Variables and Constants

As you have learned in the Lectures, a variable in C is a name used to refer to some location in memory and is the basic building block of any C program. By using a variable the contents of that memory location can be read from and written to. All values in C, including variables, have a type that defines what kind of value they are or in the case of a variable what kind of value can be stored.

For example, consider the expression `int x = 5;` in this case, the variable x is declared as an integer type using the keyword `int` and a value of 5 is stored at the location pointed to by x.

It is also possible to declare a variable as constant. This means that once the variable has been assigned its value it cannot be changed. To declare a constant the `const` keyword can be placed in front of a variable declaration, for example we can now make x constant by writing: `const int x = 5;`

Constant values declared in this fashion are still represented by a variable and have all of the scoping and type rules described in the 1st and 2nd C Lectures.

The following basic types are available in the C programming language.

Integer

The integer type is used to present whole numbers (0, 1, 2, ...) and can be declared by using the `int` keyword. Integers are a direct binary representation of the number and can be signed and unsigned. As mentioned in the lectures integer values can be of different sizes (size in this context means the number of bytes allocated to represent value) denoted by the `short, int` and `long` keywords. For more information on the maximum and minimum values that can be stored see the lecture slides.

Floating Point

The floating-point type is used to represent a limited precision real number, and can be declared using the `float` keyword. In this course only two flavours of floating point numbers will be considered: (single-precision) float and (doubleprecision) double. Usually a `float` will be represented using 32 bits while a `double` will be represented using 64 bits, the use of "single" and "double" precision referring to the relative size of the types in memory. This, however, is often dependent on the machine architecture being used and care should be taken when making assumptions. See additional course material on Moodle for more discussion of floating point number representation.

Character

Although the character type was invented to represent a single character of text, it can also be used to represent a single byte sized value (i.e. a number) as it is guaranteed to be exactly 8 bits in size by the C standard. Characters are declared using the `char` keyword and encodes a single character of text as an ASCII value (see ASCII character table at http://www.asciitable.com/). Character values are enclosed in the single quotes ('), e.g. `char myChar = 'x';`

Void

Essentially void means "no type" and is most likely seen in the return type of function (covered later) declaring that they do not return a value.

Operators

The C programming language provides a small set of operators for arithmetic, logical and utility purposes. Operators are built-in methods for manipulating values. You can find complete table of operators (Table 1) in the appendix. You will be familiar with most of the basic mathematical operators such as addition (+) or subtraction (-); some other operators, such as finding the remainder on division (%) use symbols you may not have encountered. Nevertheless, these operators follow the same precedence in C as they do in mathematics. Every C operator has its own precedence with respect to the other operators, governing the order in which the operations are undertaken. You are presented with the table of operators precedence in the appendix (Table 2). The operators at the top have the highest precedence. For example, given

```
int result;
result = 2 + 3 % 4 + 5;
```

the result stored in result depends on the order in which we apply the operators. The remainder operator % has the highest precedence, and so is applied first, giving the answer to (3%4), which is of course 3 (as 3 is less than 4 anyway). The expression after this operation looks like:

result += 2 + 3 + 5;
The next step is to add 2, 2 and 5 which results in the final effect: result = 10;
In this example, all of the literal values in the calculation are of type int, and so the result is also of that type. In general, if an operator acts on values of a given type, it will return the value also as that type (watch out for this when dividing integer types!). If you have two values of different (numeric) type applied to an operator, then the calculation will be carried out with all values converted to the "widest" type present; in general, this means the type with the largest range or precision. So, 3 * 4.0 will perform the calculation with 3 represented as a double, since 4.0 is already a double, and doubles can store a wider range of values than ints.
Note that the precedence may be changed by the use of the parentheses ( ) which, as in mathematics, increase the precedence of the part of the calculation they enclose so that it is evaluated before the parts outside. You can nest parentheses in order to give yourself as much fine control of precedence as you like, bearing in mind that over-parenthesizing can make it very hard to read a calculation.
In the C programming language most operators are 'binary', which means that the operator acts on the two values (immediately to the left and right of it). For instance if we wanted to add two numbers together and store the result in a variable you would write:
int x = 5 + 6;
The increment and decrement operators ++ and -- are examples of non-infix operators, applying to the single value either immediately to the left (postfix) or immediately to the right (prefix) of the operator. In this course, we will generally use *postfix* versions of the operators in examples, which read the value in a variable *and then* increment it (the prefix versions do this the opposite way around, resulting in calculations differing by 1 if you mix them up).
C also has a number of shorthand operators for carrying out common mathematical tasks. For example, increasing the value stored in a variable by a particular number might be written as: x = x + 5;
but using the shorthand += operator, can be expressed as: x += 5;
There are similar shorthand operators for performing the other basic arithmetic operations as updates to a variable.
For a complete list of all the available operators, see Table 1.

Generating Output
In order to be able to interact with a program, it's useful to be able to output values to the terminal (display). The printf() and puts() functions can be used to output characters to the screen. In order to use these functions, we must use an #include statement. We'll cover #include in more detail later in the course, but for now understand that the command
#include <stdio.h>
at the beginning of your C source file tells the compiler to use a file called "stdio.h" (short for "standard I/O") to get definitions for functions we'll be using.
The traditional first program when learning a new language is "Hello, World!", which simply prints the phrase "Hello, World!" to the screen. In C this would look like:

```
#include <stdio.h>
int main(void) {
    puts("Hello, World!");
    return 0;
}
```

In this example everything that is contained within the double quotes will be printed on the screen. We call this double quoted list of characters a *string*, the precise details of which will be covered later in the course. For the moment, just remember that when printing a sequence of characters to the screen they need to be contained in double quotes (").
The function puts() in the above example looks like a very neat and simple way of printing the output to the screen. However, it is not as simple as it first appears. Firstly, the puts() function automatically appends the '\n' character, this is called an "escape character". This denotes a new line character, it is not explicitly printed to the screen, but it starts a new line, so that the two adjacent puts() statements will never output the strings on the same line. . In addition, there are other escape characters that can be used, such as:
\t (tab)
\v (vertical tab)

\f (new page)
\b (backspace)
\r (carriage return)
For more control, we can use the printf() function. The 'f' stands for "formatted", and as this implies, the function gives us total control of how the string is printed. '\n' character is not appended to the output of the printf() function. You need to do it manually in order to start a new line. Secondly, printf() allows us to "build up" our output from the contents of variables or other values, which puts() does not allow. Consider a string: "We have %d cats". The puts("We have %d cats"); would merely output We have %d cats. In the case of printf() the %d indicates that some value has to be displayed at this point in the string. The part of a string that starts with % is called the format specifier, with the characters immediately following, in this case 'd', specifying what type to interpret the value given as when printing it (integer, in this case). So
printf("We have %d cats", 10); prints We have 10 cats.
See the table of format specifiers (Table 3) in the appendix.
Format specifiers also permit us to control the details of how a value is represented in text, beyond simply the type of value to use. For example, we can specify the precision and length with which to represent numerical values:
%6d decimal integer at least 6 characters wide
%8.2f float, at least 8 characters wide, two decimal digits
%.10s first 10 characters of a string
While printf() function introduces a greater flexibility the puts() function is an easy (and safe) way to output simple strings to the screen.

Conditional Branching (if-else and switch)
Conditional Branching is about controlling the flow-of-execution of a program or, equivalently, the conditional evaluation of statements. That is, choosing to follow one, or another, set of instructions dependant on the result of a test. For example, bank account security allows withdrawal of money only if a pin has been verified (*test 1*) and there is enough money in an account to cover the withdrawal (*test 2*). C provides two structures for this kind of decision making, via the if and switch conditional statements. These statements both evaluate tests based on C's representation of true (0) and false (>0) to make a decision on whether or not to take an action.
The simplest decision making statement in C is the if statement, which can be used in a number of different ways:

- A simple if statement, which executes code only if a condition is met.

```
if ( condition is true ) {
   execute statements here;
}
```

- An if-else statement, which executes statement1 if a condition is met or statement2 otherwise.

```
if ( condition is true) {
        statement1;
}
else { //otherwise the condition is false
         statement2;
}
```

- An if-else-if-else statement, which executes statement1 if a condition 1 is met or statement2 if condition 2 is subsequently met or statement3 if neither condition 1 or 2 is met.

```
if ( condition 1 ) { statement1;
}
else if (condition 2) {
         statement2;
}
else {
        statement3;}
```

Multiple else if blocks can be included between the starting if block and the final else, each with their own alternative test. The else at the end is only used if all of the alternative conditions before it are false.
Switch statements can be used to represent "multiple option" choices (these types of problem could also be solved by a large if-else-if-else statement but they soon become unwieldy and difficult to read). The *expression* in the switch statement below is evaluated to return an integer value, which is then compared to the values in each case below (from top to bottom). At the first place it matches that block of code is executed. If nothing matches, the default block is executed (the default case is strictly optional, but omitting it is not safe unless you have cases for all possible values of the expression). The general form of switch statement switch (expression) {

```
        case value1:
                statements1;
                break;
        case value2:
                 statements2;
                break;
        case value3:
                statements3;
                break;
        default:
                statements4;
}
```

Loops (while and for)
We often encounter the situation when programming that we want to repeat the same set of actions multiple times. For example: if we wanted to print the numbers between 1 and 5 in sequence, we *could* write a program that has 5 explicit printf statements:

```
#include <stdio.h>
int main() {
    printf("%d\n", 1);
    printf("%d\n", 2);
    printf("%d\n", 3);
    printf("%d\n", 4);
    printf("%d\n", 5);
}
```

Modifying this code to print the numbers 1 to 100 would require us to to copy the printf statement an additional 94 times, changing the number each time. Loop constructs give us a much better way to accomplish the same task, by simply repeating the core instruction (printf("%d n", ...)) with a different value each time. The most general purpose loop in the C programming language is called a for loop.
A for loop is usually used when the number of iterations (or times around the loop) is known . The basic syntax of a for loop is:

```
for (init; test; inc/dec) {
        statements; ...
}
```

Where init is a statement that sets up the initial state for the loop, test is the condition that has to hold true for the loop to continue (at the start of each cycle) and inc/dec is an increment or decrement operation to happen each time the statements inside the loop complete (at the end of each cycle).
For example, printing the numbers 1 to 5 could be done with a for loop that looks like:

```
#include <stdio.h> int main() {
                for (int i=1; i<=5; i++) {
                    printf("%d\n", i);
                }
}
```

If we now wanted to print the numbers 1 to 100 we could easily change just the conditional test, whilst leaving the rest unchanged. In fact, init, test and inc/dec, are each optional, and can be individually left out of a for statement definition; one can even write a "minimal" for loop of the form for(; ;);. This type of loop is known as an "infinite loop": as there's no condition to test, the statements inside it will continue to be executed until the program is aborted external or some internal check causes the loop to be exited. In general, care should be taken to make sure conditions in the loop can be met, and it's particularly important to try to avoid modifying the loop variable inside the loop itself.

A while loop is usually used if a number of iterations is unknown but there is some logical condition that needs to be met in order to exit the loop. The basic syntax of a while loop is:

```
while ( condition ) { statements; ...
}
```

We could write our example of printing the numbers 1 to 5 using a while loop:

```
#include <stdio.h>
int main() {
    int i;
     i = 1;
    while (i<=5) {
            printf("%d\n", i);
            i++;
   }
}
```

You can see here that the initial condition and increment are separate from the loop construct itself and are part of the logical flow of the program; the design of the for is clearly better for this, in terms of keeping all the "loop stuff" together. A better use of a while loop might be waiting for enough applications of some complex calculation to achieve a suitably small error:

```
#include <stdio.h>
int main() {
        int error;
        while (error > 0.00001) {
                //Some complex calculation
                error = some_calculation();
        }
}
```

Here, the error is not a "counter" that is external to the functionality of the statements in the loop, but is a direct result of each run through the loop (we assume in this case that the error will at some point become small enough that the loop condition will be met). While we could write a for loop to do the same thing, the simplicity of the while makes our intent clearer.

Handling User Input

The variety and complexity of programs that can be written increases immensely as soon as we are able to accept input from a user. To understand the technicalities of getting user input requires more C knowledge than you have at this point but we shall present a recipe that can be used, in order that you can write more interesting programs. Don't worry if you don't understand it all at the moment, we'll be covering the important points in the next few lectures.

The general form of the fgets() function is: fgets(name, size, stdin);

- name - is the name of the character array. The line, including the end-of-line character, is read into this array;
- size - the maximum number of characters to be read. fgets() reads until it gets a line complete with ending '\n' or it reads size - 1;
- stdin - specifies that we want to read our input from the STDIN of the process (which will often be the keyboard of the user);

In this case we use the sizeof() function so you don't have to figure out what to enter instead of "size" in the fgets() function. Whatever you type at the keyboard will then be contained in the variable "name", including the return character.

Once a line of text has been read from the input, the values contained in it can be extracted into variables. In this case the sscanf() function re-reads the string and breaks it into parts. The general format of a sscanf() statement is: sscanf(name, format, &variable1, ...)

- name - is the name of the character array to read from;
- format - is a format statement similar that used in printf to describe the type of each piece of data to be read;
- &variable - is a sequence of variables to store the values once the've been read from the buffer, with & appended to their names. The & lets the sscanf function modify the contents of the variables, for reasons explained in a later lecture.

Once the values have been read from the input and stored in variables they can be used like any other variable in our C program. In our example above we simply write them back out to the screen but we would, of course, generally do something less trivial with them.

Arrays

An array is a data structure that allows us to group together values of the same type. It is required to be implemented as a set of consecutive memory locations. A typical array declaration would look like: int data_list[3];

The declaration of an array is much like that of a scalar variable, with an additional suffix. The declaration includes the type specifier at the beginning (int in this case) which is the type of each of the elements in the array. The name of the array follows along with an additional parameter given by suffixing the name with square brackets. The integer value between the square brackets gives the number of elements in the array (in the above example it is 3).

To refer to a particular location or element in the array, we specify the name of the array and the position number of the particular element in the array. In the above declaration, for example, data_list[0] is the first element of the array, data_list[1] - 2^{nd} and data_list[2] - 3^{rd} element. In general, the i^{th} element of any array is refered to as array[i-1] because C starts counting indexes from 0. (It may help to think of the index as "the distance from the start of the array", rather than an ordinal number in itself.)

You should bear in mind that arrays are not identical in behaviour to the variables you have met so far. A common mistake is to assume that testing the equality of two arrays will test the pairwise equality of each of their elements - this is not the case! In fact, equality testing for array variables tests if they are stored in the same place... (more on this in due lectures). In order to compare two arrays, therefore, we need to compare each pair of elements in turn. This requires accessing the first element of each array, comparing them and then moving on to the second element of each array, etc. Generally, of course, you would write a loop to perform this test.

The arrays we have covered so far are 1-dimensional arrays in that they only have a single index associated with them. However, data is often multi-dimensional (e.g. matrices, spreadsheets, 3-D coordinate systems). Declaring an array with multiple number of square brackets ([]) after the name produces a multidimensional array. So, if you declare an array with the following: float coordinates[3][3][3];

A 3-Dimensional array will be initialised (this example can be visualised as a cube with the sides of length 3). In fact, you can declare as many dimensions as you need, but going higher than 4 dimensions is difficult to visualise/understand and it is not very practical. (In addition, C's implementation of multi-dimensional arrays can become unwieldy with more than a few indices.)

Strings

As discussed in the lectures, strings in C are represented as an array of characters. Since dealing with text can be tricky and strings are used frequently, C provides some additional features to make the string operations more convenient than most arrays. Firstly, C provides the concept of a *string literal* which is a sequence of

characters, enclosed in double-quotes ("). Note: characters are enclosed in single quotes (’). Because of this there are two ways of initialising a string:

```
char string[] = "String."; or
char string[] = {'S', 't', 'r', 'i', 'n', 'g', '.', '\0'};
```

Note that the end of the string is marked with an additional, special, character, the `null` character (`'\0'`). In the first declaration, the `null` character is automatically appended, but in the second case it needs to be added explicitly. Therefore, to hold a string of n characters, the array needs to be n+1 characters long to hold the extra `'\0'` which marks the end.

Finally (as with other arrays) strings can’t be compared, or manipulated like variables. C provides a set of useful functions that can be used to carry out operations such as string comparison. It’s important to use the standard string libraries as working with strings is tricky and can lead to errors in our code.

`strcmp(str1, str2)` function compares two strings for equality. If the strings are equal, function returns `0`.

`strlen(str)` function returns the length of `str` (a number of characters before a `'\0'` character is encountered).

`strcpy(str1, str2)` function copies the contents of the `str2` into the `str1`. You have to make sure that `str1` is large enough to store contents of the `str2`, otherwise, you may encounter unexpected behaviour. The C language does not include checks on if the array is large enough to store a particular string (this means that code can run faster, but has to rely on the programmer doing the right thing).

These functions are declared in the file “string.h”, so you need to include the following line at the beginning of your C code if you use any of these functions:

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

There are many more string functions in “string.h”. All of them, including the ones mentioned above, rely on the `null` (’\0’) character.

Structs

While arrays let us group multiple values of the same type together, it is often the case that there are natural groupings of variables of *different* types into a single record (e.g. phone book, address book, particle physics event, etc.). `structs` (short for structured types) are collections of one or more variables, possibly of different types, grouped together under a single name. The general form of a `struct` definition is:

```
struct structure_name {
      field-type field-name;
      field-type field-name; ...
} variable_name;
```

The above definition defines single variable (`variable_name`) as part of the initial definition of a struct type called `structure_name`. Naming your struct type is not necessary, but if you provide it with a name, you can define new variables of that type later on. To do so (assuming the above code also exists), you could write: `struct structure_name new_variable;`

It is also possible to declare an array of structures (and you can have arrays of variables as part of structure definitions): `struct structure_name p_array[10];`

You can initialise a `struct` type variable using the {} braces similarly to arrays (elements of the struct are initialised in the order they are declared in the definition):

```
struct year_book {
      char name[100];
      int age; int year;
} student1 = {"John", 23, 2014}; //in case of a new variable
struct year_book student2 = {"Anna", 20, 2014};
```

As with arrays, structs are only really useful if we can address their contained elements. For structs, this is done using the member-access operator, ’ **.** ’ . The syntax is `variable_name.field-name`. For example, to write a value to the “age” element in a year_book struct type variable: `student1.age = 22;`

So, you can also initialise a struct type variable by accessing its members one by one.
C also allows the programmer to define their own variable type names through the `typedef` command. The general form of it is: `typedef type-declaration;`
In other words, we can specify an alias for an existing type (including the struct type). For example,
```
typedef struct year_book new_type;
```
In this example the new type is called `new_type`. From now on, you can declare new struct variables in the following way: `new_type student3 = {''Jane'', 21, 2014 };`
This is quite often done in code to simplify the creation of structured data types.

Functions
In this course, you have written only short pieces of code so far. In general, as the amount of code in a project gets larger, it becomes much harder to understand it without adding additional structure (just as we structure prose into sentences, paragraphs, chapters and volumes as a written work becomes bigger to allow us to process it more easily). The next unit of "code structure" in C, after blocks, is the *function* (sometimes called a "procedure" or a "subroutine" in other languages). Functions allow to encapsulate commonly used code into compact, named, units that can be reused. C programs are typically written by combining new code the programmer writes with "pre-packaged" functions available in the C standard library, or in other libraries from other sources. You've already met many functions (`printf`,`scanf`,etc) and you've written at least one in every piece of code you've produced: the `main()` function.
The generic syntax for a function definition is:
```
return_type function_name(parameters list) {
     declarations;
     statements;
     return result;
}
```
The function definition has two parts.

1.Function declaration - this is the first line of the function definition. It specifies the return type of a function (the type of the returned value), function name and the types and names of the arguments that function expects to receive (this combination of things also defines the "function signature", which is the "type" of the function).

2.Function body - this is the block of code that is executed each time the function is called. This block contains all the declarations, statements, etc. that define how a function operates. Notice that a function body has to be enclosed in the curly braces . The function body is a lot like a selfcontained "subprogram" that is executed each time the function is called (some languages call functions "subroutines" for this reason)

It is important to realise that *functions* in C are *not* mathematical functions - while a function that takes arguments and returns a value may resemble a mathematical function externally, the philosophy behind them is entirely different. For example, it is sometimes desirable to have a function which does not return a particular value but rather does some other things internally (e.g. prints a new line character or gives some other output to the screen, changes value of its argument, etc.). Such a function might have a return type `void`. In this case the `return` statement is not necessary (unless you want to leave the function early) as there is nothing to return. We call the effects of a function which are not reflected in its return type *side effects*.
We can, of course, define a function which has both side effects and returns values. For example, the `printf` function from the C standard library returns a value equal to the number of characters it passed on to the screen and also outputs a string on the screen (which is its main feature in most uses).

Similarly to variables, functions have to be "declared" in the code before their first use - the compiler needs to know what their signature looks like so it can appropriately arrange for values to be passed to them. However, they do not have to be *defined* before they are used. Thus, you can declare the function without including the function body in a "function prototype", as long as you later provide a full function definition (declaration plus body) later on. That is:
```
  //function prototype
```

```
  return_type function_name(parameters list);

  /*
    Lots of Code here
  */

//function definition
  return_type function_name(parameters list) {
    declarations; statements;
    return result;
  }
```

You can think of this as being a bit like how you can declare a variable without giving it an initial value, with the caveat that you're not allowed to change the "value" of a function after the first definition.
Usually, the function prototype is declared at the beginning of a file (before the `main` function), providing the external interface for the function to be used, and the function is defined later in the code (possibly after the `main` function). In bigger projects that contain several files, functions are declared in separate file(s) and their prototypes are *included* before their calls.

Pointers

A *pointer* is a variable that stores a memory address of another variable (that is, it *points* at the location used to store that variable). In this sense, a variable name directly references a value and a pointer indirectly references a value. Pointers, like any other variables, must be declared before they can be used. The declaration

```
  int *ptr_count, count;
```

declares the value of type pointer to type int (i.e. `ptr_count` is a pointer to an integer value). Notice the dereference (or indirection) operator (`*`) before the `ptr_count`. In this statement it indicates that `ptr_count` is a pointer. Note that (`count`) is here declared as type `int` here, *not* pointer to int. Each pointer must be declared with the * prefixed to the name.
The next step is to initialise a pointer variable. This can be done either when a pointer is declared or in an assignment statement:

```
  \\at the declaration time: int var, *ptr_var = &var;
  \\or separate assignment statement: ptr_var = &var;
```

The &, or address operator, is a unary operator that returns the address (id of the location in memory) of its operand. The statements above assign the address of the variable `var` to pointer `ptr_var`. Variable `ptr_var` points to `var`. There is nothing stopping you from assigning the address of a variable of type A to a pointer to type B, but this should be avoided unless you know exactly what you're doing (the pointer doesn't know that the address you've given it is the wrong type, and will treat the value as the type it is intended to point at).
Outside of a variable declaration, the * operator has another use. It returns the value which is stored in the memory location with the given number. You can rephrase the declaration of a pointer: *"If I dereference `ptr_var` I will get an `int`"*. For example, the printf statements in the code `int x = 5, *ptr_x = &x; printf("%d\n", *ptr_x); printf("%d\n", x);` would output the same value. Below is the table of the pointer operators in use.

Pointers as Function Arguments

The function arguments we used in the previous section were always non-pointer variables. In C, variables are passed by value (call by value) meaning that their contents were *copied* to the variables named in the function scope. Thus, a function cannot modify the values in the variables used to call it. However, pointers can be passed as function arguments as well. This allows us to use a method referred to as "call by reference" as opposed to "call by value". The value in the pointer is a reference to a location in memory - thus, the

function gets to know the same location in memory and can directly modify the contents of it (even if the variable name referring to it is not in scope in the function).

Dangers of Pointers

Care needs to be taken when dealing with pointers. As you know, pointer contains an address of a memory location but C does not guarantee about the actual status or use of this location. In other words, the contents or use of an address may change. For example, if we use the address of some variable and then it gets out of the scope, the memory attached to it is no longer associated to it and may be used for other things. For example:

```
#include <stdio.h>
int *f(int b) {
      int a = 2 * b;
      return &a;
}
double f2(double a, int b) {
      double d = a * b;
      return d / 2;
}
int main() {
     int c = 3;
     int *p = f(c);
     f2(5.0, c);
     printf("%d\n", *p);
     return 0;
}
```

The result of the printf statement on the 17*th* line most likely will be unexpected. This happens because, the memory location pointed by the pointer `p` gets overwritten by the function `f2`. Even though, the pointer `p` points to the same location in memory, the contents of that memory has been altered as the variable `a` in the first function is not in scope in the main function. (Your compiler would usually give a warning about this, if you turned on warnings with -Wall).

Pointers and Arrays

Pointers and arrays are very closely related. You do not have to explicitly declare a pointer to an array because the bare name of an array is a pointer to the memory location where the first element of an array is stored. This is very convenient if we want to pass an array as a function argument. So now a function prototype might look like:

```
void function(int array[], int n);
```
or...
```
void function(int *array, int n);
```

The `function` call in the `main` function would look like: `function(array, n);`

We can get an address of an individual element of an array by using the `&` operator (e.g. `&array[3]`).

Command Line Arguments

Now that we know more about functions and pointers we can introduce a new prototype for the `main` function. We started with a signature that looks like `int main(void)`

However, the alternative, full, signature is

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

You will notice that this signature adds two parameters to the `main` function. This version of `main` allows us to make use of command line arguments (i.e. now you can write some arguments after the program name on the command line). The variable `argc` keeps track of how many command line arguments, including the name of the program, were passed on to the program (e.g. if your program accepts one command line argument the `argc` will be equal to 2). The `argv` is an array of strings that contains, in order, the text of each argument. The variable names `argc` and `argv` are not enforced, but they are called so by convention, and so changing them risks confusing other people reading your code.

Preprocessing

The C preprocessor is a separate program called by the gcc before the compiler. Put simply, the C preprocessor is a fancy text substitution tool. Pre-processing includes:

- Replace defined TOKENS by predefined text.
- Remove blocks of text (based on test).
- Copy in (*include*) contents of another file.

All preprocessor commands begin with the hash symbol (#). It must be the first non-blank character on the line. For readability, a preprocessor directives should not be indented if possible.
The most commonly used preprocessor directives are covered in the following subsections.
#define directive
The syntax of the `#define` statement is the following:

```
#define identifier replacement
```

The define directive instructs the preprocessor to replace `identifier` with `replacement` in the rest of the file. The `identifier` cannot contain spaces. It is a good practice to distinguish it from variable names by using uppercase letters, underscores, etc, so that you don't accidentally end up with the `replacement` text breaking things. The `replacement` does not have such restrictions: it may include spaces, and is essentially the rest of the line. It can be an expression, a statement, a block or simply anything. However, the preprocessor will not replace `identifier` in string literals. Keep in mind that the preprocessor directive extends only across a single line (you do not need to add semicolon (;) at the end of the directive). Although it is possible to extend a preprocessor directive through more than one line by using backslash ('\') character at the end of a line, in general this is not necessary. Below is an example demonstrating `#define` usage:

```
#include <stdio.h>
#define MAX 100
#define MIN 5
#define STRING "Hello, World!\n"
int main(void) {
      char arr[MAX] = {STRING};
      printf(STRING);
      printf("%.MINs", STRING);
      return 0;
}
```

`#define` is used three times in this example. The identifiers are "`MAX`", "`MIN`" and "`STRING`". "`MAX`" and "`MIN`" are replaced with `100` and `5` respectively while "`STRING`" is replaced with the string `"Hello, World!\n"`. You can use the gcc option -E to see how the code looks after preprocessing is done. In this case, it looks something like:

```
int main(void) {
     char arr[100] = {"Hello, World!\n"};
     printf("Hello, World!\n");
     printf("%.MINs", "Hello, World!\n");
     return 0;
}
```

The preprocessor gave the expected results on the lines 2 and 3. "MAX" was replaced with 100 and the string "Hello, World!\n" replaced the identifier "STRING". However, the line 4 is not what we expected. "MIN" did not get replaced with the integer 5. As it was mentioned before, the preprocessor does not replace identifiers within strings. The preprocessor does not warn about this: in this case, the resulting code would have generated a warning as `M` is not a valid conversion specifier, but this will not always be the case.
This example, as many other codes used before, begins with a statement:

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

The #include directive and header files
The `#include` is another preprocessor directive. When the preprocessor finds the `#include` directive it replaces it by the entire contents of the specified "header file". There are two ways of using `#include` statement: `#include <...>`

```
#include "..."
```

The difference between these two declarations is where the preprocessor will look for a file/ header. < > quotes make the preprocessor look for a file in system defined locations containing generally useful header files, while " " quotes refer to a file in the same directory as the source file. As such, < ... > should be used for including header files for third-party code, while " ... " should be used for including header files managed as part of the same project.
Essentially, a "header file" is just another file containing C source code, by convention ending in '.h'. Usually, if we have a C file called "code.c" which we want other files to be able to use the contents of, we create a header file called "code.h". "code.h" should contain function prototypes for all of the functions in "code.c" that we want other files to be able to use, and any other shared variables or definitions that the code relies on. Also, you should include comments sufficient enough for other programmers to understand what your functions do.

Conditionals (#ifdef...#endif)

You can construct conditional statements using C preprocessor directives. These directives allow to include or discard part of the code of a program if a certain condition is met. For example, `ifdef` directive forms a preprocessor equivalent of an if-else construct with else and endif. To put this in code:

```
#define __VAR__ "Hello, World!\n"
#ifdef __VAR__
      printf("Identifier __VAR__ has been defined!\n");
      printf(__VAR__);
#else
      printf("Identifier __VAR__ has not been defined!\n");
#endif
```

In the above example two `printf` statements on the lines 4 and 5 will be left in the code since the identifier `__VAR__` was defined before the conditional statement. The `printf` statement on the line 7 will be deleted before the compilation stage so that the program will not execute this statement. On the other hand, if `__VAR__` was not defined at the beginning, the two `printf` statements would be deleted and only the `printf` statement on the 7th line would be kept.

The opposite statement to the `#ifdef` is `#ifndef`. This is particularly useful in large projects which may contain nested header files. Suppose that two files (one.h and two.h) each include the same third file (three.h). In this case, if a file includes both one.h and two.h, it would end up including the contents of three.h twice, once from one.h and once from two.h. Depending on the contents of three.h, this is likely to prevent the resulting file from compiling (for example, by presenting multiple prototypes for the same function). The way around this problem is to build a check in three.h to see if it was already included. For example,

```
#ifndef _THREE_H_INCLUDED_
#define _THREE_H_INCLUDED_
more statements
#endif //_THREE_H_INCLUDED_
```

Having included these lines of code, the preprocessor will not include the statements within the "_THREE_H_INCLUDED_" `#ifndefs` twice. In the above situation, the preprocessor processes the file "one.h" first, which includes file three.h. "_THREE_H_INCLUDED_" has not been defined yet, so the preprocessor keeps all the statements between `#ifndef` and `#endif`, including the statement defining "_THREE_H_INCLUDED_". When two.h is processed, it also includes three.h, but "_THREE_H_INCLUDED_" has already been defined. Thus, statements between `#ifndef` and `#endif` are not included for a second time.

Compilation

The preprocessed source code is passed on to the Compiler. The Compiler does work of actually turning human readable source code into machine code. Firstly, the Compiler breaks the source code into syntactic units ("tokens") and then works out the meaning of the code, tagging every item (file scope, variables, functions and function calls) with a "symbol" (a name for referencing it). The resulting machine code, annotated with symbols, and prepended with a list of all symbols in the file, is called "object code".
Similarly to the Preprocessor you can stop compiling your process after the compilation stage using the gcc option -c. This will produce an object file with an extension '.o'. This file, being mostly machine code, is not

easily readable by a human. However, you can use the -S option to gcc instead, which produces a human readable assembly code representation of the same code. It may give you a sense of how source code is translated towards the machine code. (Other useful gcc options include -Wall and -g. -Wall turns on all warnings and -g turns on debugging.) Below is a short table of useful gcc options (there are many more, as the manpage for gcc demonstrates!).

Optimisation

It is possible to set the level of optimisation in the compilation process. To put it other way round, we can ask the Compiler to be "smarter" about the compilation.

Option	Description
-c	compile or assemble the source file, but do not link
-S	stop after the stage of compilation proper
-E	stop after the preprocessing stage
-ansi	compiles using ansi C (C89)
-std=c99	compiles using C99 standard
-Wall	enables all the warnings

Turning on an optimisation flag makes the Compiler to attempt to improve the performance and/or code size at the expense of compilation time and the ability to debug a program.

The Compiler performs optimisation based on knowledge it has about a program. Compiling multiple files at once to a single output file mode allows the Compiler to use information gained from all files when compiling each of them.

We specify optimisation with -On option to the Compiler. 'n' is an integer between 0 and 3. The extent and scope of the optimisation increases with n. Higher levels of optimisation can cause the compiled code to be larger, take more time to complete compilation, and, if you are doing especially 'clever' things, may change the results of the code slightly; they will almost always also improve the performance of the code compared to lower levels. Usually -O2 is a reasonable choice which guarantees efficiency of a code and relatively fast compilation.

Linking

Linking is the final step of the compilation process, and creates a single executable file from multiple object files. In other words, the Linker joins the object code from the Compiler, so that all the symbols are mapped to their representations. Firstly, it takes all object code files it is given (one for each C source file) and turns them into one large file with "consolidated" symbol index at the top. Before the object code is linked, function calls are simply a note of the symbol that corresponds to the function they want to call. The Linker looks up the symbol in the symbol index and replaces it with a call to the function code with the corresponding symbol. This also happens for variable names in the *file scope* in each file.

Errors about undefined, or conflicting definitions for, functions and variables are generated by the Linker (which is why they often appear a bit different to Compiler errors in their syntax).

At the stage of compilation the Compiler looks at one (source) file at a time. Therefore, if the Compiler cannot find a definition of a particular function it assumes that the definition is in another file. However, the Linker's business is to resolve all of the definitions needed for all of the symbols in all of the files it is combining - it will complain if it cannot find all of the definitions it needs. (Strictly, the Linker here performs only "static linking", where all of the symbols are matched up within the code of the executable itself. "Dynamic linking", where a symbol can be replaced with a reference to the object code that it will be found in, without that code being directly incorporated into the executable, is also fairly common, especially when using commonly available libraries which you can rely on being present on anyone's computer. The Linker also manages the configuration of dynamic links - although, of course, the link itself is resolved each time the executable runs.)

External libraries

The Linker allows us to link someone else's object code to ours as long as we know what symbols are in it. Useful code from other people is often distributed as *libraries*, essentially precompiled object code in a special form. If we use such external libraries we usually need to tell the linker about that. We do that by using the

gcc flag `-lname`. 'l' stands for library and `'name'` is the name of the library. Flags like this tell the linker to look for a (system) library called libname.a (or .so).
The most common example of an external library is math library. If you use a function like `sqrt()` you need to include

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

at the beginning of your code. But also you need to add the '-lm' flag to gcc in your terminal, on a Linux system.
You should notice that we did not need to add any flags when we compiled programs that included header files like `stdio.h`, `stdlib.h` and `string.h`. These header files include function prototypes for code in the *C Standard Library*. The C Standard Library (libc.a or libc.so) is linked to any C project by default without the need for special directive. While the C Standard Library itself is large, the linker will only link those parts of it necessary for the code to compile (which is also the reason why there are so many different header files that all refer to it).
On a standard Linux install, the header files covering the C Standard Library are found in /usr/include (along with other headers which are considered important for the system), which is one of the "system paths" that < > include directives search. The library file itself is usually found in /usr/lib/ or a subdirectory of this.
(The math library mentioned above is actually part of the C Standard Library, by definition, but for historical reasons is a separate library, as well as header, on Linux. If you're compling on OSX, the math library *is* in libc, so you wouldn't need the -lm.) Documentation on the version of the C Standard Library most commonly used by Linux, the Gnu C Standard Library, can be found here: https://www.gnu.org/software/libc/manual

File I/O

The C Standard Library provides functions for reading and writing to or from data stream. A stream is an abstract representation of any external source or destination for data, so the keyboard, the command line on your screen, and files on a disk are all examples of things you can work with as streams. Three files and their associated streams are automatically opened when program execution begins - the *standard input (stdin)*, the *standard output (stdout)* and the *standard error (stderr)*. For example, stdin stream enables a program to read data from the keyboard, and the stdout stream enables a program to print data on the screen. This is what we have used so far. But what we really want is to be able to read from and write to *files*.
We will start with the situation where we wish to write to *text* files. In this case data are written as a sequence of characters organised as lines, where each line is terminated by a newline '\n' character and each file ends with the *end-of-file marker (EOF)*.
The first thing provided by the C Standard Library is the (`typedef`ed) structure type, called `FILE`. It is a data structure which holds the information the standard I/O library needs to keep track of the file for you. So that functions can modify this info, we usually declare variables as *pointers* to `FILE` type. The name of a variable can (as for any variable) be anything you choose. You declare a variable to store a file pointer (often called *file handle*) like this:

```
FILE *fp_input;
```

It is quite common to have more than one file pointer at the same time. For example, you may want to read from two files, or read from one file and write to another file:

```
FILE *fp_input1, *fp_input2, *fp_output;
```

Opening a File

Like any variable, a file pointer is not any good until it is initialised to point to something. *Opening a file* associates a file pointer with the named file. Files are opened with the `fopen()` function that returns a file pointer for a specific external file. The `fopen()` function is defined in `stdio.h`, and it has this prototype:

```
FILE *fopen("file name", "file mode");
```

The "file name" is simply a name of a file that you want to process. If your program always works with the same file, you can define a file name as a constant at the beginning of your program. But, if a file name may change, you can obtain a file name through some external means, such as from the command line when a program is started, or you could arrange it in from the keyboard. The second argument "file mode" specifies what you want to do with a file. The table below lists 4 modes:

"r"	Open a text file for reading operations.
"w"	Open a text file for write operations.
"a"	Open a text file for append operations.
"r+"	Open text file for update (reading and writing).

File mode specification is a character string between double quotes, not a single character between single quotes.
If a file could not be opened, `fopen` returns the NULL pointer. Thus, it is a good practice to check if file was opened successfully.

Writing to a File (Text, Formatted I/O)
Once a file has been successfully opened in a mode allowing writing, you can write to it using `fputs` and `fprintf`. These functions are general versions of `puts` and `printf` functions which we met earlier. Note that `stdio.h` includes more functions on writing to a file.

```
fputs("Text to be written to a file\n", fp_output);
```

Function `fputs` requires you to specify the destination file pointer. If we use a stream pointing at the special file pointer `stdout`, then the string will be output to the screen, replicating the `puts` function, the only difference being that the `fputs` function does not add a newline. On success, a non-negative value is returned (equal to the number of characters written). If `fputs` fails to write to a file, it will return the special value EOF (end-of-file). Many of the I/O functions use this value to indicate a problem.
Just like `puts`, `fputs` does not interpret formatted strings. The following section describes `fprintf` function which allows formatted output.

```
fprintf(fp_output, "Values: %d %d %f\n", i1, i2, f1);
```

The first parameter (`fp_output`) is a file pointer to write to. If we specified the stream pointing to `stdout` we would get essentially the same function as `printf`. The next parameter is a format string which includes symbols to be replaced with values of some variables which are listed as the next parameter. The number of variables should be the same as the number of specifiers in a format string. Similarly to the `printf` function `fprintf` returns a number of characters it wrote in case of success. Otherwise it will return a negative value.

```
#include <stdio.h>
int main(void) {
    int x=3,
    y=1;
    char fl='G';
    FILE *fp_input;
    fp_input = fopen("data.txt", "w");
    fprintf(fp_input, "Flat %c%d %d Fun    St.\n", fl, x, y);
    fclose(fp_input);
    return 0;
}
```

Reading from a File
This section shows how to read data sequentially from a file. We will discuss two functions for this purpose: `fgets` and `fscanf`. As before, a file has to be opened successfully in a reading mode to be able to read its content.

```
fgets(string, sizeof(string), file handle);
```

We have already met `fgets` in the lab 2 when we wanted to get user input from the keyboard. In that case the "file handle" was the special file `stdin`. Now we want to be able to read from any file therefore we will use file pointers to specify a file to be read. We will also have to specify a string to copy the input to and the number of characters (at most) to read. The NULL pointer is returned if `fgets` fails to read a file. Otherwise, a pointer pointing to a string we provided will be returned.

Just like before the function `fgets` does not extract particular values from a string. To do that you should use function `sscanf` in the same way as it was described in the lab 2. Revise the section "Handling User Input" in that lab script keeping in mind that we now use file handles instead of the stdin.
Suppose that you have a file which contains a table with three columns. The first column has strings, the second - integers and the third - floating point numbers. Your task is to read this file line by line and extract these three values into variables. You could use the `fgets` function to read in one line into a string. Then you would have to use the `sscanf` function to extract a `string`, an `int` and a `float`. You should then repeat the same process until the end-of-file is reached. Your code could look like:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
	char buffer[100], string[30];
	int x;
	float y;
	FILE *fp_input;
	fp_input = fopen("list.txt", "r");
	if (fp_input == NULL) {
		printf("Error: file could not be opened!\n");
		exit(8);
}
	while (NULL != fgets(buffer, sizeof(buffer), fp_input)){
		sscanf(buffer, "%30s %d %f", string, &x, &y);
		printf("%30s %d %f\n", string, x, y);
	}
	fclose(fp_input);
	return 0;
}
```

The following section presents another function to read from files. **`fscanf`**
The same task could be approached in a slightly different way by the means of the `fscanf` function. Below is the function prototype

```
fscanf(file handle, "format string", &variable1,...);
```

`fscanf` takes three parameters: pointer to a file from which data is read (file handle), string of format specifiers and a list of pointers to variables that will take values indicated in the format string. The number of pointers to variables should be the same as the number of format specifiers.
`fscanf` returns a number of values it managed to assign, or EOF if it could not read the file.
The same task (presented in the previous section) could be rewritten in the following way:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
	char string[30];
	int x;
	float y;
	FILE *fp_input;
	fp_input = fopen("list.txt", "r");
	if (fp_input == NULL) {
		printf("Error: file could not be opened!\n");
		exit(8);
	}
	while (0 <= fscanf(fp_input, "%30s %d %f", string, &x, &y)) {
		printf("%30s %d %f\n", string, x, y);
	}
	fclose(fp_input);
	return 0;
}
```

As you can see, the body of the `while` loop now has only one line of code and we do not need to define an extra array. In summary, `fscanf` is to files as `sscanf` is to strings. You should be careful with the *scanf functions, compared to fgets, when parsing strings, in particular. The `%s` format specifier does not have a

limit on how many characters it will read (unlike fgets), and so you should always specify a number with it (which is interpreted as the number of characters to read, at most).

Closing Files

When you are finished with a file, you should *close* it.

`fclose` removes the connection between a file handle and an actual file. This function receives a *file pointer* as an argument. If function `fclose` is not called explicitly, the operating system will *usually* close the file when the program ends. However, it is a good practice to explicitly close each file as soon as it is known that the program will not reference the file again, as it allows the file to be locked for writing by other programs (and prevents you from mistakenly writing to it again later in the program). Another advantage of explicitly closing a file is that all pending writes to the file will be written to disk, ensuring that the file is in the state you expect it to be immediately.

If you want to force all writes you have asked so far to complete without closing a file, you can use function `fflush`. It takes one parameter, the file pointer to flush.

Block ("Direct") File I/O

While formatted stream I/O is useful, in that it produces files that are easy to read and edit by hand by humans, it also has downsides. The process of converting numeric values to text in order to write them is not space efficient (if an `int` is 32bits wide, and can store values with up to 9 digits, we use up 9 chars to represent it - potentially 72 bits of space in the file), and can also lead to precision loss for floating point formats. There is also, inevitably, a small cost in terms of the speed at which we can write, as every value needs to be converted before writing.

The alternative form of file stream I/O is "Direct", or "Block" I/O. In this, we directly write the values in memory to a file, without converting them to a text format beforehand. While the output file is not necessarily human readable anymore (certainly for numeric values), we both save space and also guarantee that we are recording the exact value that was in memory.

When performing direct I/O, we should be careful to open our file with the "binary" flag added to the file mode we specify; adding a `b` to the file mode we would use for formatted I/O. (This explicitly disables any special measures that the operating system may apply for text-based files - on POSIX systems, there are no such measures to disable, but on Windows systems (for example) text files are treated specially.) That is:

`"rb"`	Open a text file for reading operations.
`"wb"`	Open a text file for write operations.
`"ab"`	Open a text file for append operations.
`"rb+"`	Open text file for update (reading and writing).

Once you've opened a file appropriately, the two functions `fread` and `fwrite` let you read and write data between the file and memory.

fread

As fread accesses memory directly, it needs to be told which chunk of memory to write to. Its prototype is:

```
size_t fread (void * data, size_t size, size_t count, FILE * stream)
```

where `size_t` is an integer large enough to hold the maximum amount of memory supported on your system. (An `unsigned long` is almost always equivalent to it.) `data` is a pointer to the start of the section of memory you want to copy data to, `size` is the "unit size" of the data (so, say `sizeof(int)` if we're writing `int`s), and count is how many of those units you want to fill up. `stream` is just the file pointer to copy the data from. As the structure of the function suggests, `fread` is really designed to fill up an array with data, but we can also read individual values:

```
int i;
fread (&i, sizeof(i), 1, myfile);
```

fwrite

As fwrite is just doing the same thing as fread, but in the opposite direction, it looks very similar as a function:

```
size_t fwrite (void * data, size_t size, size_t count, FILE * stream)
```

All of the parameters mean the same thing, except that fwrite is copying *from* the memory starting at `data` *to* the file pointer `stream`.