does. When we declare an array `T array[n]`, the variable `array` can be used as a pointer to the start of the array's memory, which is also the address of its first element. This means that where we were previously using `&array[0]` we can now just use `array`.

```c
/* These are both the same */
*(array + 1) = 1;
array[1] = 1;
```

The latter option (array indexing) is almost always preferred, as the intent is much clearer.

We have seen that we can use pointer arithmetic on arrays. Since array indexing is translated to pointer arithmetic by the compiler, we can also use array indexing syntax on pointers instead of pointer arithmetic.

6.7 Memory allocation

So far the only way we have allocated memory is by declaring a new variable. The C standard library provides us with the function `malloc`[4] to allocate regions of memory at runtime whose sizes aren't known at compile time. Any memory on the *heap* (where malloc takes its memory from) is referred to as "dynamic memory".

```c
void* malloc(size_t size)
```

`void*` is a new construct to us. Without the pointer, `void` means that a function does not return any value, or does not take arguments (depending on context). A *void pointer* has a different meaning — it is a pointer which points to memory of an unknown or undecided type. C will automatically convert a pointer to any sort of value into a void pointer.

[2] An item of a specific type has a fixed size in C
[3] Technically, some compilers may optimise them differently, but we'll ignore that here.
[4] found in `stdlib.h`

The `size` argument is the size in *bytes* of the memory you want to allocate. That bit about *bytes* is very important. So far we have not dealt with sizes in terms of bytes (C handled that for us when we were doing pointer arithmetic). We need some way of finding out how many bytes of memory a certain type requires. For this we use the `sizeof(...)` operator. `sizeof` can take a variable, an expression, or a type, and will return its size in bytes[5]. On a 64-bit machine:

```
short a = 1;
short* b = &a;
int c = 2;
long d = 3;
printf("%zu\n", sizeof(char)); // 1
printf("%zu\n", sizeof(*b)); // 2
printf("%zu\n", sizeof(c + 5)); // 4
printf("%zu\n", sizeof(d)); // 8
```

A typical memory allocation using `malloc` will look like this:

```
int* dynamic = malloc(sizeof(int));
```

You **must** follow a similar pattern whenever you allocate memory unless you really know what you're doing — your should never call `malloc` without using `sizeof` to calculate the memory size.

6.8 Dynamic arrays

We know that arrays and regions of memory can be used interchangeably thanks to pointer arithmetic, and now we know how to allocate memory on-demand at runtime. All it takes is to put the two together and we have runtime-sized arrays.

```
T* mem = malloc(sizeof(T) * numElements);
```

Here we see that mem will be a pointer to a chunk of memory large enough to hold `numElements` of type `T`. This is exactly what we needed for an array! Despite `mem` not being declared as an array, it points to a piece of memory large enough for `numElements` and can be used just like an array.

This dynamic memory allocation does come at a cost however, and that cost is that you now have to *manage* this newly allocated memory. It is up to you, the programmer, to keep track of the memory, and when you're done with it you must give it back to the system. You can give memory back by calling `free` on the original pointer given to you by malloc.

```
/* Pointer to memory for 5 ints */
int* mem = malloc(sizeof(int) * 5);
doSomething(mem); // Do something with it
free(mem); // Now we're done give it back
```

Not freeing your memory after you're done may not cause any immediate problems, but every system resource is finite and eventually you will run out. When this happens your program will probably crash, which is not desirable.

[5] *Technically* according to the standard, `sizeof` and `malloc` deal in "allocation units". But the authors have not encountered a system where the allocation unit is not bytes. Still, this is another reason to do things properly (with `sizeof`) rather than taking shortcuts

Using memory after it has been freed is also another common bug, and is one cause of the dreaded "segfault". Be mindful when you're freeing something that you have not got other variables set to the same pointer.

6.9 Multidimensional dynamic arrays

Often you will need to have more than one dimension in your array, and doing the allocation for that can sometimes be tricky. The following example shows one way of allocating a two dimensional array using `malloc`: by allocating an array of rows, and then allocating each row individually. Note how the convention for `malloc`ing properly from above is followed here!

```
int rows = 10;
int cols = 20;

int** arr = malloc(sizeof(int *) * rows);
for (int i = 0; i < rows; ++i) {
    arr[i] = malloc(sizeof(int) * cols);
}
```

Since we are allocating multiple separate pieces of memory here, it is not enough to simply free `arr`. We must be careful to free every element of `arr` (the rows) before we free `arr`.

As with the nested arrays we saw earlier in the chapter, there is nothing special about our choice of `arr` as an array of rows rather than an array of columns.

The other commonly used method for dynamically-sized multidimensional arrays is to allocate a single block of memory large enough to hold every element, and calculate the indexes required for specific elements. If we wanted an array equivalent to `arr[3][7]` we would allocate a block $3 \times 7 = 21$ items long. The first seven elements correspond to `arr[0]`, the second seven to `arr[1]`, and so on. Thus the element corresponding to `arr[0][0]` would be at index `0*7+0` and the element corresponding to `arr[2][5]` would be at index `2*7+5`.

These two methods both have tradeoffs: the former is simple to use, but has more complicated allocation and deallocation routines, while the latter is easy to allocate and deallocate but it more complicated to use.

6.10 Passing arrays to functions

In C, unlike `int` and other single value types, arrays are *not* passed into functions by value. As we saw above, an array can be represented by a pointer to the start of its memory, and this is how it is passed to functions.

The example below shows how to pass an array to a function. The important thing in this example is that you always need to pass the size of the array to the function, because as was said earlier, there is no way to know the size of the memory that the pointer points to automatically. If you know for sure the size you are passing (maybe it's always 1), then you could omit the size argument.

```
void arrayFunc(int *arr, int size) {
    ...
```

```c
}


void someFunc(void) {
    int arr[20];
    otherFunc(arr, 20);
}
```

The `arr` argument to `arrayFunc` could equivalently be written as `int arr[]`. When declaring an array argument to a function, the size can be left out as it may vary between calls to the function.

6.11 Returning an array from a function

If you allocate an array using the `[]` syntax, it gets placed on the runtime stack. When your function returns, memory on the stack is available for reuse and so could easily get overwritten. This means that you **cannot** return an array which was allocated using `[]` syntax.

If you want to create an array and return it, you **must** allocate it dynamically using `malloc`. The downside is you then need a way to also return the size of the newly allocated array. You can do this using a pointer argument to the function, as shown below.

```c
int* arrayAllocator(int *size) {
    int* array = malloc(sizeof(int) * 20);
    *size = 20; // Tell the caller how many elements!

    return array;
}

void someFunc(void) {
    int size, *array;

    /* Note the use of & to make a pointer to the size
     * variable, which arrayAllocator will update.
     */
    array = arrayAllocator(&size);
    printf("Array of size %d was allocated\n", size);
}
```

As with any use of `malloc`, we must be careful to `free` the array when we are done with it.

Chapter 7

Structs

Often when you are programming you will have a number of variables all related to the same task. In a language like python or java, you may use a class to help signify this grouping, which can also have methods which operate on instances of the class (objects). In C there is no concept of a class, however it does provide a way to group variables into a common place. This is called a struct.

7.1 Defining a Struct

The syntax for defining a struct is shown below. This defines a struct whose name is myStruct. The struct contains 3 members, an `int` called `a`, a `char` called `b` and an `int *` called `c`.

```c
struct myStruct {
    int a;
    char b;
    int *c;
};
```

7.2 Declaring and accessing members

Structs can be used in place of any other type. This means you can have a struct variable, and also a struct pointer, shown below.

```c
/* var1 is of type struct myStruct */
struct myStruct var1;
/* var2 is of type struct myStruct pointer */
struct myStruct *var2;
```

If you have a variable of type struct something, then you can access the members using the `.` operator. For example using `var1` from above we could have

```c
int x = var1.a;
char y = var1.b;
int *z = var1.c;
```

If you have a pointer to a struct, then you can access the members using the `->` operator. For example using `var2` from above

```c
int x = var2->a;
char y = var2->b;
int *z = var2->c;
```

You can also assign to struct members using the same syntax.

7.3 Passing structs to functions

Often you might be in a situation where you have a struct, and you want to call a function which will use and update some part of that struct. A mistake people often make here is that they assume structs are some sort of special type like in java/python where they are passed by reference by default. This is not the case. If you pass a struct to a function, that function will be working on a *copy* of the struct. Whatever you do to it in the function will not be reflected in the calling function's version of the struct. You can return structs from a function to allow the values to be updated, but that involves a lot of copying you probably don't need.

As a general rule, if you're passing a struct to a function it's likely you want to be passing a pointer to that stuct. As was previously mentioned in the pointers section, a pointer lets the function know where *your* copy of the stuct is and allows it to update it. The following example demonstrates how this works.

```c
void structFunc(struct myStruct *s)
{
    /* Note the use of -> because we have a pointer */
    s->a = 10;
}

void otherFunc(void)
{
    struct myStruct s;

    /* Here we use . because it's not a pointer! */
    s.a = 5;
    printf("s.a before: %d\n", s.a); // prints 5
    structFunc(&s); // make a pointer to s
    printf("s.a after: %d\n", s.a); // prints 10
}
```

7.4 Self referencing structs

An interesting thing to consider is whether a struct can contain another struct of the same type. If you think about this, a struct containing the same type creates an infinite recursion when computing the size, so it is not allowed. However a struct can contain a *pointer* to its own type. Why? Because the size of a pointer is always known by the compiler. The size of the thing it points to has no impact on the memory needed to store an address.

Consider the example below. We have a struct type called `node` which has two ints and then a pointer to another struct of the same type. You will see how this can be used to implement useful data structures in a later section.

```c
struct node {
    int a;
    int b;
    struct node *next;
};
```

7.5 The size of a struct

In the above section we mentioned the compiler computing the size of a struct. If you want to use the malloc rule that was introduced earlier, you need to be able to find the size of the struct in bytes. Fortunately this works just like every other type!

```c
struct myStruct *s = malloc(sizeof(stuct myStruct) * 1);
```

One thing you need to be careful of when working with structs is making assumptions about how big it will end up being. Consider the example below. It may be intuitive to think the total size of this struct will be 5 bytes, but that is unlikely to be true unless you ask the compiler really nicely. Modern CPU's in general require data to be aligned on certain memory address boundaries in order to load them most efficiently, and the compiler will respect this. You should always ask the compiler using `sizeof` because it is the one who ultimately decides the size that is best for the CPU your code is running on. You may find that the size of the struct below is actually 8 bytes.

```c
struct s {
    char a; // 1 byte
    int b; // 4 bytes
};
```

Chapter 8

Dealing with files

In standard C, a file is represented by a `FILE*` (file pointer). The contents of this type are unimportant to the user, and you should never dereference a file pointer.

To make things convenient, the standard IO streams already have a `FILE*` globally defined in the `stdio.h` header. Standard output is available through `stdout`, standard input is available through `stdin` and standard error is available through `stderr`.

8.1 Opening a file

The first thing you will need to do before you read/write to a file is to get a `FILE*` to it. This is done using the `fopen` function.

```
FILE* fopen(char* name, char* mode)
```

The function takes two arguments, `name` and `mode`, and returns a `FILE*`. The argument `name` is self explanatory. It is a string containing the name of the file to be opened. The `mode` argument describes what mode you want to open the file in. The following `mode` values are most typical:

- **"r"**: read mode - In this mode the file is opened for reading only, attempts to write to it will fail.
- **"w"**: write mode - In this mode the file is opened for writing only, attempts to read from it will fail (be careful, this mode removes any existing content from the file!).
- **"a"**: append mode - In this mode the file is opened for appending. This is like write mode only it does not remove the existing contents.

Opening a file can fail for many reasons, such as it not existing or not having the correct permissions. When using `fopen`, failure is reported by returning `NULL`. You should always check the return value before continuing. The following example shows how to do this.

```
/* Try and open testfile.txt for reading. */
FILE* testFile = fopen("testfile.txt", "r");
```

```c
if (testFile == NULL) {
    printf("Opening file failed!\n");
}
```

8.2 Reading from a file

If you have a `FILE*` which is opened for reading, the next thing you need to do is read from it. There are a number of functions which can be used for this, however the most useful is typically `fgetc`.

```c
int fgetc(FILE* file)
```

The function takes the `FILE*` you want to read from and returns the next value in the stream. One thing that is important to note here is that while the next value in the stream is always of type `char`, `fgetc` returns type `int`. The reason for this is that when there is nothing left to read, the function needs a way to tell you. It does this by returning the value `EOF`, however it needs an extra value (outside the range of a `char`) to do this[1]. By returning an `int` it as able to do this, however you must be careful to never directly assign the result of `fgetc` to a `char`. The example below shows how to correctly read a character using `fgetc`.

```c
int next = fgetc(stdin);
char result;
if (next == EOF) {
    printf("End of file encountered, exiting!\n");
    exit(0);
} else {
  result = (char)next;
}
```

You might wonder why this code wasn't written like this:

```c
char result;
if (feof(stdin)) {
    printf("End of file encountered, exiting!\n");
    exit(0);   // This seems drastic
} else {
    result = (char)fgetc(stdin);
}
```

C IO functions do not detect EOF until after the program has tried (and failed) to read anything[2]. So even if the program had read everything in the file, it still would not know that it was at the end of the file until the next read.

Often in your program you will want to read a line of text from the file, not just a single character. It is helpful in this instance to write a small function that does this which you can reuse later on. The following is a small example of how to do this. It first checks for either end of file or newline and returns the

[1]Remember: EOF is not actually a value in the file. A special value to watch out for like this is called a "sentinel".

[2]Yes, this is like checking for the edge of a cliff by asking if you are falling, but that is how C does things.

result (making sure to terminate the string first!), otherwise it appends to new character to the string.

This example uses a fixed size buffer, it is up to the user to work out how to make this more useful. The `realloc` function may be useful for this. It may also be useful to differentiate between an empty line ending in a newline and encountering the end of the file (perhaps by returning `NULL`).

```
char* read_line(FILE* file)
{
    char* result = malloc(sizeof(char) * 80);
    int position = 0;
    int next = 0;

    while (1) {
        next = fgetc(file);
        if (next == EOF || next == '\n') {
            result[position] = '\0';
            return result;
        } else {
            result[position++] = (char)next;
        }
    }
}
```

8.3 Writing to a file

Being able to write to a file is just as important as being able to read from a file. Fortunately this is just as easy as printing to the screen! All you have to do is replace `printf` with `fprintf`.

```
int fprintf(FILE* file, char* fmt, ...)
```

The main difference between the two is that `fprintf` allows you to specify the file that it prints to.

There is one issue that you must be aware of when writing to files, and that is *buffering*. Writing to a file is typically expensive, so the C library tries to be smart and avoid doing it whenever possible. It does this by buffering all the things you've written to the file until you call the `fflush` function or close the file (coming in the next section).

```
int fflush(FILE* file)
```

While this buffering is good for performance, it can sometimes cause you to think your code is not printing something, when in fact it is just being buffered. In general, whenever you want something to appear in the file (this could be stdout), you should `fflush` it after printing.

8.4 When you're finished with a file

Some systems impose a limit on the number of files a process/user can have open at any one time. When you're finished with a file, it is good practice to close it and free up its resources. This is done using `fclose`. Closing the file also flushes any buffers associated with it if you hadn't done it already.

```
int fclose(FILE* file)
```

After calling fclose on a `FILE*` you should no longer attempt to use it again until you reassign it using `fopen`.

8.5 Complete example

Below is a complete example you can run. It uses concepts from above to read a line from the user and write it to the file "testfileio.out". This may be a useful starting point for completing some of the exercises later on.

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

// Question for reader: is this equivalent to the standard
// function fgets()?
char* read_line(FILE* file)
{
    char* result = malloc(sizeof(char) * 80);
    int position = 0;
    int next = 0;

    while (1) {
        next = fgetc(file);
        if (next == EOF || next == '\n') {
            result[position] = '\0';
            return result;
        } else {
            result[position++] = (char)next;
        }
    }
}

int main(int argc, char** argv)
{
    FILE* outputFile;
    char* input;

    outputFile = fopen("testfileio.out", "w");

    printf("Type something: ");
    fflush(stdout); /* Appear now! */

    input = read_line(stdin);

    fprintf(outputFile, "Read \"%s\" from user!\n", input);
    free(input); // Because we used malloc
    fflush(outputFile);
    fclose(outputFile);

    printf("Done!\n");

    return 0;
}
```

Chapter 9

Strings

> This part assumes that you have a good understanding of the materials on arrays and files.

So far, we have output strings (using the %s placeholder) but apart from `argv` in main have avoided string variables and string input. Strings are stored in arrays of char, however in C you can ask how long a string is but not how big an array is. This is because strings use a special character with the numeric value 0 (written as `'\0'`) to mark the end of the string. So `strlen("") == 0` that is, the length of the empty string is zero. This also means that `("")[0] == '\0'`. The *null terminator* is not counted as part of the length of the string.

For example:

```
char single[2];
single[0] = 'A';
single[1] = '\0';
```

would make `single`, the same as `"A"`. Note the use of `'` around characters and `"` to mark strings.

On the other hand,

```
char single[2];
single[0] = 'A';
single[1] = 'B';
```

would not make `single` store `"AB"`, rather it is not storing a proper string at all[1]. Important note: while strings are stored in arrays of char, that does not mean all arrays of char store strings.

Because strings are stored in arrays, it does not make sense to compare them with `==`. Instead, we use standard string functions from `<string.h>`. Compile and run the following program:

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

[1]It is actually worse than that. The most we can say, is that we can't tell whether single stores `"AB"` or not. This is because we don't know what follows the array in memory. If there just happened to be a zero byte next then, when the program looked in `single` it would see a proper string.

```c
#include <stdio.h>

int main(int argc, char** argv) {
    char msg[10];
    msg[2] = '\0';
    msg[0] = 'h';
    msg[1] = 'w';
    printf("strlen(%s)==%d\n", msg, strlen(msg));
    printf("msg==\"hw\" -> %d, strcmp(msg,\"hw\")
        ==%d\n", (msg == "hw"), strcmp(msg, "hw"));
    return 0;
}
```

In this case, both comparing with `==` and comparing with the string compare function (`strcmp`) both give zero/`false`. However, we need to be clear about what `strcmp` returns.

```c
strcmp("B", "A") == -1 // arguments are in reverse order
strcmp("B", "B") == 0 // arguments are the same
strcmp("A", "B") == 1 // arguments are in correct order
```

That is, strcmp returns **0** if strings are the same.

To copy strings, use `strcpy`: `strcpy(buffer, "hello ");` String functions which modify a string, put the destination first. To add to an existing string, use `strcat` (string concatenate): `strcat(buffer, "world");` After executing those two statements, `buffer` will contain `"hello world"` *provided that* `buffer` *was big enough.* If buffer was declared as `char buffer[12]` or bigger then we are ok. If it was smaller (eg 2 chars), then the string functions would write off the end of the array and corrupt other variables or crash (possibly)[2].

What if you want to put numbers into a string? The simplest way is with `sprintf`:

```c
char buffer[20];
sprintf(buffer, "%d + %d == %d", 3, 5, 3 + 5);
```

Will result in `buffer` storing `"3 + 5 == 8"`. As with all standard functions which modify strings, `sprintf` will put a `'\0'` on the end of the string it makes. There is also an `sscanf` to read values out of a string instead of from a `FILE*`.

If you are making a string which will be passed outside the current function, it is better to use dynamic arrays:

```c
char* getMessage1(void) {
    char* msg = malloc(sizeof(char) * 11);
    sprintf(msg, "%d + %d == %d", 3, 5, 3 + 5);
    return msg;
}

char* getMessage2(void) {
    char msg[11];
    sprintf(msg, "%d + %d == %d", 3, 5, 3 + 5);
```

[2]This possibly is the worst part. The program *might* crash or it might look fine. The behaviour might change depending on which system you ran the program on. Undefined and unpredictable behaviour like this can be very hard to find and debug.

```
    return msg;       // returning pointer to local variable
}
```

getMessage2 is dangerous because the memory used for array msg be used for other things once the function returns. Since getMessage1 allocates it's memory is on the heap and so won't be reused until it is freed.

String summary

- Strings are stored in char arrays and must be terminated properly.
- Standard functions will take care of '\0' for you. In your own code, you need to look after it.
- C won't check that the underlying storage is big enough.
- strlen(s) — length of s.
- strcpy(d, s) — make d a copy of s
- strcat(d, s) — copy s onto the end of d.

Exercise:

1. Write a function called mystrlen which does the same job as the standard strlen function. You are not permitted to call any standard C string functions.

   Test it with the following:

   ```
   #include <stdio.h>

   // your code here

   int main(int argc, char** argv) {
       if ((mystrlen("hello") == 5) && (mystrlen("") == 0)) {
           printf("Ok\n");
       } else {
           printf("No\n");
       }
       return 0;
   }
   ```

2. Write your own version of strcpy (mystrcpy) which does the same job. You are not permitted to call any standard C string functions. Write a test program to demonstrate your function is correct.

3. Write a function char* stringadd(const char* s1, const char* s2) which takes two strings and returns a new string which contains s1 followed by s2. You are not permitted to call any standard C string functions. Write a test program to confirm that your function is correct.

4. Write a function which reads a line of any length from a file, and returns it as a string. You are not permitted to use `fgets`. Write a test program to confirm that your function is correct.

Chapter 10

Simple input

Values can be read from the user with `scanf`. This function is quite powerful and we won't deal with all its capabilities here.

A call to `scanf` function looks a bit like a `printf` call:

```c
void get_and_print(void) {
    int x;
    scanf("%d", &x);
    printf("%d\n", x);
}
```

Like `printf`, `scanf` takes a string with placeholders and some variables[1]. The difference is that the variables are pointers to the variables and we are storing into the variables rather than reading from them.

The types (and number of) placeholders must match the type of pointers given the argument list. For example:

```c
char c;
scanf("%d", &c);
```

— storing an `int` into a `char` is a bad idea. Since `int`s are bigger than `char`s additional memory "outside" `c` and possibly overwrite other variables.

Compile and run the following program:

```c
#include <stdio.h>

int main(int argc, char** argv) {
    int number;
    do {
        printf("Enter a number: ");
        scanf("%d", &number);
        printf("Number was %d\n", number);
    } while (number != 0);
    return 0;
}
```

[1]Actually any expression which evaluates to a pointer will do but let's keep it simple.

Enter a few numbers, you don't need to press enter after each number, but you will need an enter before the program takes notice of your input. Enter a zero to finish.

Note: make sure you know how to stop a running program before going on.

To see a problem with this program, run the program again and give the following input: `1 2 z6`

Stop the program. The problem is that `scanf` was told to read an integer but it saw the `z` character. Because it couldn't make an integer from that, it failed. But it didn't get rid of the `z` either, so when the loop ran again, `z` was still there. Hence the infinite loop you saw. When you use functions, you should (if possible) check to see if it worked. `scanf` will return the number of variables it managed to read correctly. So in this case, if the input succeeded, we would expect a return value of 1. Replace the `scanf` call with:

```
  if (scanf("%d", &number) == 1) {
     printf("Number was %d\n", number);
  } else {
     number = 0;     // trigger the end of the loop
  }
```

Recompile the program and try the same sequence of input.

To read other types of variables, the placeholders are generally similar to the ones used by `printf`. An important exception are floating point values. Use `%e` for `float`s and `%le` for `double`s.

There are four main things to think about when using `scanf`:

1. Use the correct placeholder for the type you wish to read.
2. Pass pointers to the variables you wish to store in.
3. Be sure to check to see if `scanf` actually succeeded before using its results.
4. Remember that bad input does not disappear automatically.

Chapter 11

Linked Queue

This part assumes that you have a good understanding of the material on dynamic arrays.

In this chapter, we'll look at using structs and dynamic memory to make a linked implementation of a queue data type. A queue is a data structure (sometimes called a FIFO[1]) where the first item added to the queue is the first one which will come out of the queue. Contrasted with a stack where the next item to come off the stack is the most recently added.

For this example, we'll make our queue store `int`s.

```
struct queue_node {
    int data;
    struct queue_node* next;
};
```

Here we have started to define a struct and put a pointer to another struct of the same type inside it. This is legal in C because as soon the compiler sees `struct queue_node`, it will accept mentions of `struct queue_node` as a type. However, while we can do the above, remember that we can't do this:

```
struct queue_node {
    int data;
    struct queue_node next;
};
```

That is, declarare a struct which contains itself (note the lack of pointer). [2]

Using this we can have no items in the queue: `struct queue_node* q=0;` one item in the queue:

```
q=malloc(sizeof(struct queue_node));
q->data=7;
```

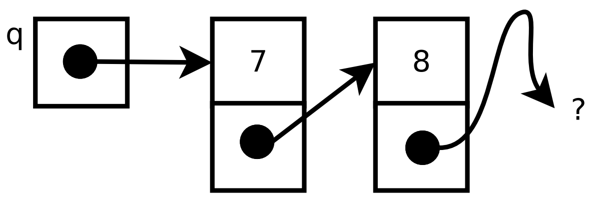
or more than multiple items:

[1]**F**irst **I**n **F**irst **O**ut

[2]The compiler won't let you actually declare a variable of a struct type until it can work out how big it is (which it doesn't know until the closing }).

```
struct queue_node* t=malloc(sizeof(struct queue_node));
t->data=8;
q->next=t;
```

all via a single pointer variable. In this case: `q->data==7` and `q->next->data==8`.



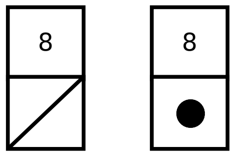
Important advice: drawing pictures is very helpful here. If you are unsure about any code involving pointers, draw a picture.

If we were sure that the queue had at least 4 items in it, we could write `q->next->next->next->data`. This brings up an important point. It is critical that the end of linked data structures be properly indicated. In this case setting the `next` member to `0`. If this "invariant" is preserved, you could output all the items in a queue like this:

```
struct queue* t=q;
while (t!=0) {
    printf("%d\n", t->data);
    t=t->next;
}
```

instruction: stop and make sure you understand exactly what the above does.

There a number of different visual conventions for indicating a null pointer and it probably doesn't matter too much which one you use provided you are consistant. Here are some examples:



Now we'll add a nicer interface around the queue_node to make it easier to use.

```
struct queue {
    struct queue_node* head;
    struct queue_node* tail;
};
```

As an optional feature, we could use a `typedef` to shorten the name.

```
typedef struct queue queue;
```

Since the queue involves memory allocation, we should have setup and teardown/cleanup functions.

```
void init_queue(queue* q);
void free_queue(queue* q);
```

The `init_queue()` function assumes that you have already has a queue struct but that it has not been initialised yet.

```
queue q;
init_queue(&q);
```

As an alternative, you could have a function which allocates a queue as well: `queue* alloc_queue()`. Other functions we will need are:

```
void add_queue(queue* q, int i);
int get_queue(queue* q);
```

Notice that this interface doesn't mention malloc even though we know it will be using it. Let's provide the implementation for these functions.

```
void init_queue(queue* q) {
    q->head=0;
    q->tail=0;
}

void free_queue(queue* q) {
    struct queue_node* t=q->head;
    while (t!=0) {
            // we need this so we can free it
        struct queue_node* temp=t; // after t has moved on
        t=t->next;
        free(temp);
    }
}
```

We will add new elements to the end (tail) of the queue and remove items from the front (head) of the queue.

```
void add_queue(queue* q, int d) {
      // don't forget the special case of an empty queue
    if (q->head==0) {
        q->tail=q->head=malloc(sizeof(struct queue_node));
        q->head->next=0;
    } else {
        struct queue_node* n=malloc(
            sizeof(struct queue_node));
        n->next=0;
        q->tail->next=n
        q->tail=n;
    }
    q->tail->dat=d;
}
```

```c

int get_queue(queue* q) {
      // check for empty queue first
   if (q->head==0) {
        // what to do here??
      return 0;
   }
   int result=head->data;
   struct queue_node* temp=q->head;
   q->head=q->head->next;
   free(temp);
     // now we need to check if we have emptied the queue
   if (q->head==0) {
      q->tail=0;
   }
   return result;
}
```

Queue summary

- Use structs to carry around all the related variables for a type.
- Write setup and cleanup functions for your struct first.
- Watch out for special cases.

Glossary

Argument a value supplied to a function to tell it what it should be doing. For example (if `sqrt` computes square roots, then in `sqrt(36)`, 36 is an argument but the same function could be called later with a different argument.

Note that arguments are distinct from the names given to the variables the arguments are stored in inside the function (see Parameters).

Array A variable, which contains multiple values intead of just one. Individual values are accessed by their `index` within the array. In most C derived languages this is indicated with brackets. eg: `scores[17]`

In C arrays, all the values have the same type and the size of the array is fixed when it is created.

Dynamic memory Memory which is allocated on the heap rather than the stack. Unlike the stack, memory on the heap is not automatically cleaned up during the program's lifetime.

Parameter A piece of information which a function needs in order to do its job. For example: the function for computing a square root could be declared as: `double sqrt(double x)`. In order to work, the function needs to know which number it should be taking the square root of. In this case, that information will be stored in a variable called `x`. Note that parameters are distinct from Arguments which are the values given in a specific call.

String A sequence of characters which can be manipulated as a unit. In C derived languages, they are usually written in code surrounded by double quotes[3]. eg: `"Welcome to C"`

[3]The widely known exception is Python, where single or double quotes can be used for strings.

Appendix A

How to attack a spec

The following is a specification for a programming task. You are not expected to actually code this. The goal instead is to look at how it could be broken into managable pieces for a sane development strategy.

Read through the following assignment spec. In Section A.3, we'll discuss it.

A.1 Introduction

Your task is to write an *ANSI-C* program (called match) which allows the user to play a game.

match will display a grid of cells, each containing a symbol. The user will nominate a cell (by entering its coordinates). If the symbol in the cell matches the symbol in any of the cells directly above, below, left or right, then the selected cell and any of its matching neighbours will be removed. This effect is also applied to neighbours of matching neighbours (and so on). As an example, suppose the grid contents (surrounded by a border) were:

```
+-----+
|ZAAAA|
|ZBBCC|
|ZWaaX|
|XXXXX|
|Zcdef|
+-----+
```

and the cell 31 was chosen (that is, Row 3 and Column 1, with the top left being 0 0). In this case, all of Row 3 would be removed as well as the last value on Row 2.

```
+-----+
|ZAAAA|
|ZBBCC|
|ZWaa.|
|.....|
|Zcdef|
+-----+
```

If a move leaves a gap in the middle of a column, the symbols above it will be moved down to fill the the gap.

```
+-----+
|.....|
|ZAAA.|
|ZBBCA|
|ZWaaC|
|Zcdef|
+-----+
```

If 4 1 was chosen, nothing would happen because the 'c' in that cell is not touching any other 'c's. If a removal leaves one or more whole columns empty, any columns to the right of the empty ones, will be moved over to fill the gap. For example, if 1 0 was chosen in the above, the result would be:

```
+-----+
|.....|
|AAA..|
|BBCA.|
|WaaC.|
|cdef.|
+-----+
```

If the grid looked like:

```
+-----+
|AAAAA|
|ZABAC|
|ZAaAX|
|XAXAX|
|ZAdAf|
+-----+
```

and 0 0 was chosen, the result would be:

```
+-----+
|.....|
|ZBC..|
|ZaX..|
|XXX..|
|Zdf..|
+-----+
```

The game ends when either all the symbols are removed, or there are no more legal moves.

A.1.1 Invocation

When run with no arguments, match should print usage instructions to stderr:

```
Usage: match height width filename
```

and exit (see the error table).

`height` and `width` must be positive integers greater than 1 and less than 1,000. `filename` must be a path to a readable file containing a valid grid (see later) with dimensions matching `height` and `width`. See the error table for what to do if these constraints are not met.

When run with valid arguments, match should read in the map file. It should then display the grid. If there are no legal moves, it should exit with the appropriate message; otherwise it should display the prompt:

```
>
```

(note, a space follows the prompt). The user should then enter either:

- the row and column they wish to remove (space separated). eg 3 2
- `w` followed immediately by a filename(or path). eg `whello` This will result in the current grid (including the border) being written to a file called "hello".

If there is no more input from the user, a message should be displayed (see error table) and the program should exit normally. If the user enters anything else, the prompt should be displayed again.

After the user has entered a legal (non-w) move, the grid should be updated and displayed again. If the game is over, display the appropriate message and exit. If not, display the prompt again and wait for another move.

A.1.2 Input file format

A valid grid file will consist of rows of symbols (and blanks) terminated by newlines (`\n`). You may use whichever characters you want for grid symbols except the following:

- `\0` — the null character. This should trigger an error in your program.
- `\n` — newlines will always be interpretted as the end of a line.
- `.` — a dot indicates a blank. You can include these in your input but the gaps will not be filled until a valid move has been completed.

A.1.3 Errors and messages

When one of the conditions in the following table happens, the program should print the error message and exit with the specified status. All error messages in this program should be sent to *standard error*. Error conditions should be tested in the order given in the table. All messages are followed by a newline.

Condition	**Exit Status**	**Message**
Program started with incorrect number of arguments	1	Usage: match height width filename
Invalid grid dimensions	2	Invalid grid dimensions
Cannot open grid file	3	Invalid grid file
Error reading grid. Eg: bad chars in input, not enough lines, short lines	4	Error reading grid contents

For a normal exit, the status will be 0 and no special message is printed.

There are a number of conditions which should cause messages to be displayed but which should not immediately terminate the program. These messages should also go to standard error.

Condition	**Action**	**Message**
Error opening file for saving grid	Prompt again	Can not open file for write
Save of grid successful	Prompt again	Save complete
End of input while waiting for user input	exit program normally	End of user input

If after a move, all the symbols have been removed, the program should display “Complete” to stdout and exit normally. If the game has reached a configuration where completion is impossible and there are no more legal moves, the program will display “No moves left” to stdout and exit normally.

A.1.4 Compilation

Your code must compile with command:
```
gcc -Wall -ansi -pedantic ass1.c -o match
```

A.2 Marks

A.2.1 Functionality criteria

- Command args — correct response to
  - incorrect number of args
  - invalid dimensions
  - invalid grid filename
- Correctly display initial grid and prompt
- Reject illegal moves on the initial grid
- Correctly process a single move:
  - Single removal from top line
  - Single removal of whole rightmost column

    - Single removal covering multiple rows/columns

- Detect no more legal moves
- Correctly save grid
- Play complete games with multiple removals

A.3 Comments

Reading the spec we can see some points:

1. There is a grid which can store letters.
2. We don't know how big the grid needs to be until the program starts running (so no fixed sized variables).
    - The grid does not change size once the game starts though.
3. There are rules about removing letters from the grid
4. There is something about files and file formats
5. There are instructions about command line arguments.
6. There are instructions about error conditions and what to do when they occur.
7. There is something about how it should compile. *Do not ignore instructions about the intended target platform. If it does not work on the client's machine, it does not work*[1].
8. Finally there are some sort of criteria.

So where to start? It seems like taking things out of the grid seems to be important so should we start there? Probably not. When working on code, you should be thinking about how you are going to make sure that what you have written is correct. That is, "how are you going to test that it complies with the spec?" Do not fall for the idea that you can just test everything at the end and it will be less work[2]. The more code you need to search, the harder finding bugs will be.

A better approach is to try to look for smaller chunks of functionality. That you can write *and test* without needing lots of other code you don't have yet. When you have completed (and tested) a new chunk of functionality, record that version somewhere (eg in version control) and then move on.

For example, look at the command line argument checking. In Section A.1.1 it says that you need to display a message when the program is run with no arguments. This is easy to write and check:

- Run with no arguments and check message.
- Run with arguments and no message.

[1]unless the client has a broken environment, but that is a separate issue.
[2]This is almost always wrong.

Section A.1.3 describes some extra errors you need to check to do with command line arguments. Some have to do with processing the contents of a file, so that might need to wait, but the rest of them should be reasonable. Again, test each possibility and save that version. The reason for storing your code after each bit works is that if you break it later[3] (adding more features, you have a working version you can go back to).

Now you should have some code you can rely on (and you can easily specify different sizes when testing other parts later).

What to do next may be a bit less obvious. You could try the file reading code (checking for errors) but you don't have anywhere to put any grid information once you've read it. In fact, the next thing to do would probably be the representation of the grid but how to test it? So you would need to work out how to to store the grid *and* how to display it. Start with creating and displaying a blank grid. Then try modifying the grid in code and displaying again to see if things appear in the correct place. This last point, can only be a temporary modification since a working program would not do this, so remove it once you are sure it works[4].

Following from there you have a number of options. Removal of letters? Saving? Loading? Game over detection? The first two are possible, but you would need to add test code to create various board states to test with, so being able to have your testing states in files and pulling them in with load would seem to be easier.

When it comes to dealing with removing letters, rather than trying to solve the whole thing in one go. Try to identify simple subcases, eg an entire column is removed and build up from there.

[3]This raises another point. Just because something worked before doesn't necessarily mean it still does. You will want to recheck earlier tests from time to time as well. This is where automated testing (sometimes mistakenly called unit testing) can be useful.

[4]Another strategy you can employ is to have one or more files which have your functions in them and then use another file which has test functions in it and call the normal functions from there.