The basics

Any program is made up of information about data and operations to perform on the data. Our C program will be made up of:

- *global variable declarations* describing the data which the whole program can use;
- *functions* describing operations on data, which can include declarations for variables only that function can use.

Comments

In addition, the program can (and should!) include comments. Anything between `/*` and `*/` is a comment, as is anything after `//` on a line. See the function example below.

Variable declarations and types

Variable declarations tell C to allocate memory for data of a particular type, with a particular name. A variable declaration consists of the *type* of the variable followed by its name (arrays are a bit different, but we'll come to those). Here are examples of variable declarations for each type. The last "word" is the name you choose, anything before that is the type:

`short a;`	integer (whole number) from -32767 to 32768
`unsigned short c;`	integer from 0 to 65535
`long someCount;`	integer from 0 to -2147483648 to 2147483647
`unsigned long ct;`	integer from 0 to 4294967295
`int c;`	same as `short`[1]
`unsigned int c;`	same as `unsigned short`[1]
`char b;`	a character of text
`bool isNoseLong;`	either `true` or `false`[2]
`float wibble;`	a floating-point number (one which can have a fractional part, like 2.5)

[1] `short` is the same as `int` only on the Arduino — if you're writing code on a PC, `int` is the same as `long`.

[2] `bool` isn't actually a C type — it's a C++ type. You're actually programming in C++ at the moment, but without using the C++ advanced facilities like classes and objects.

You can declare several variables of the same type in a single declaration by separating the names with commas:

```
int a,b,fishCount;     // declare three integers
```

It's also possible to *initialize* a variable as you declare it — as well as saying that the variable exists, you can give it a starting value:

```
int q = 60; // give q the initial value of 60
int a=3, b=5; // set a to 3 and b to 5
```

Functions

Functions are blocks of *statements* — actual instructions for the computer to do something. Functions can have data *passed in* to them by another function, and can *return* values to those functions which *call* them. Each function has a name, and a list of *parameters* (sometimes called "arguments") which are the values passed in. These become variables inside the function.

Scope of variables

Functions can also have their own variables — *local variables* — which only exist inside the function while it's running, and can't be used anywhere else. In fact, anything between curly brackets `{}` can only be used within those curly brackets. Where a variable can be used is called its *scope.*

Functions have the form:

```
returnType functionName(...parameters...) {
    .. local variable declarations ..
    .. statements ..
}
```

Here `returnType` is the type of the value which must be returned to the calling function, and `functionName` is the name of the function. If a function has the special return type `void` it doesn't return anything at all.

Here's an example of a function named `doCalc()` which takes some parameters, has a local variable, and returns a value. It's called by another function called `process()`:

```
/* function doCalc returns an integer, and takes two
   parameters - both integers, called x and y */

int doCalc(int x,int y) {

    int a; // local variable: an integer called "a"
    a = x*y; // multiply x by y and store in a
    return a-20; // return a minus 20
}

// a function called "process" which takes no
// parameters and returns nothing.

void process() {
    // declare 3 local variables
    int a,b,result;

    // call a function defined elsewhere to get values
    // for a and b
    a = getInput("first number: ");
    b = getInput("second number: ");

    // call doCalc to get the result
    result = doCalc(a,b);
}
```

Returning values

If the function has a return type which isn't `void`, it must return a value to the function which called it. To do this, use the `return` statement as shown in the example above, in `doCalc()`. Naturally, the type of this value must be the same as that given in the function's definition.

Maths etc.: expressions and operators

Whenever you assign to a variable or pass a value to a function, you can use an *expression.* These are combinations of

- **literals**: actual values like 5, "this string", 6.4 or `true`
- **variables** like `count` and `x` (provided they are local to the current block or are global)
- **calls to functions** such as `myFunction(foo,bar)`
- **operators** which act on pairs of values, like the "+" in `count+6`, or on single values like the "-" in "-20"
- **brackets** so you can control how the operators work: `(6+a)*b` is different from `6+(a*b)`

The most common binary (taking two values) operators are:

`+`	addition
`-`	subtraction
`/`	division
`+`	multiplication
`%`	remainder, or *modulus*

The most common unary (taking one value) operator is "-", which negates a value when placed in front, as in "-20" or `-(b/d)`.

Increment and decrement

These are special operators which change a variable and return either its old or new value:

`x++`	add 1 to `x` and return the *old* value
`x++`	add 1 to `x` and return the *new* value
`x--`	subtract 1 from `x` and return the *old* value
`--x`	subtract 1 from `x` and return the *new* value

Think of `x++` as "get variable, then add 1" and `++x` as "add 1, then get variable".

If statements

You can make things happen only sometimes in your functions by using `if...` statements. These have the form

```
if(condition) {
    statements...
}
```

For example

```
if(x>20) {
    y=20;
    print(x+y);
}
```

The condition is an expression which produces a *boolean* result (either true or false), or an integer — in which case the statements inside the block will happen if the result is not zero. We have special operators which return boolean values:

`==`	equals (do not confuse with `=`, which means "assign variable" !)
`!=`	not equal
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

There are also useful operators which take two boolean values:

`&&`	and — true if both sides are true
`||`	or — true if either side, or both, is true

Finally, the *not* operator negates in a logical sense, turning its expression into its logical negative. Here's an example:

```
if( (a<10) && (b>20) && !isRunning()){
    ...
}
```

can be read as "if a<10 and b>20 and the `isRunning()` function does not return true, do the statements in the block".

Single conditional statements

It's also possible to run a single statement conditionally — in this case we miss out the curly brackets:

```
if(x<20)
    print("X is small!");
```

Switch statements

Switch statements are used to run different bits of code depending on an expression which can have several different values — usually integers. They have the form:

```
switch(expression) {
    case value:
        statements...
    case value:
        statements...
    case value:
        statements...
    default:
        statements...
}
```

On meeting a switch, the system will jump to the appropriate `case`. **Important:** if we don't stop it, at the end of the case the system will keep going, *falling through* into the next case. To avoid this, we usually put a `break` statement at the end of each case, which will jump to after the curly bracket at the end of the entire `switch`.

The `default` case is a special case for any values for which we don't have an explicit case.

For example, if the expression `a/3` could have four different values (0,1,2,3) we could write

```
switch(a/3){
    case 0:
        print("The value is zero");
        break;
    case 1:
        print("The value is one");
        break;
    case 2: // fall through into 3's case
    case 3:
        print("The value is two or three");
        break;
    default:
        print("Out of range");
}
```

Loops

while loops

The first type of loop is the plain `while` loop. This runs its block until the expression is false — and the expression is always checked *at the start of the loop.* For example:

```
x=1;
while(x<=10){
    print(x);
    x++;
}
```

might print the numbers from 1 to 10, if we had a suitable `print()` function.

do..while loops

Then there is the `do..while` loop, which is similar but does the test *at the end:*

```
x=1;
do {
    print(x++); // see how x++ works
} while(x<=10);
```

will print the numbers from 1 to 10 again — but sometimes it can be important where the test goes!

for loops

The `for` loop is a kind of shorthand for a `while` loop. Instead of writing

```
start statement
while(condition){
    do something
    variable change
}
```

we can write

```
for(start statement;condition;variable change){
    do something
}
```

A typical use is counting through a sequence, so taking our earlier example

```
x=1;
while(x<=10){
    print(x);
    x++;
}
```

This can be written using a `for` loop as

```
for(i=1;i<=10;i++){
    print(x);
}
```

It's possible to declare the variable in the start statement. In this case, the variable is local to the for loop itself and cannot be used outside it:

```
for(int i=0;i<10;i++){
    ...
}
```

Leaving a loop early: break

It's possible to jump out of a loop early by using a `break` statement:

```
while(true) { // loops forever
    if(isFinished())
        break; // but this will end it
}
```

Arrays

Arrays are blocks of memory holding multiple variables of one type. To define an array variable, put the size of the array in square brackets after the name of the variable. For example:

```
int myArray[32];
```

defines an array of 32 integers. Once defined, the individual elements can be used like ordinary variables, but with the *index* of the slot you want to use in square brackets after the name:

```
int myArray[32];
for(int i=0;i<32;i++){
    myArray[i]=i*i;
}
```

will fill the array with the squares of the numbers 0..31. Note that *array indices start at zero* — not one.

It's also possible to have *multidimensional* arrays. Here's a 4x4 array:

```
int grid[4][4];
```

You can read this is being "an array of 4 arrays of 4 integers" — in other words, a 4x4 array of integers. This can be used by putting each index in its own square brackets:

```
grid[0][1] = grid[2][3] + grid[x][y];
```

Initialising arrays

The initial values of the slots in an array are undefined, but you can give them when you declare the array by putting them in curly brackets:

```
int array[5]={1,2,3,4,5};
```

will define an array of 5 integers, giving them the values 1 to 5. This works for multidimensional arrays too:

```
int boxes[2][2] = { {1,2},{3,4} };
```

Here, `box[0][0]` gets the value 1, `box[0][1]` gets the value 2, `box[1][0]` gets the value 3 and `box[1][1]` gets the value 4.

The preprocessor

Before your code is turned into machine code, the compiler runs it through a separate program called the *preprocessor*. This looks for lines which start with a hash symbol `#`, scans them for special commands, and modifies the text before it's compiled.

#define

One of the most common commands is `#define`. This will define a *preprocessor symbol*. After this, when the symbol is found anywhere in the program it will be replaced with the text it was defined as (provided the symbol is not part of another word, and is not inside a string). For example, if we had the line

```
#define PI 3.1415927
```

at the start of the program, we could write something like

```
float circ = rad*2.0*PI;
```

and it would automatically become

```
float circ = rad*2.0*3.1415927;
```

This is extremely useful for defining *constants* — values which look like variables but don't change.

#ifdef

We can also use `#define` to do *conditional compilation* — compiling parts of our program optionally. If we had some code like this:

```
x=y*10;
#ifdef DEBUG
print(x);
#endif
```

The part between `#ifdef` and `#endif` would only become part of our program if we had

```
#define DEBUG 1
```

somewhere before it. We could define `DEBUG` as anything, it would still work — what's important is that it has been defined.

#include

This is possibly the most common preprocessor command. It simply includes another file into the text of the program. You're currently using it to include the definitions of the functions for various Arduino libraries.

The Arduino

Programming on the Arduino requires you to write two functions: `loop()` and `setup()`. Neither return any values nor take any parameters. The `setup()` function runs once when the Arduino is switched on or reset. After that, the `loop()` function runs over and over again.

However, any local variables defined in `loop()` will be destroyed and recreated each time it runs — `loop()` will not remember anything from the last time it ran!

The serial port

This allows you to talk to your PC. Set up the serial port by putting

```
Serial.begin(9600);
```

inside your `setup()` function. This tells the serial port to start at a speed of 9600 bits per second. Once this is done, you can send text to the PC with

```
Serial.print(x);
```

where x is a variable, string or number. More serial port functions can be found in your lecture notes, such as functions to read data from the PC.

Timing

- **Stop the program for a time** using the `delay()` function, which takes a time in milliseconds as its only parameter.
- **Get the time** in milliseconds since the Arduino was switched on or reset using the `millis()` function.

```
void setup(){
    Serial.begin(9600);
}

void loop(){
    delay(100); // wait 1/10 second
    Serial.println(millis()); // print time since start
}
```

Reading and writing digital pins

(Skip this if you're not interested in using the pins directly.) First, add calls to the `pinMode()` function to your `setup()`. to tell the Arduino whether the pin is input or output. Then use `digitalWrite()` to write `HIGH` or `LOW` to the pin, and `digitalRead()` to read `HIGH` or `LOW` from the pin. Here's an example of an entire Arduino program to flash the LED on and off, stopping when it reads a digital HIGH from pin 5:

```
void setup(){
    pinMode(13,OUTPUT); // the output is the LED, pin 13
    pinMode(5,INPUT);   // we're reading pin 5
}

void loop(){
    if(digitalRead(5)==HIGH){
        // if pin 5 goes high, just wait
        // forever by looping a 1 second
        // delay
        while(1){
            delay(1000);
        }
    }

    digitalWrite(5,HIGH);
    delay(200);
    digitalWrite(5,LOW);
    delay(200);
}
```

The LED shield

To use the LED shield, you need to install the AberLED library and add

```
#include <AberLED.h>
```

at the start of your code, and call `AberLED.begin()` in your `setup()`.

Generally, your program will repeatedly draw images to the display. Each time round your loop, you should:

- clear the display
- draw your image
- swap the display to show it on the LED

I recommend doing all this in a separate function you define — I typically call it `render()`.

- **Clear the display** by calling `AberLED.clear()`
- **Set a pixel** by calling `AberLED.set(x,y,colour)` where colour is `RED`, `GREEN`, `YELLOW`, or `BLACK`
- **Swap the display buffers** after drawing by calling `AberLED.swap()`.

Here's an example which bounces a dot across the screen:

```
#include <AberLED.h>
void setup(){
    AberLED.begin();
}

int x=0; // position
int dx=1; // direction

void render(){
    AberLED.clear(); // clear the screen
    AberLED.set(x,4,GREEN); // draw the dot
    AberLED.swap(); // display the image we made
}

void loop(){
    x = x+dx; // move the dot

    if(x==7 || x==0) // change direction
        dx = -dx;    // if we hit the edge

    render();   // draw everything
    delay(100); // wait a bit
}
```

Input

- `getButton(n)` reads the *current state* of button n — nonzero (i.e. true) if the button is pressed;
- `getButtonDown(n)` returns nonzero (true) if the button was pressed down *sometime between the last two swap() calls.*

`getButtonDown()` is useful if you don't want a piece of code to keep repeating once the button has been pressed — you only want it to happen when it is first pressed.

Make sure you only call `AberLED.swap()` once in your loop — `getButtonDown()` will stop working properly if you don't call it, or call it several times.