PART I

C++ LANGUAGE BASICS

In order to write programs we need to learn some of the many traditional feature of the C++ programming language, including variables, operators, and control flow statements.

Your First C Program

This demonstration uses a program named HELLO.CPP, which does nothing more than display the words Hello, World! on-screen. This program, a traditional introduction to C programming, is a good one for you to learn. The source code for HELLO.CPP is in Listing 1.1. When you type in this listing, you won't include the line numbers or colons.

Listing 1.1. HELLO.CPP.

```
1: #include <iostream>
2:using namespace std;
3: int main()
4: {
5:     cout<<"Hello, World!\n";
6:     return 0;
7: }
```

The Program's Components

The following sections describe the various components of the preceding sample program. Line numbers are included so that you can easily identify the program parts being discussed.

The main() Function (Lines 4 Through 7)

The only component that is required in every C++ program is the main() function. In its simplest form, the main() function consists of the name main followed by a pair of empty parentheses (()) and a pair of braces ({}). Within the braces are statements that make up the main body of the program. Under normal circumstances, program execution starts at the first statement in main() and terminates at the last statement in main().

The #include Directive (Line 2)

The #include directive instructs the C++ compiler to add the contents of an include file into your program during compilation. An include file is a separate disk file that contains information needed by your program or the compiler. Several of these files (sometimes called header files) are supplied with your compiler. You never need to modify the information in these files; that's why they're kept separate from your source code. Include files should all have an .H extension (for example, iostream).

You use the #include directive to instruct the compiler to add a specific include file to your program during compilation. The #include directive in this sample program means "Add the contents of the file iostream"

Compilation Errors

A compilation error occurs when the compiler finds something in the source code that it can't compile. ***A misspelling, typographical error, or any of a dozen other things can cause the compiler to choke***. Fortunately, modern compilers don't just choke; they tell you what they're choking on and where it is! This makes it easier to find and correct errors in your source code.

This point can be illustrated by introducing a deliberate error into HELLO.CPP. If you worked through that example (and you should have), you now have a copy of HELLO.CPP on your disk. Using your editor, move the cursor to the end of the line containing the call to cout<<, and erase the terminating semicolon. HELLO.CPP should now look like Listing 1.2.

Listing 1.2. HELLO.CPP with an error.

```
1: #include <iostream>
2:using namespace std;
3: int main()
4: {
5:     cout<<"Hello, World!";
6:     return 0;
7: }
```

Next, save the file. You're now ready to compile it. Do so by entering the command for your compiler. Because of the error you introduced, the compilation is not completed. Rather, the compiler displays a message.

Exercises

1.Enter the following program and compile it. What does this program do? (Don't include the line numbers or colons.)

```
1: #include <iostream>
2:using namespace std;
3: int radius, area;
4:
5: int main()
6: {
7:     cout<<"Enter radius (i.e. 10): " ;
8:     cin>>radius ;
9:     area = (int) (3.14159 * radius * radius);
10:    cout<<"\n\nArea = %d\n"<<area ;
11:    return 0;
12: }
```

2. Enter and compile the following program. What does this program do?

```
1: #include <iostream>
2:using namespace std;
3: int x,y;
4:
5: int main()
6: {
7:     for ( x = 0; x < 10; x++, cout<< "\n" )
8:         for ( y = 0; y < 10; y++ )
9:             cout<< "X" ;
10:
11:    return 0;
12: }
```

Variables

You use variables in your program to hold data. A variable is an item of data named by an identifier. You must explicitly provide a name and a type for each variable you want to use in your program. The variable's name must be a legal identifier. You use the variable name to refer to the data that the variable contains. **A variable** is a name assigned to a data storage location. Your program uses variables to store various kinds of

data during program execution. In C++, a variable must be defined before it can be used. A variable definition informs the compiler of the variable's name and the type of data it is to hold. The variables type determines what value it can hold and what operations can be performed on it. To give a variable a type and a name, you write a variable declaration:

type name

A variable has scope which specifies the section of code where the variable's simple name can be used.

Data Types

Every variable must have a data type which determines the value that the variable can contain and the operations that can be performed on it.

The C++ programming language has two categories of data types: ***primitive*** and ***reference.*** A variable of primitive type contains a single value of the appropriate size and format for its type.

Primitive Data Types

Keyword	**Description**	**Size**
(Integers)		
byte	Byte-length int	8-bit two's complement
short	Short integer	16-bit
int	Integer	32-bit
long	Long integer	64-bit
(Real numbers)		
float	Single-precision floating-point	32-bit
double	Double-precision fp	64-bit
(others)		
char	A single character	16-bit
boolean	A boolean value	true or false

The program below will help you determine the size of variables on your particular computer. Don't be surprised if your output doesn't match the output presented after the listing.

A program that displays the size of variable types.

```
1:  /* SIZEOF.CPP--Program to tell the size of the CPP variable */
2:  /*          type in bytes */
```

```
3:
4:    #include <iostream>
5:    using namespace std;
6:    int main()
7:   {
8:
9:         cout<< "\nA char      is %d bytes"<<sizeof( char );
10:        cout<<"\nAn int      is %d bytes"<<sizeof( int );
11:        cout<<"\nA short     is %d bytes"<< sizeof( short );
12:        cout<<"\nA long      is %d bytes", sizeof( long );
13:        cout<< "\nAn unsigned char  is %d bytes"<<sizeof( unsigned char );
14:        cout<<"\nAn unsigned int   is %d bytes"<<sizeof( unsigned int );
15:        cout<< "\nAn unsigned short is %d bytes"<<sizeof( unsigned short );
16:        cout<<"\nAn unsigned long  is %d bytes"<<sizeof( unsigned long );
17:        cout<< "\nA float     is %d bytes"<<sizeof( float );
18:        cout<< "\nA double    is %d bytes\n"<<sizeof( double );
19:
20:        return 0;
21:   }
A char      is 1 bytes
An int      is 2 bytes
A short     is 2 bytes
A long      is 4 bytes
An unsigned char  is 1 bytes
An unsigned int   is 2 bytes
An unsigned short is 2 bytes
An unsigned long  is 4 bytes
A float     is 4 bytes
A double    is 8 bytes
```

ANALYSIS:

Although the size of the data types can vary depending on your computer platform, CPP does make some guarantees, thanks to the ANSI Standard. There are five things you can count on:
The size of a char is one byte.

The size of a short is less than or equal to the size of an int.

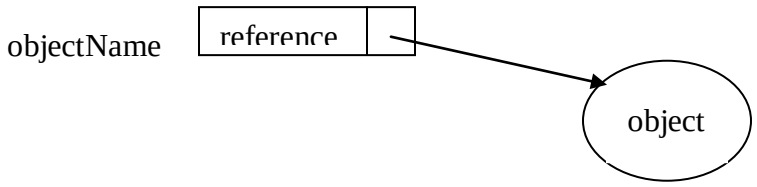
The size of an int is less than or equal to the size of a long.

The size of an unsigned is equal to the size of an int.

The size of a float is less than or equal to the size of a double

Reference types

A reference is a data element whose value is an address. **Arrays, classes, and interfaces** are reference types. The value of a reference type variable is reference to (an address of) the value or set of values represented by the variable. A reference is called a pointer, or a memory address in other languages. The C++ programming language does not support the explicit use of addresses like other programming languages do. You use the variables name instead.



Variable Names

A program refers to a variable's value by the variable name.

```
 int sum;
for (int counter = 1; counter <=10; counter ++)
 {
   sum = sum + current;
}
cout<< "Sum  = " << sum<<endl;
```

A name such as *sum* that is composed of a single identifier is called a simple name in contrast to qualified names, which a class uses to refer to a member variable that is in another object or class,

The following must hold true for a simple name:

1. The name can contain letters, digits, and the underscore character (_).

2. It must be a legal identifier
3. Must not be a keyword, a boolean literal (true or false)
4. It must be unique within its scope.
5. Case matters (that is, upper- and lowercase letters). Thus, the names count and Count refer to two different variables

SCOPE

A variable's scope is the region of a program within which the variable can be referred to by its simple name. Scope also determines when the system creates and destroys memory for the variable.

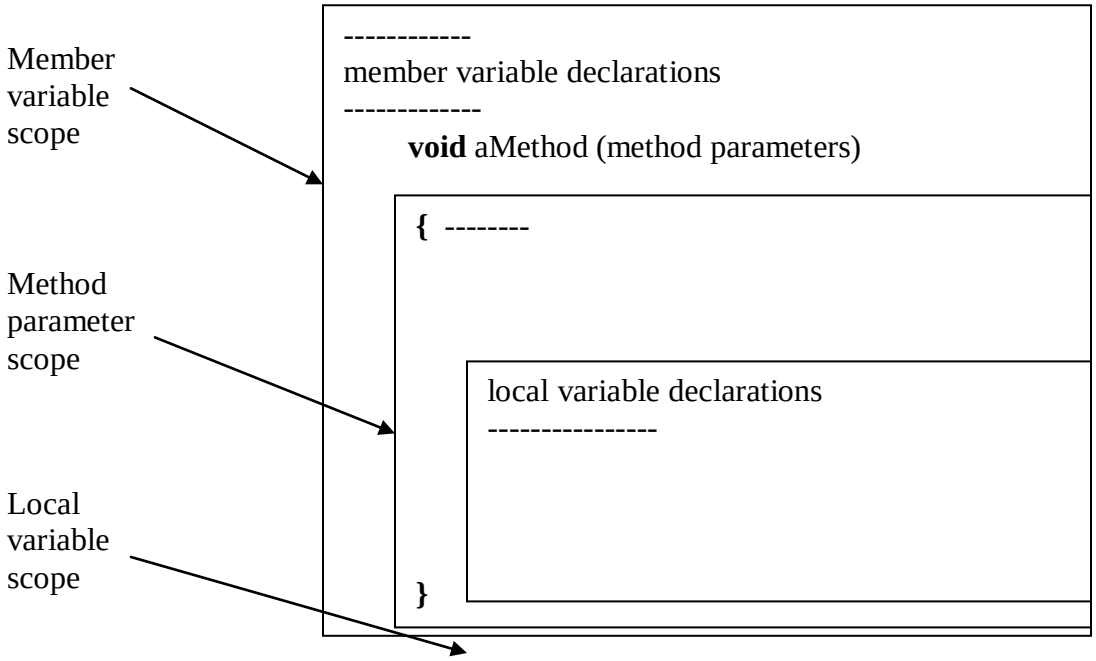
The location of the variable declaration within your program establishes its scope and places it into one of these four categories:

- member variables
- local variables
- method parameter

A member variable is a member of a class or an object, it is declared within a class but outside of any method. Its scope is the entire declaration of the class.

You declare local variables within a block of code. The scope of a local variable extends from its declaration to the end of the code block in which it was declared.

Parameters are formal arguments to methods and are used to pass values into methods. The scope of a parameter is the entire method for which it is a parameter.



}

Consider the following code sample:

```
if (…….)
{
int i = 17;
……
}
cout<< “The value of i = “ << i<<endl;
```

The final line won’t compile because the local variable is outside scope. The scope of *i* is the block of code between { and }. The *i* variable does not exist any more after closing}.

Constant Variables

Like a variable, a constant is a data storage location used by your program. Unlike a variable, the value stored in a constant can't be changed during program execution. CPP has two types of constants, each with its own specific uses.

Literal Constants

A literal constant is a value that is typed directly into the source code wherever it is needed. Here are three examples:

```
const int aConstantInteger = 100;

int count = 20;
float tax_rate = 0.28;
```

The 20 and the 0.28 are literal constants. The preceding statements store these values in the variables count and tax_rate. Note that one of these constants contains a decimal point, whereas the other does not. The presence or absence of the decimal point distinguishes floating-point constants from integer constants.

A literal constant written with a decimal point is a floating-point constant and is represented by the CPP compiler as a double-precision number. Floating-point constants can be written in standard decimal notation, as shown in these examples:

```
123.456
0.019
100.
```

Note that the third constant, 100. Is written with a decimal point even though it's an integer (that is, it has no fractional part). The decimal point causes the CPP compiler to treat the constant as a double-precision value. Without the decimal point, it is treated as an integer constant.

Floating-point constants also can be written in scientific notation. In CPP, scientific notation is written as a decimal number followed immediately by an E or e and the exponent:

1.23E2 1.23 times 10 to the 2nd power, or 123

4.08e6 4.08 times 10 to the 6th power, or 4,080,000

0.85e-4 0.85 times 10 to the -4th power, or 0.000085

A constant written without a decimal point is represented by the compiler as an integer number. Integer constants can be written in three different notations:
A constant starting with any digit other than 0 is interpreted as a decimal integer (that is, the standard base-10 number system). Decimal constants can contain the digits 0 through 9 and a leading minus or plus sign. (Without a leading minus or plus, a constant is assumed to be positive.)

A constant starting with the digit 0 is interpreted as an octal integer (the base-8 number system). Octal constants can contain the digits 0 through 7 and a leading minus or plus sign.

A constant starting with 0x or 0X is interpreted as a hexadecimal constant (the base-16 number system). Hexadecimal constants can contain the digits 0 through 9, the letters A through F, and a leading minus or plus sign.

Symbolic Constants

A symbolic constant is a constant that is represented by a name (symbol) in your program. Like a literal constant, a symbolic constant can't change. Whenever you need the constant's value in your program, you use its name as you would use a variable name. The actual value of the symbolic constant needs to be entered only once, when it is first defined.

Symbolic constants have two significant advantages over literal constants, as the following example shows. Suppose that you're writing a program that performs a variety of geometrical calculations. The program frequently needs the value ,, (3.14159) for its calculations. (You might recall from geometry class that ,, is the ratio of a circle's circumference to its diameter.) For example, to calculate the circumference and area of a circle with a known radius, you could write

```
circumference = 3.14159 * (2 * radius);
area = 3.14159 * (radius)*(radius);
```

The asterisk (*) is C++'s multiplication operator Thus, the first of these statements means "Multiply 2 times the value stored in the variable radius, and then multiply the result by 3.14159. Finally, assign the result to the variable named circumference."

If, however, you define a symbolic constant with the name PI and the value 3.14, you could write

```
circumference = PI * (2 * radius);
area = PI * (radius)*(radius);
```

The resulting code is clearer. Rather than puzzling over what the value 3.14 is for, you can see immediately that the constant PI is being used.

The second advantage of symbolic constants becomes apparent when you need to change a constant. Continuing with the preceding example, you might decide that for greater accuracy your program needs to use a value of PI with more decimal places: 3.14159 rather than 3.14. If you had used literal constants for PI, you would have to go through your source code and change each occurrence of the value from 3.14 to 3.14159. With a symbolic constant, you need to make a change only in the place where the constant is defined.

C++ has two methods for defining a symbolic constant: the #define directive and the const keyword. The #define directive is one of CPP's preprocessor directives. The #define directive is used as follows:

```
#define CONSTNAME literal
```

This creates a constant named CONSTNAME with the value of literal. Literal represents a literal constant, as described earlier. CONSTNAME follows the same rules described earlier for variable names. By convention, the names of symbolic constants are

uppercase. This makes them easy to distinguish from variable names, which by convention are lowercase. For the previous example, the required #define directive would be #define PI 3.14159

Note that #define lines don't end with a semicolon (;). #defines can be placed anywhere in your source code, but they are in effect only for the portions of the source code that follow the #define directive. Most commonly, programmers group all #defines together, near the beginning of the file and before the start of main().

a) How a #define Works

The precise action of the #define directive is to instruct the compiler as follows: "In the source code, replace CONSTNAME with literal." The effect is exactly the same as if you had used your editor to go through the source code and make the changes manually. Note that #define doesn't replace instances of its target that occur as parts of longer names, within double quotes, or as part of a program comment. For example, in the following code, the instances of PI in the second and third lines would not get changed:

```
#define PI 3.14159
/* You have defined a constant for PI. */
#define PIPETTE 100
```

b) Defining Constants with the const Keyword

The second way to define a symbolic constant is with the const keyword. **const** is a modifier that can be applied to any variable declaration. A variable declared to be const can't be modified during program execution--only initialized at the time of declaration. Here are some examples:

```
const int count = 100;
const float pi = 3.14159;
const long debt = 12000000, float tax_rate = 0.21;
```

const affects all variables on the declaration line. In the last line, debt and tax_rate are symbolic constants. If your program tries to modify a const variable, the compiler generates an error message, as shown here:

```
const int count = 100;
count = 200;        /* Does not compile! Cannot reassign or alter */
                    /* the value of a constant. */
```

What are the practical differences between symbolic constants created with the #define directive and those created with the const keyword? The differences have to do with pointers and variable scope. Pointers and variable scope are two very important aspects of C++ programming.

Now let's look at a program that demonstrates variable declarations and the use of literal and symbolic constants. The program below prompts the user to input his or her weight and year of birth. It then calculates and displays a user's weight in grams and his or her age in the year 2000.

The program that demonstrates the use of variables and constants.

```
1:   /* Demonstrates variables and constants */
2:   #include <iostream>
3:   using namespace std;
4:   /* Define a constant to convert from pounds to grams */
5:   #define GRAMS_PER_POUND 454
6:
7:   /* Define a constant for the start of the next century */
8:   const int NEXT_CENTURY = 2000;
9:
10:  /* Declare the needed variables */
11:  long weight_in_grams, weight_in_pounds;
12   int year_of_birth, age_in_2000;
13:
14:  int main()
15:  {
16:      /* Input data from user */
17:
18:      cout<<"Enter your weight in pounds: ";
19:      cin>>weight_in_pounds;
20:      cout<<"Enter your year of birth: ";
21:      cin>>year_of_birth;
22:
23:      /* Perform conversions */
24:
25:      weight_in_grams = weight_in_pounds * GRAMS_PER_POUND;
```

```
26:      age_in_2000 = NEXT_CENTURY - year_of_birth;
27:
28:      /* Display results on the screen */
29:
30:      cout<<"\nYour weight in grams ="<<weight_in_grams);
31:      cout<<"\nIn 2000 you will be"<< age_in_2000<<" Years Old\n");
32:
33:      return 0;
34:  }
Enter your weight in pounds: 175
Enter your year of birth: 1960
Your weight in grams = 79450
In 2000 you will be 40 years old
```

OPERATORS

An operator performs a function on one, two or three operands.

Arithmetic Operators

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
-	op – op 2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1/op2	Divides op1 by op2
%	op1%op2	Computer the remainder of dividing op1 by op2

Mixed Operations

Data type of result	Data Type of operands
long	Neither operand is a float or double, at least one is long
int	Neither is a float or double; neither is long
double	At least one operand is a double
float	At least one operand is a float; neither is a double

op++ increments op by 1
op-- decrements op by 1

The unary mathematical operators are so named because they take a single operand. C++ has two unary mathematical operators, listed in Table below

C++'s unary mathematical operators.

Operator	Symbol	Action	Examples
Increment	++	Increments the operand by one	++x, x++
Decrement	--	Decrements the operand by one	--x, x--

The increment and decrement operators can be used only with variables, not with constants. The operation performed is to add one to or subtract one from the operand. In other words, the statements

```
++x;
--y;
```

are the equivalent of these statements:

```
x = x + 1;
y = y - 1;
```

You should note from Table above that either unary operator can be placed before its operand (prefix mode) or after its operand (postfix mode). These two modes are not equivalent. They differ in terms of when the increment or decrement is performed:
When used in prefix mode, the increment and decrement operators modify their operand before it's used.

When used in postfix mode, the increment and decrement operators modify their operand after it's used.

An example should make this clearer. Look at these two statements:

```
x = 10;
y = x++;
```

After these statements are executed, x has the value 11, and y has the value 10. The value of x was assigned to y, and then x was incremented. In contrast, the following statements result in both y and x having the value 11. x is incremented, and then its value is assigned to y.

```
x = 10;
y = ++x;
```

Remember that = is the assignment operator, not a statement of equality. As an analogy, think of = as the "photocopy" operator. The statement y = x means to copy x into y. Subsequent changes to x, after the copy has been made, have no effect on y.

Program illustrates the difference between prefix mode and postfix mode.

UNARY.CPP: Demonstrates prefix and postfix modes.

```
1:   /* Demonstrates unary operator prefix and postfix modes */
2:
3:   #include <iostream >
4:   using namespace std;
5:   int a, b;
6:
7:   int main()
8:   {
9:       /* Set a and b both equal to 5 */
10:
11:      a = b = 5;
12:
13:      /* Print them, decrementing each time. */
14:      /* Use prefix mode for b, postfix mode for a */
15:
16:      cout<< a--<< --b;
17:      cout<< a--<< --b;
18:      cout<< a--<< --b;
19:      cout<< a--<< --b;
20:      cout<< a--<< --b;
21:
22:      return 0;
23:  }
5    4
4    3
3    2
2    1
1    0
```

Operator Precedence and Parentheses

In an expression that contains more than one operator, what is the order in which operations are performed? The importance of this question is illustrated by the following assignment statement:

x = 4 + 5 * 3;

Performing the addition first results in the following, and x is assigned the value 27:

x = 9 * 3;

In contrast, if the multiplication is performed first, you have the following, and x is assigned the value 19:

x = 4 + 15;

Clearly, some rules are needed about the order in which operations are performed. This order, called operator precedence, is strictly spelled out in C++. Each operator has a specific precedence. When an expression is evaluated, operators with higher precedence are performed first. Table below lists the precedence of C++'s mathematical operators. Number 1 is the highest precedence and thus is evaluated first.

The precedence of C's mathematical operators.

Operators	Relative Precedence
++ --	1
* / %	2
+ -	3

Relational and Conditional Operators

A relational operator compares two values and determines the relationship between them.

Operator 1	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 > = op2	op1 greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 < = op2	op1 is less than or equal to op2
= =	op1 = = op2	op1 and op2 are equal
! =	op1! = op2	op1 and op2 are not equal.

Relational operators often are used with conditional operators to construct more complex decision-making expressions.

Operator	Use	Returns true if
&&	op&&op2	op1 and op2 are both true
||	op1||op2	either op1 or op2 is true
!	!op	op is false

Assignment Operator

You use the basic assignment operator, =, to assign one value to another.

```
int y, z;
int x = 10;
y = x + 5
y = 2;
```

Expression, Statements and Blocks

Variables and operators are basic building blocks of programs. You combine literals, variables and operators to form expressions – segments of code that perform computations and return values. Certain expressions can be made into statements – complete units of execution. By grouping statements together with curly braces { and }, you create blocks of code.

Expressions

Among other things, expressions are used to compute and to assign values to variables and to help control the execution flow of a program. The job of an expression is twofold; to perform the computation indicated by the elements of the expression and to return a value that is the result of the computation.

e.g.
```
x * y*z
x + y/100
(x + y)/100

char achar = 'c'
```

When writing compound expressions, you should be explicit and indicate with parentheses which operator should be evaluated first. When operators of equal precedence appear in the same expression, the rule is to evaluate in the left–to–right order. Assignment operators are evaluated *right to left.*

Statement
A statement forms a complete unit of execution and is terminated with semicolon (;)

```
avalue = 8933.234;
avalue++;
cout<<avalue;
```

Null Statements

If you place a semicolon by itself on a line, you create a null statement--a statement that doesn't perform any action. This is perfectly legal in C. Later in this book, you will learn how the null statement can be useful.

Compound Statements // Blocks
A block is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

A compound statement, also called a block, is a group of two or more C++ statements enclosed in braces. Here's an example of a block:

```
{
   cout<<"Hello, ";
   cout<<"world!";
}
```

These are examples of expression statements. A declaration statement declares a variable e.g.

```
double aValue = 8933.234;
int    sum;
```

A control flow statement regulates the order in which statements get executed. The ***for*** loop and the ***if*** statements are both examples of control flow.

PART 2

CONTROL FLOW STATE MENTS

When you write a program, you type statements into a file. Without control flow statements, the interpreter executes these statements in the order they appear in the file from left–to–right, top to bottom. You can use control flow statements in your programs to conditionally execute statements, to repeatedly execute a block of statements, and to otherwise change the normal, sequential flow of control.

Statement Type	**Keyword**
looping	while, do-while, for
decision making	if-else, switch-case
branching	break, continue, return

The ***while*** and ***do-while*** statements

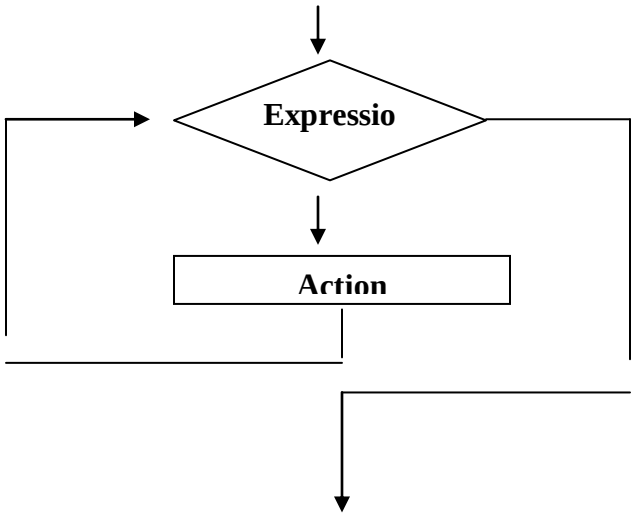
You use a ***while*** statement to continually execute a block of statements while a condition remains true.

The general syntax is

```
while (expression){
Statements
}
```

First the while statement evaluates expression, which must return a boolean value.

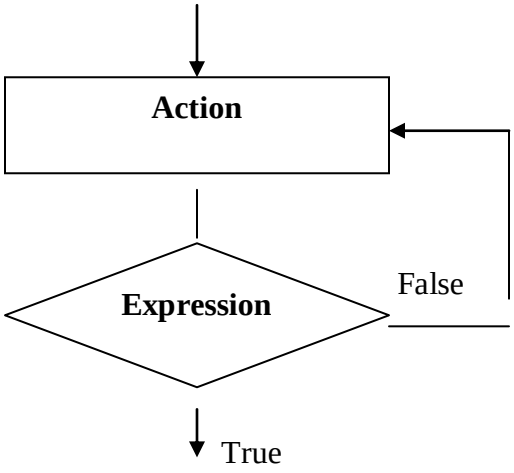
Flow-chart representation.



The ***do – while*** statement

syntax:

```
do{
        statements
  }while (expression)
```



The ***for*** statement

Syntax:

```
for (initialization; termination; update){
        Statement

        }
```

Initialization is an expression that initializes the loop – its executed once at the beginning of the loop.

Termination expression determines when to terminate the loop. When the expression evaluates to false, the loop terminates. Finally, update is an expression that gets invoked after each iteration through the loop.