UNIT 1 OVERVIEW OF C++

Structure **Page Nos.**

1.0 INTRODUCTION

In this Unit, some important features of C++ have been discussed. Most of the object-oriented features of C++ will be dealt with more elaborately in the remaining Units. However, the Object-Oriented Programming Paradigm features and the need for such a paradigm have been introduced in this Unit. Though there is an advantage of saving the time for transfer of control and storing return addresses in the case of Macros, several disadvantages are present there. These have been discussed in this Unit. The time to write a program is reduced due to presence of library routines. There are instances when we use some routines frequently which are not part of library. We usually write the routing repeatedly when we are using it in different programs. To avoid this, we can always make a library of routines. The method of making a library of our own routines has also been discussed in this Unit.

1.1 OBJECTIVES

After going through this Unit, you should be able to:

- differentiate different programming paradigms;
- define various statements of C++;
- use functions in C++;
- make a library using C++, and
- define macros in C++.

1.2 PARADIGMS OF PROGRAMMING LANGUAGES

The term *paradigm* describes a set of techniques, methods, theories and standards that together represent a way of thinking for problem solving. According to [Wegner, 1988], paradigms are **"patterns of thought for problem solving ".** Language paradigms were associated with classes of languages. First the paradigms are defined. Thereafter, programming languages according to the different paradigms are classified. The language paradigms are divided into two parts: **imperative** and

declarative paradigms as shown in the Figure 1. Imperative languages can be further as classified into **procedura**l and **object-oriented** approach. Declarative languages can classified into **functional languages** and **logical languages**. In *Figure 1*, the examples of languages in each category are also given.

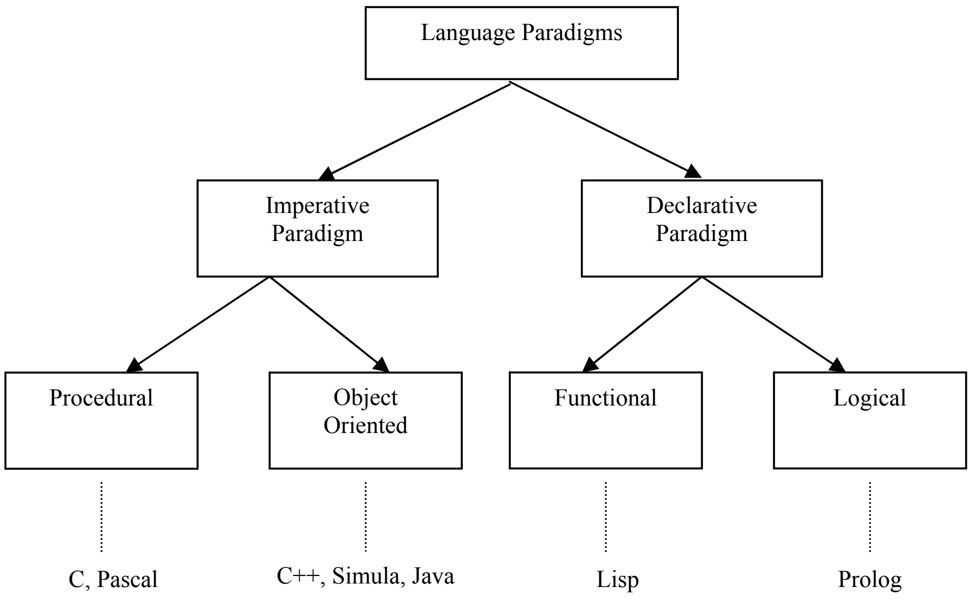


Figure 1: Language Paradigms

Imperative Paradigms: The meaning of imperative is **"expressing a command or order".** So, the programming languages in this category specify the step-by-step explanation of command. Imperative programming languages describe the details of **how** the results are to be obtained in terms of the underlying machine model. The programs specify step-by-step the entire set of transitions that the program goes through. The programs starts from an initial state, goes through the transitions and reaches a final state. Within this paradigm, we have the procedural approach and object-oriented approach.

Procedural paradigm: Procedural Languages are **statement oriented** with the variables holding values. In this language the execution of a program is modeled as a series of states of states of variable locations. We have two kinds of statements. **Non-executable** statements allocate memory, bind symbolic names to absolute memory locations and initialize memory. **Executable** statements like computation, control flow, and input/output statements. The popular programming languages in this category are *Ada, fortran, Basic, Algol, Pascal, Cobol, Modula, C, etc.*

Object-oriented paradigm: The object-oriented paradigm is centered on the concept of the object. Everything is focused on objects. Can you think what is an object? In this language, program consists of two things: first, a set of objects and second, the way they interact with each other. Computation in this paradigm is viewed as the simulation of real world entities. The popular programming languages in this paradigm are C++, Smalltalk and Java.

Declarative paradigm: In this paradigm, programs declare or specify what is to be computed without specifying how it is to be achieved. Declarative programming is also known as value-oriented programming. Declarative languages describe the relationship between variables in terms of functions and inference rules.
The language executor applies a fixed method to these relations to produce a desired result. It is mainly it is used in solving artificial intelligence and constraint-satisfaction problems. Declarative paradigm is further divided into two categories: functional and logical paradigms.

Functional paradigm: In this paradigm, a programme consist of a **collection of functions**. A function just computes and returns a value. A program consists of calling a function with appropriate arguments, but any function can make use of other functions also. The main programming languages in this category are **Lisp, ML Scheme,** and **Haskell**.

Logic paradigm: In this paradigm programs only explain what is to be computed not how to compute it. Here program is represent by a set of relationships, between objects or property of objects known as predicate which are held to be true, and a set of logic/clauses (i.e., if A is true, then B is true). Basically, logic paradigm, integrates data and control structures. The **Prolog** language is perhaps the most common example, **Mercury** language is a more modern attempt at creating a logic programming language.

Programming is an art. This art of programming has seen many evolutions. The basic idea for evolution was to develop simpler, maintainable, dependable, efficient, reusable programs. Let us now trace the evolution of object-oriented programming in slightly more detail.

1.2.1 Procedural Programming

In this paradigm, we divide a problem into sub-problems recursively and then we write a procedure for each sub-problem. Procedures communicate with each other by parameters. In this paradigm, data structures are declared locally or globally. The procedures can contain local data. Pascal and C support this programming approach. For example, if you want to implement a stack, you divide the problem into smaller problems like how to push value in a stack, how to pop a value and how to find whether stack is full or empty. Then you write functions for Push, Pop, stack empty and stack full and call these functions using its parameters to implement a stack in a program. The stack size is declared in main program using a data structure (may be an array).

1.2.2 Modular Programming

In this style, the program is divided into modules. Each module will contain all the procedures, which are related to each other and the data on which they act. Modular programming is the other name for Data hiding principle. The reason for this name is that the data in a module cannot be accessed by the procedures in another module unless it has been specified that it can be done so. Modula 2 supports this notion. For example, the complete set of operations of a stack including the data structure and functions may reside in a module.

The advantages of this paradigm are that we can have different files for a simple program. Each file can be separately compiled and executable file can be made from them. So, if there is any error in one module, the entire program need not be compiled. Only the module in which the error has occurred can be recompiled separately. This will reduce the total compilation time.

C enables modular programming by providing provision for including files and separate compilation facilities. The context of module is supported by the class concept of C++.

Data Abstraction

Data Abstraction = Modular programming + Data hiding principle.
This concept is supported by C++. In C++, classes can be defined in which the data can be specified as Private. This Private data can be accessed only by the member functions of the class. Then, we can define the class as a user defined data type which is the other name of Data Abstraction.

So, in this paradigm, we have to decide the types we want and then, we have to provide a full set of operations for each type. For example, in the data structure of stack, the data stored in it cannot be accessed or modified by any other class except the functions of stack class. Then, it can be classified as Data Abstraction.

1.2.3 Object Oriented Programming

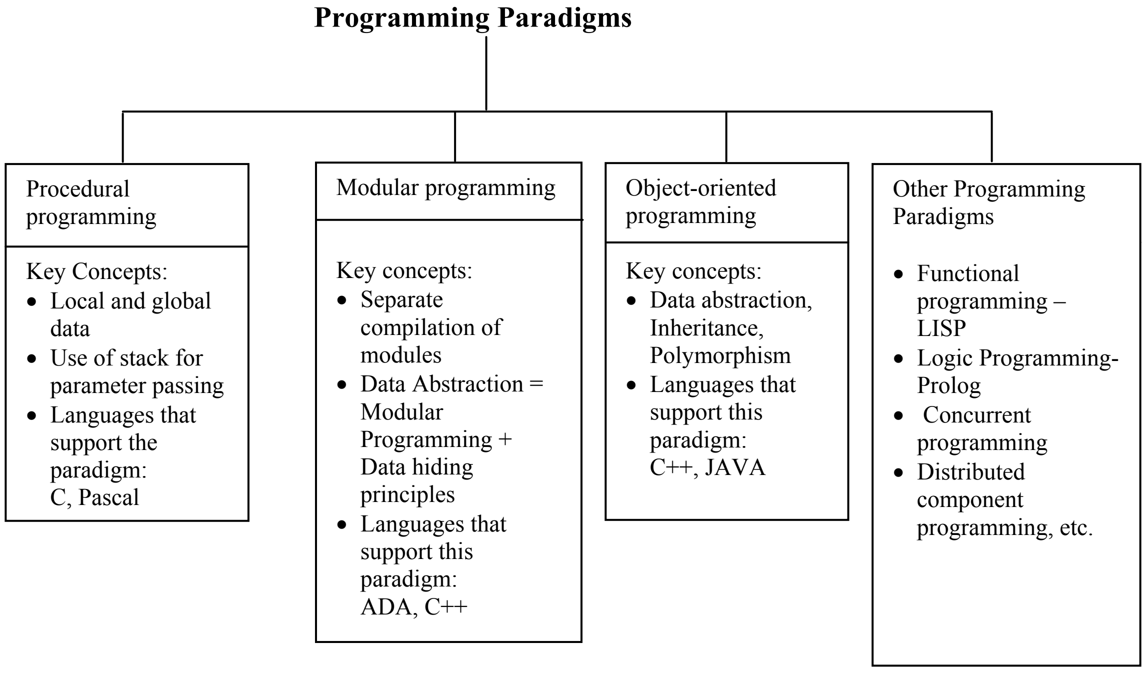
When we define data types (user defined data types) we may find commonality among them. Also, when we think of defining a new class, we may find that there is another class which is already processing most of the features of the class, we wish to have. Under such a situation, we can define a class with only additional features and explicitly state that it includes all the features of another class which has been already defined.

This concept is known as inheritance. The object-oriented programming paradigm is made up of Abstraction and Inheritance.

So, OOP = Data Abstraction + Inheritance.

In this paradigm, we have to decide the classes we want; provide a full set of operations for each class and then we have to make commonality explicit by using inheritance. These concepts will be further dealt with in more details in this Block.

A brief recapitulation of these programming paradigms are given in the following diagram:


Programming Paradigms
Procedural programming
Key Concepts:
• Local and global data
• Use of stack for parameter passing
• Languages that support the paradigm: C, Pascal
Modular programming
Key concepts:
• Separate compilation of modules
• Data Abstraction = Modular Programming + Data hiding principles
• Languages that support this paradigm: ADA, C++
Object-oriented programming
Key concepts:
• Data abstraction, Inheritance, Polymorphism
• Languages that support this paradigm: C++, JAVA
Other Programming Paradigms
• Functional programming – LISP
• Logic Programming- Prolog
• Concurrent programming
• Distributed component programming, etc.

☞ Check Your Progress 1

1) The various programming paradigms are ________________. ______________, _________________.

2) C++ supports:
   a. Procedural Programming
   b. Modular Programming
   c. Object-oriented programming

   Mark the correct answer(s).

3) Data abstraction is a ________of object-oriented programming paradigm.

1.3 C++ PROGRAMMING LANGUAGE: A RE-VISIT OF CONCEPTS OF C/C++

In this Section, we will define the term expression and discuss different type of C++ operations.

An expression is composed of one or more operations. The objects of an operation are referred to as operands. Operators represent the operations.

Operators that act on the operand are referred to as 'Unary' operators.
Example: -6 (Uniary minus).

Operators who act on two operands are referred to as 'Binary' operators.
Example: x * y (multiplication) 2+5 (addition).

The evaluation of an expression results in one or more operations, yielding a result. When two or more operations are combined, the expression is referred to as a 'Compound' Expression. The "precedence" and "associativity" of the operators determine the order of the operator evaluation.

The simplest form of an expression consists of a single literal constant of a variable. This "operand" is without an operator. The result is the operand's value. For example, here are three simple expressions:

3.14159
"melancholia"
UpperBound.

The result of 3.14159 is 3.14159. Its type is double. The result of melancholia is the address in memory of the first element of the string. Its type is char*. The result of UpperBound is its rvalue (that is the value bounded with the variable. In other words, the present value stored in the location held by this variable). Its type is determined by its definition.

Arithmetic Operators

The following table lists the Arithmetic operators:

Operators	**Function**	**Use**
*	Multiplication	expr * expr
/	Division	expr / expr
%	Modulus (Remainder)	expr % expr
+	Addition	expr + expr
-	Subtraction	expr – expr

Division between two integers results in an integer output. If the quotient contains a fractional part, it is truncated. Example:

21/6 = 3

The modulus operator (5) can be applied only to integers. The left operand of the % is divided. The divisor is the operator's right operand. Example:

```
86.3 % 5          // error: floating point operand
65 % 3            // OK : Result is 2
40 % 8            // OK: Result is 0
```

Equality, Relational and Logical Operators

The equality, relational and logical operators evaluate to either true or false. A true condition yields 1: a false condition yields. The following table lists the Equality, Relational and Logical Operators:

Operators	**Function**
!	Logical NOT
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
= =	equality
!=	inequality
&&	logical AND
| |	logical OR

Except of ! all are binary operators.

The logical AND ("&&") operator evaluates to true only if both its operators evaluate to true. The logical OR ("||") operator evaluates to true if either of its operands evaluates to true. The operators are evaluated from left to right.

Evaluation stops as soon as the truth or falsity of the expression is determined. The logical NOT ("!") operator evaluates to true, if its operand has a value of Zero. Otherwise, it evaluates to false.

Assignment operators

The left operand of the assignment operator ("=") must be an lvalue. The term lvalue is derived from the variable position to the left of the assignment operation. You might think of lvalue as meaning location value. The effect of an assignment is to store a new value in the storage associated with left operand. For example, given the following three definitions:

```
int i, *ip, ia[4];
```

the following are legal assignments:

```
ip = &i;
i=ia[0]+1;
ia[*ip]=1024;
*ip=i*2+i*a[i];
```

The data type of the result is the type of its left operand. Assignment operators can be concatenated provided that each of the operands being assigned is of the same general data type. For example,

```
int i, j;
i=j=0; // OK: each assigned 0
```

0 is assigned to j and i. The order of evaluation is right to left.

The compound assignment operator also provides a measure of notational compactness. For example,

```
int arrayprod (int ia[ ], int sz)
{
        int prod = 0;
        for (int i = 0; i < sz; ++i)
        prod * = prod [i];
        return prod;

}
```

The general syntactic form of the compound assignment operator is
a op = b
where op = may be one of the following ten operators: +=, –=,*=,/=,%=,<<=,>>=, &=, ^=,| =. Each compound operator is equivalent to the following long hand assignments:

a = a op b;
so, a += b is equivalent to a = a + b

Increment and Decrement Operators

Two special operators are used in C++, namely incrementer and decrementer. These operators are used to control the loop in an effective manner. There are two types of incrementers: Prefix incrementer (++i) and postfix incrementer (i++).

In Prefix incrementer, first it is incremented and then the operations are performed. On the other hand, in postfix incrementer, first the operations are performed and then it is incremented. However, the result of the incremented value will be same in both the cases. For example:

```
i=8;
x=++i;
cout <<x; //9 will be printed
j = 7;
y = j++;
cout << y; //7 will be printed
cout <<j; //8 will be printed.
```

The decrementer is also similar to incrementer. The symbol '--' is used for decrementing. There are two types of decrementers. They are prefix decrementer (--i) and postfix decrementer (i--). For example:

```
i=8;
x=--i;
cout <<x; //7 will be printed.
j=7;
y=j--;
cout <<y; //7 will be printed
cout <<j;//6 will be printed.
```

The sizeof operator

The 'sizeof' operator returns the size, in bytes, of an expression or type specifier. It may occur in either of the two forms:

```
sizeof (type-specifier);
sizeof expr;
```

For example,

```
sizeof (short); // returns the storage allocated to a short integer which is
                // machine dependent
```

sizeof (stack); // stack is a class. So storage occupied by it, will be returned.

The Arithmetic If Operator

The Arithmetic If operator, the only ternary operator in C++, has the following syntactic form;

```
exprl ? expr2: expr3;
```

exprl is always evaluated. If it evaluates to a true condition – that is, any non-zero value then expr2 is evaluated, otherwise expr3 is evaluated. The following program illustrates the usage of this operator:

```
# include < iostream.h>
void main ( )
{
        int i= 10, j =20;
        cout << "The larger value of : <<i<< "and" <<j<< "is" <<(i>j?i:j) << endl;
}
```

When compiled and executed, the program generates the following output:

```
The larger value of 10 and 20 is 20.
```

Comma Operator

A Comma expression is a series of expressions separated by a Comma(s). These expressions are evaluated from Left to Right. The result of a Comma expression is the value of Right most expression. In the following example, each side of the arithmetic if operator is a Comma expression. The value of the first Comma expression is 1; the value of second is 0.

```
main ( )
{ int ival = (ia!= 0) ? ix = index ( ), ia [ix] = ix, 1: (set-array (ia),0);
}
```

The above expression sets an array if it does not exist or gets and assign a value to an index element equal to the index value and returns 1.

The Bitwise Operators

A bitwise operator interprets its operand(s) as an ordered collection of bits. Each bit may contain either a 0 (off) or a 1 (on) value. A bitwise operator allows the programmer to test and set individual bits.

The operands of the bitwise operation must be of an integral type. (Is char an integral type? Yes). The following table lists the bitwise operators:

Operator	Function
~	bitwise NOT
<<	Left Shift
>>	Right Shift
&	bitwise AND
^	bitwise XOR
|	bitwise OR

The bitwise NOT operator (~) flips, the bits of its operands. Each 1 bit is set to 0 and 0 bit is set to 1. For example, unsigned char bit = o227

1	0	0	1	0	1	1	1

On applying bitwise NOT, operator will give:

0	1	1	0	1	0	0	0

The bitwise left shift operator ("<<") shifts the bits to the left keeping the same number of bits by dropping shifted bits off the end, and filling in with zeroes from the other end.

For example, let x = 33 (0010 0001) (8 bits). Now, x <<results in 0100 0010.

The bitwise Right Shift operator (">>") shifts the bits to the Right keeping the same number of bits by dropping shifted bits off the end, and filing in with zeroes from other end.

For example, let x = 33 (0010 00001) (8 bits). An operation, x>>3 result in (0000 0100).

The bitwise AND operator ("&") takes two integral operands. For each bit position, the result is a 1-bit if both operands contain 1 bit; otherwise, the result is a 0 bit. This operator is different from logical AND operator ("&&").

For example:

Unsigned char result:

Unsigned char b1= o145	0	1	1	0	0	1	0	1
Unsigned char b2= o257	1	0	1	0	1	1	1	1
Result = b1 & b2	0	0	1	0	0	1	0	1

The bitwise XOR (exclusive or) operator ('^') takes two integral operands. For each bit position, the result is a 1-bit if either but not both operands contain a 1-bit. Otherwise, the result is 0-bit. For example, the result on bi^b2 operation would be:

b1^b2 =	1	1	0	0	1	0	1	0

The bitwise OR operator ("|") takes two integral operands. For each bit position, the result is a 1-bit if either or both operands contain a 1 bit; otherwise, the result is a one bit. For example, the result on b1|b2 operation would be :

b1|b2 =	1	1	1	0	1	1	1	1

Precedence

Operator precedence is the other in which operator are evaluated in a compound expression. Operators who have the same precedence are evaluated from left to right.

:: ::	Scope resolution Global	class_name :: member ::name
.	Member selection	Object.member
→	Member selection	Pointer →member
[ ]	Subscripting	Pointer {expr}
( )	Function call	expr (expr_list)
( )	Value construction	type (expr_list)
++ --	Post-increment Post-decrement	lvalue ++ lvalue --
Sizeof	Size of object	sizeof expr

sizeof	Size of type	sizeof (type)
++	Pre-increment	++ lvalue
--	Pre-decrement	-- lvalue
~	Complement	~ expr
!	Not	!expr
-	Unary minus	-expr
+	Unary plus	+ expr
&	Address of	& lvalue
*	Dereference	*expr
new	Create (allocate)	new type
delete	Destroy (de-allocate)	delete pointer
delete[ ]	Destroy array	delete[ ] pointer
( )	Cast (type conversion)	(type)expr
*	Member selection	Object .* pointer-to-pointer
→*	Member selection	Pointer →* pointer-to-pointer
*	Multiply	expr * expr
/	Divide	expr / expr
%	Modulo (remainder)	expr % expr
+	Add (plus)	expr + expr
-	Subtract (minus)	expr-expr
<<	Shift Left	expr<<expr
>>	Shift Right	expr>>expr
<	Less than	expr < expr
<=	Less than or equal	expr <= expr
>	Less than	expr > expr
>=	Greater than or equal	expr >= expr
= =	Equal	expr = = expr
!=	Not equal	expr != expr
&	Bitwise AND	expr & expr
^	Bitwise exclusive OR	expr ^ expr
|	Bitwise exclusive OR	expr|expr
&&	Logical AND	expr && expr
| |	Logical inclusive OR	expr | | expr
? :	Conditional expression	Expr ? expr:expr
=	Simple assignment	lvalue = expr
*=	Multiply and assign	lvalue *= expr
/=	Divide and assign	lvalue /= expr
%=	Modulo and assign	lvalue %= expr
+=	Add and assign	lvalue += expr
-=	Subtract and assign	lvalue –= expr
<<=	Shift left and assign	lvalue <<= expr
>>=	Shift right and assign	lvalue >>= expr
&=	AND and assign	lvalue &= expr
|=	Inclusive OR and assign	lvalue |= expr
^=	Exclusive OR and assign	lvalue ^= expr
Throw	Throw exception	Throw expr
,	Comma (sequencing)	expr, expr

Reserved Words

The following identifiers are reserved for use as keywords, and may not be used otherwise:

asm	continue	float	new	signed	try
auto	default	For	operator	Sizeof	typedef

break	delete	friend	private	static	union
case	do	goto	protected	Struct	unsigned
catch	double	If	public	switch	virtual
char	else	inline	register	Template	void
class	enum	Int	return	This	volatile
const	extern	long	Short	Throw	while

In addition, identifiers containing a double underscore (_ _) are reserved for use by C++ implementations and standard libraries and should be avoided by users.

The ASCII representation of C++ programs uses the following characters as operators or for punctuation:

!	%	^	&	*	(	)	-	+	=	{	}	|	~
[	]	\	;	,	:	"	<	>	?	,	.	/	

And the following character combination are used as operators:

→	++	-	.*	*→	<<	>>	<=	>=	= =	!=	&&
||	*=	/=	%=	+=	-=	<<=	>>=	&=	^=	|=	::

Each is a single token.

In addition, the processor uses the following tokens:
#   # #

Type Conversion

Converting one pre-defined type into another typically will change size and/or interpretation properties of the type, but not the underlying bit pattern. The size may widen or narrow, and of course the interpretation will change. There are two ways of type conversion:

Implicit type conversion

Using assignment statement. For example,

```
long  lval = 3.14159;
      int i = lval;//i=3
```

However, such type of type conversion is restricted to pre-specified type conversions only. Such type conversion sometimes result in surprising results, for example,
int i = 2/3 will assign a zero to i.

Explicit type conversion

The notation used will be

```
type (expr)
(type)expr
```

These are referred to as typecast. For example,
Int(3.14159) result in 3

The if statement

The syntax of the if statement is as follows:
if (expression)
Statement;

The expression must be enclosed in parenthesis. The statement may be a compound statement. For example,

```
if (x > 5)
x=0;
```

The statement will be executed only if the expression is true. The syntax of the If-else statement is as follows:

```
if (expression)
        Statement-1;
else
        Statement-2;
```

If expression is true statment-1 is executed. If expression is false, statement -2 is executed. So, depending on the truth-value of expression, either statement -1 or statement - 2 is executed. For example,

```
if (x > 5)
        ++y:
else
        --z;
```

The statement 1 and 2 may be another if statement if the need be so. The if-else statement introduces a source of potential ambiguity referred to as the dangling-else. The problem occurs, when a statement contains more if-else clauses. The question is "with which **if** does the additional **else** clause properly match up? Consider,

```
if (row < 5)
        if (col < 10);
        cout << "valid";
else
        cout << "Invalid order";
```

The indentation indicates the programmer's belief that the 'else' is associated with outer 'if '. But it is wrong. To avoid this ambiguity, a rule has been made that an 'else' is associated with the last unmatched 'if '. So, the above statements will be executed in accordance with the following indentation:

```
if(row < 5)
        if (col < 10)
                cout <<"valid";
        else
                cout <<"Invalid";
```

The Switch Statement

The syntax of the switch statement is as follows:

```
switch(expression) {
        case constant-1
                statement;
        case constant-2
                statement;
                :
                :
                :
        case statement-n
```

```
            statement;
        default: statement;
        }// end of switch
```

The expression can be any valid expression except a floating-point value. The value of expression is compared with each constant for a match. The statement corresponding to matched case and the statements of the following cases are executed. If the value of expression is unmatched with all of the constant values, then the statement corresponding to default clause is executed. For example,

```
x = 5;
switch(x)
{
case 1 : cout<< "one"; break;
case 5 : cout<< "five"; break;
case 8 : cout<< "eight"; break;
default : cout<< "matching did not occur";
}
```

So, the output will be:

```
five
```

In the absence of the break statement, the output will be:

```
five    eight    matching did not occur
```

The while statement

The syntax of the while statement is of the following form:

```
while (expression)
       statement;
```

So, whenever expression becomes true, the statement will be executed.

The for statement

The syntactic form of the FOR loop is as follows:

```
for (initialisation; expression-1; expression-2)
        Statement(s);
```

The order of the evaluation is as follows:

```
initialization
expression-1
if expression is true {
        statement(s);
        expression-2;
}
else
        exit loop;
```

Both, initialization and expression-2, can be NULL statements.
Examples:

```
For (int k = 0; k < 5; ++k)
For (;value>2; ++count)
For (;colour = green;)
```

Expression -1 must always evaluate to either 1 or 0. Only when expression is true, the statement(s) is/are executed.

The do while statement

The syntactic form of the do while loop is as follows:

```
do

            Statement(s)
while (expression);
```

The order of evaluation is as follows:

Statements(s)

Expression: If the expression is true, then the statement(s) will be executed again.

So, the statement is executed before the expression is evaluated. The expression will always evaluate to True (1) or False (0). So, unlike other loop structures, the body is always executed at least once.

The break Statement

The break statement will transfer control to the first statement after the body of the latest for, while, do or switch in which it is present. For example,

```
while (1)
{
            cin >> height;
            if (height >10000)
                        break;
}
cout <<"Reduce Altitude";
```

The continue statement

The continue statement will terminate the current iteration of the while, for or do loop statement;

```
For ( i=0 ; i<st-strength; ++i)
{
if (marks [(i+1)]>=50
continue;
else
            // communicate to the student that
            //he has failed
}
```

The goto Statement

The goto statement is an unconditional jump statement. The syntax of the goto statement is as follows:

```
goto Label;
```

Both the goto and Label should appear in the same function. A colon should always follow the Label. For example,

```
void matri(int i)
{
if (i==1)
goto Multiply;
.
.
.
.
Multiply:
}
```

Constraints on goto Statement

There should be at least one statement followed by LABEL. For example,

```
{gotoX;
.
.
.
X: // a Null statement is used
}
```

Between the goto and Label statements, there should be no explicit or implicit initialised statements. The constraint is applicable only in the case of forward jumps. In the case of backward jumps, this rule is not applicable.

However, the rule regarding forward jumps is not applicable if the GOTO statement jumps on the entire block containing the initialised statement. Example:

```
goto Process
{
        i=0;
        j=k;
Process:
```

A good programming practice is to avoid goto statement in a program.

☞ Check Your Progress 2

1) What are the problems in the following program segment?

```
        main ( )
        {
        int i, j, k;
        float x;
        x=i;
        if (i>x) got mult;
        else
        j=i;
mult:
        k=i*j;
}
```

2) What will be the value of the following expressions of C++ for the values of int a = 5, b = 6, c = 7, d = 8

(a) a + b * c % d
(b) a++ * b++ * a++ * b++
(c) ++a * b--* a++* --b
(d) a & b
(e) c | d
(f) sizeof a

3) How many times will the following loops be executed for the value of int i = 7, j = 250, k = 55

(a)
```
int count = 0;
for (;i<k; i+=5)
{printf (“%d”, count);
count ++;
}
```
(b)
```
do {i++;
count ++;
} while (j>i);
```

1.4 FUNCTIONS AND FILES

In this Section, we will mainly discuss how to make our own library of functions which will be used frequently.

1.4.1 How to make a Library?

Let us assume that you want to have a function, which will receive parameters of type double, computes their sum and returns that sum which is of type double.

Also, let us have a function, which will receive an integer parameter and returns a 0 (false) if it is odd or a 1 if it is even. Let the names of above functions be sum and even. Let the files in which they are placed are sum.c and even.c.

Let us make a library with these two files and let the name of the library be lib.a. Now, applying the following sequence of steps in UNIX environments can make the library lib.a:

```
$cc - c sum.c even.c           // The result is the equivalent object files
$ar cr lib.a sum.o even.o      // An archive called lib.a is made which
                               // contains the object
                               // codes of two functions.

$ranlib lib.a    // Now; the library is indexed for faster access.
```

In the above code, $ is the system prompt.

Now, the library can be used along with any program say, bill.c in the following way:

```
$cc bill.c lib.a
```

The advantage of using a library is that only the functions which are needed will be used and the linker will look after it. If we do not archive them into a library, we have to specify the files individually, which will be cumbersome as well as error-prone.

1.4.2 Functions

In C++, any function has to be declared and defined before it is called.
The format of a function declaration is as follows:
Return-type functionname (Argument-type1, Argument-type2.......);

For example,

```
void swap(int*, int*);
```

The format of a function definition is as follows:

```
return-type         function-name(Argument-type1, Argument-type2.......)
{
//function body
}
```

For example,

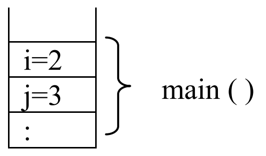
```
void swap(int *u, int *v)

//Program to swap the values of two integers u an v using third location temp.
{
        int temp;
        temp = *u;
        *u = *v;
        *v = temp;
        return;
}
```

Arguments to a function can be passed by value or by reference.

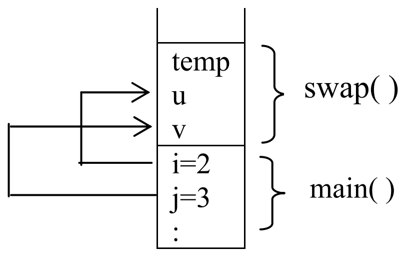
Let us demonstrate the call by value mechanism with a simple example as follows:

```
#include <iostream.h>
//swap function prototype
void swap(int, int);
void main ( )
{ int i = 2;
  int j = 3;
  swap(i, j);
  cout << i << j;
}
// A wrong swap function implemented here
void swap (int u, int v)
{       int temp = u;
        u = v;
        v = temp;
        return;
}
```

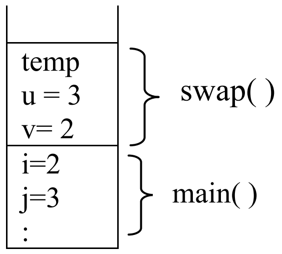
The output will be 2 (the value of i) and 3 (the value of j) that is no exchange has taken place in original values although the values of u & v must have changed. The reason is, in call by value, temporary storage will be allocated to arguments parameters.



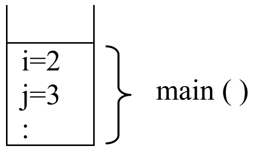
Start of main( ) before swap function execution



On call to swap function i is passed to u and j is passed to v



On complete execution of swap but before return



On return from the call the values in main ( ) remains unchanged.

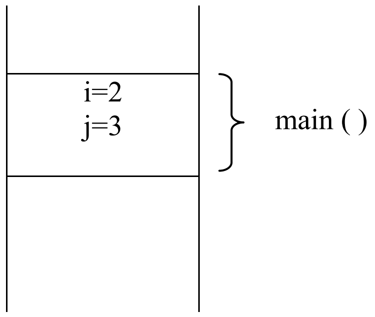
The value of i & j in main ( ) remain the same since the memory in which they reside is different from the storage locations where they are passed as arguments. This is the reason why the same values are reflected in the output.

Consider the following example for demonstration of call by reference.

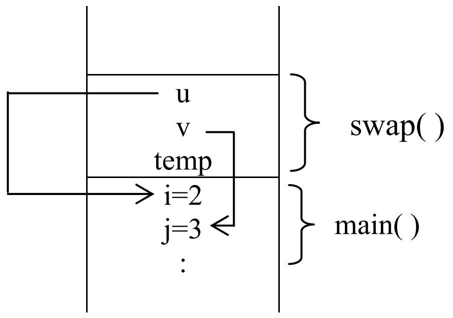
```
# include <iostream.h>

void swap (int&, int&);
// indicating call by reference
void main ( )
{       int i = 2;
        int j = 3;
        swap (i, j);
        cout < ' < "i="<<'j'<<j'
}

void swap (int& u, int& v)
{       int temp = u;
        u = v;
        v = temp;

}
```

The output will be i=3 j=2. That is the interchange in the main has occurred which was the desired purpose.

The reason is that during the compilation and execution of main ( ), the locations i and j are created and they were assigned the values 2 and 3 as follows :

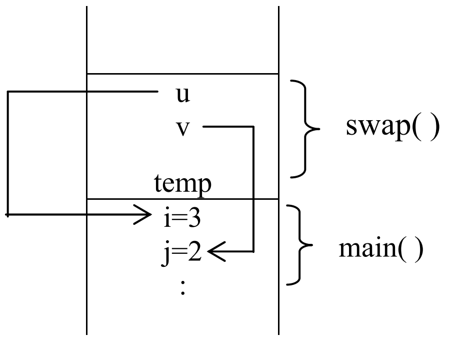


When there is a call to swap, the control will be transferred to procedure swap. Since the argument passing is by call by reference, the memory location u and v will refer to i and j. It can be illustrated as shown in the following *Figure*:



u is a pointer/alias to i and v is alias to j.

After the completion of execution of procedure swap, the control passes to the main( ) program and the location of i and j will contain 3 and 2 .



On interchange, u and v will change the values of i and j.

The locations in all the above situations are same. In this way, the output will be i=3, j=2 in call by reference.

We can declare an argument as const if we want that it should not be modified by the called function. It is also possible to change the type of the argument.
For example,

```
int check (const int & u)
{   if u > 0
          return 1;
   else return 0;}
```

At the same time, if there is need for any type conversion between a formal parameter and an actual parameter then the argument must be declared const to the reference argument. It cannot be converted if it is not declared const.

Arrays can be passed as arguments to a function in C++ as in C.

Overloading means using the same name for different operators on different types. For example, '+' is an overloaded operator in most of the languages since '+' is the sign used for addition of two integers as well as two floating point numbers. The same is the case with '-' operators.

Function names can also be overloaded in C++ as demonstrated by an example as follows:

```
void search (const char*);
void search (int*);
# include <iostream.h>
void main ( )
{         int j[5];
          char k[5];
          // code for reading elements
          search (j) ; //second function is used
          search (k) ; // first function is used
}
```

When a function name is overloaded, the overloaded functions should have different number of arguments or different types of arguments. If not, the compiler cannot take a decision regarding which function to use.

Type conversion will take place if the matching of arguments does not take place and in such cases, error messages will result. We can also have unspecified number of arguments by using va_arg, va_end, va_list, va_start of library <stdarag.h>. We can have pointers to a function as in C language.

1.4.3 Macros

A Macro can be defined as a segment of code with a name which replaces the occurrences of name in the code. For example:

```
#define Msoft Bill Gates
```

So, in the program, whenever the token "Msoft" is encountered, "Bill Gates" replaces it.

Macros can be used effectively for defining a symbolic name to a constant. It improves the readability of the program. However, the same can be done using constant declarations, for example:

Macros	**Constant Variable Definition**
Example # define PI 3.14159 # define MAXSIZE 5000	Example Const PI = 3.14159 Const MAXSIZE = 5000;
Advantage the symbol PI will be taken as value and no space will be reserved by compiler for them. But symbols cannot be identified to a type.	Symbols declared here can be identified to a type. Thus, helps in identifying compilation errors.

The capabilities of macros are far beyond just symbols. However, they should be used with caution.

- There will be problems with macros when there were recursive calls.
- Another problem is with precedence.

For example,

```
#define cube (a)      a * a * a
```

Now, let us assume that there is a statement in the program as follows:

```
ink k = cube ( k + 3);
```

Now, this statement will be expanded as follows:

int k = (k + 3 * k+ 3* k + 3)
which is certainly not $(k + 3)^3$ as per precedence of operator.

Due to these and many other reasons, C++ provides const, and template mechanisms so that the use of Macros can be minimised.

1.5 ADVANTAGES OF C + +

There are almost two dozen major object-oriented programming languages in use today. But the leading commercial OO languages are C++, Smalltalk and Java. We have discused about some of these in Unit 4 of Block 1. Let us discuss some of the reasons of success and advantages of C++ in this section. Java will be presented in CS-75 course.

Why C + + succeeded

C++ started as an extension to C language or more precisely we can say C++ started as turning C into an OOPL and it emerged out as Superset of C. But this is just the part of the reason for the success of C++. C++ has solved many other problems faced by C programmers in today's development scenarios. C++ has especially come as a major tool to the person who has made large investments in C.

The second reason is the main reason for the success of C++, which in a nutshell, is economics: It still costs to move to OOP, but C++ may cost less.

C++ is aimed at enhancing productivity. The productivity enhancement is due to design, which helps you as much as possible and do not hinder you with any arbitrary rules and requirements. It is designed to follow a practical approach aimed at benefiting the programmer.

Advantages of C++

The basic advantages of C++ can be summed up as under:

- C++ has closed many holes in the C language and provides better type checking and compile-time analysis.

- You are forced to declare functions so that the compiler can check their use. The need for the preprocessor has virtually been eliminated for value substitution and macros, which removes a set of difficult-to-find bugs.

- C++ has a feature called references that allows more convenient handling of addresses for function arguments and returned values.

- The handling of names is improved through a feature called function overloading, which allows you to use the same name for different functions. A feature called namespaces also improves the control of names. There are numerous smaller features that improve the safety of C.

- **The learning curve for C programmers is very fast:** Most of the companies already have C programmers. They do not want that their programmer becomes ineffective in a day. C++ is an extension of C, thus, reduces the learning time. In addition, C++ compiler accepts C code.

- **Efficiency:** C++ allows greater control of program performance and also allows programmers to interact with assembly code as the case with C language. Thus, C++ is quite a performance-oriented language. It sacrifices some flexibility in order to remain efficient, however, C++ uses compile-time binding, which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This makes for high run-time efficiency and small code size, but it trades off some of the power to reuse classes.

- **Systems are easier to express and understand:** Since, C++ supports object-oriented paradigm, thus, demonstrates the compatibility of good solution expression as it deals with higher-level concept like objects and classes rather than functions and data. It also produces maintainable code. The programs that are easier to understand are easier to maintain.

- **Good Library Support:** One of the fastest ways to create a program is to use already written code from the library. C++ libraries are easy to use and can be used in creating new classes, C++ guarantees proper initialization, clean up and call to library functions/classes, you can use the libraries by just knowing the message interfaces.

- **Source Code Reuse using templates:** Template feature reuses same source code with automatic modification for different classes. It is a very powerful tool that allows reuse of library code. Templates hide complexity of the code reuse for different classes on the user.

- **Error Handling:** C++ supports error-handling capabilities that catches the errors and reports them too. This feature provides control of error handling to the programmers in a similar way as being done for the libraries.

- **Programming in Large:** Many programming languages have their own limitation; some have limitations on line of code, some on recursion, etc., however, C++ provides many features that supports the programming.
  Some of these features are:
  - Templates namespaces and exception handling,
  - Strongly typed easy to use compiler,
  - Small or large programs are allowed,
  - Objects help in reducing complex problem to manageable one.

☞ Check Your Progress 3

1) What is overloading of functions? How does compiler resolve which of the overloaded function has been called?

   ……………………………………………………………………………

   ……………………………………………………………………………

   ……………………………………………………………………………

2) Implement SWAP functions using call by value and call by reference and print intermediate result to check the justification given in the section above.

....

………………………………………………………………………...

....

3) Write a macro to define a max function for two variable.

....

………………………………………………………………………...

....

1.6 SUMMARY

In this Unit, we have introduced various programming paradigms. Object-oriented programming has become popular since the paradigm perfectly fits the real life situations.

We have also discussed various control statements namely – IF THEN, IF THEN ELSE, SWITCH, WHILE, DO, FOR and the circumstances when we have to prefer a particular statement through which one can be expressed in the form of other.

We are also able to know about the declaration and use of functions. A function can have more than one return statement.

We have seen the method of defining macros and a few disadvantages of them. We will explore the "Inline functions" in the remaining sections of the Block.

In the next Unit, we will also describe the Declaration and usages of classes, concept of overloading, inheritance and also how C++ supports Object Oriented Programming.

1.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Procedural Programming, Modular Programming, Object-Oriented Programming.

2) (a) & (c)

3) Part

Check Your Progress 2

1)
   - no return type to main ( )
   - an assignment and comparison of dissimilar type not necessarily will give syntax error as C++ as it is not very strong typed language.
   - use of goto is not recommended.

2) Write and run a small C++ program and compare your theoretical results with those obtained from program.

3) Write and run a small C++ program and compare your theoretical results with those obtained from program.

Check Your Progress 3

1) More than one function with the same name, but different number of or type of arguments. The different number of arguments or type of arguments is used by compiler for resolving which function has been actually called.

2) Do it through a program and check result with different set of datas.

3) # define max (a,b) (a>b)? a:b

1.8 FURTHER READINGS

1) B. Stroustrup, *The C++ Programming Language*, third edition, Pearson/Addison-Wesley Publication.

2) N. Barkakati, *Object Oriented Programming in C++,* Prentice-Hall of India.