lec 1 + 2

Lecture 1

Introduction to C++ Programming

Objectives

- To be able to write simple computer programs in C++.
- To be able to use simple input and output statements.
- To become familiar with fundamental data types.
- To be able to use arithmetic operators.
- To understand the precedence of arithmetic operators.

Lecture 1

Introduction to C++ Programming

1.1 A Simple Program: Printing a Line of Text

We begin by considering a simple program that prints a line of text.

```
1)  // Fig. 1.1: fig01_01.cpp
2)  // A first program in C++
3)  #include <iostream>
4)
5)  int  main()
6)  {
7)     std::cout  << "Welcome to C++!\n";
8)
9)   return 0; // indicate that program ended successfully
10)  }
```

This program illustrates several important features of the C++ language. We consider each line of the program in detail. Lines 1 and 2

```
// Fig. 1.1: fig01_01.cpp
// A first program in C++
```

each begin with // indicating that the remainder of each line is a *comment.* Programmers insert comments to *document* programs and improve program readability. C++ supports two kinds of comments. The first one uses the pairs of characters /* and */ to define the comment's range. This range can be confined within the same line, or it can span several lines. The second kind of comments in C++ uses the // character pair to mark the beginning of a comment that strictly runs to the end of the same line.

Line 3

```
#include <iostream>
```

is a *preprocessor directive,* i.e., a message to the C++ preprocessor. Lines beginning with # are processed by the preprocessor before the program is compiled. This specific line tells the preprocessor to include in the program the contents of the *input/output stream header file* **<iostream>**. This file must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output.

Line 5

```
int  main()
```

is a part of every C++ program. The parentheses after **main** indicate that **main** is a program building block called a *function.* C++ programs contain one or more functions, exactly one of which must be **main.** C++ programs begin executing at function **main,** even if **main** is not the first function in the program. The keyword **int** to the left of **main** indicates that **main** "returns" an integer (whole number) value.

The *left brace,* {, (line 6) must begin the *body* of every function. A corresponding *right brace,* }, (line 10) must end the body of each function.

Line 7

```
std::cout << "Welcome to C++!\n";
```

instructs the computer to print on the screen the *string* of characters contained between the quotation marks. The entire line, including std::cout, the << *operator,* the *string* "Welcome to C++!\n" and the *semicolon* (;), is called a *statement.* Every statement must end with a semicolon (also known as the *statement terminator*). Output and input in C++ is accomplished with *streams* of characters. Thus, when the preceding statement is executed, it sends the stream of characters Welcome to C++! to the *standard output stream object*--std::cout--which is normally "connected" to the screen.

Notice that we placed std:: before cout. This is required when we use the preprocessor directive #include <iostream>. The notation std::cout specifies that we are using a name, in this case cout, that belongs to "namespace" std. Namespaces are an advanced C++ feature

The operator << is referred to as the *stream insertion operator.* When this program executes, the value to the right of the operator, the right *operand,* is inserted in the output stream. The characters of the right operand normally print exactly as they appear between the double quotes. Notice, however, that the characters \n are not printed on the screen. The backslash (\) is called an *escape character.* It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an *escape sequence.* The escape sequence \n means *newline.* It causes the *cursor* (i.e., the current screen position indicator) to move to the beginning of the next line on the screen. Some other common escape sequences are listed below.

Escape	**Sequence**	**Description**
\n	Newline.	Position the screen cursor to the beginning of the next line.
\t	Horizontal tab.	Move the screen cursor to the next tab stop.
\r	Carriage return.	Position the screen cursor to the beginning of the current line; do not advance to the next line.
\a	Alert.	Sound the system bell.
\\	Backslash.	Used to print a backslash character.
\"	Double quote.	Used to print a double quote character.

Welcome to C++! can be printed several ways. For example:

```
std::cout << "Welcome \n to\n \n C++!\n";
std::cout << "Welcome"<<" to"<<" C++!\n";
```

Line 9

```
return 0; // indicate that program ended successfully
```

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 terminated successfully

1.2 Naming Items in C++

When naming items in C++, you need to observe the following rules:

- The first character of a name must be a letter or an underscore (_).
- Subsequent characters may be underscores, letters, or digits.
- Identifiers in C++ are case-sensitive. For example, the names volume, VOLUME, VOLume, and Volume are four different identifiers.
- You cannot use reserved words, such as int, double, or static, as identifiers.

Here are examples of valid identifiers:

```
y
x
myString
HOURS_PER_DAY
HexNumber1
hex_number_1
hex1Number3
_Length
_length_
```

1.3 Declaring Variables

C++ requires that you declare variables before you use them. Typically, you declare variables at the beginning of a function's body. The general syntax for declaring a variable is:

```
// form 1
        type variableName;
// form 2
        type variableName = initialValue;
```

Figure 1-2 declares the **char-type** variables **cChar1, cChar2,** and **cChar3**. The declaration of variable **cChar1** does not include initialization. The function **main** assigns the character literal '!' to variable **cChar1**. By contrast, the function **main** declares the variable **cChar2** and initializes it with the character literal '#'. As for the variable **cChar3**, the function **main** declares it and initializes it using the value in variable **cChar2**. The output of the program confirms that variables **cChar2** and **cChar3** store the '#' character.

```
#include <iostream.h>

main()
{
 char cChar1;
 char cChar2 = '#';
 char cChar3 = cChar2;

 cChar1 = '!';

 cout << cChar1 << "\n"
     << cChar2 << "\n"
     << cChar3 << "\n";

 return 0;
}
```

Figure 1-2

1.4 Predefined Data Types

Typically, programming languages offer predefined data types to manage fundamental kinds of data, such as characters, integers, floating-point numbers, and strings. Such data types represent the building blocks for user-defined data types.

Table 1-1 shows the predefined data types in C++. Your particular compiler may support additional types. Notice that some of the examples in Table 1-1 show numbers that start with the characters 0x. This is how

hexadecimal numbers are represented in C++. For example, the decimal integers 1241 and the hexadecimal integer 0xf1 are equivalent.

Table 1-1: Predefined Data Types in C++

Data Type	Byte Size	Range	Examples
bool	1	false and true	false, true
char	1	–128 to 127	'A','@'
signed char	1	–128 to 127	23
unsigned char	1	0 to 255	250,0x1c
int (16-bit)	2	–32768 to 32767	3200, –6000
int (32-bit)	4	-2147483648 to 2147483647	–1000000, 345678
unsigned int (16 bit)	2	0 to 56635	0x00aa, 32769
unsigned int (32-bit)	4	0 to 4294967295	0xffea, 65535
Short int	2	–32768 to 32767	234
unsigned short int	2	0 to 65535	0x1e, 52000
long int	4	–2147483648 to 2147483647	0xaffaf, –64323
unsigned long int	4	0 to 4294967295	167556
float	4	3.4E–38 to 3.4E+38 and –3.4E–38 to –3.4E+38	–15.443, 22.35, 2.45e+24
double	8	1.7E–308 to 1.7E+308 and –1.7E–308 to –1.7E+308	–2.5e+100, –78.32544
long double	10	3.4E–4932 to 1.1E+4932 and –1.1E–4932 to –3.4E+4932	8.5e–3000, –9.345e+2341

The data types in Table 1-1 include such keywords as short, long, and unsigned, which are really *type modifiers*. For the sake of shortening type names, however, some of these type modifiers have become synonymous with the fuller versions of the data type names. For example, the types long, short, and unsigned are equivalent to long int, short int, and unsigned int, respectively.

1.5 The #include Directive

In order for a programming language to perform sophisticated tasks (especially those required by operating systems, complex programs, and mission-critical applications), the source code must be able to incorporate special *directives* to the compiler. These directives guide and fine-tune the actions of the compiler.

The first, and perhaps most widely used, compiler directive you'll come across is #include. This directive instructs the compiler to read a source code file and treat it as though you had typed its contents where the directive appears. The general syntax for the #include directive is:

```
// form 1
        #include <filename>
// form 2
        #include "filename"
```

The first identifier *filename* represents the name of the file to be included. The two forms of #include vary in how they lead a program to conduct searches for the include file. The first form searches for the file in the special directory for include files. The second form expands the search to incorporate the current directory.

Here are examples of using the #include directive:

```
        #include <iostream.h>
        #include "myarray.hpp"
```

The first example includes the header file IOSTREAM.H by searching for it in the directory of include files. The second example includes the header file MYARRAY.HPP by searching for it in the directory of include files as well as in the current directory.

1.6 The #define Directive

The #define directive defines macros. C++ has inherited this directive from C for the sake of software compatibility. The general syntax for the #define directive is:

```
// form 1
        #define identifierName
// form 2
        #define identifierName literalValue
// form 3
        #define identifierName(parameterList) expression
```

The first form of the #define directive is typically used to indicate that a file has been read or to flag a certain software state. In this case, the #define directive need not associate a value with the *identifierName*. The main point for such use is to determine whether or not an identifier has been defined. Here are examples of using the #define directive to define state-related identifiers:

```
        #define _IOSTREAM_H_
        #define _DEFINES_MINMAX_
```

These examples define the identifiers _IOSTREAM_H_ and _DEFINES_MINMAX_. The first example may indicate that the file IOSTREAM.H has been read. The second example might, for example, flag the compiler to define or not define certain functions. Using uppercase identifiers is a common convention and is not enforced by the compiler. If so, the fact may be worth noting.

The second form of the #define directive defines the names of constants and associates literal values (numbers, characters, strings, and so on) with

these name. The preprocessor (which automatically runs before the compiler) replaces the name of the defined identifier with its associated value. Here are examples of using the #define directive to declare constants:

```
#define MAX 100
#define ARRAY_SIZE 20
#define MINUTE_PER_HOUR 60
```

These examples define the constants MAX, ARRAY_SIZE, and MINUTE_PER_HOUR and associate the values 100, 20, and 60 with these constants, respectively.

The third form of the #define directive defines pseudo-inline functions. In this way, the directive can create macros with arguments. The preprocessor replaces the name of the defined identifier and its arguments with the associated expression. Here are a few examples:

```
#define Square(x) ((x) * (x))
#define Reciprocal(x) (1/(x))
#define Lowercase(c) (char(tolower(c))
#define Uppercase(c) (char(toupper(c))
```

These examples define the pseudo-inline functions Square, Reciprocal, Lowercase, and Uppercase.

1.7 The #undef Directive

The #undef directive counteracts the #define directive by removing the definition of an identifier. The general syntax for the #undef directive is:

#undef *identifierName*

Here is an example of using the #undef directive:

```
#define ARRAY_SIZE 100
```

```
int nArray[ARRAY_SIZE];
#undef ARRAY_SIZE
```

This code snippet performs the following tasks:

- Define the identifier ARRAY_SIZE with the #define directive
- Use the identifier ARRAY_SIZE to define the number of elements of array nArray
- Undefine identifier ARRAY_SIZE using the #undef directive

You need not use the directive #undef to undefine an identifier before redefining it with another #define directive. Simply use the second #define directive to redefine an identifier. The following code snippet demonstrates this idea:

```
// first definition of ARRAY_SIZE
    #define ARRAY_SIZE 100
    int nArray1[ARRAY_SIZE];
    #undef ARRAY_SIZE
// second definition of ARRAY_SIZE
    #define ARRAY_SIZE 10
    int nArray2[ARRAY_SIZE];
```

These statements define, use, undefine, redefine, and then reuse the identifier ARRAY_SIZE. The next code snippet, however, which lacks the #undef directive, yields the same array declarations as the earlier one:

```
// first definition of ARRAY_SIZE
    #define ARRAY_SIZE 100
    int nArray1[ARRAY_SIZE];
// second definition of ARRAY_SIZE
    #define ARRAY_SIZE 10
    int nArray2[ARRAY_SIZE];
```

1.8 Declaring Constants

C++ allows you to declare constants either using the **#define** directive or using the formal constant syntax. The general syntax for declaring a formal constant is:

const *type constantName* = *constantValue*;

The declaration of a constant resembles the declaration of an initialized variable. Declaring a constant requires the keyword **const**. If you omit the constant's type, the compiler uses the **int** data type.

Here are examples of constants:

```
const int MAX_NUM = 1000;
const int MIN_NUM = 1;
const SEC_PER_MINUTE = 60;
const char FIRST_DRIVE = 'A';
const double MIN_RATE = 0.023;
```

The first two examples declare the constants MAX_NUM and MIN_NUM and explicitly associate the int type with these constants. By contrast, the third example, which contains the declaration of constant SEC_PER_MINUTE, has the int type by omission. The fourth and fifth examples declare constants that have the types char and double, respectively. Using uppercase with these constants is a common convention and is not enforced by the compiler.

Let's look at a simple example. Figure 1-3 shows the source code for the program CONST1.CPP, which illustrates C++ constants. The program declares a character constant and uses that constant to initialize a **char**-type variable. The program also displays the values associated with the constant and the variable. Here is the output of the program in Figure 1-3:

```
Character variable is ?
Character constant is ?
```

Figure 1-3 declares the **char**-type constant **QUESTION_MARK**. This constant is associated with the question mark character. The listing also declares the **char**-type variable **cChar** and initializes it using the constant **QUESTION_MARK**. The program then displays the values in both the constant **QUESTION_MARK** and the variable **cChar**.

```
// A C++ program that illustrates declaring constants

#include <iostream.h>

main()
{
  const char QUESTION_MARK = '?';
  char cChar = QUESTION_MARK;

  cout << "Character variable is " << cChar << "\n"
       << "Character constant is " << QUESTION_MARK << "\n";

  return 0;
}
```

Figure 1-3: declaring constants

1.9 Arithmetic operators

Arithmetic operators support the manipulation of integers and floating-point numbers. Table 1-2 shows the arithmetic operators in C++.

Table 1-2: The Arithmetic Operators in C++

C++ Operator	*Role*	*Data Type*	*Example*
+	unary plus	numerical	z = +h – 2
-	unary minus	numerical	z = –1 * (z+1)
+	add	numerical	h = 34 + g
-	subtract	numerical	z = 3.4 – t
/	divide	numerical	d = m / v
*	multiply	numerical	area = len * wd
%	modulus	integers	count = w % 12

Let's look at a program that applies the arithmetic operators to variables having the integer and floating-point types. Figure 1-4 shows the source code for the OPER1.CPP program, which illustrates the arithmetic operators. The program performs the following tasks:

1. Prompt you to enter two nonzero integers
2. Apply the operators +, -, *, /, and % to your input
3. Display the integer operands and the results of the operations just described
4. Prompt you to enter two nonzero floating-point numbers
5. Apply the operators +, -, *, and / to your input
6. Display the floating-point operands and the results of the operations just described

Here is the input and output of a sample session with the program in Figure 1-4:

Enter a nonzero integer : 342

Enter another nonzero integer : 23

342 + 23 = 365

342 - 23 = 319

342 * 23 = 7866

342 / 23 = 14

342 % 23 = 20

Enter a nonzero floating-point number : 4.56

Enter another nonzero floating-point number : 12.34

4.56 + 12.34 = 16.9

4.56 - 12.34 = -7.78

4.56 * 12.34 = 56.2704

4.56 / 12.34 = 0.36953

Figure 1-4 declares three sets of variables in function **main.** The first set comprises the **int**-type variables **nNUm1** and **nNum2**. The second set of variables is made up of the **long**-type variables **lAdd, lSub, lMul, lDiv,** and **lMod**. The third set of variables includes the **double**-type variables **fX**, **fY, fAdd, fSub, fMul,** and **fDiv.**

The function **main** prompts you to enter two integers, which it then stores in variables **nNum1** and **nNum2**. The function then uses the values in these variables as the operands of the tested operators. It assigns the results of the integer operations to the **long**-type variables. I chose to use **long**-type variables (which have a wider range of values than **int**-type variables) to store the results of the operations in order to fend off possible arithmetic overflow, especially with the +, -, and * operators. The function **main** then displays the integer operands and results.
As for applying the arithmetic operators to the floating-point numbers, function **main** also prompts you to enter two numbers. The function stores your input in variables **fX** and **fY**. It then uses the values in these variables as the operands of the tested operators. It assigns the results of the floating-point operations to the **double**-type variables **fAdd, fSub, fMul,** and **fDiv**. The function **main** then displays the floating-point operands and results.
C++ supports more complicated expressions that implement more advanced mathematical equations. For example, you can write expressions such as:

```
fZ = (((3 + 2 * fX) * fX - 5) * fX - 3) * fX - 20;
fH = (2 + fX + fY) * (34.2 - fX) / (fX * fX + fY * fY);
fD = (11 + (22 + fX) * (56 - fY)) / (fX * fX + fY * fY);
```

```
#include <iostream.h>

main()
{
 int nNum1, nNum2;
 long lAdd, lSub, lMul, lDiv, lMod;
 double fX, fY;
 double fAdd, fSub, fMul, fDiv;

 // prompt for two integers
 cout << "Enter a nonzero integer : ";
 cin >> nNum1;
 cout << "Enter another nonzero integer : ";
 cin >> nNum2;
 cout << "\n";

 // apply arithmetic operators
 lAdd = nNum1 + nNum2;
 lSub = nNum1 - nNum2;
 lMul = nNum1 * nNum2;
 lDiv = nNum1 / nNum2;
 lMod = nNum1 % nNum2;
 // display operands and results
 cout << nNum1 << " + " << nNum2 << " = " << lAdd << "\n";
 cout << nNum1 << " - " << nNum2 << " = " << lSub << "\n";
 cout << nNum1 << " * " << nNum2 << " = " << lMul << "\n";
 cout << nNum1 << " / " << nNum2 << " = " << lDiv << "\n";
 cout << nNum1 << " % " << nNum2 << " = " << lMod << "\n";
 cout << "\n";

 // prompt for two floating-point numbers
 cout << "Enter a nonzero floating-point number : ";
 cin >> fX;
 cout << "Enter another nonzero floating-point number : ";
 cin >> fY;
 cout << "\n";

 // apply arithmetic operators
 fAdd = fX + fY;
 fSub = fX - fY;
 fMul = fX * fY;
 fDiv = fX / fY;
 // display operands and results
 cout << fX << " + " << fY << " = " << fAdd << "\n";
 cout << fX << " - " << fY << " = " << fSub << "\n";
 cout << fX << " * " << fY << " = " << fMul << "\n";
 cout << fX << " / " << fY << " = " << fDiv << "\n";

 return 0;
}
```

Figure 1-4: arithmetic operations

Table 1.3 Precedence of arithmetic operators.

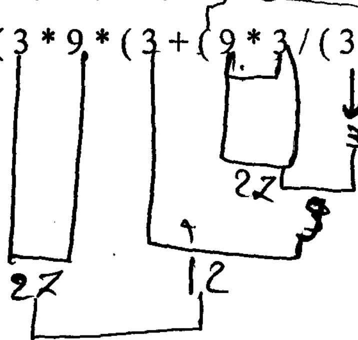
Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
* , / , or %	Multiplication Division Modulus	Evaluated second. If there are several, they are evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

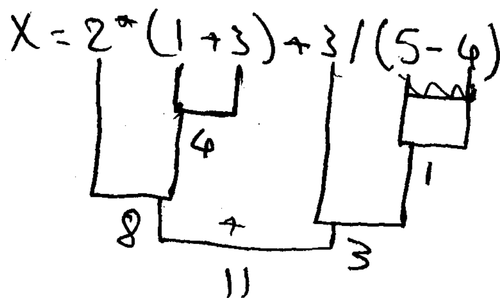
Example:

State the order of evaluation of the operators in each of the following C++ statements and show the value of x after each statement is performed.

a) x = 2*(1+3)+3/(5-4);

b) x = (3 * 9 * (3 + (9 * 3 / (3))));





1.10 Increment Operators

C++ offers the increment operators ++ and -- to support a shorthand syntax for adding or subtracting 1 from the value in a variable, respectively. The general syntax for the operator ++ is:

```
// form 1: preincrement
        ++variableName
// form 2: postincrement
        variableName++
```

The preincrement version of the operator ++ increments the value in its operand **variableName** *before* that variable supplies its value to the host expression. By contrast, the postincrement version increments the value in its operand **variableName** *after* that variable supplies its value to the host expression. If you use the increment operator in a statement that has no other operators (not even an assignment operator), then it makes no difference which form of the operator you use. Thus, these two statements have the same effect:

```
nCount++;
++nCount;
```

Here are examples of using the increment operator:

```
int nCount = 1;
int nNum;
nNum = nCount++; // nNum stores 1 and nCount stores 2
nNum = ++nCount; // nNum stores 3 and nCount stores 3
```

In this code snippet the variable **nCount** has the initial value of 1. The first statement that uses the increment operator employs the postincrement version. Consequently, the statement assigns the value in variable **nCount** to variable **nNum** and then increments the value in variable **nCount**. The result is that variable **nNum** stores 1 and variable **nCount** contains 2. The second statement that uses the increment operator employs the preincrement version. Consequently, the statement

first increments the value in variable **nCount** and then assigns the value in variable **nCount** to variable **nNum**. The result is that both variables **nNum** and **nCount** store 3.

As for the decrement operator, the general syntax for this operator is:

```
// form 1: pre-decrement
        --variableName
// form 2: post-decrement
        variableName--
```

The predecrement version of the operator -- decrements the value in its operand **variableName** *before* that variable supplies its value to the host expression. By contrast, the postdecrement version of the same operator decrements the value in its operand *after* that variable supplies its value to the host expression. If you use the decrement operator in a statement that has no other operators (including the assignment operator), then it makes no difference which form of the operator you use. Thus, the following two statements have the same effect:

```
nCount--;
--nCount;
```

Here are examples of using the decrement operator:

```
int nCount = 10;
int nNum;
nNum = nCount--; // nNum stores 10 and nCount stores 9
nNum = --nCount; // nNum stores 8 and nCount stores 8
```

In this code snippet the variable **nCount** has the initial value of 10. The first statement that uses the decrement operator employs the postdecrement version. Consequently, the statement assigns the value in variable **nCount** to variable **nNum** and then decrements the value in variable **nCount**. The result is that variable **nNum** stores 10 and variable **nCount** contains 9. The second statement that uses the decrement operator employs the predecrement version. Consequently, the statement

first decrements the value in variable **nCount** and then assigns the value in variable **nCount** to variable **nNum**. The result is that both variables **nNum** and **nCount** store 8.

Suppose j = 10;

Expression	*Evaluated As*	*Result*
j * j * j++	10 * 10 * 10	1000
j * j++ * j	10 * 10 * 10	1000
j++ * j * j	10 * 10 * 10	1000
j * j * ++j	10 * 10 * 11	1100
j * ++j * j	11 * 11 * 11	1331
++j * j * j	11 * 11 * 11	1331

1.11 Assignment Operators

If you have programmed in BASIC, Pascal, or another structured programming languages, then you have probably written expressions such as these:

```
sum = sum + x;
diff = diff - x;
scale = scale / factor;
factorial = factorial * x;
```

Each statement contains the same variable on both sides of the assignment operator. C++ supports assignment operators that combine arithmetic and bitwise operations with the assignment operator. Thus you can write the preceding statements as:

```
sum += x;
diff -= x;
scale /= factor;
factorial *= x;
```

Table 1-4 lists the arithmetic assignment operators in C++. The table also contains examples of using these operators, in addition to the long-form versions of the statements in the examples.

Table 1-4: The Arithmetic Assignment Operators in C++

C++ Operator	*Example*	*Long-Form Example*
+=	fSum += fX;	fSum = fSum + fX;
–=	fY –= fX;	fY = fY – fX;
/=	nCount /= N;	nCount = nCount / N;
*=	fScl *= fFcator;	fScl = fScl * fFactor;
%=	nBins %= nCount;	nBins = nBins % nCount;

1.12 Typecasting

C++ supports the typecasting feature (inherited from C) to allow you to explicitly convert a value from one data type into another type. The general syntax for typecasting is:

```
// form 1
(newType)expression
// form 2
newType(expression)
```

Here are examples of using the typecasting feature:

```
char cLetter = 'A'
int nASCII = int(cLetter);
long lASCII = (long)cLetter;
```

This code snippet declares and initializes the **char**-type variable **cLetter**. The code also declares the **int**-type variable **nASCII** and initializes it using the **int** typecast of variable **cLetter**. In addition, the code declares

the **long**-type variable **lASCII** and initializes it using the **long** typecast of variable **cLetter**.

static cast

Cast operators are available for any data type. The **static_cast** operator is formed by following keyword **static_cast** with angle brackets (< and >) around a data type name. The cast operator is a unary operator, i.e., an operator that takes only one operand. Here's a statement that uses a C++ cast to change a variable of type int into a variable of type char:

```
aCharVar = static_cast<char>(anIntVar);
```

Here the variable to be cast (anIntVar) is placed in parentheses and the type it's to be changed to (char) is placed in angle brackets. The result is that anIntVar is changed to type char before it's assigned to aCharVar.

Questions

(SAMS - Object-Oriented Programming in C++ Book-chapter 2)

2, 3, 4, 5, 6, 9, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 23.

2. A function name must be followed by ________.
3. A function body is delimited by ________.
4. Why is the main() function special?
5. A C++ instruction that tells the computer to do something is called a ________.
6. Write an example of a normal C++ comment and an example of an old-fashioned /* comment.
9. True or false: A variable of type char can hold the value 301.
10. What kind of program elements are the following?
    a. 12
    b. 'a'
    c. 4.28915
    d. JungleJim
    e. JungleJim()
11. Write statements that display on the screen
    a. the character 'x'
    b. the name *jim*
    c. the number 509
12. True or false: In an assignment statement, the value on the left of the equal sign is always equal to the value on the right.
14. What header file must you #include with your source file to use cout and cin?
15. Write a statement that gets a numerical value from the keyboard and places it in the variable temp.
17. Two exceptions to the rule that the compiler ignores whitespace are ________ and ________.
18. True or false: It's perfectly all right to use variables of different data types in the same arithmetic expression.
19. The expression 11%3 evaluates to ________.
20. An arithmetic assignment operator combines the effect of what two operators?

21. Write a statement that uses an arithmetic assignment operator to increase the value of the variable temp by 23. Write the same statement without the arithmetic assignment operator.
22. The increment operator increases the value of a variable by how much?
23. Assuming var1 starts with the value 20, what will the following code fragment print out?

```
cout << var1--;
cout << ++var1;
```

Exercises

1. Write a single C++ statement or line that accomplishes each of the following:
   a) Print the message "**Enter two numbers**".
   b) Assign the product of variables **b** and **c** to variable **a**.
   c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).
   d) Input three integer values from the keyboard and into integer variables **a, b** and **c**.

2. State which of the following are true and which are false. If false, explain your answers.
   a) C++ operators are evaluated from left to right.
   b) The following are all valid variable names: **_under_bar_, m928134, t5, j7, her_sales, his_account_total, a, b, c, z, z2.**
   c) The statement **cout** << **"a = 5;"**; is a typical example of an assignment statement.
   d) A valid C++ arithmetic expression with no parentheses is evaluated from left to right.
   e) The following are all invalid variable names: **3g, 87, 67h2, h22, 2h.**

3. Fill in the blanks in each of the following:
   a) What arithmetic operations are on the same level of precedence as multiplication?__________.

b) When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression?__________.

c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a__________.

4. What, if anything, prints when each of the following C++ statements is performed? If nothing prints, then answer "nothing." Assume x = 2 and y = 3.

a) cout << x;

b) cout << x + x;

c) cout << "x=";

d) cout << "x = " << x;

e) cout << x + y << " = " << y + x;

f) z = x + y;

g) cin >> x >> y;

h) // cout << "x + y = " << x + y;

i) cout << "\n";

5. Which of the following C++ statements contain variables whose values are replaced?

a) cin >> b >> c >> d >> e >> f;

b) p = i + j + k + 7;

c) cout << "variables whose values are replaced";

d) cout << "a = 5";

6. Given the algebraic equation y = ax3 + 7, which of the following, if any, are correct C++ statements for this equation?

a) y = a * x * x * x + 7;

b) y = a * x * x * (x + 7);

c) y = (a * x) * x * (x + 7);

d) y = (a * x) * x * x + 7;

e) y = a * (x * x * x) + 7;

f) y = a * x * (x * x + 7);

,Exercises' Solutions
Subject: CS503
of Lecture No. (1)

SAMS Object-oriented Programming in C++ Book Chapter2)

2- A function name must be followed by **Parentheses**:
3- A function body is delimited by **braces {}.**
4- Why is main () function special?
Answer: Because it is the first function executed when the program starts.
5- A C++ instruction that tells the computer to do something is called a **statement**.
6- Write an example of a normal C++ comment and an example of an old-fashioned /* comment.
Answer: // This program calculates the Area and Circumference of a Circle.
/* This program calculates the Area and Circumference of a Circle*/.
9- True or false: A variable of type char can hold the value 301. **(False), it holds from -128 to 127.**
10- What kind of program elements are the following?
- a- 12 **(Integer)**
- b- 'a' **(Character Constant)**
- c- 4.28915 **(Floating-point Constant)**
- d- JungleJim **(Variable Name)**

11- Write statements that display on the screen
- a- The character 'x' **(cout<<'x' ;)**
- b- The name jim **(cout<<"jim" ;)**
- c- The number 509 **(cout<<509 ;)**

12- True or false: in an assignment statement, the value on the left of the equal sign is always equal to the value on the right. (False) they're not equal until the statement is executed.
14- What header file must you #include with your source file to use cout and cin? **(<iostream.h>)**
15- Write a statement that gets a numerical value from the keyboard and places i9t in the variable temp. **(cin>>temp;)**
17- Two exceptions to rule that the compiler ignores whitespaces are **string constants and preprocessor directives.**
18- True or false: It's perfectly all right to use variables of different data types in the same arithmetic expression. **(True)**
19- The expression 11%3 evaluates to **2.**
20- An arithmetic assignment operator combines the effect of what two operators? **The assignment operator (=) and arithmetic (like+and*).**
21- Write a statement that uses an arithmetic assignment operator to increase the value of the variable temp by 23. Write the same statement without the arithmetic assignment operator. **(temp+=23 ;), (temp=temp+23 ;)**
22- The increment operator increases the value of a variable by how much? **(1).**
23- Assuming var1 starts with the value 20, what will the following code fragment print out?

```
cout<< var1--;
cout<<++var1; (2020)
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Exercises Page 25

Ex. No. 1 (Page # 25)

1-write a single C++ statement or line that accomplishes each of the following:

a) Print the message "Enter two Numbers."

Answer: cout<<"Enter two Number.";

b) Assign the product of variables b and c to variable a.

Answer: a=b*c;

c) State that a program performs a sample payroll calculation (i.e., use text that helps to document a program).

Answer: // Sample Payroll Calculation

d) Input three integers values from the keyboard and into integer variables a, b and c.

Answer:

```
int a, b, c;
cout<< "Enter Integer Values:"
cin>>a>>b>>c;
```

2- State which of the following are true and which are false. If false, explain your answer.

a) C++ operators are evaluated from left to right
Answer: (False) they're evaluated from left to right in case of equal priority.

b) The following are all valid names: _under_bar_, m928134, t5, j7, her_sales, his_account_total, a, b, c, z, z2,
Answer: (True)

c) The statement cout<<:a=5;"; is typical example of an assignment statement.
Answer: (False) this statement will be printed only as it is and not an assignment statement.

d) A valid C++ arithmetic expression with no parentheses is evaluated from left to right.
Answer: (True).

e) The following are all invalid variable names: 3g, 87, 67h2, h22, 2h.
Answer: (False) h22 is the only valid variable name and the others are not valid.

3- Fill in the blank in each of the following:

a) what arithmetic operations are on the same of level precedence as multiplications? **Division**.

b) When parentheses the **innermost parentheses**.

c) A location in the computer's memory of a program is called a **variable (s).**

4- what if anything prints when each of the following C++ statements are performed? If nothing prints, then answer "nothing" Assume x=2 and y=3.

Answer:

	Statement	Output
a)	Cout<<x;	2
b)	Cout<<x+x;	4
c)	Cout<<"x=";	x=
d)	Cout<<"x="<<x;	x=2
e)	Cout<<x+y<<"="<<y+x;	5+5

PDF created with pdfFactory Pro trial version www.pdffactory.com

f) Z=x+y; — **5** (2+3)
g) Cin>>x>>y; — **Nothing** (Input command)
h) //cout<<"x+y="<<x+y; — **Nothing** (comment command)
i) Cout<<"\n"; — **Nothing** (New Line)

5- Which of the following C++ statements contain variables whose values are replaced?

a) Cin>>b>>c>>d>>e>>f;
b) P=i+j+k+7;
c) Cout<<"Variables whose values are replaced";
d) Cout<<"a=5";

6- Given the algebraic equation y=ax3+7, which of the following if any are correct C++ statements for this equation?

a) Y=a*x*x*x+7;
b) Y=a*x*x*(x+7);
c) Y=(a*y)*x*(x+7);
d) Y=(a*x)*x*x+7;
e) Y=a*(x*x*x)+y;
f) Y=a*x*(x*x+7);

7- State the order of evaluation of the operators in each of the following C++ statements and show the value of x after each statement is performed.

a) 7 + 3 * 6 / 2 - 1;

b) X = 9 % 2 + 2 * 2 – 2 / 2;

c) X = (3 * 9 *(3+(9*3/(3))));

PDF created with pdfFactory Pro trial version www.pdffactory.com

Ex. No 8 (Page # 27)

// This program calculates the Sum, Product, Difference and Quotient of Two Given Numbers (

```
# include <iostream.h>
int main()
{
float x,y,s,p,d,q;
cout<<"Enter First Number:\n";
cin>>x;
cout<<"Enter Second Number:\n";
cin>>y;
s=x+y;
cout<<"The Sum of X+Y ="<<s;
cout<<"\n";
p=x*y;
cout<<"The Product of X*Y ="<<p;
cout<<"\n";
d=x-y;
cout<<"The Difference of X-Y ="<<d;
cout<<"\n";
q=x/y;
cout<<"The Quotient of X/Y ="<<q;

return 0;
}
```

Ex. No. 11 Page # 27

What does the following code print?

```
cout<<"*\n**\n***\n****\n*****\n";
```

Output

```
*
* *
* * *
* * * *
* * * * *
```

Ex. No. 10 Page # 27

Write a program that reads in radius of a circle and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for π. Do these calculations in output statements.

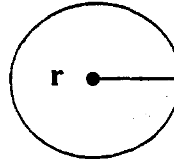
Formulas: Area = πr^2, Circumference = πr

```
// Program to Calculate the Area and Circumference of a Circle (Ex. 10 Page # 27)
#include <iostream.h>
int main()
```

```
{
float r;
float a;
float c;

const float pi=3.14159;
cout<<"Enter Radius:";
cin>>r;
a=pi*r*r;
cout<<"\n";
cout<<"Circle Area is:"<<a;
cout<<"\n";
cout<<"Enter Circumference:";
c=pi*r;
cout<<"Circle Circumference is:"<<c;
return 0;

}
```



Examples Page 18

State the order of evaluation of the operators in each of the following C++ statements and show the value of x after each statement is performed.

A) X=2*(1+3)+3/(5-4);
B) X=(3*9*(3+(9*3/(3))));

If a=2, b=3, c=1, d=4, e=5, write a single C++ statement for each of the following **expressions. State the order of evaluation & show the value of x after each statement is performed.**

1- $x = \frac{2a+2b}{3c}$

2- $x = \frac{5c+2+b}{a+b}$

3- $x = \frac{2d}{a} + \frac{c+e}{b}$

a) X=2*(1+3)+3/(5-4);

b) X=(3*9*(3+(9*3/(3))));

1-X= (2*a+2*b)/(3*c);

PDF created with pdfFactory Pro trial version www.pdffactory.com

2- X= (5*c+2+b)/(a+b);

3- X= (2*d/a)+(c+e/b);

Lecture 2

Decision Making

2.1 Relational and Logical Operators

Programs require relational and Boolean operators to create decision-making Boolean expressions. Table 2-1 shows the relational and Boolean operators in C++. Also notice that Table 2-1 contains the ***conditional assignment operator*** **?:**. It has the following syntax:

(expression) ? trueValue : falseValue

The operator yields the **trueValue** if the expression is true (or nonzero) and returns the **falseValue** otherwise. Consider how this statement uses the conditional assignment operator to assign a value to a variable:

variable = (expression) ? trueValue : falseValue;

Table 2-1: The Relational and Boolean Operators in C++

C++ Operator	*Meaning*	*Example*
&&	logical AND	k > 1 && k < 11
||	logical OR	k < 0 || k > 22
!	logical NOT	!(k > 1 && k < 10)
<	Less than	k < 12
<=	Less than or equal to	k <= 33
>	greater than	k > 45
>=	greater than or equal to	k >= 77
==	equal to	k == 32
!=	not equal to	k != 33
?:	conditional assignment	k = (k < 0)? 1 : k

logical AND:

&&	T	F
T	T	F
F	F	F

logical OR:

||	T	F
T	T	T
F	T	F

logical NOT:

!	T	F
	F	T

Example:

41 < 65 is TRUE

41 <= 29 is FALSE

65 > 29 is TRUE

41 == 65 is FALSE

41 != 29 is TRUE

41 < 65 && 65 < 29 is FALSE

41 < 65 || 65 < 29 is TRUE

!(41 <= 65) is FALSE

(0) is FALSE

(45) is TRUE

!(14%7) && !(10%7) is FALSE

2.2 The Simple if Statement

C++ offers the simple **if** statement to support single-alternative decision making. The general syntax for the simple **if** statement is:

```
// form 1
        if (condition)
          statement;
// form 2
        if (condition) {
          // sequence of statement
        }
```

The **if** statement uses the keyword **if** followed by the parentheses that contain the tested condition. If that condition is true, the program executes the statement (see form 1) or the block of statements (see form 2) that come after the tested condition. Otherwise, the program bypasses this statement (or block of statements).

Here are examples of the single-alternative **if** statement:

```
// example 1
        if (nNum < 0)
         cout << "Value is negative!\n";

// example 2
        if (i > 0 && i < 100)
         cout << "Number is in range 1 to 100\n";

// example 3
        if (nCount < 1)
         nCount = 1;
```

The first example uses the **if** statement to display a message if the value in variable **nNum** is negative. The second example employs the **if** statement to display a message when the variable **i** contains an integer in the range of 1 to 99. The third example assigns 1 to variable **nCount** if that variable contains a value that is less than 1.

Example :

Figure 2.1 shows an example for using if statement to specify if the entered number is greater than 100.

```
// demonstrates IF statement
#include <iostream>
using namespace std;
int main()
{
        int x;
        cout << "Enter a number: ";
        cin >> x;
        if( x > 100 )
                cout << "That number is greater than 100\n";
        return 0;
}
```

Here's an example of the program's output when the number entered by the user is greater than 100:

```
Enter a number: 2000
```

```
That number is greater than 100
```

If the number entered is not greater than 100, the program will terminate without printing the second line.

2.3 The if-else Statement

C++ enables the **if** statement to support dual-alternative decision making. The general syntax for the dual-alternative **if** statement is:

```
if (condition)
  // statement or block of statements
else
  // statement or block of statements
```

The dual-alternative **if** statement uses the keyword **else** to separate the two sets of statements that offer the alternative actions. If the tested condition is true, the program executes the statement or statement block that comes after the tested condition. Otherwise, program execution resumes after the keyword **else** and executes the subsequent statement or statement block.

Here are examples of the dual-alternative **if** statement:

```
// example 1
if (nNum < 0)
  cout << "Value is negative\n";
else
  cout << "Value is 0 or greater\n"

// example 2
if (i > 0 && i < 100)
   j = i * i;
else
   j = 100;

// example 3
if (nCount < 1)
```

```
        nCount = 1
      else
        nCount--;
```

The first example uses the **if** statement to determine whether or not the value in variable **nNum** is negative. If this condition is true, the **if** statement display the message "Value is negative." Otherwise, the **if** statement executes the statement in the **else** clause to display the message "Value is 0 or greater."

The second example employs the **if** statement to determine if the variable **i** contains an integer in the range of 1 to 99. If this condition is true, the statement assigns the expression **i * i** to the variable **j**. Otherwise, the **if** statement executes the **else** clause statement to assign 100 to the variable **j**.

The third example uses the **if** statement to determine whether or not the value in variable **nCount** is less than 1. If this condition is true, the **if** statement assigns 1 to variable **nCount**. Otherwise, the **if** statement executes the **else** clause statement to decrement the value in variable **nCount**.

Example:

Here's a variation of our IF example, with an else added to the if:

```
// demonstrates IF...ELSE statememt
#include <iostream>
using namespace std;
int main()
{
    int x;
    cout << "\nEnter a number: ";
    cin >> x;
    if( x > 100 )
        cout << "That number is greater than 100\n";
    else
        cout << "That number is not greater than 100\n";
    return 0;
}
```

2.4 The Multiple-Alternative if Statement

C++ also permits the **if** statement to support multiple-alternative decision making. The general syntax for the multiple-alternative **if** statement is:

```
if (condition1)
  // statement #1 or block of statements #1
else if (condition2)
  // statement #2 or block of statements #2
else if (condition3)
  // statement #3 or block of statements #3
// other else if clauses
else
  // catch-all statement or catch-all block of statements
```

The multiple-alternative **if** statement allows a routine to test a battery of conditions and take one of multiple courses of action. The **if** statement tests the Boolean expressions **condition1, condition2, condition3**, and so on in that sequence. The first condition that is true causes the runtime system to execute its associated statements. Program execution resumes after the end of the **if** statement. If none of the tested conditions are true, the program executes the statements in the catch-all **else** clause (if one is used).

Here is an example of a multiple-alternative **if** statement:

```
if (N >= 0 && N < 10)
  cout << "Variable N is a single digit\n";
else if (N >= 10 && N < 100)
  cout << "Variable N has two digits\n";
else if (N >= 100 && N < 1000)
  cout << "Variable N has three digits\n";
else if (N >= 1000)
  cout << "Variable N has four or more digits\n";
else
  cout << "Variable N is negative\n";
```

This code snippet classifies the value in variable **N** as follows:

- The condition of the **if** clause determines whether or not the variable **N** contains an integer in the range of 0 to 9.
- The first **else if** clause determines whether or not the variable **N** contains an integer in the range of 10 to 99.
- The second **else if** clause determines whether or not the variable **N** contains an integer in the range of 100 to 999.
- The third **else if** clause determines whether or not the variable **N** contains an integer equal to or greater than 1000.

Each of the **if** and **else if** clauses display a message reflecting the value in variable **N**. The catch-all **else** clause displays the message that the variable **N** contains a negative value.

Example:

Write a C++ program that accepts grade for a student then determines his graduation grade according to the incoming table:

Criteria	g > =90	90>g>=80	80>g>=70	70>g>=60	60>g
Income level	Excellent	Very good	good	pass	Fail

Interaction with the program might look like this:

Enter a grade: 85

Graduation grade: very good

```
#include <iostream.h>
int main()
{       int grade;
        cout<<"Enter grade:";
        cin>>grade;
        if(grade >=90)
                cout<<"\n Excellent"<<endl;
        else if (grade>=80)
                        cout<<"\n Very good"<<endl;
                else if (grade>=70)
                                cout<<"\n good"<<endl;
                        else if (grade>=60)
                                        cout<<"\n pass"<<endl;
                                else
                                        cout<<"\n Fail"<<endl;
        return 0;
}
```

2.5 The switch Statement

C++ offers the **switch** statement to support multiple-alternative decision making. The general syntax for the multiple-alternative **switch** statement is:

```
switch(expression)
{
 case constantExpression1:
   // statement set #1
   break;
 case constantExpression2:
   // statement set #2
   break;
...
[default:
 // catch-all statements]
}
```

The **switch** statement examines the value of the expression, which must be integer or integer-compatible (as are characters and enumerated types). The condition of the switch can be a variable, a function call, or an expression that includes constants, variables, and function calls.

The **switch** statement uses **case** labels for comparing the tested expression with different values. C++ has the following rules about the **case** labels:

- The keyword **case** is followed by a single constant (either a literal constant or a constant expression), followed in turn by a colon.
- You can include a sequence of more than one **case** label; all such labels end up executing the first sequence of statements that follows.
- A **case** label cannot list a range of constant values. Each **case** label lists only one constant.

Program execution sequentially examines the values in the **case** labels. If a **case** label value matches the tested expression, the program executes the statements that come after the case label.

Example:

Write a program that asks the user to enter the item number and prints the price of this item according to the following table.

ItemNo	1	2	3	4
Price	100	200	600	150

```
#include <iostream.h>
int main()
{
        int ItemNo;
        cout<<"Enter the item no. : ";
        cin>>ItemNo ;
        switch(ItemNo)
        {
                case 1:
                                cout<<"\n the price is 100"<<endl;
                                break;
                case 2:
                        cout<<"\n the price is 200"<<endl;
                                break;
                case 3:
                        cout<<"\n the price is 600"<<endl;
                                break;
                case 4:
                        cout<<"\n the price is 150"<<endl;
                                break;
                default:
                        cout<<"\n the item no. not found"<<endl;
        }
return 0;
}
```

Exercises

1. Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words "is larger." If the numbers are equal, print the message "These numbers are equal."

2. Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers. The screen dialogue should appear as follows:

   Input three different integers: 13 27 14
   Sum is 54
   Average is 18
   Product is 4914
   Smallest is 13
   Largest is 27

3. Write a program that reads in five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this lecture.

4. Write a program that reads an integer and determines and prints whether it is odd or even. (Hint: Use the modulus operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.)

5. Write a program that reads in two integers and determines and prints if the first is a multiple of the second. (Hint: Use the modulus operator.)

6. Write a program that inputs a five-digit number, separates the number into its individual digits and prints the digits separated from one another by three spaces each. (Hint: Use the integer division and modulus operators.) For example, if the user types in 42339 the program should print:

   4   2   3   3   9

7. Write an algorithm and a C++ program that accepts an employee salary and a tax percentage then computes the monthly net salary and determine his income level according to the incoming table:

Criteria	ns > =5000	50000>ns>=2000	ns<2000
Income level	high	moderate	low

Lecture 3

Loops

3.1 The for Loop

The general syntax for the **for** loop is:

for (*initilizationPart*; *iterationConditionPart*, *incrementPart*);

The **for** loop contains the following three parts:

1. The initialization part, which initializes the loop control variable(s). You can use single or multiple loop control variables.
2. The iteration part, which contains a Boolean expression that causes the loop to iterate as long as the expression is true
3. The increment part, which increments or decrements the loop control variable(s)

Here are examples of the **for** loop:

```
// example 1
    for (i = 0; i < 10; i++)
      cout << i << "\n";

// example 2
    for (i = 9; i >= 0; i -= 3)
      cout << (i*i) << "\n";

// example 3
    for (int i = 1; i < 100; i++)
      cout << i << "\n";

// example 4
    for (int i = 0, j = MAX; i < j; i++, j--)
      cout << (i + 2 * j) << "\n";
```

The first example initializes the loop control variable **i** to 0 and iterates as long as the value in variable **i** is less than 10. The loop increment part

increases the value of variable **i** by 1. Thus the upward-counting loop iterates 10 times with the value in variable **i** changing from 0 to 9.

The second example shows a downward-counting loop that initializes the loop control variable **i** to 9. The loop iterates as long as the value in variable **i** is not negative. The loop increment part decreases the value of variable **i** by 3. Thus the loop iterates four times with the value in variable **i** having the sequence 9, 6, 3, and 0.

The third example shows an interesting C++ feature that is related to the **for** loop: This example declares the loop control variable **i** and also initializes it to 1. The loop iterates as long as the value in variable **i** is less than 100. The loop increment part increases the value of variable **i** by 1. Thus, the upward-counting loop iterates 99 times with the value in variable **i** changing from 1 to 99.

The fourth example shows that a C++ **for** loop can declare and initialize multiple loop control variables. The loop initializes the variables **i** and **j** to 0 and **MAX**, respectively. The loop iterates as long as the value in variable **i** is less than that in variable **j**. The loop increment part increases the value in each of the variables i and j by 1.

Example:

The following program calculates and prints the sum of the even integers from 4 to 20.

```
#include <iostream.h>
int main()
{
    int sum=0;
```

```
        for (int i=4;i<=20;i=i+2)
        sum+=i;
    cout  <<sum<<endl;
    return 0;
}
```

3.2 The Open-Iteration for Loop

C++ allows any or all three parts of a **for** loop to be empty! For example, you can initialize a loop control variable *before* the loop's statement. You can also increment the loop control variable *inside* the loop's statements. What happens when a **for** loop has all three parts empty? The answer is that you get an open-iteration loop. The source code needs to initialize, increment, and test the loop iteration outside the three parts of the **for** loop. Here is a code snippet that illustrates the open-iteration **for** loop's features:

```
int i = 0; // initialize variable
int j;
for (;;) {
  cout << i << " : ";
            // test loop condition and exit if i >= 10
  if (i >= 10)
   break;
  j = i + i * i - 5;
  i++; // increment loop control variable
  cout << j << "\n";
}
```

This snippet initializes the variable **i** and uses that variable to control the iterations of the **for** loop. The loop displays the value in variable **i** and then tests the loop's condition. If the value in variable **i** is equal to or greater than 10, the loop exits using the **break** statement. The loop then calculates the value for variable **j**, increments variable **i**, and displays the

value in variable **j**. The statement **i++;** increments the loop control variable and allows iteration to progress.

3.3 The do-while Loop

The **do-while** loop iterates as long as a tested condition is true. The syntax for the **do-while** loop is:

```
do {
  // statements
} while (condition);
```

The syntax of the **do-while** loop shows that it tests the iteration condition *after* executing the loop's statement. Thus, the **do-while** loop always executes at least once. Here is an example of the **do-while** loop:

```
do
  cout << "Enter a positive integer : ";
  cin >> nNum;
while (nNum < 1);
```

This example shows a **do-while** loop that iterates as long as the value in the variable **nNum** is less than 1.

Example:

The following program calculates and prints the sum of the even integers from 4 to 20.

```
#include <iostream.h>
 int  main()
 {
        int sum=0;
        int i=4;
        do{
               sum+=i;
               i=i+2;
```

```
    }while(i<=20);
  cout <<sum<<endl;
  return 0;
}
```

3.4 The while Loop

The syntax for the **while** loop is:

```
while (condition)
 // statement or statement block
```

The syntax of the **while** loop shows that it tests the iteration condition *before* executing the loop's statement. Thus, the **while** loop will not execute if the tested condition is already false. Here is an example of the **while** loop:

```
int i = 0;
while (i * i < 1000)
 i++;
```

This example has a **while** loop that iterates as long as the squared value of variable **i** is less than 1000.

Example:

The following program finds and prints the smallest integer that is able to divide by 12 and 14

```
#include <iostream.h>
int main()
{
      int i=1;
      while(i%12!=0 || i%14!=0)
      i++;
      cout <<i<<endl;
  return 0;
}
```

3.5 Exiting Loops

The **break** statement exits the current loop. Thus to exit nested loops (more about these loops later in this chapter), you need to use a **break** statement for each loop.

Here are examples of using the **break** statement with the **do-while** and **while** loops:

```
// exit from do-while loop example
double fY, fX = 1.0;
do {
  fY = fX * fX + 10;
  if (fY > 10000.0)
    break;
  cout << "f(" << fX << ") = " << fY << "\n";
} while (fX < 100.0);

// exit from while loop example
double fY, fX = 1.0;
while (fX > 0.0 && fX < 100.0) {
  fY = fX * fX - 30;
  cout << "f(" << fX << ") = " << fY << "\n";
  if (fY > 1000.0)
    break;
};
```

The first example has a **do-while** loop that iterates as long as the value in variable **fX** is less than 100. The loop contains a statement that determines whether or not the value in the variable **fY** (which is based on the value of variable **fX**) exceeds 10,000. If this condition is true, the loop exits by executing the **break** statement in the **if** statement.

The second example has a **while** loop that iterates as long as the value in variable **fX** is positive and less than 100. The loop contains an

if statement that determines whether or not the value in the variable **fY** (which is based on the value of variable **fX**) exceeds 1000. If this condition is true, the loop exits by executing the **break** statement in the **if** statement.

3.6 Skipping Loop Iterations

C++ offers the **continue** statement to skip the remaining statements in a loop. Why skip the remaining loop statements? This condition arises when the loop statements examine a condition and conclude that the loop should not or need not proceed with executing the remaining statements. Here is an example of using the **continue** statement:

```
for (int i = -4; i < 5; i++) {
 if (i == 0)
 continue;
 double fX =  1.0 / i;
 cout << "1 / " << i << " = " << fX << "\n";
}
```

This code snippet shows a loop that displays reciprocal values. The loop has a control variable that changes values from —4 to 4, in increments of 1. The loop contains an **if** statement that determines whether or not the control variable contains 0. When this condition is true, the loop skips the remaining statements to *avoid* dividing by zero!

3.7 Nested Loops

C++ allows you to nest loops in any combination. For example, you can nest **for** loops, as shown in the following code snippet:

```
double fSum = 0;
for (int i = 10; i < 100; i++)
 for (int j = 0; j < i; j++)
  fSum += double(j * i);
```

This code snippet shows two nested **for** loops used to obtain a summation.

You can also nest different kinds of loops. Here is a code snippet that shows you nested **while** and **do-while** loops:

```
double fSum = 0;
int i = 10;
int j;
while (i < 100)
{
 j = 0;
  do {
              fSum += double(j++ * i);
      } while  (j < i);
  i++;
}
```

The nested loops obtain a summation, like the one in the example of the nested **for** loops.

Example: The following program uses two nested loops to draw the following shape

```
*********
*******
*****
***
*
*
***
*****
*******
*********
```

```
#include<iostream.h>
int main()
{
	for (int LineNumber=-9;LineNumber<=9;LineNumber=LineNumber+2)
		{
			int LineNumberTemp;
			if(LineNumber<0)
```

```
                LineNumberTemp=-1*LineNumber;
            else
                LineNumberTemp=LineNumber;

            for (int i=1;i<=LineNumberTemp;i++)
            cout<<'*';
            cout<<endl;
        }
    return 0;
}
```

3.8 Exercises

1. Identify and correct the error(s) in each of the following:

a)
```
if ( age >= 65 );
    cout << "Age is greater than or equal to 65" << endl;
  else
    cout << "Age is less than 65 << endl";
```

b)
```
if ( age >= 65 )
    cout << "Age is greater than or equal to 65" << endl;
  else;
    cout << "Age is less than 65 << endl";
```

c)
```
int x = 1, total;
while ( x <= 10 ) {
  total += x;
  ++x;
}
```

d)
```
While ( x <= 100 )
  total += x;
  ++x;
}
```

e)
```
while ( y > 0 ) {
  cout << y << endl;
  ++y;
}
```

2. What does the following program print?

```
1)  #include <iostream>
2)
3)  using std::cout;
4)  using std::endl;
5)
6)  int main()
7)  {
8)      int y, x = 1, total = 0;
9)
10)      while ( x <= 10 ) {
11)      y = x * x;
12)      cout << y << endl;
13)      total += y;
14)      ++x;
15)    }
16)
17)   cout << "Total is " << total << endl;
18)      return 0;
19)  }
```

3. Write a C++ program that utilizes looping and the tab escape sequence \t to print the following table of values:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

4. What does the following program print?

```
1)  #include <iostream>
2)
3)  using std::cout;
4)  using std::endl;
5)
6)  int main()
7)  {
8)    int count = 1;
9)
10)    while ( count <= 10 ) {
11)      cout << (count % 2 ? "****" : "++++++++")
12)         << endl;
13)      ++count;
14)    }
```

```
15)
16)    return 0;
17) }
```

5. What does the following program print?

```
1)  #include <iostream>
2)
3)  using std::cout;
4)  using std::endl;
5)
6)  int main()
7)  {
8)      int row = 10, column;
9)
10)     while ( row >= 1 ) {
11)     column = 1;
12)
13)       while ( column <= 10 ) {
14)       cout << (row % 2 ? "<" : ">");
15)       ++column;
16)     }
17)
18)     --row;
19)     cout << endl;
20)   }
21)
22)     return 0;
23) }
```

6. Determine the output for each of the following when **x** is **9** and **y** is **11** and when **x** is **11** and **y** is **9**.

a)
```
if ( x < 10 )
if ( y > 10 )
cout << "*****" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
```

b)
```
if ( x < 10 ) {
if ( y > 10 )
cout << "*****" << endl;
```

```
}
else {
cout << "#####" << endl;
cout << "$$$$$" << endl;
}
```

7. Modify the following code to produce the output shown. Use proper indentation techniques. You must not make any changes other than inserting braces. Note: It is possible that no modification is necessary.

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&" << endl;
```

a) Assuming x = 5 and y = 8 , the following output is produced.

```
@@@@@
$$$$$
&&&&&
```

b) Assuming x = 5 and y = 8 , the following output is produced.

```
@@@@@
```

c) Assuming x = 5 and y = 8 , the following output is produced.

```
@@@@@
&&&&&
```

d) Assuming x = 5 and y = 7 , the following output is produced. Note: The last three output statements after the else are all part of a compound statement.

```
#####
$$$$$
&&&&&
```

8. Write a program that reads in the size of the side of a square and then prints a hollow square of that size out of asterisks and blanks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 5, it should print

```
*****
*   *
*   *
*   *
*****
```

9. Write a program that displays the following checkerboard pattern

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

Your program must use only three output statements, one of each of the following forms:

```
cout ؛" *" >>
cout ؛' ' >>
cout << endl؛
```

10. Write a program that reads three nonzero double values and determines and prints if they could represent the sides of a triangle .

11. Write a program that reads three nonzero integers and determines and prints if they could be the sides of a right triangle.

12. Find the error(s) in each of the following:

a) For (x = 100, x >= 1, x++)

```
cout << x << endl;
```

b) The following code should print whether integer value is odd or even:

```
switch ( value % 2 )
  case 0:
     cout << "Even integer" << endl;
  case 1:
     cout << "Odd integer" << endl;
}
```

c) The following code should output the odd integers from 19 to 1:

```
for ( x = 19; x >= 1; x += 2 )
  cout << x << endl;
```

d) The following code should output the even integers from 2 to 100:

```
counter = 2;

do {
  cout << counter << endl;
  counter += 2; }
While ( counter < 100 );
```

13. Write a program that finds the smallest of several integers. Assume that the first value read specifies the number of values remaining and that the first number is not one of the integers to compare.

14 Write a program that calculates and prints the product of the odd integers from 1 to 15.

15. Write a program that prints the following patterns separately one below the other. Use for loops to generate the patterns. All asterisks (*) should be printed by a single statement of the form cout << '*';

```
(A)           (B)           (C)              (D)
*             **********    **********                *
**            *********      *********               **
***           ********        ********              ***
****          *******          *******             ****
*****         ******            ******            *****
******        *****              *****           ******
*******       ****                ****          *******
********      ***                  ***         ********
*********     **                    **        *********
**********    *                      *       **********
```

16. Assume i = 1, j = 2, k = 3 and m = 2. What does each of the following statements print? Are the parentheses necessary in each case?

a) cout << (i == 1) << endl;

b) cout << (j == 3) << endl;

c) cout << (i >= 1 && j < 4) << endl;

d) cout << (m <= 99 && k < m) << endl;

e) cout << (j >= i || k == m) << endl;

f) cout << (k + m < j || 3 - j >= k) << endl;

g) cout << (!m) << endl;

h) cout << (!(j - m)) << endl;

i) cout << (!(k > m)) << endl;

Lecture 4

User-Defined Data Types

4.1 Enumerated Types

Enumerated data types allow you to define a set of identifiers that have integer values associated with them. The associated values can be implicit (that is, automatically assigned by the compiler) or explicit (that is, you assign the numeric constants to some or all of the enumerated values). Using enumerated values replaces a set of integers with a more meaningful set of identifiers called *enumerators*. C++ allows you to declare enumerated types using the following syntax:

enum *enumeratedType* { *enumerator1*, *enumerator2*, };

The declaration of an enumerated type starts with the keyword **enum** and is followed by the name of the enumerated type identifier and the list of *enumerators*. This comma-delimited list is enclosed in braces and ends with a semicolon. Here is an example of an enumerated type that represents colors:

```
enum fewColors { clBlack, clWhite, clRed, clBlue, clGreen, clYellow };
```

The keyword *enum* starts the declaration of an enumerated type. The code snippet declares the enumerated type *fewColors* and specifies the enumerators *clBlack*, *clWhite*, *clRed*, *clBlue*, *clGreen*, and *clYellow*. The compiler assigns the value 0 to *clBlack*, 1 to *clWhite*, 2 to *clRed*, and so on. You can explicitly assign values to the enumerators as shown in the next example:

```
enum moreColors { mclBlack = 10, mclWhite, mclRed = 20, mclBlue,
        mclGreen, mclYellow };
```

This code snippet declares the enumerated type *moreColors* and includes explicit value assignments to some of the enumerators. The compiler assigns the value 10 to *mclBlack*, 11 to *mclWhite*, 20 to *mclRed*, 21 to *mclBlue*, 22 to *mclGreen*, and 23 to *mclYellow*.

Here's an example program, *DAYENUM*, that uses an enumeration for the days of the week:

```
// dayenum.cpp
// demonstrates enum types
#include <iostream>
using namespace std;
//specify enum type
enum days_of_week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
int main()
{
days_of_week day1, day2; //define variables
//of type days_of_week
day1 = Mon; //give values to
day2 = Thu; //variables
int diff = day2 - day1; //can do integer arithmetic
cout << "Days between = " << diff << endl;
if(day1 < day2) //can do comparisons
        cout << "day1 comes before day2\n";
return 0;
}
```

4.2 Structures

C++ supports the struct user-defined type, which defines *structures*. These structures are similar to records used in other programming languages. A

structure contains data members that either have a predefined data type or are themselves previously defined structures.

4.2.1 Declaring Structures

The general syntax for declaring a structure type is:

```
struct structureName
{
 type1 dataMember1;
 type2 dataMember2;
 // other data members
};
```

Let's look at a few examples. Here is a simple structure that defines the x-y coordinates of a point:

```
struct Point {
            int x;
            int y;
           };
```

The declaration defines the structure Point with the int-type data members x and y. Here is a structure that represents a complex number:

```
struct Complex {
               double x;
               double y;
              };
```

The declaration defines the structure Complex with the double-type data members x and y. Here is a structure that represents personal data:

```
struct Person
{
 char m_cFirstName[10];
```

```
  char m_cMiddleInitial;
  char m_cLastName[15];
  int m_BirthYear;
  double m_fWeight;
};
```

This declaration defines the structure Personal and declares a rich set of data members, which describe a name, a weight, and a BirthYear. Many of the data members in structure Personal are arrays of characters, which store ASCIIZ string data. Here is another example, one which uses nested structures:

```
struct Rectangle {
        Point ulc; // upper-left corner;
        Point lrc; // lower-right corner;
    };
```

This declaration defines the structure *Rectangle*, which contains the data members *ulc* and *lrc*, themselves previously defined structures.

4.2.2 Declaring Structure Variables

Declaring structure variables is no different from declaring variables with predefined types. Here is the general syntax:

```
// declaring a single variable
structureType structureVariable;
// declaring an array of structures
structureType structureArray[numerOfElemens];
```

Here are examples of declaring structure variables, using the structures that I declared in the last subsection:

```
Point Origin, StartPoint, EndPoint, Points[10];
Rectangle myRectangle;
```

```
Person Me, You, Us[30];
```

These examples declare the **Point**-type variables **Origin**, **StartPoint**, and **EndPoint**. The examples also declare the **Point**-type array **Points** to have 10 elements. They additionally declare the **Rectangle**-type variable **myRectangle**, the **Person**-type variables **Me** and **You**, and the **Person**-type array **Us**.

4.2.3 Accessing Structure Members

Accessing a data member of a structure involves using the dot access operator for a structure variable. Here is an example:

```
Point pointX;
pointX.x = 10;
pointX.y = 200;

Rectangle rectangle;
rectangle.ulc.x=20;
rectangle.ulc.y=30;
```

In the case of a pointer to a structure, use the pointer access operator -> to access the data members. Here is an example:

```
Point pointX;
Point *ptrX = &pointX;
ptrX->x = 10;
ptrX->y = 200;
```

4.2.4 Initializing Structures

C++ allows you to initialize the data members of structures. This feature resembles initializing arrays and follows similar rules. The general syntax for initializing a structure variable is:

structureType structureVariable = { *value1*, *value2*, };

The compiler assigns *value1* to the first data member of the variable *structureVariable*, *value2* to the second data member of the variable *structureVariable*, and so on. You need to observe the following rules:

- The assigned values should be compatible with their corresponding data members.
- You can declare fewer initialing values than data members. The compiler assigns zeros to the remaining data members of the structure variable.
- You cannot declare more initializing values than data members.
- The initializing list sequentially assigns values to data members of nested structures.
- The initializing list assigns values sequentially to data members that are arrays.

Keep in mind that the task of initializing structures is as simple or complex as the initialized structures themselves.

Here are examples of initializing structures:

```
struct Point
{
 double m_fX;
 double m_fY;
};

struct Rectangle
{
 Point m_UpperLeftCorner;
 Point m_LowerRightCorner;
 double m_fLength;
 double m_fWidth;
};

Point FocalPoint = { 12.4, 34.5 };
Rectangle Shape = { 100.0, 50.0, 200.0, 25.0 };

// calculate the length
Shape.m_fLength = Shape.m_UpperLeftCorner.m_fX -
            Shape.m_LowerRightCorner.m_fX;
// calculate the width
Shape.m_fWidth = Shape.m_LowerRightCorner.m_fY -
            Shape.m_UpperLeftCorner.m_fY;
```

This example declares the structures *Point* and *Rectangle*. The example also declares the *Point*-type variable *FocalPoint* and initializes its data members m_fX and m_fY with the values 12.4 and 34.5. The example further declares the *Rectangle*-type Shape and initializes the first two data members m_*UpperLeftCorner* and m_*LowerRightCorner*. Each one of these data members requires two initializing values since they have the type *Point*.

Thus, the compiler assigns the values 100.0, 50.0, 200.0, and 25.0 to *Shape.m_UpperLeftCorner.m_fX*, *Shape.m_UpperLeftCorner.m_fY*, *Shape.m_LowerRightCorner.m_fX*, and *Shape.m_LowerRightCorner.m_fY*, respectively.

4.3 Exercises

1. Write a structure specification that includes three variables—all of type int—called hrs,mins, and secs. Call this structure time.

2. When accessing a structure member, the identifier to the left of the dot operator is the name of

    a. a structure member.

    b. a structure tag.

    c. a structure variable.

    d. the keyword struct.

3. Write a definition that initializes the members of time1—which is a variable of type struct time, as defined in Question 4—to hrs = 11, mins = 10, secs = 59.

4. An enumeration brings together a group of

    a. items of different data types.

    b. related data variables.

    c. integers with user-defined names.

    d. constant values.

5. Write a statement that declares an enumeration called players with the values B1, B2, SS, B3, RF, CF, LF, P, and C.

6. A phone number, such as (212) 767-8900, can be thought of as having three parts: the area code (212), the exchange (767), and the number (8900). Write a program that uses a structure to store these three parts of a phone number separately. Call the structure phone. Create two structure variables of type phone. Initialize one, and have the user input a number for the other one. Then display both numbers. The interchange might look like this:

Enter your area code, exchange, and number: 415 555 1212
My number is (212) 767-8900
Your number is (415) 555-1212

7. A point on the two-dimensional plane can be represented by two numbers: an x coordinate and a y coordinate. For example, (4,5) represents a point 4 units to the right of the vertical axis, and 5 units up from the horizontal axis. The sum of two points can be defined as a new point whose x coordinate is the sum of the x coordinates of the two points, and whose y coordinate is the sum of the y coordinates. Write a program that uses a structure called point to model a point. Define three points, and have the user input values to two of them. Then set the third point equal to the sum of the other two, and display the value of the new point. Interaction with the program might look like this:

Enter coordinates for p1: 3 4
Enter coordinates for p2: 5 7
Coordinates of p1+p2 are: 8, 11

8. Create a structure called employee that contains two members: an employee number (type int) and the employee's compensation (in dollars; type float). Ask the user to fill in this data for three employees, store it in three variables of type struct employee, and then display the information for each employee.

Lecture 5

Functions

5.1 functions definition

All C++ functions have certain basic features. Each function has a name, a return type, and an optional parameter list. Functions can declare local constants and variables. Except for the function main you should prototype functions (that is declare them in advance). C++ functions have the following syntax:

```
returnType functionName(parameterList)
{
 // declarations

 // statements

 return expression;
}
```

Every function has a *return type* that appears before the name of the function. The *parameter list* follows the function's name and is enclosed in parentheses. The function returns a value using the **return** statement that typically appears at the end. A function may have more than one **return** statement.

The parameter list of a function may contain one or more *parameters,* which correspond to the *arguments* given the function when it is actually called. The list of parameters is comma-delimited, and each parameter has the following syntax:

parameterType[&] *parameterName*

You need to observe the following rules about the parameters of a function:

- Each parameter must have its own type. You cannot use the same type to declare multiple parameters (as you can when declaring variables).
- If a function has no parameters, the parentheses that come after the function's name contain nothing.
- The argument for a parameter is *passed by copy* (or, as it is sometimes said, "by value"), unless you insert the reference-of operator **&** after the parameter's type. When a parameter passes a copy of its argument, the function can alter only the copy of the argument used within the function itself. The original argument remains intact. By contrast, using the reference-of operator allows the argument to be *passed by reference* by declaring the parameter as a reference to its argument. In this case, the parameter becomes a special alias to its argument. Any changes the function makes to the parameter also affect the argument.
- Reference parameters take arguments that are the names of variables. You cannot use an expression or a constant as an argument to a reference parameter since an expression does not have an address as a variable does.
- Copy parameters take arguments that are constants, variables, or expressions. The type of argument must either match the type of the parameter or be compatible with it. You may use typecasting to tell the compiler how to adjust the type of the argument to match the type of the parameter.

Here are examples of functions:

```
double getSquare(double x) // one parameter
{
 return x * x;
```

```
}

double Square(double& x) // one parameter, modifies its argument
{
 x = x * x;
 return x;
}

int getMin(int nNum1, int nNum2) // two parameters
{
 return (nNum1 < nNum2) ? nNum1 : nNum2;
}

int getSmall(int nNum1, int nNum2, int nNum3); // three parameters
{
 if (nNum1 < nNum2 && nNum1 < nNum3)
  return nNum1;
 else if (nNum2 < nNum1 && nNum2 < nNum3)
  return nNum2;
 else
  return nNum3;
}
```

The first function, **getSquare**, has the return type **double** and the single **double**-type parameter **x**. The function returns the squared value of the parameter **x**. The function **getSquare** contains a single statement, namely the **return** statement.

The second function, **Square**, has the return type **double** and the single **double**-type reference parameter **x**. The function squares the value of the reference parameter and returns the new value in **x** (this value also affects the argument for function **Square**). Therefore, the function returns the squared value in two ways: first as the function's return value, and second using the reference parameter **x**.

The third function, **getMin**, has the return type **int** and the two **int**-type parameters **nNum1** and **nNum2**. The function returns the smaller of the values supplied by the arguments for the parameters. The function **getMin** has a single statement that returns the minimum number sought. This statement uses the conditional assignment operator.

The fourth function, **getMin**, has the return type **int** and the three **int**-type parameters **nNum1**, **nNum2**, and **nNum3**. The function returns the smallest value supplied by the arguments for the three parameters. The function **getSmall** uses a multiple-alternative **if-else** statement to obtain the sought-after minimum.

5.2 The Function Declaration

C++ also supports the forward declaration of functions (which is called *prototyping*). The forward declaration allows you to list the functions at the beginning of the source code. Such a list offers a convenient way to know what functions are in a source code file. In addition, using the prototypes gives the compiler advance notice of the names, return types, and parameter

lists of the various functions. You can then place the definitions of the functions in any order and not worry about the compile-time errors that occur when you call a function before you either declare it or define it. The general syntax for a function prototype is:

returnType functionName(parameterList);

Notice that the semicolon at the end is needed for the prototype but does not work in the function definition.

Here are the function declaration for the functions that are presented in the last section:

```
double getSquare(double x); // one parameter
double Square(double& x); // one parameter
int getMin(int nNum1, int nNum2); // two parameters
int getSmall(int nNum1, int nNum2, int nNum3); // three parameters
```

Example:

Write a C++ program that computes and printes the summation of the even numbers between 20 and 30. this program uses function to determine if the number is even or not.

```
#include<iostream.h>
bool even(int x);

int main()
{
        int sum=0;
        for (int i=20;i<=40;i++)
                {
                        if(even(i))
                                sum+=i;
                }
        cout<<"sum= "<<sum<<endl;
        return 0;
}
bool even(int x)
{
        if (x%2==0)
```

```
                return true;
        else
                return false;
}
```

5.3 Math Library Functions

Math library functions allow the programmer to perform certain common mathematical calculations. All functions in the math library return the data type double. To use the math library functions, include the header file <cmath>. Some math library functions are summarized in the following table. In the table, the variables x and y are of type double.

Method	Description	Example
ceil(x)	rounds **x** to the smallest integer not less than **x**	**ceil(9.2)** is **10.0** **ceil(-9.8)** is **-9.0**
cos(x)	trigonometric cosine of **x** (**x** in radians)	**cos(0.0)** is **1.0**
exp(x)	exponential function e^x	**exp(1.0)** is **2.71828** **exp(2.0)** is **7.38906**
fabs(x)	absolute value of **x**	**fabs(5.1)** is **5.1**

Lecture 6

Functions (2)

6.1 Recursive Functions

Recursion is a method in which a function obtains its result by calling itself. Successive recursive calls must pass different arguments and must reach a limit or condition where the function stops calling itself. These two simple rules prevent a recursive function from indefinitely calling itself. Conceptually, recursion is a form of iteration that does not use the formal fixed or conditional loop. Many algorithms (such as calculating factorials and performing a quicksort) can be implemented using either recursive functions or straightforward loops. Some algorithms are easier to implement using recursion. An example is the algorithm for parsing and evaluating mathematical expressions. This is because an expression may contain smaller expressions and therefore, recursion offers the best solution. In other words, the main expression may contain nested expressions. Here is an example:

Z = ((X + Y) * X)) + (X * Y) / (1 + X);

The above statement contains the nexted expressions ((X + Y) * X)), (X + Y), (X * Y), and (1 + X).

Example:
The following function calculates factorials.

```
#include<iostream.h>
int factorial(int x);

int main()
{
        int no;
```

```
        cin>>no;
            if(no<0)
            {
                cout<<" \n Invalid input"<<endl;
                return 0;
            }
            else
            cout<<" \n factorial= "<<factorial(no)<<endl;
        return 0;
}

int factorial(int x)
{
            if (x<=1) return 1;
                else
            return x*(factorial(x-1));
}
```

6.2 Default Arguments

C++ allows you to assign default arguments for parameters. The syntax for the default argument is:

parameterType parameterName = initialValue

C++ requires that you observe the following rules for declaring and using default arguments:

1. When you assign a default argument to a parameter, you must assign default arguments to all subsequent parameters.
2. You may assign default arguments to any or all parameters, as long as you obey rule number 1.
3. The default arguments feature divides the parameter list of a function into two parts. The first part contains parameters with no default arguments (this list may be empty if you assign default arguments to all parameters); the second part contains parameters with default arguments.

4. To use a default argument for a parameter, omit the argument for that parameter in a function call.
5. If you use a default argument for a parameter, you must use default arguments for all subsequent parameters. In other words, you cannot pick and choose the default arguments, because the compiler is unable to discern which argument goes to which parameter. (After all, this is programming and not black magic!)

For example, the following declaration declares the **myPower** function:

```
double myPower(double fBase, int nExponent = 2);
```

The function **myPower** has the **double**-type parameter **fBase** and the **int**-type parameter **nExponent**. The latter parameter has the default argument of 2. Thus, you can use the function myPower in this fashion:

```
double fX = 12.5;
double fXSquared = myPower(fX);
double fXCubed = myPower(fX, 3);
```

The first call to function **myPower** has only one argument. The compiler resolves this call by using the default argument of 2 for parameter **nExponent**. Consequently, the function **myPower** returns the square of the first argument's value when you omit the argument for the exponent. By contrast, the second call to function **myPower** uses the arguments **fX** and 3. In this case, the compiler does not use the default argument for parameter **nExponent**, since it has been given both arguments explicitly.

Example:

```
#include<iostream.h>
int sum(int x=2,int y=4,int z=3);
```

```
int main()
{

        int m;
        m=sum();
        cout<<"sum()="<<m<<endl;

        m=sum(1);
        cout<<"sum(1)="<<m<<endl;

        m=sum(5,6);
        cout<<"sum(5,6)="<<m<<endl;

        return 0;
}

int sum(int x,int y,int z)
{
        int n=x+y+z;
        return n;
}
```

6.3 Constant Parameters

By default, a function can alter the data passed by the arguments to its parameters. If the parameter is a not a reference parameter, then the changes made to the argument are limited to the function's scope. By contrast, if the parameter is a reference parameter, then the changes made to the argument go beyond the function's scope. You can tell the compiler that the function should not alter the argument of a parameter by declaring that parameter as a constant parameter. The declaration uses the keyword **const** and has the following general syntax:

const *parameterType*[&] *parameterName* [= *defaultArg*]

Example: The following example shows how to use pointers:

```
int x;
```

```
int* px;
cin >> x;
px = &x;
*px =*px+ 10;
cout << *px; // display value in variable nCount
```

Example: this example shows the call by reference:

```
#include<iostream.h>
void sum(int *x,int* y);

int main()
{
        int *n,*m;
        n=new int;
        m=new int;
        *n=2;
        *m=3;
        sum(n,m);
        cout<<"sum= "<<*n<<endl;

        return 0;
}

void sum(int *x ,int *y)
{
        *x=*x+*y;
        return;
}
```

To prevent the function from changing the argument we use ***const*** before the argument as follow:

```
#include<iostream.h>
int sum(const int *x, const int* y);

int main()
{
        int *n,*m;
        n=new int;
        m=new int;
        *n=2;
        *m=3;
```

```
	int p=sum(n,m);
	cout<<"sum= "<<p<<endl;

	return 0;
}

int sum(const int *x, const int *y)
{
	int z=*x+*y;
	return z;
}
```

6.4 Function Overloading

Overloaded functions in C++ are a valuable feature that allows you to declare functions in sets of versions that have the same name but different parameters. These parameters form each version's *signature*. Using an overloaded function empowers you to use the same name for a set of function versions that perform similar tasks on different data types. For example, you can define the function **Square** to obtain the squares of parameters that have the types **int**, **long**, **float**, and **double**. Here are the declarations of the overloaded function **Square**:

```
double Square(int i);
double Square(long i);
double Square(float i);
double Square(double i);
```

Each version of function **Square** has a different parameter list. When you call the function **Square**, the compiler examines the data type of the argument to decide which version of function **Square** to call.

Example:

```
#include<iostream.h>
double Square(int i);
double Square(long i);
double Square(float i);
```

```
double Square(double i);

int main()
{
        cout<<Square(1.3)<<endl;
        return 0;
}

double Square(int i)
{       return double(i*i);  }

double Square(long i)
{       return double(i*i);  }

double Square(float i)
{       return double(i*i);  }

double Square(double i)
{       return double(i*i);  }
```

Example:

```
#include<iostream.h>

int sum(int i, int j);
int sum(int i, int j, int k);
int sum(int i, int j, int k, int l);

int main()
{
        cout<<sum(2,3,5)<<endl;
        return 0;
}

int sum(int i, int j)
{  return i+j;  }

int sum(int i, int j, int k)
{  return i+j+k;  }

int sum(int i, int j, int k, int l)
{  return i+j+k+l; }
```

6.5 Exercises

1. Show the value of x after each of the following statements is performed:
   a) x = fabs(7.5)
   b) x = floor(7.5)
   c) x = fabs(0.0)
   d) x = ceil(0.0)
   e) x = fabs(-6.4)
   f) x = ceil(-6.4)
   g) x = ceil(-fabs(-8 + floor(-5.5)))

2. Write a function multiple that determines for a pair of integers whether the second integer is a multiple of the first. The function should take two integer arguments and return true if the second is a multiple of the first, false otherwise. Use this function in a program that inputs a series of pairs of integers.

3. Write a program that inputs a series of integers and passes them one at a time to function even, which uses the modulus operator to determine whether an integer is even. The function should take an integer argument and return true if the integer is even and false otherwise.

4. Write a function that displays at the left margin of the screen a solid square of asterisks whose side is specified in integer parameter **side**. For example, if **side** is **4**, the function displays

```
****
****
****
****
```

5. Write program segments that accomplish each of the following:
   a) Calculate the integer part of the quotient when integer a is divided by integer b.
   b) Calculate the integer remainder when integer a is divided by integer b.

c) Use the program pieces developed in a) and b) to write a function that inputs an integer between 1 and 32767 and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer 4562 should be printed as

4 5 6 2

6. Raising a number n to a power p is the same as multiplying n by itself p times. Write a function called power() that takes a double value for n and an int value for p, and returns the result as a double value. Use a default argument of 2 for p, so that if this argument is omitted, the number n will be squared. Write a main() function that gets values from the user to test this function.

7. Write a function called zeroSmaller() that is passed two int arguments by reference and then sets the smaller of the two numbers to 0. Write a main() program to exercise this function.

8. Write a function that takes two Distance values as arguments and returns the larger one. Include a main() program that accepts two Distance values from the user, compares them, and displays the larger.

9. Write a function called swap() that interchanges two int values passed to it by the calling program. (Note that this function swaps the values of the variables in the calling program, not those in the function.) You'll need to decide how to pass the arguments. Create a main() program to exercise the function.

10. Write a function that, when you call it, displays a message telling how many times it has been called: "I have been called 3 times", for instance. Write a main() program that calls this function at least 10 times. Try implementing this function in two different ways. First, use a global variable to store the count. Second, use a local static variable. Which is more appropriate? Why can't you use a local variable?

11. Write a function that returns the smallest of three double-precision, floating-point numbers.

12. An integer number is said to be a *perfect number* if the sum of its factors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number, because 6 = 1 + 2 + 3. Write a function **perfect** that determines whether parameter **number** is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

13. An integer is said to be prime if it is divisible by only 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

   a) Write a function that determines whether a number is prime.

   b) Use this function in a program that determines and prints all the prime numbers between 1 and 1000.

14. Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367.

15. Write a recursive function **power(base, exponent)** that, when invoked, returns

$$base^{exponent}$$

$$base^{exponent} = base \cdot base^{exponent-1}$$

$$base^{1} = base$$

16. The greatest common divisor of integers **x** and **y** is the largest integer that evenly divides both **x** and **y**. Write a recursive function **gcd** that returns the greatest common divisor of **x** and **y**. The **gcd** of **x** and **y** is defined recursively as follows: If **y** is equal to **0**, then **gcd(x, y)** is **x**; otherwise **gcd(x, y)** is **gcd(y, x % y)**, where **%** is the modulus operator.

Lecture 7

Objects and Classes

7.1 Local Variables

local variables are defined within functions:

```
#include<iostream.h>
int sum(int x,int y);
 int main()
{
int a=2;          //local variable
int b=3;          //local variable
        cout<<"sum= "<<sum(a, b)<<endl;
        return 0;
}
int sum(int x, int y)
{
        int z;  //local variable
        z=x+y;
        return z;
}
```

7.2 Global Variables

global variables are defined outside of any function. (They're also defined outside of any class, as we'll see later.) A global variable is visible to all the functions in a file (and potentially in other files). More precisely, it is visible to all those functions that follow the variable's definition in the listing. Usually you want global variables to be visible to all functions, so you put their declarations at the beginning of the listing. Global variables are also sometimes called external variables, since they are defined external to any function.

Example:

```
#include<iostream.h>
int a;            //global  variable
```

```
int b=3;          //global  variable

int sum();
 int main()
{
        a=2;

        cout<<"sum= "<<sum()<<endl;
        return 0;
}
int sum()
{
        int z;   //local variable
        z=a+b;
        return z;
}
```

7.3 Simple Classes

Let's look at a simple class that gives you a general feel for the declaration of a class and the definition of its components. The following example illustrates a simple *Employee* class:

```
#include <iostream.h>
class Employee
{
        private:
           int No;
           int Salary;

        public:
           // constructor
           Employee(int eNo,int eSalary):No(eNo), Salary(eSalary)
                {
                        //No=eNo;
                        //Salary=eSalary;
                }
           //destructor
           ~Employee()
                {
                        cout<<"\n Employee destructor \n";
                }
           void setNo(int eNo)
```

```
            { No=eNo;}
        void setSalary(int eSalary)
            {Salary=eSalary;}
        int getNo()
            {return No;}
        int getSalary()
            {return Salary;}
};

main()
{
 Employee x(2,200);
 cout<<"no= "<<x.getNo()<<endl;
 cout<<"Salary= "<<x.getSalary()<<endl;
 x.setNo(4);
 x.setSalary(300);
 cout<<"no= "<<x.getNo()<<endl;
 cout<<"Salary= "<<x.getSalary()<<endl;
 return 0;
}
```

Example:

the following example illustrates a simple *Rectangle* class:

```
#include <iostream.h>
class Rectangle
{
 private:
   double m_fLength;
   double m_fWidth;

 public:
   // constructor
   Rectangle(double fLength = 0, double fWidth = 0)
        {
        m_fLength = fLength;
        m_fWidth = fWidth;
        }
  //destructor
~Rectangle()
        {
        cout<<"\n Rectangle destructor \n";
        }
```

```
    double getLength(){ return m_fLength;}
    double getWidth(){ return m_fWidth;}
    void setLength(double fLength){m_fLength = fLength;}
    void setWidth(double fWidth){ m_fWidth = fWidth;}
    double getArea(){return m_fLength * m_fWidth;}
};

main()
{
 Rectangle rect(10.212, 20.543);

 cout << "Length = " << rect.getLength() << "\n";
 cout << "Width  = " << rect.getWidth() << "\n";
 cout << "Area   = " << rect.getArea() << "\n";
 cout << "\n";

 rect.setLength(5.142);
 rect.setWidth(15.453);

 cout << "Length = " << rect.getLength() << "\n";
 cout << "Width  = " << rect.getWidth() << "\n";
 cout << "Area   = " << rect.getArea() << "\n";

 return 0;
}
```

7.4 Static Class Data

If a data item in a class is declared as static, only one such item is created for the entire class, no matter how many objects there are. A static data item is useful when all objects of the same class must share a common item of information. A member variable defined as static has characteristics similar to a normal static variable: It is visible only within the class, but its lifetime is the entire program. It continues to exist even if there are no objects of the class. Here's an example, that demonstrates a simple static data member:

```
#include <iostream.h>
class foo
{
        private:
                static int count;
```

```
        public:
                foo()
                { count++; }
                int getcount()
                { return count; }
};

int foo::count = 0;

int main()
{
        foo f1, f2, f3;
        cout << "count is " << f1.getcount() << endl;
        cout << "count is " << f2.getcount() << endl;
        cout << "count is " << f3.getcount() << endl;
        return 0;
}
```