The Basics of Coding

Every statement (usually a statement is a line of code) must finish with a semicolon `;` - if this is missed, the compiler will complain about the **next** line, not the offending one.

Code within a pair of braces is often called a block.

All parentheses `()` braces `{}` and double or single quotes `“”` `‘’` must be **paired.** Not closing any of these confuses the compiler.

All parentheses `()` braces `{}` and double or single quotes `“”` `‘’` must be **nested**.

```
if (y > 0){
        if (x > 2){
                //some code statements here
        } //this is the end of the second if, which is completely nested inside the first if's braces
} //this is the end of the first if
```

`(“ my text “)` *not* `(“my text)”` the former is nested, with quotes inside parentheses, the latter is not

// the double slash precedes a comment – a note to the programmer/reader ignored by the compiler

the preceding code on the line is not part of the comment *// the comment only lasts until the end of that line*

/ *this is a block comment*
that might go on
for more than one line
**/*

Turning on line numbering, if that is an available option, helps with locating compiler problems.

Having an environment which automatically indents code is a big advantage. Code should be indented according to bracing level, as is the if example shown above. It is then easy for the eye to find the beginning and end of a block.

Commenting the end brace of a block with a description of what it is ending can be very helpful for choosing where to insert other code later on. When confronted with a series of end-braces like this:

```
                        }
                }
        }
}
```

it is very easy to pick the wrong one. Better:

```
                        } //end inner if
                } //end outer if
        } //end loop
} //end function
```

Storing data

Variables are names referring to stored pieces of data. The value of the data may be changed while the name remains the same.
A variable is generally of a particular data type. The common ones for beginner programmers are

int	integer
double / float	real numbers
char / character	individual character e.g. `a 3 # )` etc.
String / string	a series of characters, usually enclosed in double quotes `""`
boolean / bool	true or false value

The first time you use a variable you need to declare it – specify what type of data it will hold. Behind the scenes, this reserves the right amount of memory space for storing it.

```
int x;
String mySentence;
```

Declaring a number (`int` or `double`) variable generally gives it a default value of `0`.

Assignment

Variables get their values by assignment. The equals symbol `=` is used for assignment. In programming, do not think of equals as `"is equal to"`, but rather `"gets the value of"`.

`int x = 3;` declares a variable called x and set its value to 3.
`x = 5;` x gets the value 5
`x = x + 3;` x gets the value of what x held to start with (`5`, if these statements were performed in order) plus 3

Division happens with the / sign (operator).

```
x = x / 2;
```

Integer division loses the remainder completely.

```
x = 7 / 2; // would give x the value 3
```

If you want to keep the remainder, use the mod % operator.

```
y = 7 % 2; // would give y the value 1
```

Multiplication uses the * symbol

Expressions

An expression is any piece of program code which, when its execution terminates, returns a value. In most programming languages, expressions consist of constants, variables, operators, functions, and parentheses. The operators and functions may be built-in or user defined.
(from http://dictionary.reference.com/browse/expression)

```
(3 + 2) * 5 / 2  // is an expression
int x = (3 + 2) * 5 / 2;  // is a statement
```

In many languages, mathematical statements can be shortened.

`x = x + 1` can be shortened to `x += 1`
`y = y / 4` can be shortened to `y /= 4`

Sequence, selection and iteration

Sequence

Statements in a block are performed in order (sequence).

Often, you will have statements that should only be performed sometimes, if a condition or conditions are right. Choosing the condition is known as selection, and is commonly done with `if` and `if else` statements. Sometimes you will have statements that you want to be performed repeatedly for a known number of times, or until a condition is met. Loops are generally used for repetition; each repetition is called an iteration. There are several different kinds of loop which may be used, and choosing the best one is a bit of an art.

Selection

if

An if statement is performed when its condition is true. A condition is any legally formed expression which may be either true or false

```
if (condition){
   //do something if the condition is true
}
```

Constructing a condition:

Assuming `score` is a variable of type `int`:
A condition might be

`(score < 5)`	score is less than 5
`(score >= 5)`	score is greater than or equal to 5
`(score == 5)`	score is equal to 5
`(score != 5)`	score is not equal to 5
`(score > 5 && score < 50)`	score is less than 5 and greater than 50
`(score < 0 || score > 100)`	score is less than 0 or score is greater than 100

else

`if else`

```
if (condition){
   //do something if the condition is true
} else {
   //do something else if the condition is not true
}
```

`if else if`

```
if (condition1){
    //do something if condition1 is true
} else if (condition2){
    //do something else if condition2 is true and condition1 is not true
} else {
    //do something if neither condition1 nor condition2 is true
}
```

`nested if`

```
if (condition1){
    //do something if the condition1 is true
    if (condition2){
       //do something more if condition 2 is true and condition1 is true
    }
} else {
    //do something if condition1 is not true. condition2 is irrelevant
}
```

Iteration

The most common forms of loop are **do..while**, **while** and **for**:

```
do{

   //statements to be repeated go here

}while(condition);  //the condition is checked after the loop, so the loop is
                                                                    // performed at least once



while(condition){  //the condition is checked before the loop, so the loop may not be
                                                                          // performed at all

   //statements to be repeated go here

}


for (start condition; stop condition; increment){

   //statements to be repeated go here
}
```

examples of each:

do

```
int sum = 0;   //sum is required outside the loop block, so declared outside
int increment =  10;
do{
   sum = sum + increment;
}while(sum < 1000);
```

while

```
int doubled = 1;    //doubled is required outside the loop block, so declared outside
while( doubled < 1000){
   doubled *= 2;
}
```

for

```
int sum = 0;    //must be declared outside the block or would start
                // again from 0 with each iteration
int increment =  10;
for (int count = 0; count < 50; count++){
   sum += increment;
}
```

Common problems –

putting a semicolon after the condition
not ensuring the loop will always finish sometime.
Declaring loop finishing data or other accumulating variables inside the loop block.

Functions / methods / subroutines / modules

A series of statements which performs a particular purpose can be put in a block. This block of code can be given a name. The block should not be nested inside any other block of code. Some languages ask that they come first, before the main task / method / function.
This enables the block of code to be used repeatedly without being rewritten. All you need to do is call its name whenever you want it.

If a function / method is `void`, the code within it is performed (run executed) when the function is called (has its name mentioned in a statement).

```
//this is a function
void myFunction(){
   //code statement/s in here
}
```

example of function call

```
if(score > 50){
   myFunction();  //all the statements within myFunction will be performed before any other statement
   //other statements here
}
```

Sending data to a function

Where the function is declared, inside the parentheses might be a list of the data type and name for the incoming data.

```
void function_1(int a, int b){  //a and b are incoming data
   int c = a + b;
}
void function_2(string s){
   //code statements here might print s, or count the number of spaces in it, or check that it isn't too short . . .
}
```

and to call the function:
e.g.

```
function_1 (3, 4);
function_1(102, 3015);

int x = 30;
function_1 (x, 3 * x));

function_2("a sentence on the move");
```

The list of data with its type specified in the function itself is called a parameter list. The function can not be called without all of these items being sent to it, so the call to the function must match the parameter list with the appropriate number and type of data.

For the functions declared above:

`function_1` expects to be sent 2 integers, so any call to it must have 2 integers.
`function_2` expects to be sent 1 string, so any call to it must have 1 string.
`myFunction` expects to be sent nothing, so any call to it must have empty parentheses.

Getting data from a function

A function may return a value (just one value). In this case, the type of the value returned must be specified instead of the word void. The last statement in a return method should be a return statement which returns a piece of data of the type specified.

```
int myFunction(){
  int x = 2 + 2 ;
  return x;
}
```

OR, doing the same job:

```
int myFunction(){
  return 2 + 2;
}
```

When calling a return method, think of the value being returned replacing the call to the function in the code.
When calling a return method don't use

```
myFunction();    // as that will have no effect
```

but instead use

```
x = myFunction();  // as this will store the value returned by myFunction in the variable x
```

or

```
if(myFunction() > 5) {
    //which will calculate the result of myFunction where it is needed for the condition,
    //but does not store the result
```

Many languages have libraries of predefined functions which can be used.

Scope

In most modern languages, **where** a variable is declared defines where it can be used. If it is declared inside a block of code it will not be able to be used outside of that block. If it is declared outside of any block, customarily at the top of the program, it is available throughout the whole program.

```
int a;
codeblock3{    //note codeblock is a function header or an if or loop
      int z;
      codeblock1{
            int x;
      }
      codeblock2{
            int y;
      }
}
codeblock4{

}
```

The variable `z` can be used in `codeblock1`, `codeblock2` and `codeblock3`.
The variable `x` can be used just in `codeblock1`.
The variable `y` can be used just in `codeblock2`.
The variable `a` can be used in `codeblock1`, `codeblock2`, `codeblock3` and `codeblock4`.

Common error: If you declare a variable twice (use the same name in different places), you need to be careful to do it for the right reasons. The code below compiles, but may not give the programmer the right answer.

```
int z;  // this z will be set to the default value of 0
codeblock3{
      int z = 3;  // this z is only available between the braces
      // here, any access to z will find the closest z, and return 3.
}
// here, any access to z will return 0
```

FIXED:

```
int z;  // this z will be set to the default value of 0
codeblock3{
      z = 3;  // this sets a new value for z
      // here, any access to z will find the closest z, and return 3.
}
// here, any access to z will return 3
```

There are times when it is OK to use the same name for 2 variables e.g. if two functions both require a counter of some sort, but the counter is completely contained within the function.

```
codeblock1{
      int count;  //e.g. to count the number of times the user has played a game
      do{  //in a loop
            playGame();
            count++;      // the count is not needed outside of the block
      while (count < 5) //repeat until count reaches some number
}
codeblock2{
      int count;   //e.g. counts the number of items read from a data source in order to
                   //calculate an average – the average may be required outside of the block,
                   // but the count isn't
}
```

Arrays

It is common for a series of related data to be stored together. It can then be processed together by a loop. An array is an example of a data structure, and there are others.

An array is a series of data of the same data type.

An array needs to be declared with a special syntax e.g. `int myArray[];`

The individual pieces of data in an array are called elements. The elements have a position number, or index. The data starts at index number 0. There is always some way of retrieving the length of the array e.g. `ArrayLen(myArray)`

myArray

index	0	1	2	3	4	5	6	7	8	9
element stored	6	3	8	10	9	3	3	7	3	1

To set the value of 1 array element, use its index in square brackets.

```
myArray[3] = 9; // stores the number 9 in  myArray index position 3 (will overwrite 10)
```

```
myArray[0] = 1; // changes the first element of the array from 6 to 1
```

To get an elements value out of the array, once again use its index in square brackets:

```
int value1 = myArray[2];
```

```
if (myArray[0] < 5){
```

Looping through an array

The power of data structures lies in the ability to loop through them sequentially.

```
for (int i = 0; i < ArrayLen(myArray); i++){
      myArray[i] += 10; //adds 10 to every element of the array, and stores it
}
```