SESSION 1-2: Operators and Expressions

1-2.1 Assignment Statements

An assignment statement is a means by which identifiers are given values or a way of changing the value stored in a variable. The assignment statement always has a variable (receiving field) to the left of the assignment sign (=) and an expression on the right hand side. The expression can be a variable and/or constants with operators. Operators are special symbols used for mathematical, assignment and logical expressions. The syntax of arithmetic statements is as follows

```
receiving_field=expression;
```

where the *receiving_field* is the identifier to be assigned a value and expression is the value to be assigned to the receiving field. Examples are as follows:

```
profit = SellingPrice – CostPrice;      exression
XchangeRate = 18000;                    constant
Hold = department;                      variable
```

In an assignment statement such as the two above, the left part of the = is called the **lvalue** and the right hand side is called the **rvalue**. The lvalue should always be a variable and never a constant.

To initialise a number of variables to the same value, one can also use the following syntax. This is also true for assigning a value or a constant to a number of variables.

```
variable1=variable2=variable3=…=value;
```

where *variable1*, *variable2*, *variable3*, etc are variables and value is their initialisation.

For example, we can have

```
sumx=sumy=sumxy=sumxsqd=0.0;
```

will initialise *sumx*, *sumy*, *sumxy* and *sumxsqd* to zeros. Thus the above statement is equivalent to the following:

```
sumx=0;
sumy=0;
sumxy=0;
sumxsqd=0;
```

SHORTHAND ASSIGNMENT STATEMENT

There is a shorthand notation for writing assignment statements by combining the assignment operator (=) and arithmetic operators such that the receiving variable or field can have its value changed by adding, subtracting, dividing and multiplying by a constant or an arithmetic expression. Examples are as follows:

count + = 1; is same as writing count = count +1;

NetPay + = grossPay-Tax is same as NetPay = NetPay + (grossPay-Tax)

Question: What can you say about the following three program statements?

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

Answer: They are equivalent.

Question: The statement A=++B is equivalent to A=B++. Is this true or false;
Answer: False.

1-2.2 Arithmetic/Mathematical expressions

The syntax of a mathematical expression is similar to that of an assignment statement. It combines variables and/or constants using arithmetic operators such as the + and -. The syntax for arithmetic expression is as follows:

variable = operand1 operator1 operand2 […operandn];

where *variable* is the receiving field, the *operand1*, *operand2*, etc are identifiers or constants and the *operator1*, *operator2*, etc are one of the following operators shown in Table 2.1.

Table 1.1: Arithmetic Operators

Operator	Use for
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Modulus

The operators *, / and % have the same order of precedence and are higher than the + and – which also have the same order of precedence. During evaluation of an expression, the compiler calculates expressions in the following order;

1. Expressions in brackets or parenthesis are evaluated first starting from the innermost set of parentheses and then to the outermost.
2. Multiplication, division and modulus are evaluated next from left to right
3. Addition and subtraction are evaluated from left to right.

The following points must be observed when writing arithmetic statements or expressions.

1. Generally, no two arithmetic operators must be next or adjacent to one another. For example the expression y = 5 * x + -z; is invalid because the operators + and - are next to one another. Note that in C++, the compilers will accept such statements as valid as the negative sign before the identifier z is seen as a uniary operator and not the arithmetic

operator, minus. For example y=5*x + /y will cause the error message "*expected primary-espression before '/' token*" to be displayed (when using the bloodshed c++ compiler). Therefore, whenever it becomes necessary for two operators to be next to each other, use parenthesis to separate them. There are instances where it is valid to write an expression such as count++ but we shall discuss this later.

1. No two variables or constants must be next to one another. In other words multiplication is not implied in C++ when two or more variables are next to one another. Whenever two or more variables are next to one another C++ assumes the variables that are next to each other to be a different identifier and will therefore display as error message such as

   "error C2065: 'PayRateHours' : undeclared identifier"

   if *PayRate* and *Hours* are two different identifiers. For example, if you are given that the product of three numbers say x, y and z is A, we can represent this statement mathematically as xyz = A or A = xyz. Here, the multiplication is implied between the variables x, y and z. In C++, we need not assume that multiplication is implied and therefore the above in C++ should be written as A = x * y * z; Note also that in mathematics, the A could be written such that it is either to the left or the right hand side of the replacement sign but in C++ and programming in general it can only appear to the left hand side because the meaning of = in C++ is different from that of mathematics. If an arithmetic operator appears on the right side of an arithmetic expression such as

   NetPay-Deductions = GrossPay;

   The compiler will display the error message

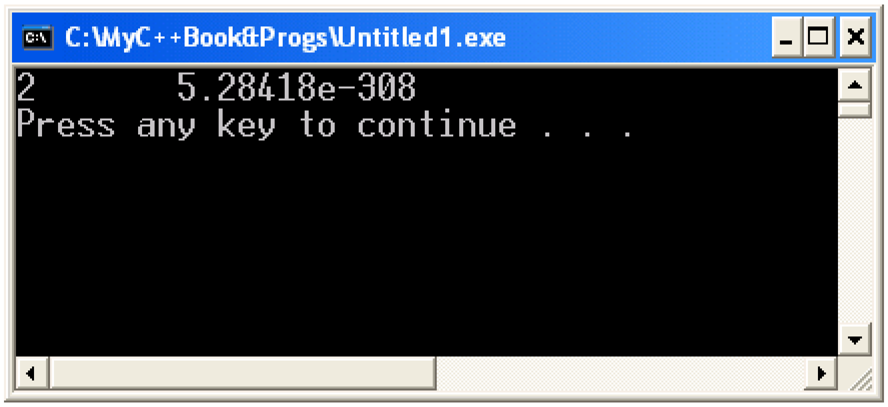
   "*error C2106: '=' : left operand must be l-value*".

2. Any variable used in an arithmetic expression must have previously been declared. They must also be initialized or might have been assigned a value if they are to appear to the right hand side of the = sign. In other words, any variable that appears to the right of the assignment sign should have a value before it is used in an expression. The warning "*warning C4700: local variable 'PayRate*' used without having been initialized" is displayed if the identifier *PayRate* is being used on the right hand side of the assignment operator (=) without having been initialized or assigned a value. Let us consider the following complete C++ programme:

```
#include<iostream>
```

```
using namespace std;
int main()
{
    int b;
    double c;
  cout <<b <<"\t" <<c;
 cout <<endl;
 system("pause");
 return 0;
}
```

Note that the identifiers b and c have been declared to be of type int and double respectively without any of them being intialised, however, on running the program the figure 2.1 was obtained. Figure 2.1 shows that the value of b is 2 and that of c is 5.28418e-308. Ask yourself where did the computer get these values from. This explains the reason why you need to initialize identifiers before they are used in expressions in which they are to appear at the right side of the assignment sign (=).



```
2       5.28418e-308
Press any key to continue . . .
```

Figure 2.1: Output Screen Showing Values of Uninitialised Variables

4. Real and integer quantities, if possible, should not be mixed up in the same expression. The reason is that although the compiler would not indicate that this is an error but only display a warning message, the result of your calculation may result in a truncation or the final answer may be different from what one would have expected. For example, one will expect the expression (17/18)*18+1.0 to give 18 as that is what one will get by using a calculator, however, the computer will return 1.0 instead. See if you can figure out why. If it happens that real and integer quantities are to be mixed up in the same expression then type casting should be applied.

5. Computations must be explicitly stated using operators. The use of parenthesis in an expression does not mean multiplication as it is in mathematics. For example, in mathematics 3(X-6) means 6 should be subtracted from X and the result multiplied by 3.

This is wrong in C++ and the compiler will display the error message "**error C2064: term does not evaluate to a function**". The correct form of this in C++ is 3*(X-6).

EFFECT OF ASSIGNING INT VARIABLES TO REAL VARIABLES AND VICE VERSA

Whenever an attempt is made at assigning *real* type value to an *int* (integer) variable, the C++ compiler truncates the fractional or decimal part of the value and converts the remaining part to int before storing the result into the memory cell of the int variable. Normally, if *x* is defined as an *int* variable then when a statements such as x=5.2 is entered as one of the program statements, the compiler displays a warning message such as *converting to 'int' from 'double'* during program compilation to let the programmer know that there may be a possible trunction.

On the other hand, whenever an attempt is made at assigning an integer value to a real variable, the C++ compiler places a decimal point and a zero (.0) at the end of the integer value to convert it to a real number before assigning the result to the memory cell for the real variable. For example, if the result of 12/8 is to be assigned to a real variable, the computer first evaluates the 12/8 to give 1. Remember an integer divided by an integer gives an integer result. Thus, before the Compiler assigns the value 1 to the real variable, it will make it 1.0 and then assign to the memory cell of the real variable.

1-2.3 Assignation Property

C++ has an assignation property where the *rvalue* can also contain the assignment operator. For example,

```
NetPay =GrossPay-(Deductions=Tax+Insurance);
```

The above expression means that the sum of *Tax* and *Insurance* be assigned to *Deductions*. The *NetPay* is then assigned *GrossPay* and finally *Deductions* is deducted from the value in *NetPay*. Please note that if the part involving the second assignment sign is not enclosed in bracket, the C++ compiler displayes the error message "*non-lvalue in assignement*". This means that the above statement is equivalent to the following:

```
Deductions = Tax + Insurance;
NetPay = GrossPay-Deductions;
```

Question: What will be displayed to the screen when the following program is run?

```
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
#define tab4 "    "
#define NextLine '\n'

int main()
{   int GP=5000,NP,Tax=100,Insurance=50,Deductions;
    NP=GP-(Deductions=2*(Tax+Insurance));
    cout <<GP <<tab4
         <<NextLine <<Deductions
         <<NextLine <<Tax <<NextLine;
    system("PAUSE");
    return 0;
}
```

Answer:

```
5000
300
100
Press any key to continue . . .
```

1-2.4 Operators

RELATIONAL OPERATORS

A logical expression uses logical operators to compare two or more variables. The result of a logical expression can either be 1 or 0 for true or false respectively. Operands in a logical expression are compared using their ASCII collating sequence, and an operand is said to be greater than another operand if it appears later in the ASCII collating sequence. If the operands being compared contain more than one character or digit each, the comparison is performed character by character (or digit by digit) and from left to right basis till a pair is found to be different or when all pairs have been compared. The valid logical operators are as given in Table 2.2.

Table 2.2: Relational Operators

Operator	Meaning
= =	Equal to
!=	Not equal to

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

For example to compare the content of var1 and var2 for equality will be as follows:

```
                    if  (var1==var2)
                              action;
```

where *action* is a statement specifying the action to take when var1 and var2 are the same or equal.

LOGICAL OPERATORS

The C++ Compiler has three logical operators. These are &&, || and !. A logical operator is used to connect two relational expressions. The operators && and || are binary operators since they appear between two relational expressions, that is they are used in forming compound conditions. The operator || (or) is the disconjunction and the && (and) is the conjunction. The operator ! is uniary because it precedes a single relational operand. Table 2.3 shows the results returned by the && and || operators. P and Q are two conditions and the values under them indicate a result that can be returned. For example P && Q returns true if and only if both P and Q each evaluates to true.

Table 2.3: Local Operators and Values Returned

P	Q	P && Q	P || Q	!P
T	T	T	T	**F**
T	F	F	T	**F**
F	T	F	T	**T**
F	**F**	**F**	**F**	**T**

BITWISE OPERATORS

These are bit manipulation operators that operate on the machine-dependent bit representation of integral operands.

Table 2.4: Bitwise Operators

Bitwise operator	Meaning
~	Unary one's complement
<<	Left shift
>>	Right shift
&	And

^	Exclusive or
|	Or

Please note that all the operators except the ~ are binary operators since they appear between two operands. The ~ is a uniary operator since it precedes a single operand. Table 2.4 shows the results returned by the operators when used.

Table 2.5: Bitwise Operators and Values Returned

P	Q	P & Q	P | Q	P ^ Q	~P	p<<1	p>>1
1	1	1	1	0	-2	10 (2)	**0**
1	0	0	1	1	-2	10 (2)	**0**
0	1	0	1	1	-1	00 (0)	**0**
0	**0**	**0**	**0**	**0**	**-1**	**00 (0)**	**0**

The following program illustrates how to use bit manipulation operators. Note that a left shift of a number is equivalent to multiplying the number by 2 while a right shift is equivalent to dividing by 2. When a number is shifted to the left, a zero is added at the end and hence why the number gets multiplied by two. Note that for bits the digits are either zero or one, that is the numbers are usually in base 2. See if you can also explain why the unary one's complement of 2 is -3.

```
    #include<iostream>
      using namespace std;
    int main()
    {
            int y,x=2;
            cout <<"Shifting 2 to the left 8 times gives " <<(x<<8) <<endl;
            cout <<"Shifting 2 to the right 1 time gives " <<(x>>1) <<endl;
            cout <<"The unary one's complement of 2 is " <<~x <<endl;
            cout<<"The exclusive or of 1 and 0 is " <<(1^0) <<endl;
            cout<<"The 1 or 0 is " <<(1|0) <<endl;
            return 0;
}
```

Program output:

```
        Shifting 2 to the left 8 times gives 512
        Shifting 2 to the right 1 time gives 1
        The unary one's complement of 2 is -3
        The exclusive or of 1 and 0 is 1
        The 1 or 0 is 1
        Press any key to continue
```

PRIORITY OF OPERATORS

Table 2.6 gives the order of priority of the C++ operators that one can use. We have not yet treated some of them and we will do so at the appropriate time. If an expression contains a number of operators, then those in parenthesis would be evaluated first. After that operators are evaluated in the following order with the highest one at the top. If operators of the same hierarchy appear in the same expression or if the same operator appears a number of times in the same expression then the evaluation will be from left to right except for the operators with priority 3, 11 and 12.

Table 2.6: Priority of C++ Operators

Priority	Operator(s)	Description/class
1	::	Scope
2	( ) [ ] -> . sizeof()	
3	++ --	Increment/decrement
	~	Ones complement
	!	Unary NOT
	& *	Pointer reference and dereference
	type(n)	Type casting of n to type
	+	Unary plus
	-	Unary minus
4	* / %	Arithmetic operators
5	+ -	Arithmetic operators
6	<< >>	Left and right bit shifting
7	< <= >=	Relational operators
8	== !=	Relational operators
9	& ^ |	Bitwise and, exclusive or and or
10	&& ||	Logical and and or
11	?:	Conditional operator
12	+= -= *= /= %= >>= <<= &= ^= |= =	Assignation operators
13	,	Separator

Self Assessment 1-2

1. What are some of the important rules that one must take into account when writing an arithmetic expression?

2. What are the different types of operators in C++ programming language?

3.Gie two examples of a typical arithmetic expression in C++ and explain what the expression could be used for?

SESSION 2-2: Input and Output Concepts

This deals basically with statements that are used to accept input data from the keyboard and output to the screen or the console. The path travelled by data in a program is usually referred to as a stream. If data is a source (keyboard or a file) into a program then we have input stream. If on the other hand the data is sent out of a program to a destination (printer or console) then we have output stream. We will also look at output formatting, that is how one can specify the way output of a program should be displayed.

2-2.1 Input Statement

The input statements in C++ are used to receive data from the keyboard or to read data from a file. For now, we shall concern ourselves with accepting data from the keyboard (using the **cin** statement). For reading from a file we shall consider this when we treat files. The syntax of the input statement for accepting inputs from the keyboard is as follows:

```
cin>>variable1 >>variable2 [ …>>variablen];
```

where *variable1*, *variable2*, etc are identifiers. The 'cin' is usually pronounced as 'C in' . For example, to include an input statement in a program so that it accepts values into the variables principal, rate and time, we will write this as follows:

```
cin>> principal  >>rate  >>time;
```

Note that only one terminating semi colon is required as the entire cin>> statement is taken to be a single statement. Note also that the above statement can be written in two other ways as follows;

(1) cin>> principal

```
            >> rate
             >> time;

(2)     cin>> principal;
        cin>> rate;
        cin>>time;
```

Even though in (1) above, the input statement has been spread over three lines using only one *cin* the three lines are considered to be only one statement and hence why only one semicolon is used at the end of the third line. In (2), each line has its on cin>> and that makes them different input statements and therefore why each statement has a terminating semicolon.

Using the input statement in (1) as an example, when program execution reaches the cin>> statement, it waits for inputs to be entered from the keyboard. The first data value is assigned to the first variable, the second value to the second variable, etc. The data values are read and stored in the variables only when the enter key is pressed. The input data must be separated by one or more spaces if they are to be entered on the same line or by line breaks if they are entered on different lines. Note that during variable assignment of input data, the computer skips spaces and line breaks until it finds the next input data.

WHAT HAPPENS DURING INPUT VIA KEYBOARD

When entering inputs via the keyboard during program execution, care must be taken in entering the correct data and also when to enter a data. When the compiler is reading data entered at the keyboard during program execution, all spaces are skipped until a character or a digit is found. For numeric, when a digit is found, the compiler continues to read subsequent digits until a space or a non digit character is encountered except for the decimal point in the case of real numbers. If there are variables that have not been assigned values, then the compiler will continue with the search for the data for the next variable by skipping whitespace (spaces and carriage returns). This continues until all variables have been assigned data. If there are more data than variables in a current read statement, the extra data items are left for the next input statement to access. If on the other hand there are fewer data than variables in an input statement, then the compiler pauses program execution until the required number of data items is entered. If a wrong data type is entered for a variable, the compiler is likely to terminate program execution or truncate the given data depending on the C++ compiler used. For example, if a variable is declared to store integers and the input is entered as character, the Bloodshed C++ compiler will not flag this as error but then when an output statement is used to display the content of the variable, the value 575 is displayed irrespective of the non digit character entered by the user. For a variable declared as float and double the values displayed will be 8.05747e-43 and 1.22015e311 respectively. If a variable is declared to be of integer type and a real value is enetered, only the integer part of the data will be assigned to the integer variable. These values may change depending on your compiler. For example if age is declared as an int variable and the value entered for it or assigned to it is 30.25 then only the 30 will be stored in age. The .25 will be available to the next input statement. Thus for the following input statement, 30 will be displayed for cedis and 0.25 for pesewas if cedis and pesewas are declared to be of int and double respectively:

```
cin >> cedis >>pesewas;
```

Note that for a char variable, if a digit is entered, the digit is read and stored in the variable but as a character hence it cannot be used in arithmetic computations. Also, if more than one character is entered, only the first character will be read and stored in the variable. For example, if letter is declared as char then giving it the value 7 and adding 10 to it will cause the alphabet 'A' to be displayed as the value stored in letter. Let us consider the following question.

2-2.2 Output Statement

C++ uses the cout<< statement to display messages and data to the screen. The ‘cout’ is usually pronounced ‘c out’. For now, we shall look at displaying information to the screen. The general syntax is as follows:

```
cout<<variable1 <<variable2,…variablen;
```

where *variable1*, *variable2*, etc may be a prompt (string constant) or a variable whose content is to be displayed. Just like the cin, the cout variables can also be spread over a number of lines using a single cout or a cout for each variable. Thus, cout in addition to the format given above can also be written in any of the following formats:

```
(1)     cout<<variable1
            <<variable2
                  :
            <<variablen;
(2)     cout<<variable1;
        cout<<variable2;
              :
        cout<<variablen;
```

Note that when a cout is used to display information to the screen, it does not move the cursor to the next row and neither does it move it before it displays the information. It allows subsequent cout statements to write on the current (same) line and moves to continue on the next line or row only when the current row is used up. It is possible to instruct the Computer to display outputs from cout statements on separate lines. This is done by using either the ‘\n’ (new-line character) or the endl (**end** of **l**ine) statement. Examples of using the ‘\n’ are as follows:

```
(1)     cout<<’Enter your name\n’;
(2)     cout<<’\n and then press the enter key’;
```

In example (1), the message ‘Enter your name’ is first displayed and the cursor is then moved to the next line for the next cout statement to write on the new line. In example (2), the cursor is moved to the next line before the message ‘and then press the enter key’ is displayed ensuring that the message is written on a new line.

The following example also illustrates how the endl statement is used:

```
cout<<’The average age is ‘
     << averageAge
     <<endl;
```

In the above example, the message ‘The average age is’ will be displayed on the current line. It will then be followed by the value stored in the variable averageAge. The endl part instructs the computer to advance to the next line so that the next cout starts on a new line.

For those who are good and comfortable at using BASIC or visual Basic, you can redefine your input and output statements such that you can use input and print respectively. To do so you must include the following directives in your program:

```
#define input cin>>
#define print cout <<
```

Having included the above two statements in your program, you can write any of the following as valid statements:

```
print "Please enter a positive integer";
input number;
```

OUTPUT LAYOUT

It is important that a programmer controls the way program output should be. Failure to do so normally results in an output that may not be suitable for the user. For example, data that has to be on different lines will appear on the same line, data values may not appear under their correct heading etc. There are a number of formatting characters that can be used with the cout function to specify how the layout of an output should be. Table 2.7 shows the commonly used formatting characters.

Table 2.7: Output Layout Controlling Characters

Character	Use for
\a	Alert
\b	Backspace
\f	Form feed
\r	Carriage return
\n	New line

\t	Tab
\v	Vertical tab
\’	Single quotes
\”	Double quotes
\\	Backslash
\0	Null character

For example, to specify that the content of a variable num be displayed on a new line at the second tab position the output statement required is as follows:

```
cout <<“\n \t\t” <<num;
```

Note that the above formatting characters can usually appear anywhere in a cout output string.

2-2.3 Output Formatting of Numbers (Using Manipulators)

In C++, integer values are usually displayed in a way that users have no problem with. However, the way values of double and float types are usually displayed may not be what a user wants. For example, if the variable salary is declared as double and has the value 100.28 at the time the cout statement is used to display its content to the screen, the output for example could be 100.280000 or may even be displayed in a scientific notation as 1.00280000e02. Since the salary is holding an amount of money, one will prefer if the value is displayed to 2 decimal places and also in a non scientific form. Output formatting is done using manipulators. To use a manipulator that takes an argument or a parameter in a program, the header file iomanip.h must be included in the program, that is #include<iomanip> should be included in the directives. This header file is not needed if the manipulator does not take in a parameter. The formats for using manipulators are as follows:

(i) For the manipulators setw, setfill and the setprecision we have

```
cout>>manipulator(argument)…..
```

where manipulator is either setw, setfill or setprecision. The argument is an argument as defined below. Example: cout<< setw(10) <<100;

(ii) For the other manipulators, we have

```
cout.manipulator;
```

where *manipulator* is a manipulator other than those listed in (i) above.

Table 2.8 shows the commonly used manipulators:

Table 2.8: Commonly Used Output Manipulators

Manipulator	Uses	Affects
setw(n)	This is used to specify the minimum number of digits to be displayed for numbers. If a number has more integer	Only the number to be displayed

	digits than as specified with the setw, the whole number is printed by ignoring the width set by the setw.	immediately after the setw(n)
setprecision(n)	It is used to set precision to n where *n* is an integer. If n is less than the number of digits in the number then the number is printed in e-notation form (e.g. 1.2e+002). If n equals the number of integer digits then only the integer part is printed. Hence n should normally be large so as to print the entire number. Note that if n is greater than the total number of digits the number will be displayed without leading spaces.	All numbers to be displayed after this statement unless another setprecision statement is encountered in the same program.
precision(n)	Same as setprecision.	Affects only the floating point number immediately following this statement.
Fixed	When included in a program allows fixed point notation to be used for floating point numbers. By default, precision with fixed is set to 6.	All floating point numbers after this statement unless formatting is changed to scientific.
scientific	When included in a program format floating point numbers to be displayed in scientific notation.	All floating point numbers after this statement unless formatting is changed to fixed.
left	This works only after setw(n). It is used to left justify both integer and real values. If a number to be displayed has fewer digits than as specified by setw(n), the number is printed and prefix by spaces to make up the required width.	Only the number to be displayed immediately after the left manipulator.
Right	This works only after setw(n). It is used to right justify both integer and real values. If a number to be displayed has fewer digits than as specified by setw(n), the number is printed and with spaces	Only the number to be displayed immediately after the right manipulator

	added at the end to make up the required width.	
setfill("ch")	This works only after setw(n). When used replaces spaces with the character ch to make up the number of digits required to be displayed for a number. For example cout <<setw(8) <<setfill('*') << n will *****123 if n has a value 123 and is required to be printed as left justified	Only the number to be displayed immediately after the setfill manipulator.
Fill	Same as the setfill	Only the number to be displayed immediately after the setfill manipulator
width(n)	This manipulator has the same effect as setw(n).	Only the number to be displayed immediately after the width manipulator
showpoint	Causes the decimal point to be displayed for all floating point numbers by displaying at least one decimal digits	Affects all floating point numbers
showpos	Causes the sign (+ or -) of a number to be displayed	Affects all numbers after this statement

Note that you can combine some of the manipulators in the same cout statement by separating them with the | sign. For example, if we want all floating point numbers to have their sign and the decimal to be displayed, we can write the following:

```
cout.setf(ios::showpos|ios::showpoint);
```

Table 2.9 shows examples of output statements using some of the manipulators above. This is to enable you to know how the manipulators are used.

Table 2.9: Examples of Output Manipulators Effects

Example	Output
cout.width(10);cout <<123;	123
cout <<setw(10) <<setfill('0') <<123;	0000000123
cout <<setw(10);cout.left;cout <<123;	123
cout.precision(3);cout <<123.01;	123
cout.precision(2);cout <<123.01;	1.2e+002

cout <<setw(2);cout <<123.01	123.01
cout<<setprecision(3) <<123.01	123

2-2.4 Other Manipulators

In addition to the above manipulators, there are other manipulators that are not necessary for output formatting. For example the skipws manipulator is used for input. Table 2.10 shows some of the other important manipulators.

Manipulator	Meaning
Skipws	When set causes leading whitespace characters (spaces, tabs and newlines) to be discarded during input.
Noskipws	When set causes leading whitespaces not to be discarded
Boolalpha	When set allows true or false to be input or output for Boolean identifiers.
Noboolalpha	When set prevents true or false to be used for Boolean identifiers during input and output
setbase(n);	Used to set the base of numbers. The n can be 7, 10 or 16
showbase, noshowbase	The showbase turns on the showbase flag during output while the noshowbase turns it off
uppercase, nouppercase	When set causes the e in scientific notation or the x in hexadecimal numbers to be displayed in upper case. It affects all floating point numbers and hexadecimal numbers. By default it is no set. The nouppercase set the uppercase off if it has been set on.
Ends	When used outputs a null
Flush	Flushes an output stream
Internal	Turns the internal flag on
unitbuf, nounitbuf	For turning the unitbuf on and off
setioflags(f),resetioflags(f)	Used to set and reset input and output flags
Oct	All integers after this are converted to base 8. Example cout<<oct <<100 gives 144

Hex	All integers after this are converted to base 16. Example cout<<hex <<100 gives 64
Dec	All integers after this are converted to base 10 if either the hex or the oct had been used to change the base.

Specifying the format for display of values of the type double can be done using the following statements:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(n);
```

where *n* is a positive integer specifying the number of decimal places to be printed. The first two statements need not be repeated in a program when the number of decimal places required changes. However, the last statement can be repeated a number of times if necessary and should only be repeated if one intends to change the number of decimal places required.

CREATING YOUR OWN MANIPULATORS

C++ allows programmers to customise input/output system by creating your own manipulator functions. Creating your own manipulators is best when the same sequence of I/O operations are used very often. In creating a manipulator of your own, you may combine one or more of the C++ manipulators and then refer to them all by a common name. User defined manipulators are functions and hence are defined in the same way as user defined functions are defined. The main difference here is that the data type of the result to be returned and the argument is ostream. To create an output manipulator the syntax is as follows:

```
ostream &manipulatorName(ostream &stream)
{
  //your manipulator code here
 return stream;
}
```

where *manipulatorName* is the name of the manipulator and *stream* is a reference to the invoking system.

The manipulator returns a stream. Note that even though the manipulator has a single argument which is a reference to the stream that it operates on, no argument is used when the manipulator is called.

Similarly, to create an input manipulator, the syntax is as follows:

```
istream &manupulatorName(istream &stream)
{
  //your manipulator code here
 return stream;
}
```

where variables have the same meaning as above.

As an example, we will define our own manipulator such that when called will set the field width, the precision and fill character to 15, 2 and '-' respectively. In addition, we want the sign of the numbers to be displayed. Let us call this manipulator myformat. Thus myformat in a program will be defined as follows:

```
ostream &myformat(ostream &stream)
{
  stream.width(15);
  stream.precision(2);
  stream.fill('-');
  return stream;
}
```

To use the myformat to display the content of the double identifier, amount the required output statement will be as follows:

```
cout <<myformat <<amount;
```

As mentioned above, user defined manipulators are functions so sticking to our style of using functions in a program, we will need a function prototype and the function definition for the manipulator. We will now consider a small program making use of user defined manipulators.

Question: Write a program that can compute the interest amount on a given loan amount (P) at a certain interest rate (R) over a certain period of time (I). The interest, I is calculated as follows:

$$I = \frac{PxRxT}{100}$$

The output should be all inputs and the interest calculated with appropriate captions. Each output should be to 2 decimal places written in a fixed format. The principal or loan amount should be displayed with a field width of 15 and if the number of digits is less that 15, the number should be preceded with '*'s to make up the filed width.

Solution: The bolded lines in the program show the manipulator function prototype and the function definition.

Program listing:

```
#include <iostream>
#include <cstdlib>
using namespace std;

ostream &format(ostream &stream);
int main()
{
  double prin, rate, time, interest;
  cout <<"Please enter the principal: ";
  cin>>prin;
  cout <<"                 the rate : ";
  cin>>rate;
  cout <<"                 the time : ";
  cin>>time;
  interest=prin*rate*time/100;
  cout <<"\n For a principal of " <<format <<prin <<" cedis at an interest"
       <<"\n rate of " <<rate <<"% for a period of " <<time
       <<"years the interest amount"
       <<"\n will be " <<interest <<" cedis" <<endl;
  system("PAUSE");
  return 0;
}
ostream &format(ostream &stream)
{
   stream.width(15);
   stream.precision(2);
   stream.fill('*');
   stream.setf(ios::fixed);
   return stream;
}
```

Program run:

```
Please enter the principal: 12437689
                 the rate : 25.6
                 the time : 12.5

 For a principal of ****12437689.00 cedis at an interest
 rate of 25.60% for a period of 12.50years the interest amount
```

```
will be 39800604.80 cedis
Press any key to continue . . .
```

Self Assessment 2-2

1. Explain why when ***cin*** statement is used in a program, the compiler will report that the identifier ***cin*** has not be declared even though the ***cin*** is a valid instruction?

2. Explain how a variable storing a decimal number can display the value as a currency or an amount?

3. Using examples, give two syntax of manipulators that do not require an argument.

4. With an illustration explain what is meant by a assignation property?

Learning Track Activities

Unit Summary

1. We have learnt in this Unit the various C++ operators and some of the important things we need to know about the operators. We learnt that in any give expression operators of higher priority are evaluated before a lower one. And where two or more operators, be it the same or different, have the same priority, the order of evaluation of the operators will be from left to right except for the exponentiation operator.

2. We have also learnt in this Unit how mathematical and logical expressions are written in C++. Generally, a valid mathematical expression should have an l-value and an r-value to the left and right of the assignment operator respectively. The l-value is the identifier that is to hold the result of the expression while the r-value consists of operators, identifiers and constants that evaluates to the expected result. For example in *netPay = GrossPay-TotalDeductions* the l-value is the *netPay* and the r-value is the *GrossPay-Deductions*.

3. Input and output statements are very important in any programming language. We have so far learnt the input and output statements that are used to accept inputs from the keyboard (*cin*) or to display information to the screen (*cout*).

4. There are a number of manipulators that one can use with the *cout* statement to format program output. These manipulators are very important. For example, a float identifier whose value is 255.89898 and is meant to a monitory value may be displayed as either 255.89898 or in its scientific form instead of 255.90 since monetary values are to two decimal places.

Unit Assignments 2

[insert here details of self-assessment]