Chapter Two
Basic Concepts of C++ Programming

2.1. Variables, Constants, Initializing Variables

2.1.1. Variables

A variable is a symbolic name for a memory location in which data can be stored and subsequently recalled. Variables are used for holding data values so that they can be utilized in various computations in a program.

- A variable is a reserved place in memory to store information in.
- Variables are used for holding data values so that they can be used in various computations in a program.

All variables have three important properties:

- Data Type: a type which is established when the variable is defined. (e.g. integer, real, character etc). Data type describes the property of the data and the size of the reserved memory
- Name: a name which will be used to refer to the value in the variable. A unique identifier for the reserved memory location
- Value: a value which can be changed by assigning a new value to the variable.

2.1.1.1. Variable Declaration

Variables can be created in a process known as declaration. Declaring a variable means defining (creating) a variable. You create or define a variable by stating its type, followed by one or more spaces, followed by the variable name and a semicolon. The variable name can be virtually any combination of letters, but cannot contain spaces and the first character must be a letter or an underscore.

Syntax: Datatype Variable_Name;

Variable names cannot also be the same as keywords used by C++. Legal variable names include x, J23f, and myAge. Good variable names tell you what the variables are for; using good names makes it easier to understand the flow of your program. The following statement defines an integer variable called myAge:

int myAge;

IMPORTANT-Variables must be declared before used!

As a general programming practice, avoid such horrific names as J23qrsnf, and restrict single-letter variable names (such as x or i) to variables that are used only very briefly. Try to use expressive names such as myAge or howMany.

A point worth mentioning again here is that C++ is case-sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable named age is different from Age, which is different from AGE.

- The name of a variable sometimes is called an identifier which should be unique in a program.
- Certain words are reserved by C++ for specific purposes and cannot be used as identifiers.

Creating More Than One Variable at a Time

You can create more than one variable of the same type in one statement by writing the type and then the variable names, separated by commas. For example:

Int myAge, myWeight; // two int variables

long area, width, length; // three longs

As you can see, myAge and myWeight are each declared as integer variables. The second line declares three individual long variables named area, width, and length. However keep in mind that you cannot mix types in one definition statement.

2.5.1.2. Signed and Unsigned.

- Signed integers are either negative or positive. Unsigned integers are always positive.
- Because both signed and unsigned integers require the same number of bytes, the largest number (the magnitude) that can be stored in an unsigned integer is twice as the largest positive number that can be stored in a signed integer.

E.g.: Lets us have only 4 bits to represent numbers

Unsigned					***Signed***				
Binary				Decimal	Binary				Decimal
0	0	0	0	→0	0	0	0	0	→0
0	0	0	1	→1	0	0	0	1	→1
0	0	1	0	→2	0	0	1	0	→2
0	0	1	1	→3	0	0	1	1	→3
0	1	0	0	→4	0	1	0	0	→4
0	1	0	1	→5	0	1	0	1	→5
0	1	1	0	→6	0	1	1	0	→6
0	1	1	1	→7	0	1	1	1	→7
1	0	0	0	→8	1	0	0	0	→0
1	0	0	1	→9	1	0	0	1	→ -1
1	0	1	0	→10	1	0	1	0	→ -2
1	0	1	1	→11	1	0	1	1	→ -3
1	1	0	0	→12	1	1	0	0	→ -4
1	1	0	1	→13	1	1	0	1	→ -5
1	1	1	0	→14	1	1	1	0	→ -6
1	1	1	1	→15	1	1	1	1	→ -7

- In the above example, in case of unsigned, since all the 4 bits can be used to represent the magnitude of the number the maximum magnitude that can be represented will be 15 as shown in the example.
- If we use signed, we can use the first bit to represent the sign where if the value of the first bit is 0 the number is positive if the value is 1 the number is negative. In this case we will be left with only three bits to represent the magnitude of the number. Where the maximum magnitude will be 7.
- Because you have the same number of bytes for both signed and unsigned integers, the largest number you can store in an unsigned integer is twice as big as the largest positive number you can store in a signed integer. An unsigned short integer can handle numbers from 0 to 65,535. Half the numbers represented by a signed short are negative, thus a signed short can only represent numbers from -32,768 to 32,767.

Example: A demonstration of the use of variables.

```
2: #include <iostream.h>
3:
4: intmain()
5: {
6: unsigned short int Width = 5, Length;
7: Length = 10;
8:
9: // create an unsigned short and initialize with result
10: // of multiplying Width by Length
11: unsigned short intArea = Width * Length;
12:
13: cout<< "Width:" << Width << "\n";
14: cout<< "Length: " << Length <<endl;
15: cout<< "Area: " << Area <<endl;
16: return 0;
17: }
```

Output: Width:5
Length: 10
Area: 50

Line 2 includes the required include statement for the iostream's library so that cout will work. Line 4 begins the program.

On line 6, Width is defined as an unsigned short integer, and its value is initialized to 5. Another unsigned short integer, Length, is also defined, but it is not initialized. On line 7, the value 10 is assigned to Length.

On line 11, an unsigned short integer, Area, is defined, and it is initialized with the value obtained by multiplying Width times Length. On lines 13-15, the values of the variables are printed to the screen. Note that the special word endl creates a new line.

2.1.2. Constants

- A constant is any expression that has a fixed value.
- Like variables, constants are data storage locations in the computer memory. But, constants, unlike variables their content cannot be changed after the declaration.
- Constants must be initialized when they are created by the program, and the programmer can't assign a new value to a constant later.
- In C++, we have two ways to declare a symbolic constant. These are using the #define and the const key word.

2.1.2.1. Defining constants with #define:

- The #define directive makes a simple text substitution.
- The define directive can define only integer constants

  E.g.: #define studentPerClass 15

- In our example, each time the preprocessor sees the word studentPerClass, it inserts 15 into the text.

2.1.2.2. Defining constants with the const key word:

- Here, the constant has a type, and the compiler can ensure that the constant is used according to the rules for that type.

    E.g.: const unsigned short intstudentPerClass = 15;

2.1.3. Initializing Variables

- When a variable is assigned a value at the time of declaration, it is called variable initialization.
- This is identical with declaring a variable and then assigning a value to the variable immediately after declaration.
- The syntax: DataType variable name = initial value;

    e.g. int a = 0;
    or: int a;
    a=0;

2.2. Data Types

When you define a variable in C++, you must tell the compiler what kind of variable it is: an integer, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable.

Basic (fundamental) data types in C++ can be conveniently divided into numeric and character types. Numeric variables can further be divided into integer variables and floating-point variables. Integer variables will hold only integers whereas floating number variables can accommodate real numbers.

Both the numeric data types offer modifiers that are used to vary the nature of the data to be stored. The modifiers used can be short, long, signed and unsigned.

The data types used in C++ programs are described in the following table. This table shows the variable type, how much room it takes in memory, and what kinds of values can be stored in these variables. The values that can be stored are determined by the size of the variable types.

Table 2.1. Data types and their ranges

Type	Size	Values
unsigned short int	2 bytes	0 to 65,535
short int(signed short int)	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int(signed long int)	4 bytes	-2,147,483,648 to 2,147,483,647
Int	2 bytes	-32,768 to 32,767
unsigned int	2 bytes	0 to 65,535
signed int	2 bytes	-32,768 to 32,767
Char	1 byte	256 character values
Float	4 bytes	3.4e-38 to 3.4e38
Double	8 bytes	1.7e-308 to 1.7e308
long double	10 bytes	1.2e-4932 to 1.2e4932

2.3. Assigning Values to Variables

You assign a value to a variable by using the assignment operator (=). Thus, you would assign 5 to Width by writing:

```
int Width;
Width = 5;
```

You can combine these steps and initialize Width when you define it by writing:

```
Int Width=5;
```

Initialization looks very much like assignment, and with integer variables, the difference is minor. The essential difference is that initialization takes place at the moment you create the variable.

Just as you can define more than one variable at a time, you can initialize more than one variable at creation. For example:

```
// create two int variables and initialize them

int width = 5, length = 7;
```

This example initializes the integer variable width to the value 5 and the length variable to the value 7. It is possible to even mix definitions and initializations:

```
int myAge = 39, yourAge, hisAge = 40;
```

This example creates three type int variables, and it initializes the first and third.

2.4. Expressions, Comments, Statements, Identifier, Keywords

2.4.1. Expressions

An expression is a computation which yields a value. It can also be viewed as any statement that evaluates to a value (returns a value).

E.g.: the statement 3+2; returns the value 5 and thus is an expression.

- Some examples of an expression:

E.g.1:
3.2 returns the value 3.2
PI float constant that returns the value 3.14 if the constant is defined.
secondsPerMinute integer constant that returns 60 if the constant is declared
E.g.2: complicated expressions:
x = a + b;
y = x = a + b;
The second line is evaluated in the following order:
1. add a to b.
2. assign the result of the expression a + b to x.
3. assign the result of the assignment expression x = a + b to y.

2.4.2. Comments

- A comment is a piece of descriptive text which explains some aspect of a program.

- Program comments are text totally ignored by the compiler and are only intended to inform the reader how the source code is working at any particular point in the program.

C++ provides two types of comment delimiters:

- Single Line Comment: Anything after // {double forward slash} (until the end of the line on which it appears) is considered a comment.
    - E.g.: cout<<var1; //this line prints the value of var1
- Multiple Line Comment: Anything enclosed by the pair /* and */ is considered a comment.

E.g.:

```
/*this is a kind of comment where
Multiple lines can be enclosed in
one C++ program */
```

Comments should be used to enhance (not to hinder) the readability of a program. The following points, in particular, should be noted:

- A comment should be easier to read and understand than the code which it tries to explain. A confusing or unnecessarily-complex comment is worse than no comment at all.
- Over-use of comments can lead to even less readability. A program which contains so much comment that you can hardly see the code can by no means be considered readable.
- Use of descriptive names for variables and other entities in a program, and proper indentation

2.4.3. Statements

Statements represent the lowest-level building blocks of a program. Roughly speaking, each statement represents a computational step which has a certain side-effect. (A side-effect can be thought of as a change in the program state, such as the value of a variable changing because of an assignment.) Statements are useful because of the side-effects they cause, the combination of which enables the program to serve a specific purpose. A running program spends all of its time executing statements. The order in which statements are executed is called flow control (or control flow). This term reflect the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement. Flow control in a program is typically sequential, from one statement to the next, but may be diverted to other paths by branch statements.

Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program. Like many other procedural languages, C++ provides different forms of statements for different purposes. Declaration statements are used for defining variables. Assignment-like statements are used for simple, algebraic computations. Branching statements are used for specifying alternate paths of execution, depending on the outcome of a logical condition. Loop statements are used for specifying computations which need to be repeated until a certain logical condition is satisfied. Flow control statements are used to divert the execution path to another part of the program.

- In C++, a statement controls the sequence of execution, evaluates an expression, or does nothing (the null statement).
- All C++ statements end with a semicolon.

E.g.: x = a + b; //The meaning is: assign the value of the sum of a and b to x.

- White spaces: white spaces characters (spaces, tabs, new lines) can't be seen and generally ignored in statements. White spaces should be used to make programs more readable and easier to maintain.
- Blocks: a block begins with an opening French brace ({) and ends with a closing French brace (}).

2.4.4. Identifiers

An identifier is name associated with a function or data object and used to refer to that function or data object. An identifier must:

- Start with a letter or underscore
- Consist only of letters, the digits 0-9, or the underscore symbol _
- Not be a reserved word

For the purposes of C++ identifiers, the underscore symbol, _, is considered to be a letter. Its use as the first character in an identifier is not recommended though, because many library functions in C++ use such identifiers. Similarly, the use of two consecutive underscore symbols, _ _, is forbidden.

The following are valid identifiers

Length	days_in_year	DataSet1	Profit95
Int	_Pressure	first_one	first_1

Although using _Pressure is not recommended.

The following are invalid:

days-in-year	1data	int	first.val
throw	my__best	No##	bestWish!

Although it may be easier to type a program consisting of single character identifiers, modifying or correcting the program becomes more and more difficult. The minor typing effort of using meaningful identifiers will repay itself many fold in the avoidance of simple programming errors when the program is modified.

2.4.5. Keywords

Reserved/Key words have a unique meaning within a C++ program. These symbols, the reserved words, must not be used for any other purposes. All reserved words are in lowercase letters. The following are some of the reserved words of C++.

Table 2.2. Keywords in C++

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while

Notice that main is not a reserved word. However, this is a fairly technical distinction, and for practical purposes you are advised to treat main, cin, and cout as if they were reserved as well.

2.5. Operators and Operator Precedence

Operators

- An operator is a symbol that makes the machine to take an action.
- Different Operators act on one or more operands and can also have different kinds of operators.
- C++ provides several categories of operators, including the following:

Assignment operator (=).

- ✓ The assignment operator causes the operand on the left side of the assignment statement to have its value changed to the value on the right side of the statement.
- ✓ Syntax: Operand1=Operand2;
- ✓ Operand1 is always a variable
- ✓ Operand2 can be one or combination of:

  - A literal constant: Eg: x=12;
  - A variable: Eg: x=y;
  - An expression: Eg: x=y+2;

Compound assignment operators (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=).

- ✓ Compound assignment operator is the combination of the assignment operator with other operators like arithmetic and bit wise operators.
- ✓ The assignment operator has a number of variants, obtained by combining it with other operators. E.g.: value += increase; is equivalent to value = value + increase;

  a -= 5; is equivalent to a= a– 5;
  a /= b; is equivalent to a= a/ b;
  price *= units + 1 is equivalent to price = price * (units + 1);

- ✓ And the same is true for the rest.

Arithmetic operators (+, -, *, /, %).

- ✓ Except for remainder or modulo (%), all other arithmetic operators can accept a mix of integers and real operands. Generally, if both operands are integers then, the result will be an integer. However, if one or both operands are real then the result will be real.
- ✓ When both operands of the division operator (/) are integers, then the division is performed as an integer division and not the normal division we are used to.
- ✓ Integer division always results in an integer outcome.
- ✓ Division of integer by integer will not round off to the next integer

  E.g.:
  9/2 gives 4 not 4.5
  -9/2 gives -4 not -4.5

- ✓ To obtain a real division when both operands are integers, you should cast one of the operands to be real. E.g.: int cost = 100;
  Int volume = 80;
  Double unitPrice = cost/(double)volume;

- ✓ The module(%) is an operator that gives the remainder of a division of two integer values. For instance, 13 % 3 is calculated by integer dividing 13 by 3 to give an outcome of 4 and a remainder of 1; the result is therefore 1.

  E.g.: a = 11 % 3 ,a is 2

  Relational operator (==, !=, > , <, >=, <=).

- ✓ In order to evaluate a comparison between two expressions, we can use the relational operator.
- ✓ The result of a relational operator is a bool value that can only be true or false according to the result of the comparison. E.g.:

  (7 = = 5) would return false or returns 0
  (5 > 4) would return true or returns 1
- ✓ The operands of a relational operator must evaluate to a number. Characters are valid operands since they are represented by numeric values. For E.g.:

  'A' < 'F' would return true or 1. it is like (65 < 70)

  Logical Operators (!,&&, ||):

- ✓ Logical negation (!) is a unary operator, which negates the logical value of its operand. If its operand is non zero, it produce 0, and if it is 0 it produce 1.
- ✓ Logical AND (&&) produces 0 if one or both of its operands evaluate to 0 otherwise it produces 1.
- ✓ Logical OR (||) produces 0 if both of its operands evaluate to 0 otherwise, it produces 1. E.g.:

  !20 //gives 0
  10 && 5 //gives 1
  10 || 5.5 //gives 1
  10 && 0 // gives 0

N.B. In general, any non-zero value can be used to represent the logical true, whereas only zero represents the logical false.

Increment/Decrement Operators: (++) and (--)

The auto increment (++) and auto decrement (--) operators provide a convenient way of, respectively, adding and subtracting 1 from a numeric variable. E.g.:

if a was 10 and if a++ is executed then a will automatically changed to 11.

Prefix and Postfix:

- ✓ The prefix type is written before the variable. Eg (++ myAge), whereas the postfix type appears after the variable name (myAge ++).
- ✓ Prefix and postfix operators can not be used at once on a single variable:

  E.g.: ++age-- or --age++ or ++age++ or - - age - - is invalid

- ✓ In a simple statement, either type may be used. But in complex statements, there will be a difference.
- ✓ The prefix operator is evaluated before the assignment, and the postfix operator is evaluated after the assignment.

E.g. int k = 5;

(auto increment prefix) y= ++k + 10; //gives 16 for y
(auto increment postfix) y= k++ + 10; //gives 15 for y
(auto decrement prefix) y= --k + 10; //gives 14 for y
(auto decrement postfix) y= k-- + 10; //gives 15 for y

Conditional Operator (?:)

The conditional operator takes three operands. It has the general form:

Syntax:

operand1 ?operand2 : operand3

- ✓ First operand1 is a relational expression and will be evaluated. If the result of the evaluation is nonzero (which means TRUE), then operand2 will be the final result. Otherwise, operand3 is the final result.

E.g.: General Example
Z=(X<Y? X : Y)

This expression means that if X is less than Y the value of X will be assigned to Z otherwise (if X>=Y) the value of Y will be assigned to Z.

E.g.:
int m=1,n=2,min;
min = (m < n? m : n);
The value stored in min is 1.
E.g.:
(7 = = 5 ? 4: 3) returns 3since 7 is not equal to 5

Comma Operator (,).

- ✓ Multiple expressions can be combined into one expression using the comma operator.
- ✓ The comma operator takes two operands. Operand1,Operand2
- ✓ The comma operator can be used during multiple declaration, for the condition operator and for function declaration, etc
- ✓ It the first evaluates the left operand and then the right operand, and returns the value of the latter as the final outcome.

E.g.
int m,n,min;
int mCount = 0, nCount = 0;
min = (m < n ? (mCount++ , m) : (nCount++ , n));

- ✓ Here, when m is less than n, mCount++ is evaluated and the value of m is stored in min. otherwise, nCount++ is evaluated and the value of n is stored in min.

The sizeof() Operator.

- ✓ This operator is used for calculating the size of any data item or type.
- ✓ It takes a single operand (e.g. 100) and returns the size of the specified entity in bytes. The outcome is totally machine dependent. E.g.:

a = sizeof(char)
b = sizeof(int)
c = sizeof(1.55) etc

Explicit type casting operators.

- ✓ Type casting operators allows you to convert a datum of a given type to another data type.

E.g.
int i;
float f = 3.14;
i = (int)f; equivalent to i = int(f);

Then variable i will have a value of 3 ignoring the decimal point

Operator Precedence

The order in which operators are evaluated in an expression is significant and is determined by precedence rules. Operators in higher levels take precedence over operators in lower levels.

Table 2.3.operator precedence

Level	Operator	Order
Highest	++ -- (post fix)	Right to left
	sizeof() ++ -- (prefix)	Right to left
	* / %	Left to right
	+ -	Left to right
	<<= >>=	Left to right
	== !=	Left to right
	&&	Left to right
	||	Left to right
	? :	Left to right
	= ,+=, -=, *=, /=,^= ,%=, &= , |= ,<<= ,>>=	Right to left
	,	Left to right

E.g.
a = = b + c * d
c * d is evaluated first because * has a higher precedence than + and = =.
The result is then added to b because + has a higher precedence than = =
And then == is evaluated.

- ✓ Precedence rules can be overridden by using brackets.

E.g. rewriting the above expression as:

a = = (b + c) * d causes + to be evaluated before *.

- ✓ Operators with the same precedence level are evaluated in the order specified by the column on the table of precedence rule.

E.g. a = b += c the evaluation order is right to left, so the first b += c is evaluated followed by a = b.

2.6. Debugging and Programming Errors

Programming is a complex process, and since it is done by human beings, it often leads to errors.This makes debugging a fundamental skill of any programmer as debugging is an intrinsic part of programming. Programming errors are called bugs and going through the code, examining it and looking for something wrong in the implementation (bugs) and correcting them is called debugging.

Often times the program doesn't work as planned, and won't compile properly. Even the best programmers make mistakes, being able to identify what you did wrong is essential. There are two types of errors that exist; those that the C++ compiler can catch on its own, and those that the compiler can't catch. Errors that C++ can catch are known as compiler-time errors. Compiler-time errors should be relatively easy to fix, because the compiler points you to where the problem is. All that garbage that's spit out by the compiler has some use. Here's an example.

```
1 int main()
2 {
3 return 0
4 }
```

Your compiler should generate an error something like… \main.cpp(3) : error C2143: syntax error : missing ';' before '}' Compiler errors differ from compiler to compiler, but it's all going to generally be the same. Now lets take this compiler error apart. The first part of it \main.cpp(4) says that the error is in the file main.cpp, on line 4. After that is error C2143: That's the compiler specific error code. After that the error states syntax error : Which tells you that you messed up some syntax. So you must not have typed something right. Then it tells you missing ';' before '}' There's a missing semi-colon before a closing bracket. Acknowledging a compiler error should be as easy as that. The other type of error that C++ doesn't catch is called a run-time error. Run-time errors are errors often much more tricky to catch.

There's several ways that one can debug a program. The two that I use most often are the WRITE technique, and single-step debugging. The WRITE technique involves creating output statements for all of the variables, so you can see the value of everything.

For smaller program, the WRITE technique works reasonably well, but as things get larger, it's harder to output all your variables, and it just starts to seem like a waste of time. Instead, we'll rely on the debugger. A debugger is a tool built into most development environments (and although they differ, most debuggers work on the same principles.). A programmer controls the debugger through commands by the means of the same interface as the editor. You can access these commands in menu items or by using hotkeys. The debugger allows the programmer to control the execution of his/her program. He/she can execute one step at a time in the program, he/she can stop the program at any point, and he/she can examine the value of variables.

2.7. Basic Input and Output in C++, Formatted Input-Output

The most common way in which a program communicates with the outside world is through simple, character-oriented Input/Output (IO) operations. C++ provides two useful operators for this purpose: >> for input and << for output. The following example illustrates the use of >> for input and << for output.

```
1: #include <iostream.h>
2: int main (void)
3: {
```

```
4: int workDays = 5;
5: float workHours = 7.5;
6: float payRate, weeklyPay;
7:
8: cout<< "What is the hourly pay rate? ";
9: cin>>payRate;
10:
11: weeklyPay = workDays * workHours * payRate;
12: cout<< "Weekly Pay = ";
13: cout<<weeklyPay;
14: cout<< '\n';
15:}
```

Line 8 outputs the prompt 'What is the hourly pay rate? ' to seek user input. Line 9 reads the input value typed by the user and copies it to payRate. The input operator >> takes an input stream as its left operand (cin is the standard C++ input stream which corresponds to data entered via the keyboard) and a variable (to which the input data is copied) as its right operand.

When run, the program will produce the following output (user input appears in bold):

What is the hourly payrate? 33.55
WeeklyPay=1258.125

Both << and >> return their left operand as their result, enabling multiple input or multiple output operations to be combined into one statement.

- Cout is an object used for printing data to the screen.
- To print a value to the screen, write the word cout, followed by the insertion operator also called output redirection operator (<<) and the object to be printed on the screen.
- Syntax: Cout<<Object;
- The object at the right hand side can be:
    - A literal string: "Hello World"
    - A variable: a place holder in memory
- Cin is an object used for taking input from the keyboard.
- To take input from the keyboard, write the word cin, followed by the input redirection operator (>>) and the object name to hold the input value.
- Syntax: Cin>>Object
- Cin will take value from the keyboard and store it in the memory. Thus the cin statement needs a variable which is a reserved memory place holder.
- Both << and >> return their right operand as their result, enabling multiple input or multiple output operations to be combined into one statement. The following example will illustrate how multiple input and output can be performed: E.g.:

Cin>>var1>>var2>>var3;

Here three different values will be entered for the three variables. The input should be separated by a space, tan or newline for each variable. '

Cout<<var1<<", "<<var2<<" and "<<var3;

Here the values of the three variables will be printed where there is a "," (comma) between the first and the second variables and the "and" word between the second and the third.