Basic C++

Introduction

This workshop will cover the following topics:

- Basic Structure of a C++ Program
- Comment
- Variable
- Operator
- Control Structure
  - Sequence
  - Selection
  - Repetition
- Functions

Basic Structure of a C++ Program

First, let's see a simple program:

```
/* Author: Zhuming Chen
 * Date: 02/09/2002
 * Description:
     This is a simple C++ program named simpleProgram.cpp, which prompts user
     for two integers, adds them together, then outputs the sum
 * Input: a, b – two integers
 * Output: c – sum of a and b
*/

 #include <iostream>

 int main() {

    // Declare variables
    int a = 0;
    int b = 0;
    int c = 0;

    // Prompt user for two integers
    cout << "Please enter one integer: ";
    cin >> a;
    cout << "Please enter another integer: ";
    cin >> b;

   // Add the two integers together
   c = a + b;

   // Output the result
   cout << "The sum of " << a <<  " and " << b << " is " << c << endl;

   // Indicate that the program ended successfully
   return 0;
 }
```

In this program, we can see that the basic elements for a C++ program are:

`#include <filename>`

Sentences that begin with a pound sign are directives for the preprocessor. They are not executable code lines but indications for the compiler. The **`#include`** directive causes a copy of a specified file named **`filename`** to be included in your source code. It normally used to include standard header files such as `iostream`, `iomanip`. A header file usually contains declarations and definitions, such as function prototypes, classes, and structures. All preprocessor directives begin with #. In this case, the sentence `#include <iostream>` tells the compiler's preprocessor to include the `iostream` standard header file, which includes the declarations of the basic standard input-output library in C++.

`int main()`

This `main()` function is the point by where all C++ programs begin their execution. It is independent from whether it's at the beginning, at the end of or in the middle of the code – its content is always the first to be executed when a program starts. It is essential that all C++ programs have a `main()` function

`cin/cout`

`cin` is the standard input stream in C++ (usually the keyboard). It reads info from a device such as keyboard, disk, then stores the info into memory. `cout` is the standard output stream in C++ (usually the screen). It transfers info from memory to some device, like screen, printer. C++ I/O occurs in streams of bytes. A stream is simply a sequence of bytes. In input operation, the bytes flow from a device (e.g. keyboard) to main memory. In output operations, bytes flow from main memory to a device (e.g. screen)

`return 0;`

The C++ keyword `return` is one of several means we will use to exit a function. When the `return` statement is used at the end of `main()` as shown here, the value 0 indicates that the program has terminated successfully. This is the most usual way to terminate a C++ program.

Comments

Comments are used to document programs and improve program readability. They also help other people read and understand your program. Comments do not cause the computer to perform any action when the program is run.

Normally, there are two ways to put comments in a C++ program.

- // -- single line comment: each line begin with //. It discards everything from where // is found to the end of that same line
- /* */ -- block comment: -- all content contained inside /* */ are comments. It discards everything between /* and */, with the possibility to include multiple lines.

If you include your comments within the source code of your program without using the comment characters //, or /* */, the compiler will take them as if they were C++ instructions and most likely, it will show up one or more error messages.

Variable (use example `simpleProgram.cpp`)

A variable is a location in the computer's memory where a value can be stored for use by a program. That is a place to store a piece of information. Just as you might store a friend's phone number in your own memory, you can store this information in a computer's memory. Variables are your way of accessing your computer's memory.

Identifier (variable name)

A variable name is any valid identifier. An identifier is a series of characters consisting of letters, digits, and underscores. For example: `integer3`, `studentName`, `book`, `zip_code`. In the example `simpleProgram.cpp`, `a`, `b`, and `c` are identifiers.

An identifier does not begin with a digit, also cannot match any language's key word (e.g. `return`, `case`, `char`, `continue`, `class`, `if`, `else`).

Note: variable names are "case-sensitive" (i.e., the variable "`myNumber`" is different from the variable "`MYNUMBER`" which is different from the variable "`mYnUmBeR`"). Variable names can't have spaces.

Data type of variables

Before storing information into computer memory, we need to know what kind of information the computer will store (i.e., will it store a number, or a text-string, or something else) and how we plan to refer to the variable (i.e., the variable's name). A variable type is a description of the kind of information a variable will store.
C++ is a strongly typed language. Instead of declaring a variable as a number, you must say whether it will store integer or decimal. Some common used date types are: `char`, `int`, `float`, `double`, `bool`.

Declaration of variables

To use a variable in C++, we must declare it first by specifying what the data type is, and what the name is. All the variables that we are going to use must have been declared before used.

The syntax is:
`Data-type data-identifier(name);`
For example:
```
int myAge;
```
All this does is tell the computer that you plan to use an integer, and that the integer's name is `myAge`.

Sometimes, we can declare several variables of the same type in the same line separating the identifiers with commas. In the `simpleProgram.cpp`, we can write the declaration
```
int a = 0;
int b = 0;
int c = 0;
```
as
```
int a = 0 , b = 0, c = 0;
```
which has exactly the same meanings as that in `simpleProgram.cpp`. They all declare three variables `a`, `b` and `c` of type `int` and initialize them to 0.

It is not required to initialize variables while declaring them in C++, but it is recommendable to initialize them since sometimes you might get some unexpected values called garbage.
The syntax for initializing a variable is:

```
datatype dataidentifier = initialvalue;
```

`const`

With the `const` prefix, you can declare constants with a specific type exactly as you would do with a variable, just put the keyword `const` before the data type:

```
const int length = 100;
const char aLetter = 't';
```

After declaring `const`, the value of variable cannot be changed.

Operator

In C++, operators are a set of keywords and signs that are not part of the alphabet but available in all keyboards. They are the basis of C++ programming language.

Assignment operators in C++ (=)

The assignment operator is the equals sign (=). It serves to assign a value to a variable. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side. So, when you write `x = 5`, the operator takes the expression on the right, `5`, and stores it in the variable on the left, `x`.

It is necessary to emphasize that the assignation operation always takes place from right to left and never at the inverse.

```
a = b;
```

assigns to variable **`a`** (*lvalue*) the value that contains variable **`b`** (*rvalue*) independently of the value that was stored in **`a`** at that moment. Consider also that we are only assigning the value of **`b`** to **`a`** and that a later change of **`b`** would not affect the new value of **`a`**.

For example, if we take this code (with the evolution of the variables' content in green color):

```
int a, b;     // a:?  b:?
a = 10;       // a:10 b:?
b = 4;        // a:10 b:4
a = b;        // a:4 b:4
b = 7;        // a:4 b:7
```

it will give us as result that the value contained in **`a`** is **`4`** and the one contained in **`b`** is **`7`**. The final modification of **`b`** has not affected **`a`**, although before we have declared **`a = b;`** (right-to-left rule).

Increase and decrease operator

The *increase* operator (**`++`**)/ *decrease* operator (**`--`**) increases or reduces by 1 the value stored in a variable, and is equivalent to **`+=1`**/**`-=1`**, respectively. Thus:

```
a++;
a+=1;
a=a+1;
```

are all equivalent in its functionality: the three increase by 1 the value of **a**.

Its existence is due to that in the first C compilers the three previous expressions produced different executable code according to which one was used. Nowadays this type of code optimization is generally done automatically by the compiler.

A characteristic of this operator is that it can be used both as a *prefix* or as a *suffix*. That means it can be written before the variable identifier (**++a**) or after (**a++**) and, although in so simple expressions like **a++** or **++a** they have exactly the same meaning, in other operations in which the result of the *increase* or *decrease* operation is evaluated as another expression they may have an important difference in their meaning: In case that the increase operator is used as a *prefix* (**++a**) the value is increased before the expression is evaluated and therefore the increased value is considered in the expression; in case that it is used as a *suffix* (**a++**) the value stored in **a** is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the expression. Notice the difference:

Example 1

```
b=3;
a=++b;
// a is 4, b is 4
```

Example 2

```
b=3;
a=b++;
// a is 3, b is 4
```

In Example 1, **b** is increased before its value is copied to **a**. While in Example 2, the value of **b** is copied to **a** and **b** is later increased.

Arithmetic operators in C++

In addition to the Boolean operators, C++ has a number of arithmetic operators. They are:

Arithmetic operators		
Name	**Symbol**	**Sample usage**
addition	+	`int sum = 4 + 7`
subtraction	-	`float difference = 18.55 - 14.21`
multiplication	*	`float product = 5 * 3.5`
division	/	`int quotient = 14 / 3`
module ("mod")	%	`int remainder = 10 % 6`

They all probably look familiar with the exception of **mod (%)**. The mod is simply the remainder produced by dividing two integers. In the example shown in the table above, if we treat `10 / 6` as an integer division, the quotient is 1 (rather than 1.666) and the remainder is 4. Hence, the variable `remainder` will get the value 4.

Relational operators in C++ (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the Relational operators. As specified by the ANSI-C++ standard, the result of a relational operation is a **bool** value that can only be **true** or **false**, according to the result of the comparison.

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other. Here is a list of the relational operators that can be performed in C++:

Relational operators			
Name	**Symbol**	**Sample usage**	**Result**
is less than	<	`bool result = (4 < 7)`	`true`
is greater than	>	`bool result = (3.1 > 3.1)`	`false`
is equal to	==	`bool result = (11 == 8)`	`false`
is less than or equal to	<=	`bool result = (41.1 <= 42)`	`true`
is greater than or equal to	>=	`bool result = (41.1 >= 42)`	`false`
is not equal to	!=	`bool result = (12 != 12)`	`false`

Be aware. Operator == (two equal signs) is not the same as operator = (one equal sign), the first (==) is a relational operator of equality that compares whether both expressions in the two sides of the operator are equal to each other, and the second (=) is an assignation operator (assigns the right side of the expression to the variable in the left).

Logical operators in C++ (&&, ||, !)

`&&` -- and

The "and" operator is used by placing the "and" symbol, `&&`, in between two Boolean values. "and" requires both inputs to be true in order for the output to be true, if one of the inputs is false, the output will be false. It can have multiple operands. Let's see an example:

```
( 5 > 3 ) && ( 5 < 6 ) will return true ( true && true )
( 5 < 3 ) && ( 5 < 6 ) will return false ( false && true )
```

`||` -- or

The "or" operator is used by placing the "or" symbol, ||, in between two Boolean values. "or" requires at least one input to be true in order for the output to be true, if at least one of the inputs is true, the output will be true, otherwise the result will be false. It can have multiple operands. Let's see an example:

```
( 5 > 3 ) || ( 5 > 6 ) will return true ( true || false )
( 5 < 3 ) || ( 5 < 6 ) will return false ( false && true )
```

First Operand **a**	Second Operand **b**	Result **a && b**	Result **a || b**
true	true	**true**	**true**
true	false	**false**	**true**
false	true	**false**	**true**
false	false	**false**	**false**

`!` -- not

The "not" operator is used by placing the "not" symbol, !, before a Boolean value. It has only one operand, located at its right, and the only thing that it does is to invert the value of it.

```
! ( 5 > 3 ) will return false since the expression (5 > 3) is true (!true)
! ( 5 < 3 ) will return true since the expression (5 < 3) is false (!false)
```

It is perfectly legal in C++ to use boolean operators on variables, which are not booleans. In many compilers previous to the publication of the ANSI-C++ standard, as well as in the C language, the relational operations did not return a **`bool`** value **`true`** or **`false`**, rather they returned an **`int`** as result with a value of **`0`** in order to represent **"false"** and a value different from **`0`** (generally **`1`**, but non-zero is OK) to represent **"true"**. Let's look at a contrived example.

```
int hours = 4;
int minutes = 21;
int seconds = 0;

bool timeIsTrue = hours && minutes && seconds;
```

Since `hours` evaluates to be `true`, `minutes` evaluates to be `true`, and `seconds` evaluates to be `false`, the entire expression `hours && minutes && seconds` evaluates to be `false`.

Conditional operator (`?`).

The conditional operator evaluates an expression and returns a different value according to the evaluated expression, depending on whether it is *true* or *false*. Its format is:

`condition ? result1 : result2`

if *`condition`* is **`true`** the expression will return *`result1`*, if not it will return *`result2`*.

`7==5 ? 4 : 3`	returns **3** since **7** is not equal to **5**.
`7==5+2 ? 4 : 3`	returns **4** since **7** is equal to **5+2**.
`5>3 ? a : b`	returns **`a`**, since **5** is greater than **3**.
`a>b ? a : b`	returns the greater one, **`a`** or **`b`**.

Control Structure

Control Statements, then, are ways for a programmer to control what pieces of the program are to be executed at certain times. The syntax of Control statements is very similar to regular English, and to choices that we make every day. A program is usually not limited to a linear sequence of instructions. During its process it may break, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what and how has to perform our program.

In C++, there are three structures: *sequence structure, selection structure and repetition structure*.

Sequence structure

It's straightforward, sequential execution. Normally, statements in a program are executed one after the other in the order in which they are written. `simpleProgram.cpp` is an example of sequence structure: `get integer a and b -> do c = a + b -> display c`

Selection structure

A selection structure is used to choose among alternative courses of action. C++ provides three types of selection structures: `if, if/else, switch`.

- **`if`**

The `if` selection structure either performs (selects) an action if a condition is true or skips the action if the condition is false. It is used to execute an instruction or block of instructions only if a condition is fulfilled. Its form is:

```
if (condition)
{
   // code to execute if condition is true
}
```

where *`condition`* is the expression that is being evaluated. If this condition is **`true`**, *`code`* is executed. If it is false, *`code`* is ignored (not executed) and the program continues on the next instruction after the conditional structure.
For example, the following code fragment prints out **`x is 100`** only if the value stored in variable **`x`** is indeed 100:

```
if (x == 100)
  cout << "x is 100";
```

If we want more than a single instruction to be executed in case that *`condition`* is **`true`** we can specify a *block of instructions* using curly brackets `{ }`:

```
if (x == 100) {
  cout << "x is ";
  cout << x;
}
```

- **`if/else`**

The `if/else` selection structure performs an action if a condition is true and performs a different action if the condition is false. An **if statement** has the form:

```
if (condition)
{
   // code to execute if condition is true
}
else
{
   // code to execute if condition is false
}
```

where *condition* is a boolean value or an expression that is used to determine which code block is executed, and the curly braces act as "begin" and "end" markers.

Here is a full C++ program as an example:

```
//include this file for cout
#include <iostream>

int main() {
  // define two integers
  int x = 3;
  int y = 4;

  //print out a message telling which is bigger
  if (x > y) {
    cout << "x is bigger than y" << endl;
  }  else {
    cout << "x is smaller than y" << endl;
  }
  return 0;
```

```
}
```

In this case *condition* is equal to "(x > y)" which is equal to "(3 > 4)" which is a *false* statement. So the code within the **else** clause will be executed. The output of this program will be:

```
x is smaller than y
```

If instead the value for x was 6 and the value for y was 2, then *condition* would be "(6 > 2)" which is a *true* statement and the output of the program would be:

```
x is bigger than y
```

The *if/else* structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the present value stored in **x** is positive, negative or none of the previous, that is to say, equal to zero.

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Remember that in case we want that more than a single instruction is executed we must group them in a *block of instructions* by using curly brackets **{ }**.

- **switch**

The switch selection structure performs one of many different actions depending on the value of an expression. The syntax of the *switch* instruction is a bit peculiar. Its objective is to check several possible constant values for an expression, something similar to what we did at the beginning of this section with the linking of several *if* and *else if* sentences. Its form is the following:

```
switch (expression) {
  case constant1:
    block of instructions 1
    break;
  case constant2:
    block of instructions 2
    break;
  .
  .
  .
  default:
    default block of instructions
  }
```

It works in the following way: **switch** evaluates *expression* and checks if it is equivalent to *constant1*, if it is, it executes *block of instructions 1* until it finds the **break** keyword, moment at which the program will jump to the end of the *switch* selective structure.
If *expression* was not equal to *constant1* it will check if *expression* is equivalent to *constant2*. If it is, it will execute *block of instructions 2* until it finds the **break** keyword. Finally, if the value of *expression* has not matched any of the previously specified constants (you may specify as many **case** sentences as values you want to check), the

program will execute the instructions included in the **default:** section, if this one exists. The **default** clause is optional, but it is good programming practice to use it.

Both following code fragments are equivalent:

***switch* example**	***if-else* equivalent**

`switch (x) {`
`  case 1:`
`  cout << "x is 1";`
`  break;`
`  case 2:`
`  cout << "x is 2";`
`  break;`
`  default:`
`  cout << "value of x unknown";`
`  }`
	`if (x == 1) {`
	`  cout << "x is 1";`
	`  }`
	`else if (x == 2) {`
	`  cout << "x is 2";`
	`  }`
	`else {`
	`  cout << "value of x unknown";`
	`  }`

Notice the inclusion of the **break** instructions at the end of each block. The **break** keyword means "jump out of the switch statement, and do not execute any more code." This is necessary because if for example we did not include it after *`block of instructions 1`* the program would not jump to the end of the switch selective block (**}**) and it would follow executing the rest of blocks of instructions until the first appearing of the **break** instruction or the end of the switch selective block. To show how this works, examine the following piece of code:

```
int value = 0;
switch(input){
  case 1:
    value+=4;
  case 2:
    value+=3;
  case 3:
    value+=2;
  default:
    value++;
}
```

If *input* is 1 then 4 will be added to *value*. Since there is no **break** statement, the program will go on to the next line of code which adds 3, then the line of code that adds 2, and then the line of code that adds 1. So *value* will be set to 10!

It's not necessary to include curly brackets **{ }** in each of the cases. It can also be useful to execute one same block of instructions for different possible values for the expression evaluated, for example:

```
switch (x) {
    case 1:
    case 2:
    case 3:
      cout << "x is 1, 2 or 3";
      break;
    default:
      cout << "x is not 1, 2 nor 3";
    }
```

Notice that **`switch`** can only be used to compare an expression with different constants. Thus we cannot put variables (**`case (n*2):`**) or ranges (**`case (1..3):`**) because they are not valid constants. If you need to check ranges or values that are not constants use a concatenation of **`if`** and **`else if`** sentences.

Repetition structure (loops)

A repetition structure allows the programmer to specify an action is to be repeated while some condition remains true. There are three types of repetition structures, namely, `while`, `do/while` and `for`. The repetition structure is usually called loop.

- **`while`**

The **while statement** has the format:

```
while(condition) {
   statement
}
```

where *condition* is a boolean statement that is checked each time after the final "}" of the `while` statement executes. If the *condition* is true then the `while` statement executes again. If the *condition* is false, the `while` statement does not execute again. For example, let's say that we wanted to write all the even numbers between 1 and 10 to the screen. The following is a full C++ program that does that.

```
// include this file for cout
#include <iostream>

int main(){
   // this variable holds the present number
   int current_number = 2;

   // while loop that prints all even numbers between
   // 11 and 23 to the screen
   while (current_number < 10){
      cout << current_number << endl;
      current_number += 2;
   }
   cout << "all done" << endl;

   return 0;
}
```

The preceding example prints the value of `current_number` to the screen and then adds 2 to its value. As soon as the value of the variable `current_number` goes above 23, the while loop exits and the next line is executed. The output of the preceding program would be:

2
4
6
8
all done

- **`do/while`**

The **do/while statement** has the format:

```
do {
   statement
} while(condition);
```

Its functionality is exactly the same as the *while* loop except that *condition* in the *do-while* is evaluated after the execution of *statement* instead of before, granting at least one execution of *statement* even if *condition* is never fulfilled.

Note: there is a semicolon (;) after the `while (condition)`. Without it, some error will show up while compiling the program.

We can rewrite the `while` loop example like:

```
// include this file for cout
#include <iostream>

int main(){
   // this variable holds the present number
   int current_number = 2;

   // while loop that prints all even numbers between
   // 1 and 10 to the screen
   do {
      cout << current_number << endl;
      current_number += 2;
   } while (current_number < 10);
   cout << "all done" << endl;

   return 0;
}
```

The result would be the same as that of `while` loop.

- **for**

The `for` statement has the following format:

```
for (expression1; expression2; expression3) {
   statement
}
```

where *expression1* initializes the loop's control variable, *expression2* is the loop-continuation condition, which is the condition tested to see if the loop is executed again, and *expression3* describes how the control variable is changed on each execution of the loop.

It works the following way:

1. *expression1* is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. *expression2* is checked, if it is **true** the loop continues, otherwise the loop finishes and *statement* is skipped.
3. *statement* is executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets **{ }**.

4. finally, whatever is specified in the *expression3* is executed and the loop gets back to step 2.

In most cases the `for` structure can be represented with an equivalent while structure as follows:

```
expression1;
while (expression2) {
   statement;
   expersstion3;
}
```

Let's rewrite the example of displaying the even number between 1 and 10 using `for` structure:

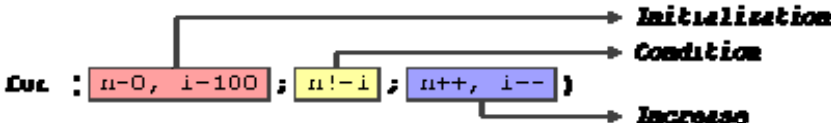
```
// include this file for cout
#include <iostream>

int main(){
   for (int current_number=2; current_number<10; current_number+=2) {
      cout << current_number << endl;
   }
   cout << "all done" << endl;

   return 0;
}
```

We will get the same result as we did in `while`, `do/while` loop.

Optionally, using the comma operator (**,**) we can specify more than one instruction in any of the fields included in a **for** loop, like in *expression1*, for example. The comma operator (**,**) is an instruction separator, and serves to separate more than one instruction where only one instruction is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
for ( n=0, i=100 ; n!=i ; n++, i-- ) {
  // whatever here...
}
```

This loop will execute 50 times if neither **n** nor **i** are modified within the loop:



n starts with **0** and **i** with **100**, the condition is **(n!=i)** (that **n** be not equal to **i**). Beacuse **n** is increased by one and **i** decreased, the loop's condition will become `false` after the 50th loop, when both **n** and **i** will be equal to 50.

Note: We must consider that the loop has to end at some point, therefore, within the block of instructions (loop's *statement*) we must provide some method that forces *condition* to become false at some moment, otherwise the loop will continue looping forever.

Function

A function is a block of instructions that is executed when it is called from some other point of the program. It allows the programmer to modularize a program. A major advantage of function is its reusability – using existing functions as building blocks to create new programs.

Each program we have presented has consisted of a function called `main()`, which is called standard library functions to accomplish its tasks. We now consider how programmers write their own customized functions.

Each function in C++ has a function prototype, and function definition. Let's see an example first:

```
/* This program modifies the program simpleProgram.cpp, itdescribes how to
   create and call a function in C++
 */

#include <iostream>

int add(int, int); // function prototype

int main() {
   // Declare variables;
   int a = 0, b = 0;

   cout << "Please enter an integer: ";
   cin >> a;
   cout << "Please enter another integer: ";
   cin >> b;
   cout << endl;

   int c = add(a,b); // function call

   cout << "The sum of " << a << " and " << b << " is " << c << endl;

   return 0;

}

// function definition
/* Name: add()
 * Description:
     This function receives two parameters, add them together, then return
     the sum.
 * Input: i, j -- two integers
 * Output: k -- return the sum of i and j
 */
int add( int i, int j ) {
   return i + j;
}
```

Function prototype

In the above example,

```
   int add(int, int);
```

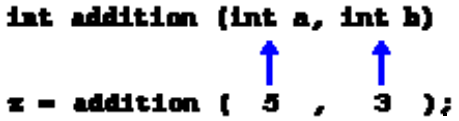
is a *function prototype*, which tells the compiler the name of the function, the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which these parameters are expected. The compiler uses function prototypes to validate function calls. In this case, the name of the function is `add`,

the date returned by the function is `int`, the number of parameters the function expects to receive is 2, the types of parameters are `int`. The compiler refers to the function prototype to check that calls to `add` contain the correct return type, correct number of arguments, and correct arguments are in correct order.

The function prototype is not required if the function definition appears before the function ‘s first use in the program. In such a case, the function definition also acts as the function prototype. For example:

```
/* funcnoprot.cpp
 * This program shows function without prototype
 */

 #include <iostream>

 // function addition
 int addition (int a, int b) {
    int r;
    r=a+b;
    return (r);
 }

 int main ()  {
    int z;
    z = addition (5,3);
    cout << "The result is " << z << endl;
    return 0;
 }
```

In this example, we can see some similarity between the structure of the call to the function and the declaration of the function itself:



The parameters have a clear correspondence. Within the **`main`** function we called to **`addition`** passing two values: **`5`** and **`3`** that correspond to the **`int a`** and **`int b`** parameters declared for the function **`addition`**.

At the moment at which the function is called from **`main`**, the control is lost by **`main`** function and passed to function **`addition`**. The value of both parameters passed in the call (**`5`** and **`3`**) are copied to the local variables **`int a`** and **`int b`** within the function.

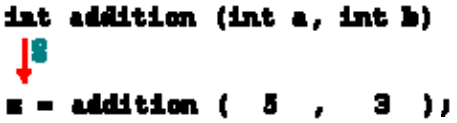
Function **`addition`** declares a new variable (**`int r;`**), and by means of the expression **`r=a+b;`**, it assigns to **`r`** the result of **`a`** plus **`b`**. Because the passed parameters for **`a`** and **`b`** are **`5`** and **`3`** respectively the result is **`8`**.

The following line of code:

```
return (r);
```

finalizes function **`addition`**, and returns the control back to the function that has called it (**`main`**) following the program by the same point in which it was interrupted by the call to

addition. But additionally, **return** was called with the content of variable **r** (**return (r);**), that at that moment was 8, so this value is said to be **returned** by the function.



The value returned by a function is the value given to the function when it is evaluated. Therefore, **z** will store the value returned by **addition (5, 3)**, that is **8**. To explain it somehow you can imagine that the call to a function (**addition (5,3)**) is literally replaced by the value it returns (**8**).

The following line of code in **main** is:

```
cout << "The result is " << z;
```

which displays the result on the screen.

Function definition

Function definition is the implementation of a function, it defines what the task is and how the function does its task. The format of a function definition is

```
return-value-type function-name (parameter-list ) {
   declarations and statements
}
```

The *function-name* is any valid identifier. The *return-value-type* is the data type of the result returned from the function to the caller. The *return-value-type* **void** indicates that a function does not return a value. The *parameter-list* is a comma-separated list containing the declarations of the parameters received by the function when it is called. If a function does not receive any values, *parameter-list* is **void** or simply left empty. A type must be listed explicitly for each parameter in the parameter list of a function. The declaration and statements in braces form the function body. In example `func.cpp`, the function definition looks like:

```
int add( int i, int j ) {
   return i + j;
}
```

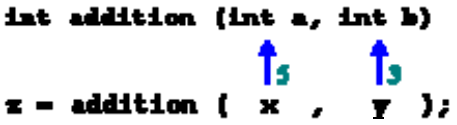
it receives two parameters, i and j, adds them, then returns the sum.

Arguments passed *by value*

Until now, in all the functions we have seen, the parameters passed to the functions have been passed *by value*. This means that when calling to a function with parameters, what we have passed to the function were values but never the specified variables themselves. For example, suppose that we called to our first function **addition** using the following code:

```
int x=5, y=3, z;
z = addition ( x , y );
```

What we did in this case was to call function **addition** passing the values of **x** and **y**, that means **5** and **3** respectively, not the variables themselves.



This way, when function **addition** is being called the value of its variables **a** and **b** become **5** and **3** respectively, but any modification of **a** or **b** within the function **addition** will not affect the values of **x** and **y** outside it, because variables **x** and **y** were not passed themselves to the function, only their values.

The following are the examples, which will be used for Basic C++ workshop:

```
/* simpleProgram.cpp
 * This is simple C++ program
 */

#include <iostream>

int main() {
   // Declare variables
   int a = 0;
   int b = 0;
   int c = 0;

   // Prompt user for two integers
   cout << "Please enter one integer: ";
   cin >> a;
   cout << "Please enter another integer: ";
   cin >> b;

  // Add the two integers together
  c = a + b;

  // Output the result
  cout << "The result is: "
       << a <<  " + " << b << " = " << c << endl;

  // Exit the program
  return 0;
}


/* incrementAndDecrement.cpp
 * This program shows how ++ works
 */

#include <iostream>

int main() {

   int b=3;
//   int a=++b;

   int a=b++;
   cout << "b is: " << b << "\na = ++b is: " << a << endl;
```

```
   int c = b;
   cout << "c is " << c << endl;
   return 0;
}


/* logicalOperator.cpp
 * This program shows how logical operators work
 */

#include <iostream>

int main() {

   if ( 5 < 6 && 5 > 3 )
      cout << "A: true" << endl;

   if ( 5 > 6 && 5 < 3 )
      cout << "B: fasle" << endl;


   if ( 5 < 6 || 5 > 3 )
      cout << "C: true" << endl;

   if ( 5 > 6 || 5 > 3 )
      cout << "D: false" << endl;

   bool test1 = 5 < 3;
   bool test2 = 5 > 3;

   if ( !(test1 == true) )
      cout << "test1 is not true." << endl;
   else
      cout << "test1 is not false." << endl;

   if ( !(test2 == true) )
      cout << "test2 is not true." << endl;
   else
      cout << "test2 is not false." << endl;

   return 0;
}


/* switchWithoutBreak.cpp
 * This program shows switch without break
 */

#include <iostream>

int main() {
   int value = 0;

   int input = 0;
   cout << "Please enter a number: ";
   cin >> input;
   cout << endl;
```

```
   switch (input) {
     case 1:
       value+=4;
     case 2:
       value+=3;
     case 3:
       value+=2;
     default:
       value++;
  }
   cout << "The value is: " << value << endl;

   return 0;
}


/* whileLoop.cpp */

// include this file for cout
#include <iostream>

int main(){
   // this variable holds the present number
   int current_number = 2;

   // while loop that prints all even numbers between
   // 1 and 10 to the screen
   while (current_number < 10){
      cout << current_number << endl;
      current_number += 2;
   }
   cout << "all done" << endl;

   return 0;
}


/* doWhile.cpp */

// include this file for cout
#include <iostream>

int main(){
   // this variable holds the present number
   int current_number = 2;

   // while loop that prints all even numbers between
   // 1 and 10 to the screen
   do {
      cout << current_number << endl;
      current_number += 2;
   } while (current_number < 10);
   cout << "all done" << endl;

   return 0;
}
```

```
/* forLoop.cpp */

// include this file for cout
#include <iostream>

int main(){
   for (int current_number=2; current_number<10; current_number+=2) {
      cout << current_number << endl;
   }
   cout << "all done" << endl;

   return 0;
}


/* function.cpp
/* This program modifies the program simpleProgram.cpp,
 * itdescribes how to create and call a function in C++
 */

#include <iostream>
// function prototype
int add(int, int);

int main() {
   // Declare variables;
   int a = 0, b = 0;

   cout << "Please enter an integer: ";
   cin >> a;
   cout << "Please enter another integer: ";
   cin >> b;
   cout << endl;

   // function call
   int c = add(a,b);

   cout << "The sum of " << a << " and " << b << " is " << c << endl;

   return 0;

}

// function definition
/* Name: add()
 * Description:
      This function receives two arguments, add them together,
then return the sum.
 * Input: i, j -- two integers
 * Output: k -- return the sum of i and j
 */
int add( int i, int j ) {
   return i + j;
}
```

```
/* funcnoprot.cpp
 * This program shows function without prototype
 */

 #include <iostream>

 // function addition
 int addition (int a, int b) {
   int r;
   r=a+b;
   return (r);
 }

 int main ()
 {
   int z;
   z = addition (5,3);
   cout << "The result is " << z << endl;
   return 0;
 }


/* funcbyvalue.cpp
 * This program shows function call-by-value
 */

#include <iostream>
// function prototype
int addition(int, int);

int main() {

   int x = 5, y = 3, z = 0;
   z = addition ( x , y ); // function call
   cout << "x is " << x << ", y is " << y << endl;
   cout << "z is " << z << endl;
}

// function definition
int addition(int a, int b) {
   int c = a + b;
   cout << "in function, before modification: " << endl;
   cout << "a is " << a << ", b is " << b << endl;
   a = 10, b = 20;
   cout << "in function, after modification: " << endl;
   cout << "a is " << a << ", b is " << b << endl;
   c = a + b;
   return c;
}
```