Chapter 2

BASIC ELEMENTS IN C++

This chapter gets you started immediately writing some simple C++ programs and helps you to understand some basic elements in any high level programming language such as data types, arithmetic operators, output statements and assignment statements.

2.1 PROGRAM STRUCTURES

Modular Programs

A large program should be organized as several interrelated segments, arranged in a logical order: The segments are called *modules*. A program which consists of such modules is called a modular program.

In C++, modules can be *classes* or *functions*.

We can think of a *function* as a program segment that transforms the data it receives into a finished result.

Each function must have a name. Names or *identifiers* in C++ can made up of any combination of letters, digits, or underscores selected according to the following rules:

- Identifiers must begin within an uppercase or lowercase ASCII letter or an underscore (_).
- You can use digits in an identifier, but not as the first character. You are not allowed to use special characters such as $, &, * or %.
- Reserved words cannot be used for variable names.

Examples:

DegToRad	intersect	addNums
FindMax1	_density	slope

Examples of invalid identifiers:

1AB3
E%6
while

Note: C++ is a case-sensitive language (i.e. upper and lower case characters are treated as different letters).

The *main()* function

The *main()* function is a special function that runs automatically when a program first executes.

All C++ programs must include one *main()* function. All other functions in a C++ program are executed from the *main()* function.

The first line of the function, in this case *int main()* is called a *function header line*.

The function header line contains three pieces of information:

1. What type of data, if any, is returned from the function.
2. The name of the function
3. What type of data, if any, is sent into the function.

```
int main()
{

   program statements in here
   return 0;
}
```

Note: The line

```
return 0;
```

is included at the end of every *main* function. 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 terminates successfully.

The cout Object

The *cout* object is an output object that sends data given to it to the standard output display device.

To send a message to the *cout* object, you use the following pattern:

```
cout << "text";
```

The *insertion operator*, <<, is used for sending text to an output device.

The text portion of the statement is called a *text string*. Text string is text that is contained within double quotation marks.

Consider the following program.

Example 2.1.1

```
#include <iostream.h>
int main()
{
  cout << "Hello world!";
  return 0;
}
```

The output of the above program:

```
Hello world!
```

Preprocessor Directives

Before you can use any runtime libraries in your program, you must first add a *header-file* into your program, using the *#include* statement. A *header file* is a file with an extension of *.h* that is included as part of a program and notifies the compiler that a program uses run-time libraries.

One set of classes you will use extensively in the next few chapters is the *iostream* classes. The *iostream* classes are used for giving C++ programs input capabilities and output capabilities.

The header file for the iostream class is *iostream.h*.

The *#include* statement is one of the several preprocessor directives that are used with C++.

The *preprocessor* is a program that runs before the compiler. When it encounters an *#include* statement, the preprocessor places the entire contents of the designated file into the current file.

Preprocessor directives and *include* statements allow the current file to use any of the *classes*, *functions*, *variables*, and *other code* contained within the included file.

Example: To include the *iostream.h* file you use the following statement:

```
#include <iostream.h>
```

An *i/o manipulator* is a special function that can be used with an i/o statement. The *endl* i/o manipulator is part of the *iostream* classes and represents a *new line* character.

Example:

```
cout << "Program type: console application" << endl;
cout << "Create with: Visual C++ "<< endl;
cout << "Programmer: Don Gesselin" << endl;
```

All statements in C++ must end with a semicolon. Large statements can span multiple lines of code.

Example:

```
cout << "Program type: console application,"
     << "Create with: Visual C++ "
     << "Programmer: Don Gesselin";
```

Comments

Comments are lines that you place in your code to contain various type of remarks. C++ support two types of comments: line and block.

C++ line comments are created by adding two slashes (//) before the text you want to use as a comment.

Block comments span multiple lines. Such comments begin with /* and end with the symbols */.

Example:

```
void main()
{
   /*
     This line is part of the block comment.
     This line is also part of the block
      comment.
    */
    cout << "Line comment 1 ";
    cout << "Line comment 2 ";
    // This line comment takes up an entire line.
}
```

All programs should contain comments. They are remarks, insights, wisdom in code without affecting the program. The compiler ignores comments.

2.2 DATA TYPES AND OPERATORS

2.2.1 Data Types

A *data type* is the specific category of information that a variable contains.

There are three basic data types used in C++: integers, floating point numbers and characters.

Integers

An integer is a positive or negative number with no decimal places.

- 259 -13 0 200

Floating Point Numbers

A *floating point* number contains decimal places or is written using exponential notations.

-6.16 -4.4 2.7541 10.5

Exponential notation, or scientific notation is a way of writing a very large numbers or numbers with many decimal places using a shortened format.

2.0e11 means 2*1011

C++ supports three different kinds of floating-point numbers:

- float (i.e. single precision numbers),
- double (i.e. double precision numbers)
- long double.

A double precision floating-point number can contain up to 15 significant digits.

The Character Data Type

To store text, you use the character data type. To store one character in a variable, you use the *char* keyword and place the character in single quotation marks.

Example:

```
char cLetter = 'A';
```

Escape Sequence

The combination of a *backlash* (\) and a special character is called an *escape sequence*. When this character is placed directly in front of a select group of character, it tells the compiler to escape from the way these characters would normally be interpreted.

Examples:

\n move to the next line
\t move to the next tab

The bool Data Type

The C++ *bool* type can have two states expressed by the built-in constants *true* (which converts to an integral one) and *false* (which converts to an integral zero). All three names are keywords. This data type is most useful when a program must examine a specific condition and, as a result of the condition being either true or false, take a prescribed course of action.

2.2.2 Arithmetic Operators

Most programs perform arithmetic calculations. Arithmetic operators are used to perform mathematical calculations, such as addition, subtraction, multiplication, and division in C++.

Operator	Description
+	Add two operands
-	Subtracts one operand from another operand
*	Multiplies one operand by another operand
/	Divides one operand by another operand
%	Divides two operands and returns the remainder

A simple arithmetic expression consists of an arithmetic operator connecting two operands in the form:

operand operator operand

Examples:

3 + 7
18 – 3
12.62 + 9.8
12.6/2.0

Example 2.2.1

```
#include <iostream.h>
int main()
{
  cout << "15.0 plus 2.0 equals "       << (15.0 + 2.0) << '\n'
       << "15.0 minus 2.0 equals "      << (15.0 - 2.0) << '\n'
       << "15.0 times 2.0 equals "      << (15.0 * 2.0) << '\n'
       << "15.0 divided by 2.0 equals " << (15.0 / 2.0) << '\n';
  return 0;
}
```

The output of the above program:

```
15.0 plus 2.0 equals  17
```

15.0 minus 2.0 equals 13
15.0 times 2.0 equals 30
15.0 divided by 2.0 equals 7.5

Integer Division

The division of two integers yields integer result. Thus the value of 15/2 is 7.

Modulus % operator produces the remainder of an integer division.

Example:

9%4 is 1
17%3 is 2
14%2 is 0

Operator Precedence and Associativity

Expressions containing multiple operators are evaluated by the priority, or *precedence*, of the operators.

Operator precedence defines the order in which an expression evaluates when several different operators are present. C++ have specific rules to determine the order of evaluation. The easiest to remember is that multiplication and division happen before addition and subtraction.
The following table lists both precedence and associativity of the operators.

Operator	Associativity
unary -	Right to left
* / %	Left to right
+ -	Left to right

Example: Let us use the precedence rules to evaluate an expression containing operators of different precedence, such as 8 + 5*7%2*4. Because the multiplication and modulus operators have a higher precedence than the addition operator, these two operations are evaluated first (P2), using their left-to-right associativity, before the addition is evaluated (P3). Thus, the complete expression is evaluated as:

```
8 + 5 * 7 % 2 * 4 =
  8 + 35 % 2 * 4 =
      8 + 1 * 4  =
        8 + 4    = 12
```

Expression Types

An expression is any combination of operators and operands that can be evaluated to yield a value. An expression that contains only integer values as operands is called an *integer expression*, and the result of the expression is an integer value. Similarly, an expression containing only floating-point values (single and double precision) as operands is called a *floating-point expression*, and the result of the expression is a floating point value (the term *real expression* is also used).

2.3 VARIABLES AND DECLARATION STATEMENTS

One of the most important aspects of programming is storing and manipulating the values stored in *variables*. A variable is simply a name chosen by the programmer that is used to refer to computer storage locations. The term *variable* is used because the value stored in the variable can change, or vary.

Variable names are also selected according to the rules of identifiers:

- Identifiers must begin with an uppercase or lowercase ASCII letter or an underscore (_).
- You can use digits in an identifier, but not as the first character. You are not allowed to use special characters such as $, &, * or %.
- Reserved words cannot be used for variable names.

Example: Some valid identifiers

my_variable
Temperature
x1
x2
_my_variable

Some invalid identifiers are as follows:

%x1 %my_var @x2

We should always give variables meaningful names, from which a reader might be able to make a reasonable guess at their purpose. We may use comments if further clarification is necessary.

Declaration Statements

Naming a variable and specifying the data type that can be stored in it is accomplished using *declaration statement*. A declaration statement in C++ has the following syntax:

type name;

The *type* portion refers to the data type of the variable.

The data type determines the type of information that can be stored in the variable.

Example:

```
int sum;
long datenem;
double secnum;
```

Note:

1. A variable must be declared before it can be used.
2. Declaration statements can also be used to store an initial value into declared variables.

Example:

```
int num = 15;
float grade1 = 87.0
```

Variable declarations are just the instructions that tell the compiler to allocate memory locations for the variables to be used in a program.

A variable declaration creates a memory location but it is *undefined* to start with, that means it's empty.

Example 2.3.1

```
#include <iostream.h>
int main()
{
  float price1 = 85.5;
  float price2 = 97.0;
  float total, average;

  total = price1 + price2;
  average = total/2.0;  // divide the total by 2.0
  cout << "The average price is " << average << endl;
  return 0;
}
```

The output of the above program:

```
The average price is 91.25
```

Let notice the two statements in the above program:

```
total = price1 + price2;
average = total/2.0;
```

Each of these statements is called an *assignment statement* because it tells the computer to assign (store) a value into a variable. Assignment statements always have an equal (=)

sign and one variable name on the left of this sign. The value on the right of the equal sign is assigned to the variable on the left of the equal sign.

Display a Variable's Address

Every variable has three major items associated with it: its data type, its actual value stored in the variable and the address of the variable. The value stored in the variable is referred to as the variable's contents, while the address of the first memory location used for the variable constitutes its address.

To see the address of a variable, we can use *address operator*, &, which means "the address of ". For example, *&num* means the address of *num*.

Example 2.3.2

```
#include <iostream.h>
int main()
{
  int a;

  a = 22;
  cout << "The value stored in a is " << a << endl;
  cout << "The address of a = " << &a << endl;
  return 0;
}
```

The output of the above program:

```
The value stored in a is 22
The address of a = 0x0065FDF4
```

The display of addresses is in hexadecimal notation.

2.4 INTEGER QUANTIFIERS

Portable languages like C++ must have flexible data type sizes. Different applications might need integers of different sizes. C++ provides *long integer*, *short integer*, and *unsigned integer* data types. These three additional integer data types are obtained by adding the quantifier *long*, *short* or *unsigned* to the normal integer declaration statements.

Example:

```
long int days;
unsigned int num_of_days;
```

The reserved words *unsigned int* are used to specify an integer that can only store nonnegative numbers.

The *signed* and *unsigned* quantifiers tell the compiler how to use the sign bit with integral types and characters (floating-point numbers always contain a sign). An unsigned number does not keep track of the sign and thus has an extra bit available, so it can store positive numbers twice as large as the positive numbers that can be stored in a signed number.

Data type	Storage	Number Range
short int	2 bytes	-32768 to 32767
unsigned int	2 bytes	0 to 65535
long int	4 bytes	-2,147,483,648 to -2,147,483,647

When you are modifying an *int* with *short* or long, the keyword *int* is optional.

Now all the built-in data types provide by C++ are given in the following list, ordered descendingly by the size of the data types.

Data types
---
long double
double
float
unsigned long
long int
unsigned int
int
short in
char

Data Type Conversions

An expression containing both integer and floating point operands is called a *mixed mode expression.*

Example:

```
        int a;
float x = 2.5;
        a = x + 6;          // x + 6 is a mixed mode expression
```

The general rules for converting integer and floating point operands in mixed mode arithmetic expressions were presented as follows:

1. If both operands are either character or integer operands:

a. when both operands are character, short or integer data types, the result of the expression is an integer value.
b. when one of the operand is a long integer, the result is a long integer, unless one of the operand is an unsigned integer. In the later case, the other operand is converted to an unsigned integer value and the resulting value of the expression is an unsigned value.

2. If any one operand is a floating point value:
   a. when one or both operands are floats, the result of the operation is a float value;
   b. when one or both operands are doubles, the result of the operation is a double value;
   c. when one or both operands are long doubles, the result of the operation is a long double value;

Notice that converting values to lower types can result in incorrect values. For example, the floating point value 4.5 gives the value 4 when it is converted to an integer value. The following table lists the built-in data types in order from "highest type" to "lowest type".

Determining Storage Size

C++ provides an operator for determining the amount of storage your compiler allocates for each data type. This operator, called the *sizeof()* operator.

Example:

```
sizeof(num1)
sizeof(int)
sizeof(float)
```

The item in parentheses can be a variable or a data type.

Example 2.4.1

```
// Demonstrating the sizeof operator
#include <iostream.h>
int main()
{
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  long double ld;
  cout << "sizeof c = " << sizeof(c)
      << "\tsizeof(char) = " << sizeof( char )
      << "\nsizeof s = " << sizeof(s)
```

```
        << "\tsizeof(short) = " << sizeof( short )
        << "\nsizeof i = " << sizeof (i)
        << "\tsizeof(int) = " << sizeof( int )
        << "\nsizeof l = " << sizeof(l)
        << "\tsizeof(long) = " << sizeof( long )
        << "\nsizeof f = " << sizeof (f)
        << "\tsizeof(float) = " << sizeof(float)
        << "\nsizeof d = " << sizeof (d)
        << "\tsizeof(double) = " << sizeof(double)
        << endl;
   return 0;
}
```

The output of the above program:

```
sizeof c = 1        sizeof(char) = 1
sizeof s = 2        sizeof(short) =  2
sizeof i = 4        sizeof(int) =  4
sizeof l = 4        sizeof(long) =  4
sizeof f = 4        sizeof(float) =  4
sizeof d = 8        sizeof(double) =  8
```