UNIT-I
OBJECT ORIENTED THINKING

DIFFERENT PARADIGMS FOR PROBLEM SOLVING

Paradigm definition

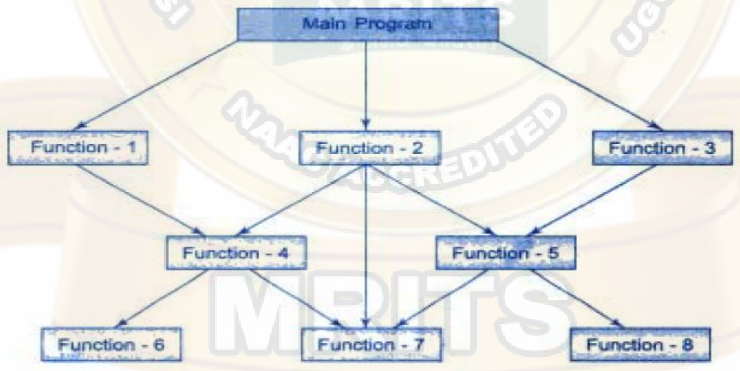
A paradigm is a way in which a computer language looks at the problem to be solved.

Evolution of Paradigms

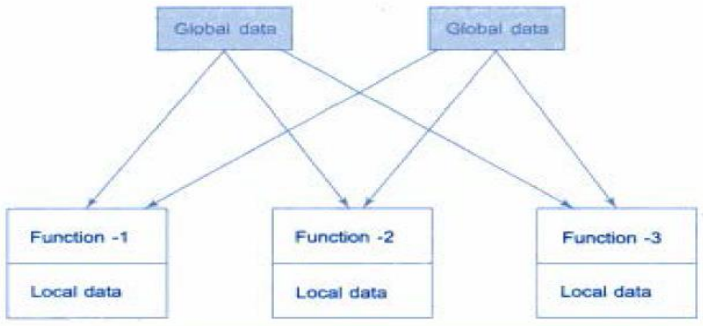
Since the invention of computers, many programming approaches have been developed. The primary motivation of doing so is to handle the increasing complexity of programs and to make them reliable and maintainable.

The following are the different paradigms for problem solving,

1. **Monolithic programming**
   - This is the main technique used in 1980's.
   - The program is written with a single function. A program is not divided into parts i.e. statements are written in sequence.
   - When the program size increases it failed to show the desired result in terms of bug free, easy to maintain and reusable programs.
   - The concept of sub programs does not exist, and hence is useful for small programs.
2. **Procedure Oriented Programming**
   - It basically consists of writing a list of instructions for the computer to follow and break down the code and organize these instructions into manageable segments or groups known as functions.
   - In this, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks.



   - In a multi function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its local data.



Drawbacks

- Global data are more vulnerable to an inadvertent change by a function.

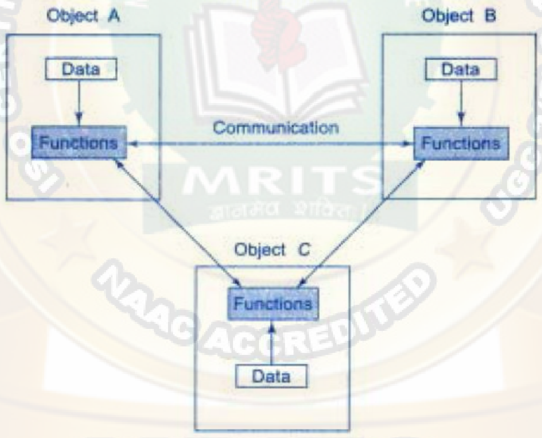
- In a large program it is very difficult to identify what data is used by which function. This provides an opportunity for bugs to creep in.
- It does not model real world problems very well. This is because functions are action oriented and do not really corresponding to the elements of the problem.

Characteristics

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs *top·down* approach in program design.

3. **Object Oriented Programming**

- This is the most recent concept among programming paradigms.
- It is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as template for creating copies of such modules on demand.
- The major motivating factor in the invention of object oriented approach is *to* remove some of the flaws encountered in the procedural approach.



Features

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- *Follows bottom-up* approach in program design.

NEED FOR OBJECT-ORIENTED PARADIGM

- The object oriented programming paradigm is a methodology for producing re usable software components.
- It promotes efficient design and development of software systems using reusable components that can be quickly and safely assembled into larger systems.
- It produces reusable code/objects because of encapsulation and inheritance.

- The data is protected because it can be altered only by the encapsulated methods.
- It is more efficient to write programs which use pre-defined objects.
- The storage structure and/or procedures within an object type could be altered if required without affecting programs that make use of that object type.
- New functions can easily be added to objects by using inheritance
- The code produced is likely to contain fewer errors because pretested objects are being used.
- Less maintenance effort will be required by the developer because objects can be reused.

DIFFERENCE BETWEEN OBJECT ORIENTED PROGRAMMING AND PROCEDURE ORIENTED PROGRAMMING

S.NO	Object oriented Programming	Procedure Oriented Programming
1	Emphasis is on data	Emphasis is on doing things
2	Programs are divided into Objects	Programs are divided into Functions
3	Employs Botton up approach	Employs Top down approach
4	Modification potential is high	Modification potential is low
5	Data is hidden and cannot be accessed by external functions	Data is open and can be accessed by any functions
6	Suitable for solving big problems	Not Suitable for solving big problems
7	It needs more memory than POP	It needs less memory
8	Supports Polymorphism, Inheritance, abstraction and Encapsulation	Does not supports Polymorphism, Inheritance, abstraction and Encapsulation
9	Example languages are C++, Java	Example languages are C,VB,FORTRAN, COBOL

OVERVIEW OF OOP CONCEPTS

1. **Abstraction**
   - *Abstraction* refers to the act of representing essential features without including the background details or explanations.
   - Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost and *functions* to operate on these attributes.
   - They encapsulate all the essential properties of the objects that are to be created.
   - The attributes are sometimes called *data numbers* because they hold information.
   - The functions that operate on these data are sometimes called *methods or member functions.*
   - Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

2. **Encapsulation**

- *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created.
- When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation.
- Within an object, code, data, or both may be *private* to that object or *public*.
- Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object.
- When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

3. **Polymorphism**
   - Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods."
   - In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.
   - For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push( )** and **pop( )**, that can be used for all three stacks.
   - Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the *general interface*.

4. **Inheritance**
   - *Inheritance* is the process by which one object can acquire the properties of another object.
   - This is important because it supports the concept of *classification*. If you think about it, most knowledge is made manageable by hierarchical classifications.
   - For example, a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

C++ BASICS

Origin of C++

C++ began as an expanded version of C. The C++ extension was invented by **Bjarne Stroupstrup in 1979** at **Bell laboratories** He initially called the new language as "C **with Classes**" but renamed it as "**C++**" in 1983.

STRUCTURE OF A C++ PROGRAM

Most C++ programs has the following general form,

```
#include
base class declarations
derived class declarations
non-member function prototypes
int main()
{

        //.....
}
non member function definition
```

Sample C++ Program

```
#include<iostream.h>
int main()
{
        int i;
        cout<< “enter a number”:
        cin>>i;
        cout<<i<<” squared is“<<i*i<<endl;
        return 0;
}
```

Output:

```
enter a number 10
10 squared is 100
```

Header File

Header <iostream> is included. This header supports C++ style of I/O operations.

Input Operator

cin>>i;

This is an input statement and causes the program to wait for the user to type in a number.

Operator >> is known as **extraction or getfrom** operator. It takes the value from the keyboard and assigns it to the variable on its right. Similar to scanf() inn C.

Output Operator

cout<< “enter a number”:

This causes the string in quotation marks to be displayed on the screen.

Operator << is called **insertion or putto** operator. It inserts (or sends) the contents of the variable on its right to the object on its left. Similar to print() in C.

DATA TYPES

There are 7 different data types in C++.They are

1. Character(char)
2. Integer(int)
3. Floating-point(float)
4. Double floating-point(double)
5. Valueless(void)
6. Boolean(bool)

7. Wide Character(Wchar_t)

6. bool Data Type

C++ defines a built-in Boolean type called **bool**. Objects of type **bool** can store only the values **true** or **false**, which are keywords defined by C++. Automatic conversions take place which allow **bool** values to be converted to integers, and vice versa. Specifically, any non-zero value is converted to **true** and zero is converted to **false**. The reverse also occurs; **true** is converted to 1 and **false** is converted to zero.

General form: bool b1=true;

7.Wide Characters

C++ define wide characters which are 16 bits long. To specify a wide character precede the character with an **L**.

General form: wchar_t wc;
wc = L'A';

Here, **wc** is assigned the wide-character constant equivalent of A. The type of wide characters is **wchar_t**. In C, this type is defined in a header file and is not a built-in type. In C++, **wchar_t** is built in.

Program 1: Write a C++ program to demonstrate different data types available in C++

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

int main ()
{
  bool b = true;
  wchar_t w = L'A';
  int i;
  char ch;
  float fl;
  double d1;

  cout << "Enter a character: ";
  cin >> ch;
  cout << "\nYou entered: " << ch;

  cout << "\n\nEnter a floating-point number: ";
  cin >> fl;
  cout << "\nYou entered: " << fl;

  cout << "\n Enter a integer";
  cin >> i;
  cout << "\n You entered :" << i;

  cout << "\n Enter a double";
  cin >> d1;
  cout << "\n You entered :" << d1;

  cout << "\n boolean value is :" << b;
  cout << "\n Wide character value::" << w << '\n';

  return 0;
}
```

Output:

```
Enter a character: a

You entered: a

Enter a floating-
point number: 3345

You entered: 3345
 Enter a integer56

 You entered :56
 Enter a double57

 You entered :57
 boolean value is :1
 Wide character value::65
```

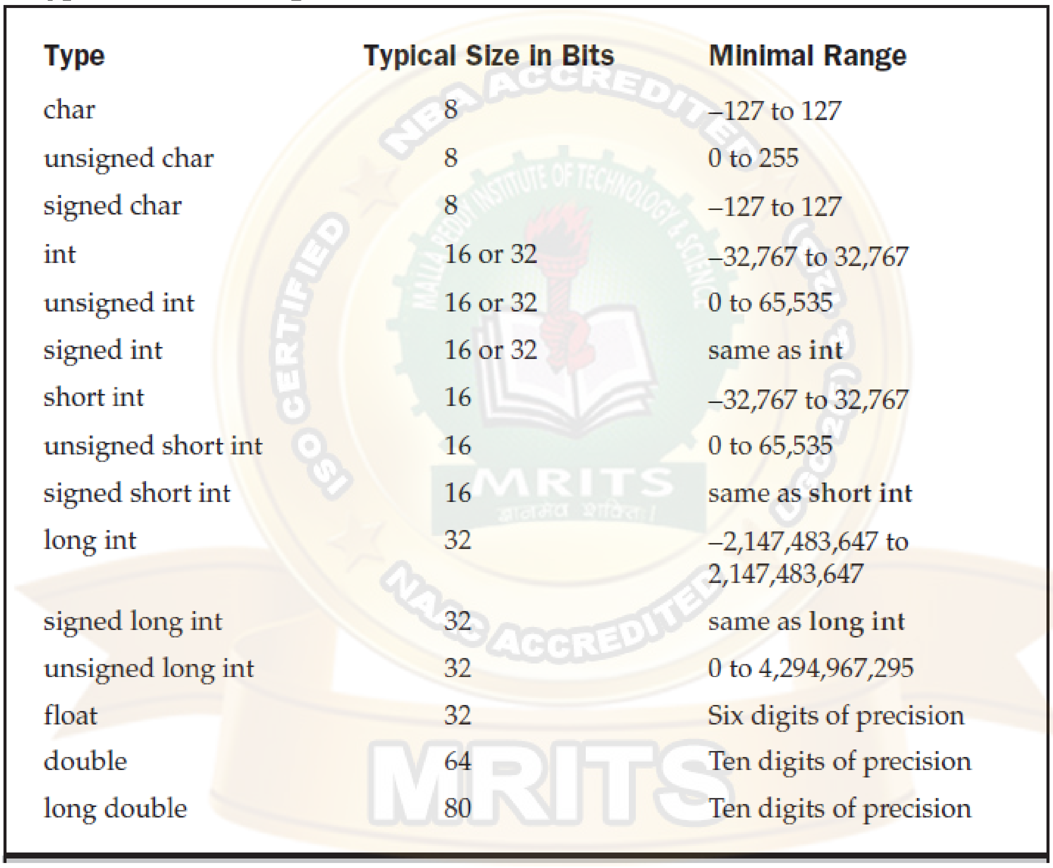
Program 2: Write a C++ program to know sizes of different data types available in C++

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

int main()
{
        cout << "Size of char : " << sizeof(char) << " byte" << endl;

         cout << "Size of int : " << sizeof(int) << " bytes" << endl;

        cout << "Size of short int : " << sizeof(short int)  << " bytes" << endl;

        cout << "Size of long int : " << sizeof(long int)   << " bytes" << endl;

        cout << "Size of signed long int : " << sizeof(signed long int)  << " bytes" << endl;

        cout << "Size of unsigned long int : "<< sizeof(unsigned long int)<< " bytes" << endl;

        cout << "Size of float : " << sizeof(float)   << " bytes" <<endl;

        cout << "Size of double : " << sizeof(double)   << " bytes" << endl;

        cout << "Size of wchar_t : " << sizeof(wchar_t)  << " bytes" <<endl;

        return 0;
}
```

Output:

```
Size of char : 1 byte
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar_t : 4 bytes
```

Data types size and Range



Type	Typical Size in Bits	Minimal Range
char	8	–127 to 127
unsigned char	8	0 to 255
signed char	8	–127 to 127
int	16 or 32	–32,767 to 32,767
unsigned int	16 or 32	0 to 65,535
signed int	16 or 32	same as int
short int	16	–32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	same as short int
long int	32	–2,147,483,647 to 2,147,483,647
signed long int	32	same as long int
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	80	Ten digits of precision

Modifying the Basic Types

Except for type **void**, the basic data types may have various modifiers preceding them. You use a *modifier* to alter the meaning of the base type to fit various situations more precisely.

You can apply the modifiers **signed**, **short**, **long**, and **unsigned** to integer base types. You can apply **unsigned** and **signed** to characters. You may also apply **long** to **double**.

The list of modifiers is shown here:

1. signed
2. unsigned
3. long
4. short

VARIABLES

A *variable* is a named location in memory that is used to hold a value that may be modified by the program.

Declaration of a variable

All variables must be declared before they can be used.

General form: *type variable_list;*

Here, *type* must be a valid data type plus any modifiers, and *variable_list* may consist of one or more identifier names separated by commas.

Examples:

```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Initialization of a variable

We can assign a value to a variable.

General form: *variable= expression;*

Example: i=10;

We can initialize a variable at the time of declaration.

General form: type variable= *expression;*

Example: int i=10;

Where Variables Are Declared

Variables will be declared in three basic places:

1. Inside functions (local variables)
2. In the definition of function parameters(formal parameters)
3. And outside of all functions (global variables)

1. Local Variables

Variables that are declared inside a function are called *local variables*. Local variables may be referenced only by statements that are inside the block in which the variables are declared. In other words, local variables are not known outside their own code block.

Local variables exist only while the block of code in which they are declared is executing. That is, a local variable is created upon entry into its block and destroyed upon exit. The most common code block in which local variables are declared is the function.

For example, consider the following two functions:

```
void func1(void)
{
int x;
x = 10;
}


void func2(void)
{
int x;
x = -199;
}
```

The integer variable **x** is declared twice, once in **func1( )** and once in **func2( )**. The **x** in **func1( )** has no bearing on or relationship to the **x** in **func2( )**. This is because each **x** is known only to the code within the block in which it is declared.

Program 1: Write a C++ program to demonstrate Local Variables

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

int main()
{
float f;
double d;

cout << "Enter two floating point numbers: ";
cin >> f >> d;

cout << "Enter a string: ";
char str[80]; // str declared here, just before 1st use
cin >> str;

cout <<"printing received values"<<endl<< f << " " << d << " " << str;
return 0;
}
```

Output:

```
Enter two floating point numbers: 9.9

17.21
Enter a string: keerthi
printing received values
9.9 17.21 keerthi
```

Important difference between C and C++

An important difference between C and C++ is when local variables can be declared. In C89, you must declare all local variables used within a block at the start of that block. You cannot declare a variable in a block after an "action" statement has occurred. For example, in C89, this fragment is incorrect:

```
/* Incorrect in C89. OK in C++. */
int f()
{
int i;
i = 10;
int j; /* won't compile as a C program */
j = i*2;
return j;
}
```

In a C89 program, this function is in error because the assignment intervenes between the declaration of **i** and that of **j**. However, when compiling it as a C++ program, this fragment is perfectly acceptable. In C++ (and C99) you may declare local variables at any point within a block—not just at the beginning.

2. Formal Parameters

If a function is to use arguments, it must declare variables that will accept the values of the arguments. These variables are called the *formal parameters* of the function. They behave like any other local variables inside the function. As shown in the following program fragment, their declarations occur after the function name and inside parentheses:

```
/* Return 1 if c is part of string s; 0 otherwise */
int is_in(char *s, char c)
{
while(*s)
if(*s==c) return 1;
else s++;
return 0;
}
```

The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character specified in **c** is contained within the string **s**; 0 if it is not.

3. Global Variables

Unlike local variables, *global variables* are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program's execution. You create global variables by declaring them outside of any function. Any expression may access them, regardless of what block of code that expression is in.

Program 2: Write a C++ program to demonstrate Global Variables

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

int count; /* count is global */
void func1(void);
void func2(void);
int main(void)
{
count = 100;
func1();
return 0;
}
void func1(void)
{
int temp;
temp = count;
cout<<"count is (from func1)"<< count; /* will print 100 */
func2();
}

void func2(void)
{
```

```
int temp;
temp = count;
cout<<"count is (from func2)"<< count; /* will print 100 */
}
```

Output:

```
count is (from func1)100count is (from func2)100
```

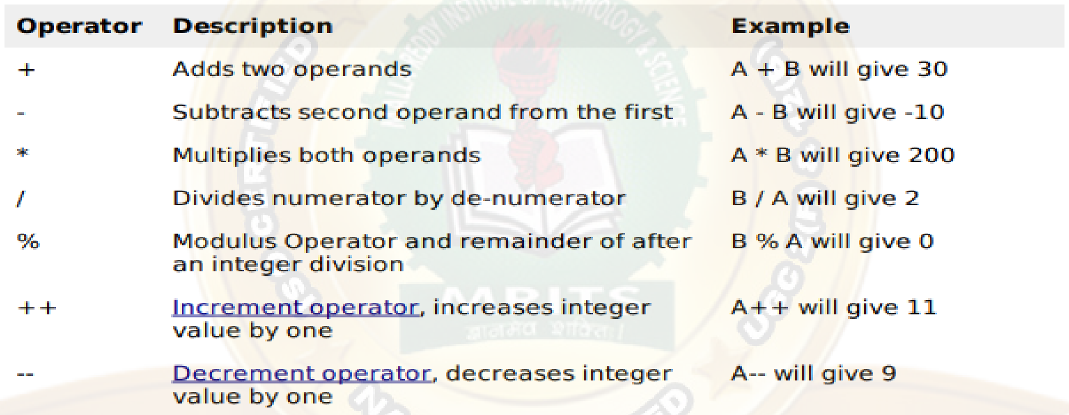
OPERATORS

C++ is rich in built-in operators. There are four main classes of operators: *arithmetic*, *relational*, *logical*, and *bitwise*. In addition, there are some special operators for particular tasks.

1. Arithmetic Operators

These are defined to perform basic arithmetic operations. The operators **+**, **−**, *****, and **/** work as they do in most other computer languages. You can apply them to almost any built-in data type.

Assume variable A holds 10 and variable B holds 20



Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Both the increment and decrement operators may either precede (prefix) or follow (postfix) the operand. For example,

```
        x = x+1;
can be written
        ++x;
or
        x++;
```

There is, however, a difference between the prefix and postfix forms when you use these operators in an expression. When an increment or decrement operator precedes its operand, the increment or decrement operation is performed before obtaining the value of the operand for use in the expression. If the operator follows its operand, the value of the operand is obtained before incrementing or decrementing it. For instance,

```
x = 10;
y = ++x;
```

sets **y** to 11. However, if you write the code as

```
x = 10;
y = x++;
```

y is set to 10. Either way, **x** is set to 11; the difference is in when it happens.

Precedence of the Arithmetic Operators

highest	++ – –
	– (unary minus)
	* / %
lowest	+ –

PROGRAM 3: ARITHEMATIC OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        int num1, num2, res;

        cout<<"Enter any two number: ";
        cin>>num1>>num2;

        res = num1 + num2;
        cout<<"\n";
        cout<<num1<<" + "<<num2<<" = "<<res<<endl;

        res = num1 - num2;
        cout<<num1<<" - "<<num2<<" = "<<res<<endl;

        res = num1 * num2;
        cout<<num1<<" * "<<num2<<" = "<<res<<endl;

        res = num1 / num2;
        cout<<num1<<" / "<<num2<<" = "<<res<<endl;

        res = num1 % num2;
        cout<<num1<<" % "<<num2<<" = "<<res<<endl;

        getch();
}
```

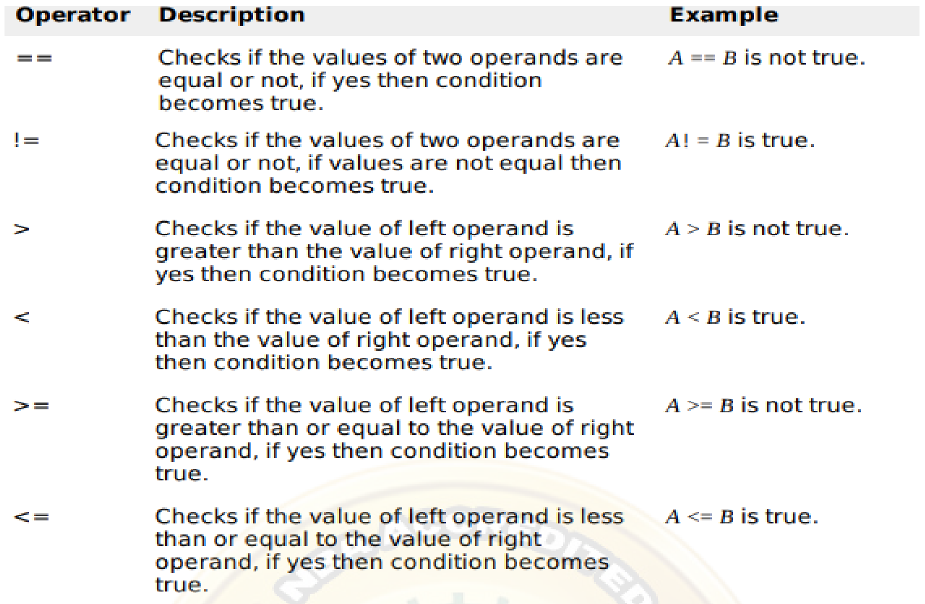
OUTPUT:

```
Enter any two number: 2
3

2 + 3 = 5
2 - 3 = -1
2 * 3 = 6
2 / 3 = 0
2 % 3 = 2
```

2. Relational Operators

Relational operators refer to the relationships that values can have with one another. The result of relational operators is either true or false.

Assume variable A holds 10 and variable B holds 20



Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	$A == B$ is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	$A != B$ is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	$A > B$ is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	$A < B$ is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	$A >= B$ is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	$A <= B$ is true.

PROGRAM 4: RELATIONAL OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        int p, q;
        int res;
        cout<<"Enter any two number: ";
        cin>>p>>q;

        cout<<"\n";
        cout<<"p    q    p<q    p<=q    p==q    p>q    p>=q    p!=q\n\n";

        res = p<q;
        cout<<p<<"    "<<q<<"    "<<res<<"      ";
        res = p<=q;
        cout<<res<<"      ";
        res = p==q;
        cout<<res<<"      ";
        res = p>q;
        cout<<res<<"     ";
        res = p>=q;
        cout<<res<<"      ";
        res = p!=q;
        cout<<res<<endl;

        getch();
}
```

OUTPUT:

```
Enter any two number: 3 4
p    q    p<q    p<=q    p==q    p>q    p>=q    p!=q
3    4    1      1       0       0      0       1
```

3. Logical Operators

Logical refers to the ways these relationships can be connected or combined. The result of logical operators is either true or false.

Assume variable A holds 1 and variable B holds 0

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	A && B is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	A||B is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!A && B is true.

The truth table for the logical operators is

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Precedence of the Relational and Logical operators:

Highest	!
	> >= < <=
	== !=
	&&
Lowest	||

PROGRAM 5: LOGICAL OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        int res;

        res = (6 <= 6) || (5 <3);
        cout<<res<<endl;

        res = (6 <= 6) && (5 < 3);
        cout<<res<<endl;

        res = !(6 <= 6);
        cout<<res<<endl;

        res = !(5 > 9);
        cout<<res<<endl;

        getch();
}
```

OUTPUT:

```
1
0
0
1
```

4. Bitwise Operators

C++ supports a full complement of bitwise operators. *Bitwise operation* refers to testing, setting, or shifting the actual bits in a byte or word, which correspond to the **char** and **int** data types and variants. You cannot use bitwise operations on **float**, **double**, **long double**, **void**, **bool**, or other, more complex types.

Bitwise operations most often find application in device drivers—such as modem programs, disk file routines, and printer routines — because the bitwise operations can be used to mask off certain bits, such as parity.

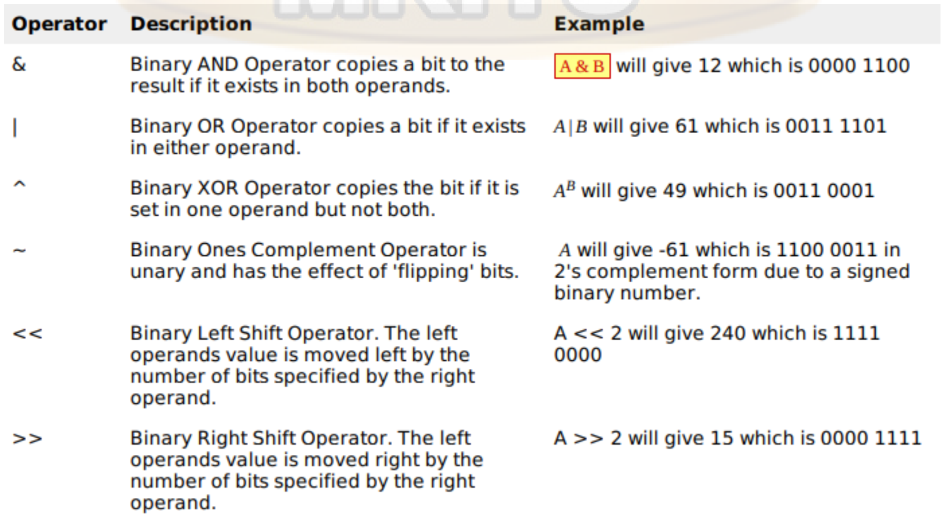
Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

You can combine several operations together into one expression, as shown here:
10>5 && !(10<9) || 3<=4

Assume variable A holds 60 and variable B holds 13



Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	A & B will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	A|B will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	A^B will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	A will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

PROGRAM 6: BITWISE OPERATORS

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        unsigned int a = 60;      // 60 = 0011 1100
        unsigned int b = 13;      // 13 = 0000 1101
        int c = 0;

        c = a & b;            // 12 = 0000 1100
        cout<<"a = 0011 1100 (60)\tand\tb = 0000 1101 (13)\n\n";
        cout<<"a & b = "<<c<<endl;

        c = a | b;            // 61 = 0011 1101
        cout<<"a | b = "<<c<<endl;

        c = a ^ b;            // 49 = 0011 0001
        cout<<"a ^ b = "<<c<<endl;

        c = ~a;               // -61 = 1100 0011
        cout<<"~a = "<<c<<endl;

        getch();
}
```

OUTPUT:

```
a = 0011 1100 (60)      and     b = 0000 1101 (13)

a & b = 12
a | b = 61
a ^ b = 49
~a = -61
```

Special operators

a. The ? Operator

C++ contains a very powerful and convenient operator that replaces certain statements of the if-then-else form. The ternary operator **?** takes the general form

Exp1 ? Exp2 : Exp3;

where *Exp1*, *Exp2*, and *Exp3* are expressions.

The **?** operator works like this: *Exp1* is evaluated. If it is true, *Exp2* is evaluated and becomes the value of the expression. If *Exp1* is false, *Exp3* is evaluated and its value becomes the value of the expression.

For example,

```
        x = 10;
        y = x>9 ? 100 : 200;
```

y is assigned the value 100. If **x** had been less than 9, **y** would have received the value 200.

b. &(the address of) Pointer Operators

It is a unary operator that returns the memory address of its operand.

Example: m = &count;

places into **m** the memory address of the variable **count**.

c. *(at address) Pointer Operators

It is a unary operator that returns the value of the variable located at the address that follows it.

For example, if **m** contains the memory address of the variable **count**,

q = *m;

places the value of **count** into **q**. Now **q** has the value 100 because 100 is stored at location 2000, the memory address that was stored in **m**.

d. sizeof

sizeof is a unary compile-time operator that returns the length, in bytes, of the variable or parenthesized type-specifier that it precedes.

Example: sizeof(int) will display 4

e. The Comma Operator

The comma operator strings together several expressions. The left side of the comma operator is always evaluated as **void**. This means that the expression on the right side becomes the value of the total comma-separated expression.

For example,

x = (y=3, y+1);

first assigns **y** the value 3 and then assigns **x** the value 4.

f. The Dot (.) and Arrow (>) Operators

The . (dot) and the >(arrow) operators access individual elements of structures and unions. In C++, the dot and arrow operators are also used to access the members of a class.

The dot operator is used when working with a structure or union directly. The arrow operator is used when a pointer to a structure or union is used.

```
struct employee
{
char name[80];
int age;
float wage;
} emp;
struct employee *p = &emp; /* address of emp into p */
```

you would write the following code to assign the value 123.23 to the **wage** member of structure variable **emp**:

```
emp.wage = 123.23;
```

However, the same assignment using a pointer to **emp** would be

```
p->wage = 123.23;
```

g. The [ ] and ( ) Operators

Parentheses are operators that increase the precedence of the operations inside them.

Square brackets perform array indexing

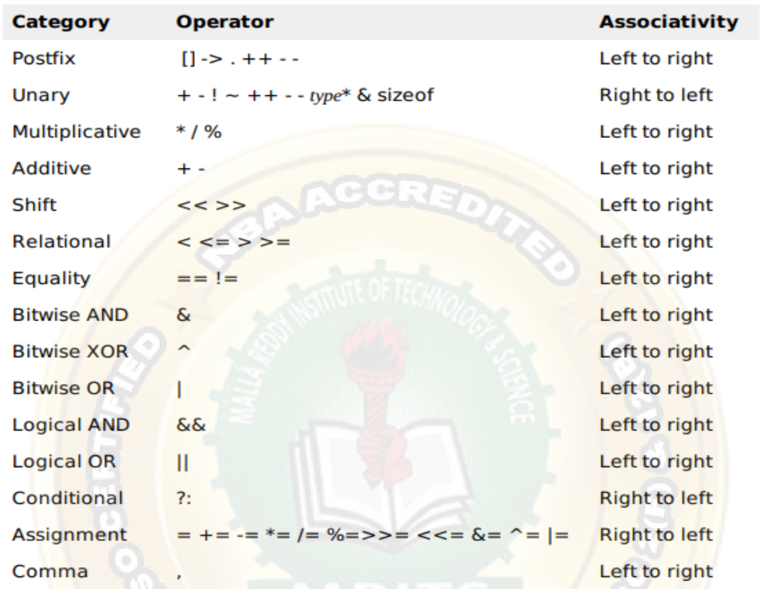
```
s[3] = 'X';
```

h. The Assignment Operator

You can use the assignment operator within any valid expression. C++ uses a single equal sign to indicate assignment

General form : *variable_name = expression;*

Operators Precedence in C++



Category	Operator	Associativity
Postfix	[] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - *type** & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

EXPRESSIONS

An *expression* in C++ is any valid combination of Operators, constants, and variables. Expressions 1nay be of the following seven types:

- Constant expressions
- Integral expressions
- Float expressions
- Point.er expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

Constant Expressions

Constant Expressions consist of only constant va1ues.

Examples: 15
20 + 5 / 2.0
'x'

Integral Expressions

Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.

Examples: m
m * n -5
m • 'x'

5 + int(2.0)
where m and n are integer variables.

Float Expressions
Float Expressions are those which, after all conversions, produce floating-point results.
Examples: x + y
x * y / 10
5 * float(10)
10.75
where x and y are floating-point variables.

Pointer Expressions
Pointer Expressions produce address values.
Examples: &m
ptr
ptr + l
"xyz"
where m is a variable and ptr is a pointer.

Relational Expressions
Relational Expressions yield results of type bool which takes a value true or false.
Examples: x<=y
a+b == c+d
m+-n > 100
When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as ***Boolean expressions.***

Logical Expressions
Logical Expressions combine two or more relationa1 expressions and produces bool type results.
Examples: a>b && x0010
x==10 || y==5

Bitwise Expressions
Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.
Examples: x << 3 *// Shift three bit position to left*
y >>1 // *Shift* one *bit position to right*
Shift operators are often used for multiplication and division by powers of two.
ANSI C++ has introduced what ore termed as *operator keyword* that can be used as alternative representation for operator symbols.

Special Assignment Expressions
Chained Assignment
x=(y=10);
or
x=y=10;
First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a =b =12.34;
```

is illegal. This may be written as

```
float a=12.34,b=12.34
```

Embedded Assignment

```
x= (y= 50) + 10 ;
```

(y = 50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50+ 10 = 60 is assigned to x. This statement is identical to

```
y = 50;
x =y + 10;
```

Compound Assignment

Like C, C+-+ supports a *compound assignment operator* which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x= x + 10;
```

may be written as

```
x+= 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
Variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that
variable1 = variable op variable2;

ORDER OF EVALUATION OF EXPRESSIONS

C++ does not specify the order in which the sub expressions of an expression are evaluated. This leaves the compiler free to rearrange an expression to produce more optimal code. However, it also means that your code should never rely upon the order in which sub expressions are evaluated. For example, the expression

```
x = f1() + f2();
```

does not ensure that f1( ) will be called before f2( ).

TYPE CONVERSION IN EXPRESSIONS

When constants and variables of different types are mixed in an expression, they are all converted to the same type. The compiler converts all operands up to the type of the largest operand, which is called ***type promotion***.

First, all char and short int values are automatically elevated to int. (This process is called *integral promotion*.) Once this step has been completed, all other conversions are done operation by operation, as described in the following type conversion algorithm:

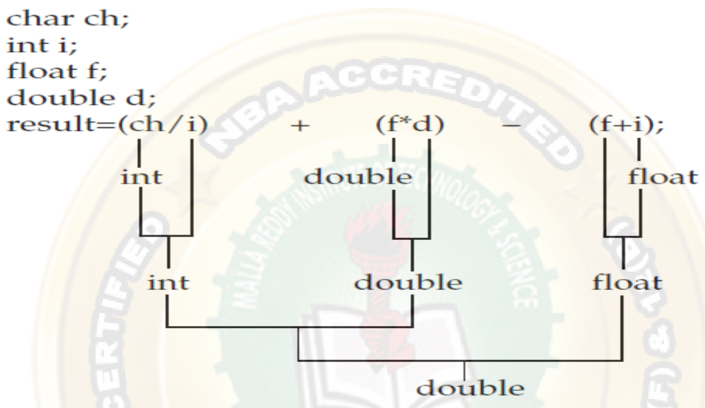
```
IF an operand is a long double
THEN the second is converted to long double
ELSE IF an operand is a double
THEN the second is converted to double
ELSE IF an operand is a float
THEN the second is converted to float
ELSE IF an operand is an unsigned long
```

THEN the second is converted to unsigned long
ELSE IF an operand is long
THEN the second is converted to long
ELSE IF an operand is unsigned int
THEN the second is converted to unsigned int

Additional special case: If one operand is long and the other is unsigned int, and if the value of the unsigned int cannot be represented by a long, both operands are converted to unsigned long.

Example:

First, the character ch is converted to an integer. Then the outcome of ch/i is converted to a double because f*d is double. The outcome of f+i is float, because f is a float. The final result is double.



```
char ch;
int i;
float f;
double d;
result=(ch/i)   +   (f*d)   -   (f+i);
```

Casts

You can force an expression to be of a specific type by using a *cast*.

General form : *(type) expression*

where *type* is a valid data type.

Example:

To make sure that the expression x/2 evaluates to type float, write (float) x/2

PROGRAM: TYPE CONVERSION

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        float res;
        float f1=15.5, f2=2;

        res = (int)f1/(int)f2;
        cout<<res<<endl;

        res = (int)(f1/f2);
        cout<<res<<endl;

        res = f1/f2;
        cout<<res;
        getch();
}
```

OUTPUT:

```
7
7
7.75
```

FLOW CONTROL STATEMENTS

A) SELECTION STATEMENTS

C++ supports two types of selection statements:

a) if
b) switch.

if

The general form of the **if** statement is

```
if (expression)
        statement;
else
        statement;
```

where a *statement* may consist of a single statement, a block of statements, or nothing (in the case of empty statements). The **else** clause is optional.

If *expression* evaluates to true (anything other than 0), the statement or block that forms the target of **if** is executed; otherwise, the statement or block that is the target of **else** will be executed, if it exists. Remember, only the code associated with **if** or the code associated with **else** executes, never both.

The conditional statement controlling **if** must produce a scalar result. A *scalar* is an integer, character, pointer, or floating-point type. In C++, it may also be of type **bool**.

PROGRAM: IF

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

int main ()
{
  int magic;                    /* magic number */
  int guess;                    /* user's guess */
  magic = rand ();              /* generate the magic number */
  cout<<"Guess the magic number: "<<endl;
  cin>>guess;
  if (guess == magic)
    cout<<"** Right **";
  return 0;
}
```

OUTPUT:

```
Guess the magic number:
777
```

PROGRAM: IF ELSE IF

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

int main ()
{
  int magic;                          /* magic number */
  int guess;                          /* user's guess */
  magic = rand ();                    /* generate the magic number */
  cout << "Guess the magic number: " << endl;
  cin >> guess;
  if (guess == magic)
    cout << "** Right **";
   else
    cout<<"Wrong";
  return 0;
}
```

OUTPUT:

```
Guess the magic number:
897
Wrong
```

PROGRAM: IF –ELSE-IF STATEMENT

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        int num;
        cout<<"Enter a number: ";
        cin>>num;
        if(num%2==0)
        {
                cout<<"You entered an even number";
        }
        else
        {
                cout<<"You entered an odd number";
        }
        getch();
}
```

OUTPUT:

```
Enter a number: 21
You entered an odd number
```

NESTED ifs

A nested **if** is an **if** that is the target of another **if** or **else**. Nested **if**s are very common in programming. In a nested **if**, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

Example,

```
    if(i)
    {
        if(j) statement 1;
        if(k) statement 2; /* this if */
        else statement 3; /* is associated with this else */
    }
    else statement 4; /* associated with if(i) */
```

Standard C++ suggests that at least 256 levels of nested **if**s be allowed in a C++ program which 15 in C language.

PROGRAM: NESTED IF

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

int main ()
{
  int magic;                    /* magic number */
  int guess;                    /* user's guess */
  magic = rand ();              /* generate the magic number */
  cout << "Guess the magic number: " << endl;
  cin >> guess;
  if (guess == magic)
    {
      cout<<"** Right **"<<endl;
      cout<<"  is the magic number\n"<< magic;
    }
  else
    {
      cout<<"Wrong "<<endl;
      if (guess > magic)
          cout<<"too high\n";
      else
          cout<<"too low\n";
    }

    return 0;
}
```

OUTPUT:

```
Guess the magic number:
1721
Wrong
too low
```

The if-else-if Ladder

A common programming construct is the *if-else-if ladder*, sometimes called the *if-else-if staircase* because of its appearance.

Its general form is

```
	if (expression) statement;
	else
	if (expression) statement;
	else
	if (expression) statement;
	...
	else statement;
```

The conditions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final **else** is executed. That is, if all other conditional tests fail, the last **else** statement is performed. If the final **else** is not present, no action takes place if all other conditions are false.

PROGRAM: IF-ELSE-IF LADDER

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
	char ch;
	float a, b, result;
	cout<<"Enter any two number: ";
	cin>>a>>b;
	cout<<"\n"<<"Enter the operator(+, -, *, /) : ";
	cin>>ch;
	cout<<"\n";
	if(ch=='+')
	{
		result=a+b;
	}
	else if(ch=='-')
	{
		result=a-b;
	}
	else if(ch=='*')
	{
		result=a*b;
	}
	else if(ch=='/')
	{
		result=a/b;
	}
	else
	{
		cout<<"Wrong Operator..!!.. exiting...press a key..";
```

```
                getch();
                exit(1);
        }
        cout<<"\n"<<"The calculated result is : "<<result<<"\n";
        getch();
}
```

OUTPUT:

```
Enter any two number: 2
3
Enter the operator(+, -, *, /) : +
The calculated result is : 5
```

PROGRAM 4: IF-ELSE-IF LADDER

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

int main ()
{
  int magic;                    /* magic number */
  int guess;                    /* user's guess */
  magic = rand ();              /* generate the magic number */
  cout << "Guess the magic number: " << endl;
  cin >> guess;
  cout<<magic<<endl;
  if (guess == magic)
    {
      cout<<"** Right ** "<<endl;
      cout<<" is the magic number"<< magic<<endl;
    }
  else if (guess > magic)
    cout<<"Wrong, too high"<<endl;
  else
    cout<<"Wrong, too low"<<endl;
  return 0;
}
```

OUTPUT:

```
Guess the magic number:
67
1804289383
Wrong, too low
```

SWITCH

C++ has a built-in multiple-branch selection statement, called **switch**, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.
The general form of the **switch** statement is

```
        switch (expression)
        {
                case constant1:
                        statement sequence
                break;
                case constant2:
                        statement sequence
                break;
                case constant3:
                        statement sequence
                break;
                        ...
                default
                        statement sequence
        }
```

The *expression* must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of *expression* is tested, in order, against the values of the constants specified in the **case** statements. When a match is found, the statement sequence associated with that **case** is executed until the **break** statement or the end of the **switch** statement is reached. The **default** statement is executed if no matches are found. The **default** is optional and, if it is not present, no action takes place if all matches fail.

Standard C++ recommends that *at least* 16,384 **case** statements be supported! Which is at least 257 **case** statements in C language. In practice, you will want to limit the number of **case** statements to a smaller amount for efficiency.

There are three important things to know about the switch statement:

- The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of relational or logical expression.
- No two case constants in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch may have case constants that are the same.
- If character constants are used in the switch statement, they are automatically converted to integers.

The switch statement is often used to process keyboard commands, such as menu selection.

PROGRAM: SWITCH

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

int main()
{
        int dow;
        cout<<"Enter number of week's day (1-7): ";
        cin>>dow;
        switch(dow)
        {
                case 1 : cout<<"\nSunday";
                        break;
                case 2 : cout<<"\nMonday";
```

```
                        break;
                case 3 : cout<<"\nTuesday";
                        break;
                case 4 : cout<<"\nWednesday";
                        break;
                case 5 : cout<<"\nThursday";
                        break;
                case 6 : cout<<"\nFriday";
                        break;
                case 7 : cout<<"\nSaturday";
                        break;
                default : cout<<"\nWrong number of day";
                        break;
        }
        getch();
}
```

OUTPUT:

```
Enter number of week's day (1-
7): 4

Wednesday
```

NESTED SWITCH STATEMENTS

You can have a **switch** as part of the statement sequence of an outer **switch**. Even if the **case** constants of the inner and outer **switch** contain common values, no conflicts arise. For example, the following code fragment is perfectly acceptable:

```
        switch(x)
        {
                case 1:
                        switch(y)
                        {
                                case 0: printf("Divide by zero error.\n");
                                break;
                                case 1: process(x,y);
                        }
                break;
                case 2:
                ......
```

.

.COMPARISON OF IF AND SWITCH

The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of relational or logical expressions.

B) ITERATION STATEMENTS

In C++, and all other modern programming languages, iteration statements (also called *loops*) allow a set of instructions to be executed repeatedly until a certain condition is reached. This condition may be predefined (as in the **for** loop), or open-ended (as in the **while** and **do-while** loops).

1. **The for Loop**

The general design of the **for** loop is reflected in some form or another in all procedural programming languages. However, in C++, it provides unexpected flexibility and power.

General form:

```
for(initialization; condition; increment)
{
        statement;
}
```

The *initialization* is an assignment statement that is used to set the loop control variable.

The *condition* is a relational expression that determines when the loop exits.

The *increment* defines how the loop control variable changes each time the loop is repeated.

You must separate these three major sections by semicolons.

The **for** loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the **for**.

PROGRAM: FOR LOOP

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

int main ()
{
  int x;
  for (x = 1; x <= 10; x++)
    {
      cout << x<<endl;

    }
  return 0;
}
```

OUTPUT:

```
1
2
3
4
5
6
7
8
9
10
```

In **for** loops, the conditional test is always performed at the top of the loop. This means that the code inside the loop may not be executed at all if the condition is false to begin with.

for Loop Variations

Several variations of the **for** are allowed that increase its power, flexibility, and applicability to certain programming situations.

One of the most common variations uses the comma operator to allow two or more variables to control the loop.

For example, the variables **x** and **y** control the following loop, and both are initialized inside the **for** statement:

```
for(x=0, y=0; x+y<10; ++x)
{
        y = getchar();
        y = y - '0'; /* subtract the ASCII code for 0
        from y */
        .
        .
        .
}
```

Commas separate the two initialization statements. Each time the loop repeats, **x** is incremented and **y**'s value is set by keyboard input. Both **x** and **y** must be at the correct value for the loop to terminate. Even though **y**'s value is set by keyboard input, **y** must be initialized to 0 so that its value is defined before the first evaluation of the conditional expression.

PROGRAM: MULTIPLE LOOP VARIABLES

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

int main ()
{
  for (int i = 0, j = 0; i < 3; i++, j++)
    {
      cout << "i: " << i << " j: " << j << endl;
    }
  return 0;
}
```

OUTPUT:

```
i: 0 j: 0
i: 1 j: 1
i: 2 j: 2
```

PROGRAM: PATTERN PRINTING

```
#include <iostream>
#include<conio.h>
using namespace std;

int  main()
{
        int i, j;
        for(i=0; i<5; i++)
        {
                for(j=0; j<=i; j++)
                {
                        cout<<"* ";
                }
                cout<<"\n";
        }
        getch();
}
```

OUTPUT:

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

Interesting trait of the for loop

- The conditional expression does not have to involve testing the loop control variable against some target value. In fact, the condition may be any relational or logical statement. This means that you can test for several possible terminating conditions.

  For example, you could use the following function to log a user onto a remote system. The user has three tries to enter the password. The loop terminates when the three tries are used up or the user enters the correct password.

  ```
  void sign_on(void)
  {
          char str[20] = "";
          int x;
          for(x=0; x<3 && strcmp(str, "password"); ++x)
          {
                  cout<<"Enter password please:";
                  gets(str);
          }
          if(x==3) return;
                  /* else log user in ... */
  }
  ```

  This function uses **strcmp( )**, the standard library function that compares two strings and returns 0 if they match.
- Each of the three sections of the **for** loop may consist of any valid expression. The expressions need not actually have anything to do with what the sections are generally used for.

PROGRAM: FOR LOOP VARIATIONS

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

int sqrnum (int num);
int readnum (void);
int prompt (void);
int
main (void)
{
```

```
  int t;
  for (prompt (); t = readnum (); prompt ())
    {
      sqrnum (t);
    }
  return 0;
}

int
prompt (void)
{
  cout << "Enter a number: " ;
  return 0;
}

int
readnum (void)
{
  int t;
  cin >> t;
  return t;
}

int
sqrnum (int num)
{
  cout << "SQUARE is " <<num * num<<endl;
  return num * num;
}
```

OUTPUT:

```
Enter a number: 17
SQUARE is 289
Enter a number: 21
SQUARE is 441
Enter a number: 0
```

- Pieces of the loop definition need not be there. In fact, there need not be an expression present for any of the sections— the expressions are optional.

  For example, this loop will run until the user enters **123**:

  ```
  for(x=0; x!=123; )
  cout<<x;
  ```

  The increment portion of the **for** definition is blank. This means that each time the loop repeats, **x** is tested to see if it equals 123, but no further action takes place. If you type **123** at the keyboard, however, the loop condition becomes false and the loop terminates.

- The initialization of the loop control variable can occur outside the **for** statement. This most frequently happens when the initial condition of the loop control variable must be computed by some complex means as in this example:

  ```
  gets(s); /* read a string into s */
  ```

```
if(*s) x = strlen(s); /* get the string's length */
else x = 10;
for( ; x<10; )
{
        cout << x;
        ++x;
}
```

The initialization section has been left blank and **x** is initialized before the loop is entered.

The Infinite Loop

Although you can use any loop statement to create an infinite loop, **for** is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty:

```
for( ; ; )
        cout<<"This loop will run forever.\n";
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the **for(;;)** construct to signify an infinite loop.

Actually, the **for(;;)** construct does not guarantee an infinite loop because a **break** statement, encountered anywhere inside the body of a loop, causes immediate termination. Program control then resumes at the code following the loop, as shown here:

```
ch = '\0';
for( ; ; )
{
        ch = getchar(); /* get a character */
        if(ch=='A') break; /* exit the loop */
}
cout<<"you typed an A";
```

This loop will run until the user types an **A** at the keyboard.

for Loops with No Bodies

A statement may be empty. This means that the body of the **for** loop (or any other loop) may also be empty. You can use this fact to improve the efficiency of certain algorithms and to create time delay loops.

Removing spaces from an input stream is a common programming task. For example, a database program may allow a query such as "show all balances less than 400." The database needs to have each word fed to it separately, without leading spaces. That is, the database input processor recognizes "**show**" but not " **show**". The following loop shows one way to accomplish this. It advances past leading spaces in the string pointed to by **str**.

```
for( ; *str == ' '; str++) ;
```

As you can see, this loop has no body—and no need for one either.

Time delay loops are often used in programs. The following code shows how to create one by using **for**:

```
for(t=0; t<SOME_VALUE; t++) ;
```

2. **The while Loop**

The second loop available in C/C++ is the **while** loop.

General form:

while(*condition*)

```
{
        statement;
}
```

where *statement* is either an empty statement, a single statement, or a block of statements.

The *condition* may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line of code immediately following the loop.

The following example shows a keyboard input routine that simply loops until the user types **A**:

```
char wait_for_char(void)
{
        char ch;
        ch = '\0'; /* initialize ch */
        while(ch != 'A') ch = getchar();
        return ch;
}
```

while loops check the test condition at the top of the loop, which means that the body of the loop will not execute if the condition is false to begin with. This feature may eliminate the need to perform a separate conditional test before the loop.

PROGRAM : WHILE LOOP

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
        unsigned long num, fact=1;
        cout<<"Enter a number: ";
        cin>>num;
        while(num)
        {
                fact = fact*num;
                num--;
        }
        cout<<"The factorial of the number is "<<fact;
        getch();
}
```

OUTPUT:

```
Enter a number: 34
The factorial of the number is 4926277576697053184
```

PROGRAM 20: CHECK PALINDROME OR NOT

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

```
using namespace std;

int  main()
{
        int num, rem, orig, rev=0;
        cout<<"Enter a number : ";
        cin>>num;
        orig=num;
        while(num!=0)
        {
                rem=num%10;
                rev=rev*10 + rem;
                num=num/10;
        }
        if(rev==orig)  // check if original number is equal to its reverse
        {
                cout<<"Palindrome";
        }
        else
        {
                cout<<"Not Palindrome";
        }
        getch();
}
```

OUTPUT:

```
Enter a number : 2345
Not Palindrome
```

Interesting trait of the while loop

- If several separate conditions need to terminate a **while** loop, a single variable commonly forms the conditional expression. The value of this variable is set at various points throughout the loop.
  In this example,

```
        void func1(void)
        {
                int working;
                working = 1; /* i.e., true */
                while(working)
                {
                working = process1();
                if(working)
                working = process2();
                if(working)
                working = process3();
                }
        }
```

  Any of the three routines may return false and cause the loop to exit.
- There need not be any statements in the body of the **while** loop.
  For example,

```
while((ch=getchar()) != 'A') ;
```

will simply loop until the user types **A**. If you feel uncomfortable putting the assignment inside the **while** conditional expression, remember that the equal sign is just an operator that evaluates to the value of the right-hand operand.

3. The do-while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop always executes at least once.

General form:

```
do
{
        statement;
} while(condition);
```

The **do-while** loop iterates until *condition* becomes false.

Perhaps the most common use of the **do-while** loop is in a menu selection function. When the user enters a valid response, it is returned as the value of the function. Invalid responses cause a reprompt.

PROGRAM: DO-WHILE

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{

        int num, l=0;
        cout<<"Enter a number: ";
        cin>>num;
        cout<<"\nIncrementing & Printing the number, 10 times:\n";
        do
        {
                num++;
                cout<<num<<"\n";
                l++;
        }while(l<10);
        getch();
}
```

OUTPUT:

```
Enter a number: 04
Incrementing & Printing the number, 10 times:

5
6
7
8
```

```
9
10
11
12
13
14
```

PROGRAM: FINDING AREA, PERIMETER AND DIAGONAL OF A RECTANGLE

```
#include <iostream>
#include<conio.h>
#include<math.h>
using namespace std;

int main()
{
        char ch, ch1;
        float l, b, peri, area, diag;
        cout<<"Rectangle Menu";
        cout<<"\n 1. Area";
        cout<<"\n 2. Perimeter";
        cout<<"\n 3. Diagonal";
        cout<<"\n 4. Exit\n";
        cout<<"\nEnter your choice: ";
        do
        {
                cin>>ch;
                if(ch == '1' || ch == '2' || ch == '3')
                {
                        cout<<"Enter length & breadth: ";
                        cin>>l>>b;
                }
                switch(ch)
                {
                        case '1' : area = l * b ;
                                cout<<"Area = "<<area;
                                break ;
                        case '2' : peri = 2 * (l + b);
                                cout<<"Perimeter = "<<peri;
                                break;
                        case '3' : diag = sqrt((l * l) + (b * b));
                                cout<<"Diagonal = "<<diag;
                                break;
                        case '4' : cout<<"Breaking..Press a key..";
                                getch();
                                exit(1);
                        default : cout<<"Wrong choice !!!!";
                                cout<<"\nEnter a valid one";
                                break;
                }     //end of switch
                cout<<"\nWant to enter more (y/n) ? ";
                cin>>ch1;
                if(ch1 == 'y' || ch1 == 'Y')
                cout<<"Again enter choice (1-4): ";
        }while(ch1 == 'y' || ch1 == 'Y') ;      //end of DO-WHILE loop
        getch();
}
```

OUTPUT:

```
Rectangle Menu
 1. Area
 2. Perimeter
 3. Diagonal
 4. Exit

Enter your choice: 1
Enter length & breadth: 2
3
Area = 6
Want to enter more (y/n) ? n
```

DECLARING VARIABLES WITHIN SELECTION AND ITERATION STATEMENTS

In C++ (but not C89), it is possible to declare a variable within the conditional expression of an **if** or **switch**, within the conditional expression of a **while** loop, or within the initialization portion of a **for** loop. A variable declared in one of these places has its scope limited to the block of code controlled by that statement. For example, a variable declared within a **for** loop will be local to that loop.

Example: declares a variable within the initialization portion of a **for** loop:

```
        /* i is local to for loop; j is known outside loop. */
int j;
for(int i = 0; i<10; i++)
j = i * i;
/* i = 10; // *** Error *** -- i not known here! */
```

Here, **i** is declared within the initialization portion of the **for** and is used to control the loop. Outside the loop, **i** is unknown.

C++, then you can also declare a variable within any conditional expression, such as those used by the **if** or a **while**.

Example: ,

```
if(int x = 20)
{
        x = x - y;
        if(x>10) y = 0;
}
```

declares **x** and assigns it the value 20. Since this is a true value, the target of the **if** executes. Variables declared within a conditional statement have their scope limited to the block of code controlled by that statement. Thus, in this case, **x** is not known outside the **if**.

C) JUMP STATEMENTS

C++ has four statements that perform an unconditional branch: **return**, **goto**, **break**, and **continue**. Of these, you may use **return** and **goto** anywhere in your program. You may use the **break** and **continue** statements in conjunction with any of the loop statements.

1. The return Statement

The **return** statement is used to return from a function. It is categorized as a jump statement because it causes execution to return (jump back) to the point at which the call to the function was made. A **return** may or may not have a value associated with it. If **return** has a value associated with it, that value becomes the return value of the function.

In C89, a non-**void** function does not technically have to return a value. If no return value is specified, a garbage value is returned. However, in C++ (and in C99), a non-**void** function *must* return a value. That is, in C++, if a function is specified as returning a value, any **return** statement within it must have a value associated with it.

General form: return *expression*;

The *expression* is present only if the function is declared as returning a value. In this case, the value of *expression* will become the return value of the function.

You can use as many **return** statements as you like within a function. However, the function will stop executing as soon as it encounters the first **return**. The **}** that ends a function also causes the function to return. It is the same as a **return** without any specified value. If this occurs within a non-**void** function, then the return value of the function is undefined.

A function declared as **void** may not contain a **return** statement that specifies a value. Since a **void** function has no return value, it makes sense that no **return** statement within a **void** function can return a value.

2. The goto Statement

It is used for jumping to a specific location. The **goto** statement requires a label for operation. (A *label* is a valid identifier followed by a colon.) Furthermore, the label must be in the same function as the **goto** that uses it—you cannot jump between functions.

General form:

statement is
goto *label*;
...
label:
where *label* is any valid label either before or after **goto**.

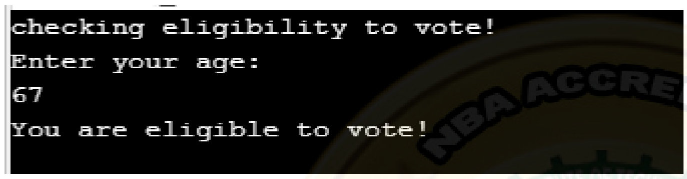
Example: you could create a loop from 1 to 100 using the **goto** and a label, as shown here:

```
x = 1;
loop1:
x++;
if(x<100) goto loop1;
```

PROGRAM: GOTO

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

int main()
{
  ineligible:
    cout<<"checking eligibility to vote!\n";
    cout<<"Enter your age:\n";
     int age;
     cin>>age;
     if (age < 18){
           goto ineligible;
     }
     else
     {
           cout<<"You are eligible to vote!";
     }
}
```

OUTPUT: _



```
checking eligibility to vote!
Enter your age:
67
You are eligible to vote!
```

3. The break Statement

The **break** statement has two uses.

- You can use it to terminate a **case** in the **switch** statement.
- You can also use it to force immediate termination of a loop, bypassing the normal loop conditional test.

When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.

For example,

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

int main(void)
{
int t;
for(t=0; t<100; t++) {
cout<< t;
if(t==10) break;
}
return 0;
}
```

Programmers often use the **break** statement in loops in which a special condition can cause immediate termination.

A **break** causes an exit from only the innermost loop.

Example:

```
        for(t=0; t<100; ++t)
        {
                count = 1;
                for(;;)
                {
                        cout<<count;
                        count++;
```

```
                    if(count==10) break;
            }
    }
```

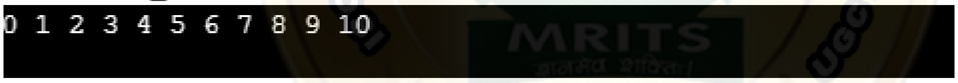
prints the numbers 1 through 10 on the screen 100 times. Each time execution encounters break, control is passed back to the outer **for** loop.

A **break** used in a **switch** statement will affect only that **switch**. It does not affect any loop the **switch** happens to be in.

PROGRAM: BREAK

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

int main(void)
{
int t;
for(t=0; t<100; t++)
{
printf("%d ", t);
if(t==10) break;
}
return 0;

}
```

OUTPUT:



```
0 1 2 3 4 5 6 7 8 9 10
```

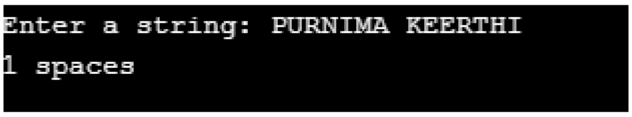
4. **The continue Statement**

The **continue** statement works somewhat like the **break** statement. Instead of forcing termination, however, **continue** forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** causes the conditional test and increment portions of the loop to execute. For the **while** and **do-while** loops, program control passes to the conditional tests. For example, the following program counts the number of spaces contained in the string entered by the user:

PROGRAM: CONTINUE

```
include <iostream>
using namespace std;

int main(void)
{
char s[80], *str;
int space;
printf("Enter a string: ");
gets(s);
str = s;
for(space=0; *str; str++)
{
if(*str != ' ') continue;
space++;
}
printf("%d spaces\n", space);
return 0;
}
```

OUTPUT:



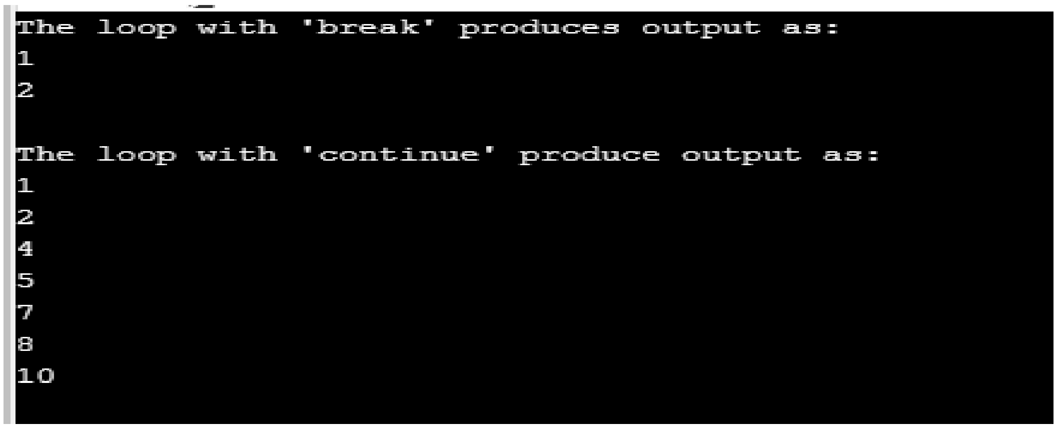
```
Enter a string: PURNIMA KEERTHI
1 spaces
```

Each character is tested to see if it is a space. If it is not, the **continue** statement forces the **for** to iterate again. If the character *is* a space, **space** is incremented.

PROGRAM: BREAK AND CONTINUE

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

int main()
{
  cout<<"The loop with \'break\' produces output as:\n";
        for(int i=1; i<=10; i++)
        {
                if((i%3)==0)
                        break;
                else
                        cout<<i<<endl;
        }
        cout<<"\nThe loop with \'continue\' produce output as:\n";
        for(int i=1; i<=10; i++)
        {
                if((i%3)==0)
                        continue;
                else
                        cout<<i<<endl;
        }

}
```

OUTPUT:



```
The loop with 'break' produces output as:
1
2

The loop with 'continue' produce output as:
1
2
4
5
7
8
10
```

ARRAYS

Definition: An *array* is a collection of variables of the same type that are referred to through a common name. A specific element in an array is accessed by an index. In C++, all arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Single-Dimension Arrays

Single-dimension arrays are essentially lists of information of the same type that are stored in contiguous memory locations in index order.

General form: *type var_name[size]*;

Like other variables, arrays must be explicitly declared so that the compiler may allocate space for them in memory. Here, *type* declares the base type of the array, which is the type of each element in the array, and *size* defines how many elements the array will hold.

Example: To declare a 100-element array called balance of type double, use this statement:

```
double balance[100];
```

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

Example: balance[3] = 12.23;

assigns element number 3 in balance the value 12.23.

In C++, all arrays have 0 as the index of their first element. Therefore, when you write

```
char p[10];
```

you are declaring a character array that has ten elements, p[0] through p[9].

The amount of storage required to hold an array is directly related to its type and size. For a single-dimension array, the total size in bytes is computed as shown here:

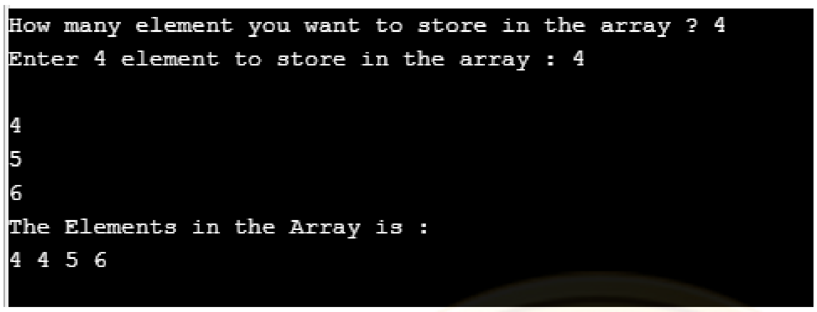
total bytes = sizeof(base type) x size of array

C++ has no bounds checking on arrays. You could overwrite either end of an array and write into some other variable's data or even into the program's code. As the programmer, it is your job to provide bounds checking where needed.

PROGRAM 31: SINGLE DIMENSIONAL ARRAY

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

int main()
{
  int arr[50], n;
        cout<<"How many element you want to store in the array ? ";
        cin>>n;
        cout<<"Enter "<<n<<" element to store in the array : ";
        for(int i=0; i<n; i++)
        {
                cin>>arr[i];
        }
        cout<<"The Elements in the Array is : \n";
        for(int  i=0; i<n; i++)
        {
                cout<<arr[i]<<" ";
        }
}
```

OUTPUT:

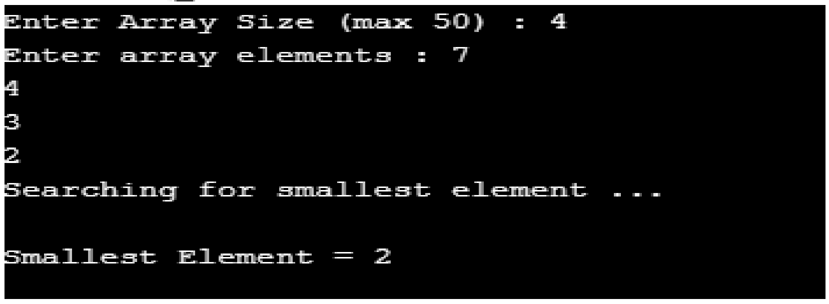


```
How many element you want to store in the array ? 4
Enter 4 element to store in the array : 4

4
5
6
The Elements in the Array is :
4 4 5 6
```

PROGRAM 32: LARGEST ELEMENTS IN ARRAY

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

int main()
{
  int small, arr[50], size, i;
        cout<<"Enter Array Size (max 50) : ";
        cin>>size;
        cout<<"Enter array elements : ";
        for(i=0; i<size; i++)
        {
                cin>>arr[i];
        }
        cout<<"Searching for smallest element ...\n\n";
        small=arr[0];
        for(i=0; i<size; i++)
        {
                if(small>arr[i])
                {
                        small=arr[i];
                }
        }
        cout<<"Smallest Element = "<<small;
}
```

OUTPUT:



```
Enter Array Size (max 50) : 4
Enter array elements : 7
4
3
2
Searching for smallest element ...

Smallest Element = 2
```

Two-Dimensional Arrays

C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, essentially, an array of one-dimensional arrays.

To declare a two-dimensional integer array d of size 10,20, you would write

int d[10][20];

C++ places each dimension in its own set of brackets.

A two-dimensional array, the following formula yields the number of bytes of memory needed to hold it:

bytes = size of 1st index x size of 2nd index x sizeof(base type)

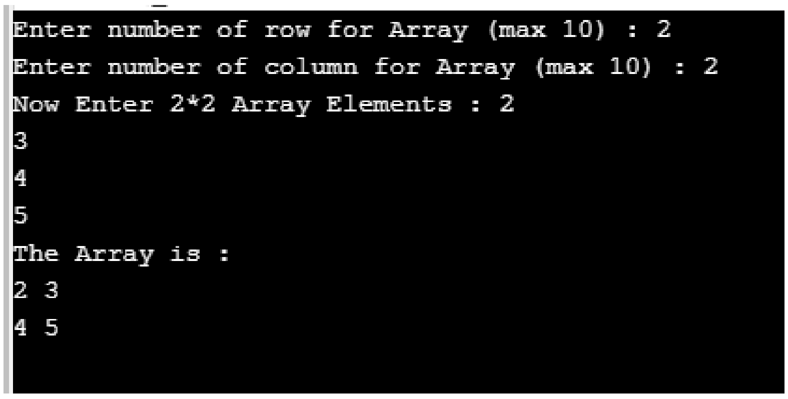
Therefore, assuming 4-byte integers, an integer array with dimensions 10,5 would have 10 x 5 x 4 or 200 bytes allocated.

Two-dimensional arrays are stored in a row-column matrix, where the first index indicates the row and the second indicates the column. This means that the rightmost index changes faster than the leftmost when accessing the elements in the array in the order in which they are actually stored in memory.

PROGRAM 38: TWO DIMENSIONAL ARRAYS

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

int main()
{
  int arr[10][10], row, col, i, j;
        cout<<"Enter number of row for Array (max 10) : ";
        cin>>row;
        cout<<"Enter number of column for Array (max 10) : ";
        cin>>col;
        cout<<"Now Enter "<<row<<"*"<<col<<" Array Elements : ";
        for(i=0; i<row; i++)
        {
                for(j=0; j<col; j++)
                {
                        cin>>arr[i][j];
                }
        }
        cout<<"The Array is :\n";
        for(i=0; i<row; i++)
        {
                for(j=0; j<col; j++)
                {
                        cout<<arr[i][j]<<" ";
                }
                cout<<"\n";
        }
}
```

OUTPUT: _



```
Enter number of row for Array (max 10) : 2
Enter number of column for Array (max 10) : 2
Now Enter 2*2 Array Elements : 2
3
4
5
The Array is :
2 3
4 5
```

PROGRAM 39: ADD TWO MATRICES

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

int main()
{
  int r, c, a[100][100], b[100][100], sum[100][100], i, j;
   cout << "Enter number of rows (between 1 and 100): ";
   cin >> r;
   cout << "Enter number of columns (between 1 and 100): ";
   cin >> c;
   cout << endl << "Enter elements of 1st matrix: " << endl;
   // Storing elements of first matrix entered by user.
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
      {
          cout << "Enter element a" << i + 1 << j + 1 << " : ";
          cin >> a[i][j];
      }
   // Storing elements of second matrix entered by user.
   cout << endl << "Enter elements of 2nd matrix: " << endl;
   for(i = 0; i < r; ++i)
      for(j = 0; j < c; ++j)
      {
          cout << "Enter element b" << i + 1 << j + 1 << " : ";
          cin >> b[i][j];
      }
   // Adding Two matrices
   for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
           sum[i][j] = a[i][j] + b[i][j];
   // Displaying the resultant sum matrix.
   cout << endl << "Sum of two matrix is: " << endl;
   for(i = 0; i < r; ++i)
       for(j = 0; j < c; ++j)
       {
           cout << sum[i][j] << "  ";
           if(j == c - 1)
```

```
            cout << endl;
        }
    return 0;
}
```

OUTPUT:

```
Enter number of rows (between 1 and 100): 2
Enter number of columns (between 1 and 100): 2

Enter elements of 1st matrix:
Enter element a11 : 1
Enter element a12 : 2
Enter element a21 : 3
Enter element a22 : 4

Enter elements of 2nd matrix:
Enter element b11 : 4
5
Enter element b12 : Enter element b21 : 7
Enter element b22 : 7

Sum of two matrix is:
5  7
10  11
```

Multidimensional Arrays

C++ allows arrays of more than two dimensions. The exact limit, if any, is determined by your compiler.

General form: *type name*[*Size1*][*Size2*][*Size3*]. . .[*SizeN*];

Arrays of more than three dimensions are not often used because of the amount of memory they require. For example, a four-dimensional character array with dimensions 10,6,9,4 requires 10 * 6 * 9 * 4 or 2,160 bytes. If the array held 2-byte integers, 4,320 bytes would be needed. If the array held doubles (assuming 8 bytes per double), 17,280 bytes would be required.

The storage required increases exponentially with the number of dimensions. For example, if a fifth dimension of size 10 was added to the preceding array, then 172,
800 bytes would be required.

In multidimensional arrays, it takes the computer time to compute each index. This means that accessing an element in a multidimensional array can be slower than accessing an element in a single-dimension array.

Array Initialization

C++ allows the initialization of arrays at the time of their declaration.

General form: *type_specifier array_name*[*size1*]. . .[*sizeN*] = { *value_list* };

The *value_list* is a comma-separated list of values whose type is compatible with *type_specifier*. The first value is placed in the first position of the array, the second value in the second position, and so on. Note that a semicolon follows the }.

Example: A 10-element integer array is initialized with the numbers 1 through 10:

int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
This means that i[0] will have the value 1 and i[9] will have the value 10.

Character arrays that hold strings allow a shorthand initialization that takes the form:
char *array_name*[*size*] = "*string*";
Example, this code fragment initializes str to the phrase "I like C++".
char str[11] = "I like C++";
This is the same as writing char str[11] = `{'I', ' ', 'l', 'i', 'k', 'e',' ', 'C', '+', '+', '\0'};`

Multidimensional arrays are initialized the same as single-dimension ones.
Example: The following initializes sqrs with the numbers 1 through 10 and their squares.

```
int sqrs[10][2] = {1, 1,
                   2, 4,
                   3, 9,
                   4, 16,
                   5, 25,
                   6, 36,
                   7, 49,
                   8, 64,
                   9, 81,
                   10, 100
                  };
```

When initializing a multidimensional array, you may add braces around the initializers for each dimension. This is called *subaggregate grouping*.
Example: Here is another way to write the preceding declaration.

```
int sqrs[10][2] = { {1, 1},
                    {2, 4},
                    {3, 9},
                    {4, 16},
                    {5, 25},
                    {6, 36},
                    {7, 49},
                    {8, 64},
                    {9, 81},
                    {10, 100}
                  };
```

When using subaggregate grouping, if you don't supply enough initializers for a given group, the remaining members will be set to zero automatically.

Unsized Array Initializations

Imagine that you are using array initialization to build a table of error messages, as shown here:

```
char e1[12] = "Read error\n";
char e2[13] = "Write error\n";
char e3[18] = "Cannot open file\n";
```

As you might guess, it is tedious to count the characters in each message manually\ to determine the correct array dimension. Fortunately, you can let the compiler automatically calculate the dimensions of the arrays. If, in an array initialization statement, the size of the array is not specified, the C++ compiler automatically creates an array big enough to hold all the initializers present. This is called an *unsized array*. Using this approach, the message table becomes

```
        char e1[] = "Read error\n";
        char e2[] = "Write error\n";
        char e3[] = "Cannot open file\n";
```

Given these initializations, this statement

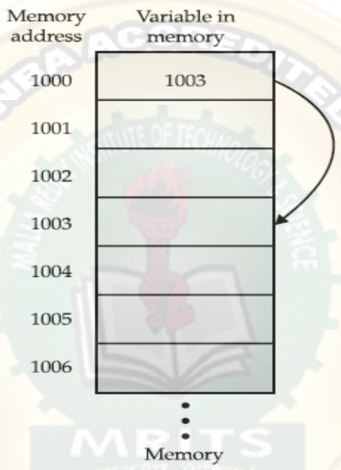
```
        cout<<"has length \n", e2, sizeof(e2);
```

will print

```
        Write error has length 13
```

POINTERS

Definition: A *pointer* is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to *point to* the second.



Pointer Variables

If a variable is going to hold a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name.

General form: *type *name;*

where *type* is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by *name.*

The Pointer Operators

There are two special pointer operators: * and &.

& Operator

The & is a unary operator that returns the memory address of its operand.

Example: m = &count;

places into m the memory address of the variable count. This address is the computer's internal location of the variable. It has nothing to do with the value of count. You can think of & as returning "the address of." Therefore, the preceding assignment statement means "m receives the address of count."

To understand the above assignment better, assume that the variable count uses memory location 2000 to store its value. Also assume that count has a value of 100. Then, after the preceding assignment, m will have the value 2000.

*** Operator**

The second pointer operator, *, is the complement of &. It is a unary operator that returns the value located at the address that follows.

Example: If m contains the memory address of the variable count,

```
q = *m;
```

places the value of count into q. Thus, q will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in m. You can think of * as "at address." In this case, the preceding statement means "q receives the value at address m."

In C++, it is illegal to convert one type of pointer into another without the use of an explicit type cast.

PROGRAM 43: POINTERS

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

int main()
{
    int var1 = 3;
   int var2 = 24;
   int var3 = 17;
   cout << &var1 << endl;
   cout << &var2 << endl;
   cout << &var3 << endl;
}
```

OUTPUT:

```
0x7ffe74f8b134
0x7ffe74f8b138
0x7ffe74f8b13c
```

PROGRAM 44: POINTERS

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

int main()
{
   int *pc, c;

   c = 5;
   cout << "Address of c (&c): " << &c << endl;
   cout << "Value of c (c): " << c << endl << endl;
   pc = &c;    // Pointer pc holds the memory address of variable c
   cout << "Address that pointer pc holds (pc): "<< pc << endl;
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

   c = 11;    // The content inside memory address &c is changed from 5 to 11.
   cout << "Address pointer pc holds (pc): " << pc << endl;
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
   *pc = 2;
```

```
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;
    return 0;
}
```

OUTPUT:

```
Address of c (&c): 0x7ffe72819c04
Value of c (c): 5

Address that pointer pc holds (pc): 0x7ffe72819c04
Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7ffe72819c04
Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7ffe72819c04
Value of c (c): 2
```

Pointer Expressions

1. **Pointer Assignments**

As with any variable, you may use a pointer on the right-hand side of an assignment statement to assign its value to another pointer.

PROGRAM 47: POINTER ASSIGNMENTS

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

int main()
{
  int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
cout<<p2; /* print the address of x, not x's value! */
return 0;
}
```

OUTPUT:

```
0x7ffd862f1e5c
```

2. **Pointer Arithmetic**

There are only two arithmetic operations that you may use on pointers: addition and subtraction. To understand what occurs in pointer arithmetic, let p1 be an integer pointer with a current value of 2000. Also, assume integers are 2 bytes long.

After the expression

p1++;

p1 contains 2002, not 2001. The reason for this is that each time p1 is incremented, it will point to the next integer. The same is true of decrements. For example, assuming that p1 has the value 2000, the expression

```
p1--;
```

causes p1 to have the value 1998.

You are not limited to the increment and decrement operators. For example, you may add or subtract integers to or from pointers. The expression

```
p1 = p1 + 12;
```

makes p1 point to the twelfth element of p1's type beyond the one it currently points to.

Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed: You may subtract one pointer from another in order to find the number of objects of their base type that separate the two. All other arithmetic operations are prohibited. Specifically, you may not multiply or divide pointers; you may not add two pointers; you may not apply the bitwise operators to them; and you may not add or subtract type float or double to or from pointers.

PROGRAM 48: POINTER ARITHMETIC: INCREMENTING POINTERS

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

int main()
{
   int var[5] = {10,20,30,40,50}; //Array Declaration.
        int *ptr; //pointer point to int.

        ptr = var; // let us have array address in pointer.

        for (int i = 0; i < 5; i++) //Loop to show Address and Value:
        {
          cout << "\n Address of var[" << i << "] = " <<ptr<<endl; //show the Address:
          cout << "Value of var[" << i << "] = "<<*ptr<<endl; //show the value:

       ptr++; // point to the next location (incrementation)
        }


    return 0;
  }
```

OUTPUT:

```
Address of var[0] = 0x7fff8ea2bc90
Value of var[0] = 10

 Address of var[1] = 0x7fff8ea2bc94
Value of var[1] = 20

 Address of var[2] = 0x7fff8ea2bc98
Value of var[2] = 30
```

```
 Address of var[3] = 0x7fff8ea2bc9c
Value of var[3] = 40

 Address of var[4] = 0x7fff8ea2bca0
Value of var[4] = 50
```

PROGRAM 49: POINTER ARITHMETIC: DECREMENTING POINTERS

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

int main()
{
   int var[5] = {10,20,30,40,50}; //Array Declaration.
   int *ptr; //pointer point to int.

   ptr = &var[4]; //ptr point to the last address of Array:

   for (int i = 5; i > 0; i--) //Loop to show Address and Value:
   {
         cout << "\n Address of var[" << i << "] = " <<ptr<<endl; //show the Address:
         cout << "Value of var[" << i << "] = "<<*ptr<<endl; //show the value:

         ptr--; // point to the Previous location:
    }
   return 0;
  }
```

OUTPUT:

```
Address of var[5] = 0x7ffec8e97200
Value of var[5] = 50

 Address of var[4] = 0x7ffec8e971fc
Value of var[4] = 40

 Address of var[3] = 0x7ffec8e971f8
Value of var[3] = 30

 Address of var[2] = 0x7ffec8e971f4
Value of var[2] = 20

 Address of var[1] = 0x7ffec8e971f0
Value of var[1] = 10
```

3. Pointer Comparisons

You can compare two pointers in a relational expression. For instance, given two pointers p and q, the following statement is perfectly valid:

```
if(p<q)
cout<<"p points to lower memory than q\n";
```

Generally, pointer comparisons are used when two or more pointers point to a common object, such as an array

PROGRAM 50: POINTER COMPARISION

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

int main()
{
    int var[5] = {10,20,30,40,50}; //Array Declaration.
   int *ptr; //pointer point to int.

   ptr = var; //ptr point to the first address of Array:
   int i=0; //Variable to use in while loop:

   while(ptr <= &var[4]) //Loop to show Address and Value:
   {
      for(int i=0; i<5; i++)
      {
            cout << "\n Address of var[" << i << "] = "
                 <<ptr<<endl; //show the Address:

            cout << "Value of var[" << i << "] = "
                 <<*ptr<<endl; //show the value:

            ptr++; // point to the Next location:
       }
    }
     return 0;
   }
```

OUTPUT:

```
Address of var[0] = 0x7ffe055b21f0
Value of var[0] = 10

 Address of var[1] = 0x7ffe055b21f4
Value of var[1] = 20

 Address of var[2] = 0x7ffe055b21f8
Value of var[2] = 30

 Address of var[3] = 0x7ffe055b21fc
Value of var[3] = 40

 Address of var[4] = 0x7ffe055b2200
Value of var[4] = 50
```

Pointers and Arrays

There is a close relationship between pointers and arrays. Consider this program fragment:

```
char str[80], *p1;
p1 = str;
```

Here, p1 has been set to the address of the first array element in str. To access the fifth element in str, you could write

```
str[4]
or
*(p1+4)
```

C++ provides two methods of accessing array elements: pointer arithmetic and array indexing. Although the standard array-indexing notation is sometimes easier to understand, pointer arithmetic can be faster. Since speed is often a consideration in programming, C/C++ programmers commonly use pointers to access array elements.

These two versions of putstr( )—one with array indexing and one with pointers

```
/* Index s as an array. */
void putstr(char *s)
{
register int t;
for(t=0; s[t]; ++t) putchar(s[t]);
}

/* Access s as a pointer. */
void putstr(char *s)
{
while(*s) putchar(*s++);
}
```

Arrays of Pointers

Pointers may be arrayed like any other data type. The declaration for an int pointer array of size 10 is

```
int *x[10];
```

To assign the address of an integer variable called var to the third element of the pointer array, write

```
x[2] = &var;
```

To find the value of var, write

```
*x[2]
```

Initializing Pointers

After a nonstatic local pointer is declared but before it has been assigned a value, it contains an unknown value. There is an important convention that most C/C++ programmers follow when working with pointers: A pointer that does not currently point to a valid memory location is given the value null (which is zero).

All pointers, when they are created, should be initialized to some value, even if it is only zero. A pointer whose value is zero is called a null pointer.

If the pointer is initialized to zero, you must specifically assign the address to the pointer.

```
pCount = &Count; // assign the address to the pointer (NO * is present)
```

It is also possible to assign the address at the time of declaration.

```
int *pCount = &Count;  //declare and assign an integer pointer
```

Another variation on the initialization theme is the following type of string declaration:

```
char *p = "hello world";
```

PROGRAM 51: ARRAY OF POINTERS: LAB PROGRAM

FUNCTIONS

Functions are the building blocks of C and C++ and the place where all program activity occurs

General Form:

ret-type function-name(*parameter list*)
{
    body of the function
}

The *ret-type* specifies the type of data that the function returns. A function may return any type of data except an array. The *parameter list* is a comma-separated list of variable names and their associated types that receive the values of the arguments when the function is called. A function may be without parameters

All function parameters must be declared individually, each including both the type and name.

General form:

f(*type varname1, type varname2, . . . , type varnameN)*

For example,

```
f(int i, int k, int j) /* correct */
f(int i, k, float j) /* incorrect */
```

PROGRAM 52: FUNCTIONS

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

int add(int, int);
int main()
{
   int num1, num2, sum;
   cout<<"Enters two numbers to add: ";
   cin >> num1 >> num2;
   // Function call
   sum = add(num1, num2);
   cout << "Sum = " << sum;
   return 0;
}
// Function definition
int add(int a, int b)
{
   int add;
   add = a + b;
```

```
    // Return statement
    return add;
}
```

OUTPUT:

```
Enters two numbers to add: 5
4
Sum = 9
```

Scope Rules of Functions

The *scope rules* of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. Each function is a discrete block of code. A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function because the two functions have a different scope.

Variables that are defined within a function are called *local* variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls.

In C++ you cannot define a function within a function.

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

```
        /* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s, char c)
{
        while(*s)
        if(*s==c)   return 1;
        else   s++;
        return 0;
 }
```

Parameter Passing

In a computer language, there are two ways that arguments can be passed to a subroutine.

a. ***Call by value***

This method copies the *value of* an argument into the formal parameter of the subroutine. In this case, changes made to the parameter have no effect on the argument.

By default, C/C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function.

PROGRAM 53: CALL BY VALUE

```
#include <iostream>
#include<conio.h>
using namespace std;

void swap(int a, int b)
```

```
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}

int main()
{
   int a = 100, b = 200;

   swap(a, b);  // passing value to function
   cout<<"Value of a"<<a;
   cout<<"Value of b"<<b;
   getch();
   return 0;
}
```

OUTPUT:

```
Value of a100Value of b200
```

b. Call by reference

In this method, the *address* of an argument is copied into the parameter. Inside the subroutine, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Creating a Call by Reference

You can create a call by reference by passing a pointer to an argument, instead of the argument itself. Since the address of the argument is passed to the function, code within the function can change the value of the argument outside the function. Pointers are passed to functions just like any other value.

Example:

```
void swap(int *x, int *y)
{
        int temp;
        temp = *x; /* save the value at address x */
        *x = *y; /* put y into x */
        *y = temp; /* put x into y */
}
```

swap( ) is able to exchange the values of the two variables pointed to by **x** and **y** because their addresses (not their values) are passed. Thus, within the function, the contents of the variables can be accessed using standard pointer operations, and the contents of the variables used to call the function are swapped.

PROGRAM 54: CALL BY REFERENCE

```
#include<iostream>
using namespace std;
#include<conio.h>

void swap (int *a, int *b)
```

```
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

int main ()
{
  int a = 100, b = 200;

  swap (&a, &b);                    // passing value to function
  cout << "Value of a" << a;
  cout << "Value of b" << b;
  return 0;
}
```

OUTPUT:

```
Value of a200Value of b100
```

C++ allows you to fully automate a call by reference through the use of reference parameters.

Reference parameters: refer References

Function declaration (Function Prototypes)

In C++ all functions must be declared before they are used. This is normally accomplished using a *function prototype*.

General form

type func_name(type parm_name1, type parm_name2,. . .,
type parm_nameN);

Example: void sqr_it(int *i); /* prototype */

The only function that does not require a prototype is **main( ),** since it is the first function called when your program begins.

In C++, an empty parameter list is simply indicated in the prototype by the absence of any parameters.

Example: int f(); /* C++ prototype for a function with no parameters */

However, in C this prototype means something different. An empty parameter list simply says that *no parameter information* is given.

In C, when a function has no parameters, its prototype uses **void** inside the parameter list.

Example: here is **f( )**'s prototype as it would appear in a C program.

float f(void);

This tells the compiler that the function has no parameters, and any call to that function that has parameters is an error. In C++, the use of **void** inside an empty parameter list is still allowed, but is redundant.

*In C++, **f( )** and **f(void)** are equivalent.*

Default Function Arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization.

Example: This declares **myfunc( )** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
    // ...
}
```

Now, **myfunc( )** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero.

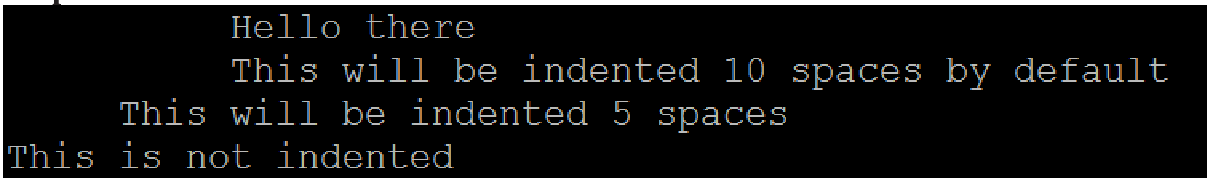
One reason that default arguments are included in C++ is because they provide another method for the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function contains more parameters than are required for its most common usage. Thus, when the default arguments apply, you need specify only the arguments that are meaningful to the exact situation, not all those needed by the most general case. For example, many of the C++ I/O functions make use of default arguments for just this reason.

A default argument can also be used as a flag telling the function to reuse a previous argument. To illustrate this usage, a function called **iputs( )** is developed here that automatically indents a string by a specified amount.

Program: Default argument

```
#include <iostream>
using namespace std;
/* Default indent to -1. This value tells the function
to reuse the previous value. */
void iputs(char *str, int indent = -1);
int main()
{
    iputs("Hello there", 10);
    iputs("This will be indented 10 spaces by default");
    iputs("This will be indented 5 spaces", 5);
    iputs("This is not indented", 0);
    return 0;
}
void iputs(char *str, int indent)
{
    static int i = 0; // holds previous indent value
    if(indent >= 0)
        i = indent;
    else // reuse old indent value
        indent = i;
    for( ; indent; indent--) cout << " ";
    cout << str << "\n";
}
```

Output:



```
          Hello there
          This will be indented 10 spaces by default
     This will be indented 5 spaces
This is not indented
```

When you are creating functions that have default arguments, it is important to remember that the default values must be specified only once, and this must be the first time the function is declared within the file.

All parameters that take default values must appear to the right of those that do not. For example, it is incorrect to define **iputs( )** like this:

```
// wrong!
void iputs(int indent = -1, char *str);
```

Once you begin to define parameters that take default values, you cannot specify a non defaulting parameter. That is, a declaration like this is also wrong and will not compile:

```
int myfunc(float f, char *str, int i=10, int j);
```

Program: Default arguments

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

class cube
{
        int x, y, z;
        public:
        cube(int i=0, int j=0, int k=0)
        {
                x=i;
                y=j;
                z=k;
        }
        int volume()
        {
                return x*y*z;
        }
};

int main()
{
        cube a(2,3,4), b;
        cout << a.volume() << endl;
        cout << b.volume();
        return 0;
}
```

Advantages

There are two advantages to including default arguments,

1. First, they prevent you from having to provide an overloaded constructor that takes no parameters.

For example, if the parameters to **cube( )** were not given defaults, the second constructor shown here would be needed to handle the declaration of **b** (which specified no arguments).

```
cube() {x=0; y=0; z=0}
```

2. Second, defaulting common initial values is more convenient than specifying them each time an object is declared.

Inline Functions

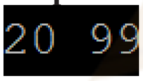
There is an important feature in C++, called an *inline function* that is commonly used with classes.

In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the

Program: Inline Function

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

inline int max(int a, int b)
{
return a>b ? a : b;
}

int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

Output:



```
20 99
```

The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

Inline is actually just a *request*, not a command, to the compiler. The compiler can choose to ignore it.

Recursion

In C/C++, a function can call itself. A function is said to be *recursive* if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called *circular definition*.
A simple example of a recursive function is **factr( )**,

```
/* recursive */
int factr(int n) {
int answer;
if(n==1) return(1);

answer = factr(n-1)*n; /* recursive call */
return(answer);
}
/* non-recursive */
int fact(int n) {
int t, answer;
answer = 1;
for(t=1; t<=n; t++)
answer=answer*(t);
return(answer);
}
```

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

PROGRAM 59: RECURSION

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
long int factorial(int);
long int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
long int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

OUTPUTS:

```
Enter any number: -1
Factorial of a number is: -1
Enter any number: 0
Factorial of a number is: 1
Enter any number: 17
Factorial of a number is: 355687428096000
```

Advantage

The main advantage to recursive functions is that you can use them to create clearer and simpler versions of several algorithms. For example, the Quicksort algorithm is difficult to implement in an iterative way.

Also, some problems, especially ones related to artificial intelligence, lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively.

Pointers to Functions

A particularly confusing yet powerful feature of C++ is the *function pointer*. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

You obtain the address of a function by using the function's name without any parentheses or arguments.

PROGRAM 60: POINTERS TO FUNCTION

```
#include <iostream>
#include<conio.h>
#include <string.h>
using namespace std;

void check(char *a, char *b,
int (*cmp)(const char *, const char *));
int main(void)
{
char s1[80], s2[80];
int (*p)(const char *, const char *);
p = strcmp;
cout<<"enter 2 strings";
gets(s1);
gets(s2);
check(s1, s2, p);
return 0;
}
void check(char *a, char *b,
int (*cmp)(const char *, const char *))
```

```
{
cout<<"Testing for equality.\n";
if(!(*cmp)(a, b)) cout<<"Equal";
else cout<<"Not Equal";
}
```

OUTPUT:

```
enter 2 strings hj
hj
Testing for equality.
Equal
```

STRINGS

C++ supports two types of strings.

a. Null-terminated string

Null-terminated string is a null-terminated character array. (A null is zero.) Thus a null-terminated string contains the characters that comprise the string followed by a null. This is the only type of string defined by C, and it is still the most widely used. Sometimes null-terminated strings are called *C-strings*.

When declaring a character array that will hold a null-terminated string, you need to declare it to be one character longer than the largest string that it is to hold. For example, to declare an array **str** that can hold a 10-character string, you would write

```
char str[11];
```

This makes room for the null at the end of the string.

When you use a quoted string constant in your program, you are also creating a null-terminated string. A *string constant* is a list of characters enclosed in double quotes.
For example,
"hello there"

You do not need to add the null to the end of string constants manually—the compiler does this for you automatically.

Null-terminated strings cannot be manipulated by any of the standard C++ operators. Nor can they take part in normal C++ expressions. For example, consider this fragment:

```
char s1[80], s2[80], s3[80];
s1 = "Alpha"; // can't do
s2 = "Beta"; // can't do
s3 = s1 + s2; // error, not allowed
```

As the comments show, in C++ it is not possible to use the assignment operator to give a character array a new value (except during initialization), nor is it possible to use the + operator to concatenate two strings. These

b. C++String Class

C++ also defines a string class, called **string**, which provides an object-oriented approach to string handling prevents such errors.

There are three reasons for the inclusion of the standard **string** class: consistency (a string now defines a data type), convenience (you may use the standard C++ operators), and safety (array boundaries will not be overrun).

C/C++ supports a wide range of functions that manipulate null-terminated strings.
The most common are

Name	**Function**
strcpy(*s1*, *s2*)	Copies *s2* into *s1*.
strcat(*s1*, *s2*)	Concatenates *s2* onto the end of *s1*.
strlen(*s1*)	Returns the length of *s1*.
strcmp(*s1*, *s2*)	Returns 0 if *s1* and *s2* are the same; less than 0 if *s1*<*s2*; greater than 0 if *s1*>*s2*.
strchr(*s1*, *ch*)	Returns a pointer to the first occurrence of *ch* in *s1*.
strstr(*s1*, *s2*)	Returns a pointer to the first occurrence of *s2* in *s1*.

These functions use the standard header file **string.h**. (C++ programs can also use the C++-style header **<cstring>**.)

PROGRAM 61: STRING TO READ A WORD

```
#include <iostream>
#include<conio.h>
#include <string.h>
using namespace std;

int main()
{
   char str[100];
   cout << "Enter a string: ";
   cin >> str;
   cout << "You entered: " << str << endl;
   cout << "\nEnter another string: ";
   cin >> str;
   cout << "You entered: "<<str<<endl;
   return 0;
}
```

OUTPUT:

```
Enter a string: KEERTHI
You entered: KEERTHI

Enter another string: KEERTHI
You entered: KEERTHI
```

PROGRAM 63: STRING USING STRING DATA TYPE

```
#include <iostream>
#include<conio.h>
#include <string.h>
using namespace std;

int main()
{
    // Declaring a string object
   string str;
   cout << "Enter a string: ";
   getline(cin, str);
   cout << "You entered: " << str << endl;
```

```
    return 0;
}
```

OUTPUT:

```
Enter a string: GOOD EVENING
You entered: GOOD EVENING
```

PROGRAM 64: STRING FUNCTIONS

```
#include<iostream>
using namespace std;
#include<conio.h>
#include <string.h>

int main ()
{
  char s1[80], s2[80];
  cout << "enter 2 strings";
  gets (s1);
  gets (s2);
  cout << "lengths: \n" << strlen (s1) << endl << strlen (s2) << endl;
  if (!strcmp (s1, s2))
    cout << "The strings are equal" << endl;
  strcat (s1, s2);
  cout << s1 << endl;
  strcpy (s1, "This is a test.\n");
  cout << s1;
  if (strchr ("hello", 'e'))
    cout << "e is in hello\n";
  if (strstr ("hi there", "hi"))
    cout << "found hi";
  return 0;
}
```

OUTPUT:

```
enter 2 stringshello
hello
lengths:
5
5
The strings are equal
hellohello
This is a test.
e is in hello
found hi
```

STRUCTURES

A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. A *structure declaration* forms a template that may be used to create structure objects (that is, instances of a structure). The variables that make up the structure are called *members*. (Structure members are also commonly referred to as *elements* or *fields*.)

General form:

```
struct struct-type-name
{
        type member-name;
        type member-name;
        type member-name;
                ..
} structure-variables;
```

where either *struct-type-name* or *structure-variables* may be omitted, but not both.

Example:

```
struct addr
{
        char name[30];
        char street[40];
        char city[20];
        char state[3];
        unsigned long int zip;
} addr_info, binfo, cinfo;
```

At this point, *no variable has actually been created*. Only the form of the data has been defined. When you define a structure, you are defining a compound variable type, not a variable. Not until you declare a variable of that type does one actually exist. In C, to declare a variable (i.e., a physical object) of type **addr**, write

```
struct addr addr_info;
```

This declares a variable of type **addr** called **addr_info**. In C++, you may use this shorter form.

```
addr addr_info;
```

In C++, once a structure has been declared, you may declare variables of its type using only its type name, without preceding it with the keyword **struct.** The reason for this difference is that in C, a structure's name does not define a complete type name. In fact, Standard C refers to a structure's name as a *tag*. In C, you must precede the tag with the keyword **struct** when declaring variables. However, in C++, a structure's name is a complete type name and may be used by itself to define variables.

When a structure variable (such as **addr_info**) is declared, the compiler automatically allocates sufficient memory to accommodate all of its members.

Accessing Structure Members

Individual members of a structure are accessed through the use of the **.** operator (usually called the *dot operator*).

The structure variable name followed by a period and the member name references that individual member. The general form for accessing a member of a structure is

structure-name.member-name

Therefore, to print the ZIP code on the screen, write

```
printf("%lu", addr_info.zip);
```

Structure Assignments

The information contained in one structure may be assigned to another structure of the same type using a single assignment statement. That is, you do not need to assign the value of each member separately.

```
struct {
int a;
int b;
} x, y;
x.a = 10;
y = x; /* assign one structure to another */
```

PROGRAM 66: STRUCTURE ASSIGNMENT

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

int main()
{
   struct {
int a;
int b;
} x, y;
x.a = 10;
y = x; /* assign one structure to another */
cout<<y.a;
return 0;
}
```

OUTPUT:

```
10
```

Arrays of Structures

To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type **addr**, defined earlier, write

```
struct addr addr_info[100];
```

To access a specific structure, index the structure name. For example, to print the ZIP code of structure 3, write

```
printf("%lu", addr_info[2].zip);
```

Passing Structures to Functions

a. Passing Structure Members to Functions

When you pass a member of a structure to a function, you are actually passing the value of that member to the function. Therefore, you are passing a simple variable

For example,

consider this structure:

```
struct fred
{
char x;
int y;
float z;
char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x); /* passes character value of x */
```

```
func2(mike.y); /* passes integer value of y */
func3(mike.z); /* passes float value of z */
func4(mike.s); /* passes address of string s */
func(mike.s[2]); /* passes character value of s[2] */
```

If you wish to pass the *address* of an individual structure member, put the **&** operator before the structure name. For example, to pass the address of the members of the structure **mike**, write

```
func(&mike.x); /* passes address of character x */
func2(&mike.y); /* passes address of integer y */
func3(&mike.z); /* passes address of float z */
func4(mike.s); /* passes address of string s */
func(&mike.s[2]); /* passes address of character s[2] */
```

b. Passing Entire Structures to Functions

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that any changes made to the contents of the structure inside the function to which it is passed do not affect the structure used as an argument.

```
/* Define a structure type. */
struct struct_type {
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
printf("%d", parm.a);
}
```

As this program illustrates, if you will be declaring parameters that are structures, you must make the declaration of the structure type global so that all parts of your program can use it. For example, had **struct_type** been declared inside **main( )** (for example), then it would not have been visible to **f1( ).**

PROGRAM 67: PASSING A STRUCTURE TO A FUNCTION

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

struct struct_type
{
int a, b;
char ch;
} ;
void f1(struct struct_type parm);
int main(void)
{
struct struct_type arg;
arg.a = 1000;
f1(arg);
return 0;
}
void f1(struct struct_type parm)
{
cout<<parm.a;
}
```

OUTPUT:

```
1000
```

PROGRAM 68: PASSING A STRUCTURE TO A FUNCTION

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

struct Person
{
   char name[50];
   int age;
   float salary;
};
void displayData(Person);   // Function declaration
int main()
{
   Person p;
   cout << "Enter Full name: ";
   cin.get(p.name, 50);
   cout << "Enter age: ";
   cin >> p.age;
   cout << "Enter salary: ";
   cin >> p.salary;
   // Function call with structure variable as an argument
   displayData(p);
   return 0;
}
void displayData(Person p)
{
   cout << "\nDisplaying Information." << endl;
   cout << "Name: " << p.name << endl;
   cout <<"Age: " << p.age << endl;
   cout << "Salary: " << p.salary;
}
```

OUTPUT:

```
Enter Full name: ABC
Enter age: 23
Enter salary: 25000

Displaying Information.
Name: ABC
Age: 23
Salary: 25000
```

Structure Pointers

C++ allows pointers to structures just as it allows pointers to any other type of variable.

Declaring a Structure Pointer

Like other pointers, structure pointers are declared by placing ***** in front of a structure variable's name. For example, assuming the previously defined structure **addr**, the following declares **addr_pointer** as a pointer to data of that type:

```
struct addr *addr_pointer;
```

Remember, in C++ it is not necessary to precede this declaration with the keyword **struct**.

Using Structure Pointers

There are two primary uses for structure pointers: to pass a structure to a function using call by reference, and to create linked lists and other dynamic data structures that rely on dynamic allocation.

When a pointer to a structure is passed to a function, only the address of the structure is pushed on the stack. This makes for very fast function calls. A second advantage, in some cases, is when a function needs to reference the actual structure used as the argument, instead of a copy. By passing a pointer, the function can modify the contents of the structure used in the call.

To find the address of a structure, place the **&** operator before the structure's name. For example, given the following fragment:

```
struct bal {
float balance;
char name[80];
} person;
struct bal *p; /* declare a structure pointer */
```

then

```
p = &person;
```

places the address of the structure **person** into the pointer **p**.

To access the members of a structure using a pointer to that structure, you must use the -> operator. For example, this references the **balance** field:

```
p->balance
```

The -> is usually called the *arrow operator*, and consists of the minus sign followed by a greater-than sign. The arrow is used in place of the dot operator when you are accessing a structure member through a pointer to the structure.

PROGRAM 69: POINTER STRUCTURES

```
#include <iostream>
```

```
using namespace std;

struct Distance
{
   int feet;
   float inch;
};
int main()
{
   Distance *ptr, d;
   ptr = &d;

   cout << "Enter feet: ";
   cin >> (*ptr).feet;
   cout << "Enter inch: ";
   cin >> (*ptr).inch;

   cout << "Displaying information." << endl;
   cout << "Distance = " << (*ptr).feet << " feet " << (*ptr).inch << " inches";
   return 0;
}
```

OUTPUT:

```
Enter feet: 10
Enter inch: 2
Displaying information.
Distance = 10 feet 2 inches
```

REFERENCES

C++ contains a feature that is related to the pointer called a *reference*. A reference is essentially an implicit pointer. There are three ways that a reference can be used: as a function parameter, as a function return value, or as a stand-alone reference.

Reference Parameters

Probably the most important use for a reference is to allow you to create functions that automatically use call-by-reference parameter passing. Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function

By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing. First, you can explicitly pass a pointer to the argument. Second, you can use a reference parameter. For most circumstances the best way is to use a reference parameter.

PROGRAM 70: REFERENCES
WITHOUT REFERENCE

```
// Manually create a call-by-reference using a pointer.
```

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

void neg(int *i);
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(&x);
cout << x << "\n";
return 0;
}
void neg(int *i)
{
*i = -*i;
}
```

WITH REFERENCE

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

void neg(int &i); // i now a reference
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(x); // no longer need the & operator
cout << x << "\n";
return 0;
}
void neg(int &i)
{
i = -i; // i is now a reference, don't need *
}
```

OUTPUT:

```
10 negated is -10
```

Returning References

A function may return a reference

simple program:

```
#include <iostream>
using namespace std;
char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

One thing you must be careful about when returning references is that the object being referred to does not go out of scope after the function terminates.

PROGRAM 72: RETURNING REFERENCES

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

char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

OUTPUT:

```
HelloXThere
```

C++'S DYNAMIC ALLOCATION OPERATORS

C++ provides two dynamic allocation operators: **new** and **delete**. C++ also supports dynamic memory allocation functions, called **malloc( )** and **free( ).** These are included for the sake of compatibility with C.

The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**.

General forms:

p_var = new *type*;
delete *p_var*;

Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception will be generated. This exception is defined in the header **<new>**. Your program should handle this exception and take appropriate action if a failure occurs

The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

Although **new** and **delete** perform functions similar to **malloc( )** and **free( )**, they have several advantages.

1. First, **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use the **sizeof** operator. Because the size is computed automatically, it eliminates any possibility for error in this regard.
2. Second, **new** automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using **malloc( )**. Finally, both **new** and **delete** can be overloaded, allowing you to create customized allocation systems.

Initializing Allocated Memory

You can initialize allocated memory to some known value by putting an initialize after the type name in the **new** statement. Here is the general form of **new** when an initialization is included:

General Form*:* *p_var* = new *var_type (initializer);*

PROGRAM 73: DYNAMIC MEMORY ALLOCATION AND DEALLOCATION TO HOLD AN INTEGER

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

int main()
{
        int *p;
        try {
                p = new int; // allocate space for an int
        } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
}
        *p = 100;
        cout << "At " << p << " ";
        cout << "is the value " << *p << "\n";
        delete p;
        return 0;
}
```

OUTPUT:

```
At 0x162cc20 is the value 100
```

PROGRAM 74: DYNAMIC MEMORY ALLOCATION AND DEALLOCATION WHICH GIVES THE ALLOCATED INTEGER AN INITIAL VALUE

```
#include <iostream>

using namespace std;

int main()
{
        int *p;
        try {
        p = new int (87); // initialize to 87
        } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
}
        cout << "At " << p << " ";
        cout << "is the value " << *p << "\n";
        delete p;
        return 0;
}
```

OUTPUT:

```
At 0xc14c20 is the value 87
```

Allocating Arrays

You can allocate arrays using **new.**

General form: *p_var* = new *array_type [size];*

Here, *size* specifies the number of elements in the array.

To free an array, use this form of **delete:**

General form: delete [ ] *p_var*;

Here, the **[ ]** informs **delete** that an array is being released.

PROGRAM: DYNAMIC MEMORY ALLOCATION AND DEALLOCATION OF ARRAYS

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

int main()
{
        int *p, i;
        try {
        p = new int [10]; // allocate 10 integer array
        } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
        }
        for(i=0; i<10; i++ )
        p[i] = i;
        for(i=0; i<10; i++)
        cout << p[i] << " ";
        delete [] p; // release the array
        return 0;
}
```

OUTPUT:

```
0 1 2 3 4 5 6 7 8 9
```

The Placement Form of new

There is a special form of **new**, called the *placement form*, that can be used to specify an alternative method of allocating memory. It is primarily useful when overloading the **new** operator for special circumstances.

General form: *p_var* = new (*arg-list*) *type*;

Here, *arg-list* is a comma-separated list of values passed to an overloaded form of **new**.

PREPROCESSOR DIRECTIVES

You can include various instructions to the compiler in the source code of a C/C++ program. These are called *preprocessor directives*

C++ preprocessor is virtually identical to the one defined by C. The main difference between C and C++ in this regard is the degree to which each relies upon the preprocessor. In C, each preprocessor directive is necessary. In C++, some features have been rendered redundant by newer and better C++ language elements.

All preprocessor directives begin with a # sign. In addition, each preprocessing directive must be on its own line.

For example,

```
#include <stdio.h> #include <stdlib.h>
```

will not work.

The preprocessor contains the following directives:

#define

The **#define** directive defines an identifier and a character sequence (i.e., a set of characters) that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as a *macro name* and the replacement process as *macro replacement*.

General form : #define *macro-name char-sequence*

Notice that there is no semicolon in this statement.

Example:

```
#define LEFT 1
#define RIGHT 0
```

#if , #elif , #else , #endif

Perhaps the most commonly used conditional compilation directives are the **#if**, **#else**, **#elif**, and **#endif**. These directives allow you to conditionally include portions of code based upon the outcome of a constant expression.

General form of #if :

```
#if constant-expression
statement sequence
#endif
```

If the constant expression following **#if** is true, the code that is between it and **#endif** is compiled. Otherwise, the intervening code is skipped. The **#endif** directive marks the end of an **#if** block.

For example,

```
/* Simple #if example. */
#include <stdio.h>
#define MAX 100
int main(void)
{
#if MAX>99
cout<<"Compiled for array greater than 99.\n";
#endif
return 0;
```

}
This program displays the message on the screen because **MAX** is greater than 99.

The **#elif** directive means "else if" and establishes an if-else-if chain for multiple compilation options. **#elif** is followed by a constant expression. If the expression is true, that block of code is compiled and no other **#elif** expressions are tested. Otherwise, the next block in the series is checked.

General form for #elif :

```
#if expression
        statement sequence
#elif expression 1
        statement sequence
#elif expression 2
        statement sequence
#elif expression 3
        statement sequence
#elif expression 4
.
.
.
#elif expression N
        statement sequence
#endif
```

#error

The **#error** directive forces the compiler to stop compilation. It is used primarily for debugging.

General form : #error *error-message*

The *error-message* is not between double quotes. When the **#error** directive is encountered, the error message is displayed, possibly along with other information defined by the compiler.

#ifdef , #ifndef

Another method of conditional compilation uses the directives **#ifdef** and **#ifndef**, which mean "if defined" and "if not defined," respectively.

General form of #ifdef:

```
#ifdef macro-name
        statement sequence
#endif
```

If *macro-name* has been previously defined in a **#define** statement, the block of code will be compiled.

General form of #ifndef:

```
#ifndef macro-name
        statement sequence
#endif
```

If *macro-name* is currently undefined by a **#define** statement, the block of code is compiled.

Both **#ifdef** and **#ifndef** may use an **#else** or **#elif** statement.

For example,

```
#include <stdio.h>
#define TED 10
int main(void)
{
#ifdef TED
cout<<"Hi Ted\n";
#else
cout<<"Hi anyone\n";
#endif
#ifndef RALPH
cout<<"RALPH not defined\n";
#endif
return 0;
}
```

You may nest **#ifdef**s and **#ifndef**s to at least eight levels in Standard C. Standard C++ suggests that at least 256 levels of nesting be supported.

#include

The **#include** directive instructs the compiler to read another source file in addition to the one that contains the **#include** directive. The name of the additional source file must be enclosed between double quotes or angle brackets.
For example,

```
#include "stdio.h"
#include <stdio.h>
```

both instruct the compiler to read and compile the header for the C I/O system library functions.

Include files can have **#include** directives in them. This is referred to as *nested includes*. The number of levels of nesting allowed varies between compilers. However, Standard C stipulates that at least eight nested inclusions will be available. Standard C++ recommends that at least 256 levels of nesting be supported.

#undef

The **#undef** directive removes a previously defined definition of the macro name that follows it. That is, it "undefines" a macro.
General form : #undef *macro-name*

For example,

```
#define LEN 100
#define WIDTH 100
char array[LEN][WIDTH];
#undef LEN
#undef WIDTH
/* at this point both LEN and WIDTH are undefined */
```

Both **LEN** and **WIDTH** are defined until the **#undef** statements are encountered. **#undef** is used principally to allow macro names to be localized to only those sections of code that need them.

#line

The **#line** directive changes the contents of **_ _LINE_ _** and **_ _FILE_ _ ,** which are predefined identifiers in the compiler. The **_ _LINE_ _** identifier contains the line number of the currently compiled line of code. The **_ _FILE_ _** identifier is a string that contains the name of the source file being compiled.

General form: #line *number* "*filename*"

where *number* is any positive integer and becomes the new value of **_ _LINE_ _ ,** and the optional *filename* is any valid file identifier, which becomes the new value of **_ _FILE_ _. #line** is primarily used for debugging and special applications.

For example, the following code specifies that the line count will begin with 100. The cout statement displays the number 102 because it is the third line in the program after the **#line 100** statement.

For example, the following code specifies that the line count will begin with 100. The cout statement displays the number 102 because it is the third line in the program after the **#line 100** statement.

```
#include <stdio.h>
#line 100 /* reset the line counter */
int main(void) /* line 100 */
{ /* line 101 */
cout<<__LINE__; /* line 102 */
return 0;
}
```

#pragma

#pragma is an implementation-defined directive that allows various instructions to be given to the compiler. For example, a compiler may have an option that supports program execution tracing. A trace option would then be specified by a **#pragma** statement.

PROGRAM: PREPROCESSOR DIRECTIVES (#define)

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

using namespace std;

#define PI 3.14159

int main () {
  cout << "Value of PI :" << PI << endl;

  return 0;
}
```

OUTPUT:

```
Value of PI :3.14159
```

UNIT-II NOTES

C++ Classes and Abstraction:

Class definition:

It is a collection of data members & data functions like structure.

Class Structure & Class objects:

The general form of a class declaration is,

```
class class_name
    {
        private data & functions
        public:
        public data & functions
    }object name list;
```

Ex:

```
    #define SIZE 100
    //This creates a class stack
    Class stack
        {
            int stck[SIZE];
            int tos;
        public:
            void init();
            void push(int i);

            int pop();

        };
```

- A class may contain private as well as public parts. By default, all items are private.
- For ex. stck & tos are private.
- These private data members only access the functions that are data members.
- All other variables, or functions defined after public can be accessed by all other functions in the program.
- The rest of your program accesses an object through its public functions.
- The functions init(),push() & pop() are called member functions.
- The variables stck & tos are called member variables(data member).
- Only member functions have access to the private members of their class.

  Only, init(), push() & pop() may access stck & tos.
- Once you have defined a class, you can create an object of that type by using the calss name.

  ```
  stack mystack;
  ```
- Mystack is a instance of stack.
- Class is a logical abstraction & while object is real(that ias, an object exists inside the memory of the computer).
- Inside the class stack, members functions were identiofied using their prototypes.
- When it comes time to actually code a function that is the member of a class, you must tell the compiler which class the

function belongs to by qualifying it name with the name of a class of which it is member.

- For ex., here is one way to code the push() function,

```
void stack::push(int i)
            {
                    if(tos==SIZE)
                            {
                                    cout<<"stack is full\n";
                                    return;
                            }
                    stck[tos]=i;
                    tos++;
            }
```

- The **::** is called the scope resolution operator.
- It tells the compiler that this version of push() belongs to the stack class.
- Or, this push() is in stacks scope.
- In C++, several different classes can use the same function same. The compiler knows which function belongs to which class because of the scope resolution operator.
- When you refer to member of a class from a piece of code that is not part of the calss, you must always do so in conjunction with an object of that class to do so,

```
            stack stack1,stack2;
            stack1.init();
```

the remaining code,

```
void stack::init(){

            tos=0;

      }
int stack::pop()
      {
            if(tos==0){
                  cout<<"stack underflow \n";
                  return 0;
            }
            tos--;
            return stck[tos];
      }
int main()
      {
            stack stack1,stack2;  // create two stack objects
            stack1.init();
            stack2.init();
            stack1.push(1);
            stack2.push(2);
            stack1.push(3);
            stack2.push(4);
```

```
        cout<<stack1.pop()<<" ";
        cout<<stack1.pop()<<" ";
        cout<<stack2.pop()<<" ";
        cout<<stack2.pop()<<" ";
        return 0;
    }
```

Output:

```
        3 1 4 2
```

The this pointer:

i. It is used to retrieve the pointer address.
   Like, we use '&' operator in C to retrieve the address. In C++, we use **this** pointer to know the current object address.

ii. The this pointer is used to distinguish data member from local variables. When both are declared with the same.
   - usually, the variables are of two types, static & non-static variables.
   1. Every non-static member of C++, having local variable called this.
   2. the static member never contains this.
   - it(this) returns address in hexadecimal format.

```
i. #include<iostream>
   using namespace std;
   class test
       {
        int a,b;
    public:
        void show()
            {
                a=10;
                b=10;
                cout<<”object Address is:  ”<<this;//returns
//the current object address
                cout<<”a=”<<this->a<<endl;
                cout<<”b=”<<this->b<<endl;
            }
    };
    int main()
        {
            test t;
            t.show();
            return 0;
        }
```

ii. **this** pointer is used to distinguish data member & arguments.

```
#include<iostrem>
using namespace std;
```

```
class test
        {
                int a,b;
            public:
                void show(int a,int b)
                    {
                    a=a;
                    b=b;
                    }
                void display()
                    {
                        cout<<a<<endl;
                        cout<<b;
                    }
        };
        int main()
            {
                test t;
                t.show(10,20);
                t.display();
}
```

- In every programming, the local variables has the priority. Here, in show() method both data member & local variables has same name & data members are not recognized.
- In order to distinguish the data members & local variables we use **this** pointer.
- The statement,

```
a=a;
b=b;
can be written as,
this->a=a;    or (*this).a=a;
this->b=b;    or (*this).b=b;
```

Friend classes:

- It is possible for one class to be a friend of another class.
- When this is the case, the friend class & all of its functions have access to the private members defined within the other class.
- For ex.
- // using friend class

```
#include<iostream>
using namespace std;
class TwoValues
        {
                int a;
                int b;
        public:
                TwoValues(int i,int j){
                    a=i;
                    b=j;
                }
                friend class Min;
        };
        class Min
                {
                        public:
                                int min(TwoValues x);
                };

        int Min::min(TwoValues x)

                {

                        return x.a<x.b?x.a:x.b;

                }

        int main()

                {

                        TwoValues ob(10,20);

                        Min m;

                        cout<<m.min(ob);

                        return 0;

                }
```

Output:

```
        10
```

- In this ex., class Min has access to the private variables a & b declared within the TwoValues class.
- It is critical to understand that when one class is a friend of another, it only has access names defined within the other class. It does not inherit the other class.
- Specifically , the members of the first class do not become members of the friend class.

Static class members:

- Both function & data members of a class can be made static.

Static Data Members:

A static member variable has certain special characteristics,

- It is initialized to zero when the first object of its class is created.
- Only one copy of that member is created for the entire class & is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is the entire program.
- Static variables are normally used to maintain values common to the entire class.
- For ex., a static data member can be occurrences of all the objects.

```
#include<iostream>
using namespace std;
class item
        {
                static int count;
                int number;
        public:
```

```
        void getdata(int a)
                {
                        number=a;
                        count++;
                }

        void getcount(void)

                {

                        cout<<"count";

                        count<<count<<"\n";

                }

};
int item::count;
int main()
        {
                item a,b,c;

                a.getcount(); // display count
                b.getcount();
                c.getcount();

                a.getdata(100); // getting data into object a
                b.getdata(200); //getting data into object b
                c.getdata(300);// getting data into object c

                cout<<"After reading data"<<"\n";

                a.getcount(); // display count
                b.getcount();
        c.getcount();
        return 0;
    }
Output:
    count:0
    count:0
    count:0
    After reading data
    count:3
    count:3
    count:3
```

Static Member Functions:

- A member function that is declared static has the following properties,
  i. A static function can have access to only other static members(functions, or variables) declared in the same class.
  ii. A static member function can be called using the class name.
     class_name::function_name();
     below program illustrates the implementations of these characteristics.
- The static function showcount() displays the number of objects created till that moment.
- A count of member of objects created is maintained by the static variable count.
- The function showcode() displays the code number of each object.

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

```
class test
        {
                int code;
                static int count;
        public:
                void setcode(void)
                        {
                                code=++count;
                        }
                void showcode(void)
                        {
                                cout<<”object number:”<<code<<”\n”;
                        }
                static void showcount(void)
                        {
                                cout<<”count:”<<count<<”\n”;
                        }
        };
        int test::count;
        int main()
                {
                        test t1,t2;
                        t1.setcode();
                        t2.setcode();
                        test::showcount();
                        test t3;
                        t3.setcode();
                        test::showcount();
                        t1.showcode();
                        t2.showcode();
                        t3.showcode();
                        return 0;
                }
```

Output:

```
Count:2

Count:3

Object number:1

Object number:2

Object number:3
```

const Member Functions:

- If a member functions does not alter any data in the class,
- Then we may declare it as a const member function as follows.
  void mul(int,int) const;
  double get_balance() const;
- The qualifier const is appended to the function prototypes(in both declaration & definition).
- The compiler will generate an error message if such functions try to alter the data values.

Constructor & Destructor:

Introduction:

- A.input(); // Initializes variables of object A.
- X.getdata(100,299.95);

- These function calls, can not be used to initialize the member variables at the time of object creation.
- Like built in types, we can also create user-defined datatypes.
- This means that we can initialize a class type variables(objects) when it is declared.
- As same as, initialization of an ordinary variable.
- When variable goes out of scope, it will be destroyed, but it is not happened with the objects.
- C++ provides a special function called the constructor which enables an object to initialize itself when it is created.

Constructor:

- A constructor is a special member function whose task is to initialize the objects of its class.
- It is name is the same as the class name.
- It is invoked whenever an object of its associated class is created.
- A constructor is declared & defined as follows,

```
class integer
        {
                int m;
                int n;
        public:
                integer(void);  //constructor declared
                ------------------
                ------------------
        };
```

```
Integer::integer(void)  //constructor defined

        {
                m=0;

                n=0;

        }
```

- intger int1;   // not only creates object, initializes its data //member to m & n zero.
- There is no need to write any statement to invoke the constructor function(like normal member functions).
- A constructor that accepts no parameter is called the default constructor.
- The default constructor for class a is A::A();
  If no such constructor is defined, the compiler supports the default constructor.

Characteristics of constructor function:

- They should be declared in the public function.
- They are invoked automatically when the objects are created.
- They do not return types, not even void.
- They can't be inherited.
- Like C++ functions, they can have default arguments.
- We can not refer to their address.

Parameterized Constructor:

- In previous, we initialized, the data members of all the objects to zero.
- In practice, we have to initialize the data member of object with different value when they are created.
- This can be achieved by passing arguments constructor.
- The constructor which takes arguments called parameterized constructor.
  class integer

```
        {
                int m;
                int n;
        public:
                integer(int x,int y);
                --------------------------
                --------------------------
        };
Intger::integer(int x,int y)
        {
                m=x;
                n=y;
        }
```

- Here,
  integer int1;   //doesn't works
- We have to pass arguments, to the constructor. This can be done in two ways.
  (i). By calling constructor explicitly
  `integer int1=integer(0,100);`
  (ii). By calling the constructor implicitly
  `integer int1(0,100);`
  Ex. This program defines a called point that stores the x & y coordinates of a point.

```
#include<iotsream>
class point
        {
                int x;
                int y;
        public:
                point(int a,int b)
                        {
                                x=a;
                                y=b;
                        }

                void display()

                        {

                                Cout<<"C"<<x<<","<<y<<")\n";

                        }

        };

    int main()

        {

                point p1(1,1); //invokes parameterized constructor

                point p2(5,10);

                cout<<"Point p1=";

                p1.display();

                cout<<"Point p2=";

                p2.display();

                return 0;

        }
```

Output:

```
    Point p1=(1,1)
```

};

Point p2=(5,10)

Multiple Constructor in a Calss:

```
class integer
        {
                int m;
                int n;
        public:
                integer()   //constructor1
                        {
                                m=0;
                                n=0;
                        }
                integer(int a,int b)

                        {

                                m=a;

                                n=b;

                        }
                integer(integer &i)

                        {

                                m=i.m;

                                n=i.n;

                        }

        };
int main()
```

```
{
        integer I1;
        integer I2(20,40);
        integer I3(I2);
        return 0;
}
```

Constructor with Default Arguments:

- It is possible to define constructor with default arguments.
- For ex. The constructor complex can be declared as,
  complex(float real,float imag=0);
  the default value of argument imag is zero.
- complex c(5.0); // real=5.0,imag=0
- complex c(2.0,3.0);
  //real=2.0, imag=3.0
- the difference between default constructor & default argument constructor.
- The default argument constructor can be called with either one argument or no arguments.

Copy constructor:

- A copy constructor is used to declare & initialize an object from another object.
  Integer I2(I1);
  //The above statement define object I2 & initializes it to the values of I1.
- Another form is,

  Integer I2=I1;

- integer(integer &i);

```
#include<iostream>
using namespace std;
class code
        {
                int id;
        public:
                code()
                        {
                        }
                code(int a)
                        {
                                id=a;
                        }
                code(code &x)  //Copy constructor
                        {
                                id=x.id;
                        }
                void display()
                        {
                                cout<<id;
                        }
        };
int main()
```

```
{
        code A(100);  //object created & initialized
        code B(A);  //copy constructor called
        code C=A; // copy constructor called
        code D;  // D is created & not initialized
        cout<<"\n id of A:";
        A.display();
        cout<<"\n id of B";
        B.display();
        cout<<"\n id of C";
        C.display();
        cout<<"\n id of D";
        D.display();
        return 0;
}
```

Output:

```
Id of A:100
Id of B:100
Id of C:100
Id of D:100
```

Destructor:

- it is used to destroy the objects that have been created by a constructor.
- Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.
- For ex. The destructor for the class integer can be defined as shown below,

```
~integer()
        {
        }
```

- A destructor never takes any argument nor does it return value.
- It will be invoked implicitly by the compiler upon exist from the program(or block or function etc).
- The ex. Below illustrates that the destructor has been invoked implicitly by the compiler.

```
#include<iostream>
using namespace std;
int count=0;
class test
        {
        public:
                test()

                        {

                        count++;
                        cout<<"\n Constructor Msg:object number";

                        }

                ~test()

                        {
```

```
                    cout<<"\n Destructor Msg:object
                    number"<<count<<"destroyed";

                    count—;

                    }

        };

int main()

        {

        cout<<"Inside the main block.";

        cout<<"\n\n Creating first object T1..";

        test T1;

        {

                cout<<"\n \n Inside Block..";

                cout<<"\n \n Creating two more objects T2  & T3..";

                test T2,T3;

                cout<<"\n\nLeaving Block1..";

        }

                cout<<"\n \n Back inside the main block";

                return 0;

        }
```

Output:

```
Inside the main block

Creating first object T1
```

```
Constructor Msg:Object number 1 created

Inside block 1

Creating two more objects T2 and T3

Constructor Msg:Object number 2 created

Constructor Msg:Object number 3 created

Leaving Block1..

Destructor Msg:Object number 3 destroyed

Destructor Msg:Object number 2 destroyed

Back inside the main block

Destructor Msg:object number 1 destroyed
```

Dynamic creation & destruction of objects:

- C++ supports dynamic memory allocation & de-allocation.
- C++ allocates memory & initializes the member variables.
- A object can be created at run-time; such an object is called a dynamic object.
- The construction & destruction of the dynamic object is explicitly done by an programmer.
- The new & delete operator are used to allocate & de-allocate memory to such objects.
- A dynamic object can be created using the new operator as follows.
  ptr=new classname;
- The new operator returns the address of the object created & it is stored in the pointer ptr.
- The variable ptr is a pointer object of the same class.

- The member variable of the object can be accessed using the pointer -> (arrow) operator.
- A dynamic object can be destroyed using delete operator as follows.
  delete ptr;
- The delete operator destroys the object pointed by the pointer ptr.
- It also invokes the destructor of a class.
- The following program explains the creation & destruction of dynamic objects.

```
#include<iostream>
using namespace std;
class data
        {
                int x;
                int y;
        public:
                data()
                        {
                        cout<<"\n Constructor";
                        x=10;
                        y=50;
                        }

                ~data()

                        {

                        cout<<"\n Destructor ";

                        }

                void display()
```

```
                    {
                              cout<<"\n x="<<x;
                              cout<<"\n y="<<y;
                    }
          };
int main()
          {
                    data *d; // declaration of object pointer
                    d=new data; //dynamic object
                    d->display();
                    delete d; //deleting the dynamic object
                    return 0;
          }
```

Output:

```
Constructor
x=10
y=50
Destructor
```

Data Abstraction:

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.

- Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC seperates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.
- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.

- Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC seperates the implementation details from the external interface.

C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.

Abstraction using classes:

An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files:

An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

// program to calculate the power of a number.

```
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
  int n = 4;
  int power = 3;
  int result = pow(n,power); // pow(n,power) is the  power function
  cout << "Cube of n is : " <<result<< endl;
  return 0;
}
```

Output:

```
Cube of n is : 64
```

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

Let's see a simple example of data abstraction using classes.

```
#include <iostream>
using namespace std;
class Sum
        {
                private:
                         int x, y, z; // private variables
                public:
                        void add()
                                {
                                cout<<"Enter two numbers: ";
                                cin>>x>>y;
                                z= x+y;
                    cout<<"Sum of two number is: "<<z<<endl;
                                }
        };
int main()
        {
        Sum sm;
        sm.add();
        return 0;
        }
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

Advantages Of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

Data Abstraction & ADT

In this topic, you'll learn about abstract data types (ADTs), how they are used to support data abstraction, and how to create an ADT in C++. You'll also be able view working code examples.

- When you start your car, you don't need to know the intricate workings of the starter motor. All you need to do is turn the key to initiate the sequence.
- If successful, the engine will turn over. This real-world example highlights the programming concept of **data abstraction**, which allows a programmer to protect/hide the implementation of a process and only gives the keys to other functions or users.
- You only need to know enough about a given function to run it but don't need to know (or care) about how the internal code works.

To take this a step further, we can create entire data types.

- An **abstract data type** (or ADT) is a class that has a defined set of operations and values.
- In other words, you can create the starter motor as an entire abstract data type, protecting all of the inner code from the user. When the user wants to start the car, they can just execute the start() function.
- In programming, an ADT has the following features:

- An ADT doesn't state how data is organized, and
- It provides only what's needed to execute its operations

- An ADT is a prime example of how you can make full use of data abstraction and data hiding.
- This means that an abstract data type is a huge component of object-oriented programming methodologies: enforcing abstraction, allowing data and encapsulation.

One of the most common ADTs is the stack. Let's take a look at it in action.

Stack Data Type Example

- In this example, we're creating an object-oriented C++ program that creates an abstract data type in the form of a **stack**, in which items can be pushed onto the top and popped off the top.
- The program simulates your browsing history. Each page you visit is stored in a stack. As you click 'back' on the browser, you are removing/viewing the top element of the stack. As you go forward you are pushing them back onto the stack.

This example also shows a little different method for declaring functions from a class. Notice how they're declared inside the class but created outside of it. Stacks are very common, so let's see how they can work for us.

```
#include <iostream>

using namespace std;

//limit size of browser stack to 100

#define MAX_SIZE 100

class Stack {

 public:
  int top;

  int size[MAX_SIZE];

   //constructor

  Stack() {

   //no top yet

   top = -1;

  }

  //function declarations

  bool push(int page);

  int pop();

  bool is_empty();

};

//Functions created below

//are we empty?

bool Stack::is_empty() {

 return (top < 0);

}

//pop from stack (back button)

int Stack::pop() {

 if(top < 0) {

  cout << "Nothing here...";
  return 0;

 } else {

  int page = size[top--];

  return page;

 }

}

//push onto stack

bool Stack::push(int page) {

 if(top >= MAX_SIZE) {

  cout << "Can't anymore, Jim";

  return false;

 } else {

  size[++top] = page;

  return true;

 }

}

//Finally, create the main function to create a new instance of the
//stack, which you can see appearing here:

int main( ) {

 //new stack

 Stack pages;
 pages.push(5);

 pages.push(10);

 pages.push(15);

 pages.push(20);

 cout << " Page " << pages.pop() << " popped from stack " << endl;

 cout << " Page " << pages.pop() << " popped from stack " << endl;

 return 0;

}
```

Output:

```
Page 20 popped from stack

Page 15 popped from stack
```

Describing Objects Using ADTs

_ An abstract data type (ADT) is a set of objects and an associated set of operations on those objects

_ Common examples of ADTs:

 User-defined types: stacks, queues, trees, lists

- stack

Values: Stack elements

Operations: create, dispose, push, pop, is_empty, is_full, etc.

-queue

Values: Queue elements

Operations: create, dispose, enqueue, dequeue, is_empty, is_full, etc.

-tree search structure

Values: Tree elements.

Operations: insert, delete, find, size, traverse (in-order, post-order, pre-order,level-order), etc.

Information Hiding:

In above ADT Stack, we can achieve information hiding by declaring data members top & size[MAX_SIZE] as private.

UNIT-III
INHERITANCE

INHERITANCE

DEFINITION: Inheritance is a process in which a new class known as derived class is created from another class called base class.

DEFINING A CLASS HIERARCHY

In C++, inheritance is supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific. The process involves first defining a *base class*, which defines those qualities common to all objects to be derived from the base. The base class represents the most general description. The classes derived from the base are usually referred to as *derived classes*. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Inheritance is one of the cornerstones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.

C++'s support of inheritance is both rich and flexible.

DEFINING THE BASE AND DERIVED CLASSES

Base Class: A class that is inherited is referred to as a base class.
Derived Class: The class that does the inheriting is called the *derived class*. Further, a derived class can be used as a base class for another derived class.

ACCESS TO THE BASE CLASS MEMBERS

When a class inherits another, the members of the base class become members of the derived class.

General Form of Class Inheritance:

```
class derived-class-name : access base-class-name
 {
        // body of class
};
```

The access status of the base-class members inside the derived class is determined by *access*. The base-class access specifier must be public, **private**, or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier.

a. Public base class access specifier

When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

PROGRAM:

```
#include <iostream>
```

```
using namespace std;

class base
{
  int i, j;
public:
  void set (int a, int b)
  {
    i = a;
    j = b;
  }
  void show ()
  {
    cout << i << " " << j << "\n";
  }
};

class derived:public base
{
  int k;
public:
    derived (int x)
  {
    k = x;
  }
  void showk ()
  {
    cout << k << "\n";
  }
};

int main ()
{
  derived ob (3);
  ob.set (1, 2);          // access member of base
  ob.show ();                 // access member of base
  ob.showk ();                // uses member of derived class
  return 0;
}
```

OUTPUT:

```
1 2
3
```

b. Private base class access specifier

When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.

PROGRAM:

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

// This program won't compile.
class base
{
  int i, j;
public:
  void set (int a, int b)
  {
    i = a;
    j = b;
  }
  void show ()
  {
    cout << i << " " << j << "\n";
  }
};

// Public elements of base are private in derived.
class derived:private base
{
  int k;
public:
    derived (int x)
  {
    k = x;
  }
  void showk ()
  {
    cout << k << "\n";
  }
};

int main ()
{
  derived ob (3);
  ob.set (1, 2);              // error, can't access set()
  ob.show ();                           // error, can't access show()
  return 0;
}
```

OUTPUT:

```
main.cpp: In function ‘int main()’:
main.cpp:48:15: error: ‘void base::set(int, int)’ is inaccessible within this context
   ob.set (1, 2);  // error, can't access set()
              ^
main.cpp:18:8: note: declared here
   void set (int a, int b)
```

```
        ^~~
main.cpp:48:15: error: ‘base’ is not an accessible base of ‘derived’
   ob.set (1, 2);  // error, can't access set()
               ^
main.cpp:49:12: error: ‘void base::show()’ is inaccessible within this context
   ob.show ();   // error, can't access show()
            ^
main.cpp:23:8: note: declared here
   void show ()
        ^~~~
main.cpp:49:12: error: ‘base’ is not an accessible base of ‘derived’
   ob.show ();   // error, can't access show()
            ^
```

c. Protected base class access specifier

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, non-member elements of the program.

Access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

A private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class.

PROGRAM:

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

// This program won't compile.
class base
{
protected:
  int i, j;                              // private to base, but accessible by derived
public:
  void set (int a, int b)
  {
    i = a;
    j = b;
  }
  void show ()
  {
    cout << i << " " << j << "\n";
  }
};
```

```
class derived:public base
{
  int k;
public:
// derived may access base's i and j
  void setk ()
  {
    k = i * j;
  }
  void showk ()
  {
    cout << k << "\n";
  }
};

int main ()
{
  derived ob;
  ob.set (2, 3);                    // OK, known to derived
  ob.show ();                       // OK, known to derived
  ob.setk ();
  ob.showk ();
  return 0;
}
```

OUTPUT:

```
2 3
6
```

Protected Base-Class Inheritance

It is possible to inherit a base class as **protected.** When this is done, all public and protected members of the base class become protected members of the derived class.

PROGRAM:

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

// This program won't compile.
class base
{
protected:
  int i, j;                         // private to base, but accessible by derived
public:
  void setij (int a, int b)
  {
    i = a;
    j = b;
  }
  void showij ()
  {
    cout << i << " " << j << "\n";
  }
};

// Inherit base as protected.
class derived:protected base
{
  int k;
public:
// derived may access base's i and j and setij().
  void setk ()
  {
    setij (10, 12);
    k = i * j;
  }
// may access showij() here
  void showall ()
  {
    cout << k << " ";
    showij ();
  }
};

int main ()
{
  derived ob;
// ob.setij(2, 3); // illegal, setij() is
// protected member of derived
  ob.setk ();                    // OK, public member of derived
  ob.showall ();                 // OK, public member of derived
// ob.showij(); // illegal, showij() is protected
// member of derived
  return 0;
}
```

OUTPUT:

```
120 10 12
```

DIFFERENT FORMS OF INHERITANCE

The following are the different types of inheritance,

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance
6. Multipath Inheritance

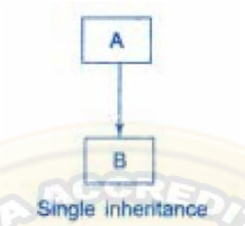
1. Single Inheritance

It is a process of creating new class called derived class from existing base class. The derived class inherits the member functions and variables of the existing base class.

General form:

```
class derived-class-name : access base-class-name
{
        // body of class
};
```

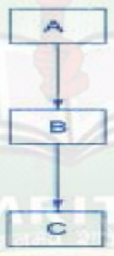
PROGRAM: ABOVE 3 PROGRAMS



Single inheritance

2. Multilevel Inheritance

When a derived class is used as a base class for another derived class, any protected member of the base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class.



Multilevel inheritance

PROGRAM:

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

class base
{
protected:
  int i, j;
public:
  void set (int a, int b)
  {
    i = a;
    j = b;
  }
  void show ()
  {
    cout << i << " " << j << "\n";
  }
};

// i and j inherited as protected.
class derived1:public base
```

```
{
  int k;
public:
  void setk ()
  {
    k = i * j;
  }                                   // legal
  void showk ()
  {
    cout << k << "\n";
  }
};

// i and j inherited indirectly through derived1.
class derived2:public derived1
{
  int m;
public:
  void setm ()
  {
    m = i - j;
  }                                   // legal
  void showm ()
  {
    cout << m << "\n";
  }
};

int main ()
{
  derived1 ob1;
  derived2 ob2;
  ob1.set (2, 3);
  ob1.show ();
  ob1.setk ();
  ob1.showk ();
  ob2.set (3, 4);
  ob2.show ();
  ob2.setk ();
  ob2.setm ();
  ob2.showk ();
  ob2.showm ();
  return 0;
}
```

OUTPUT:

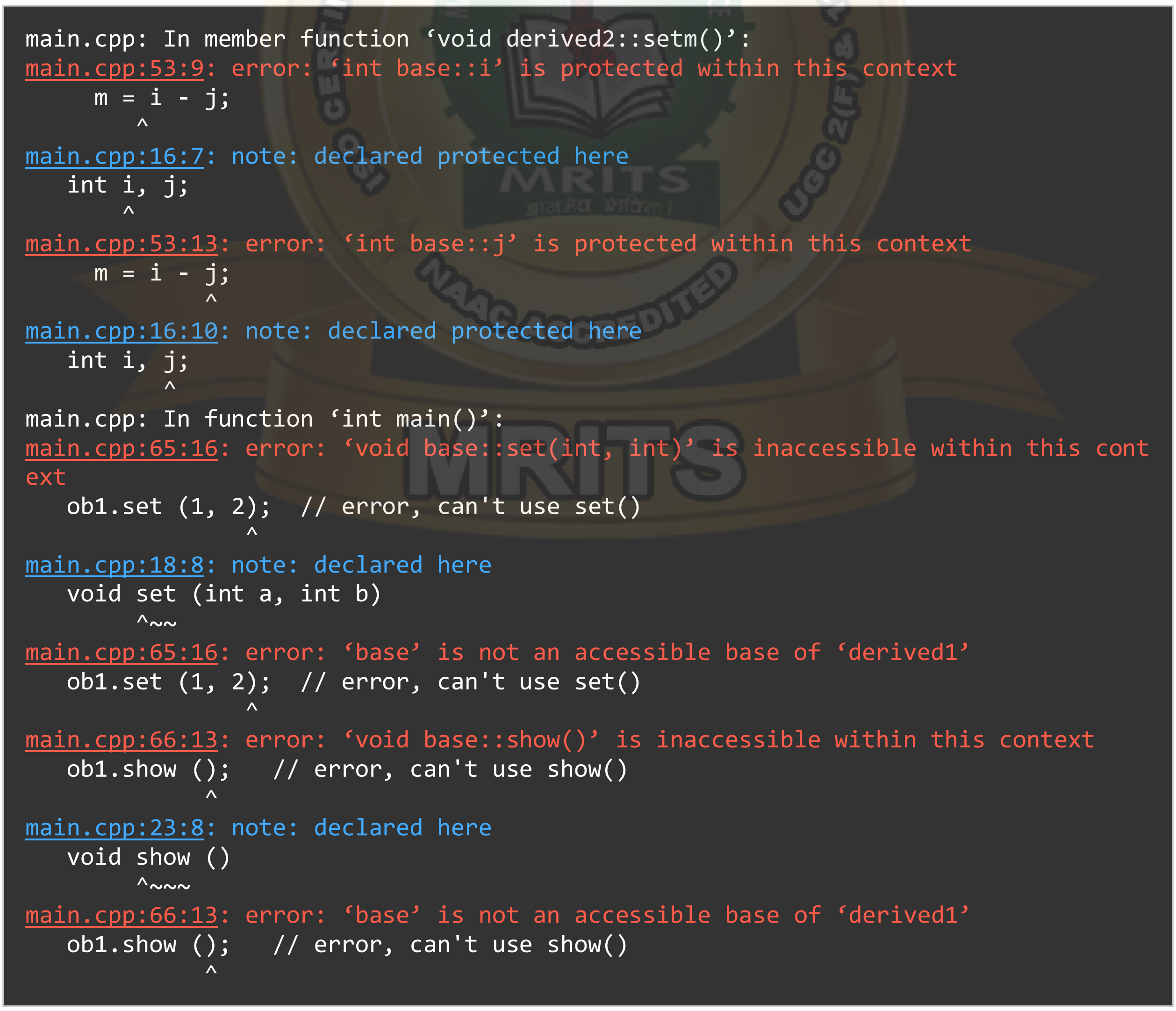
```
2 3
6
3 4
12
```

If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2**. (However, **i** and **j** would still be accessible by **derived1**.)

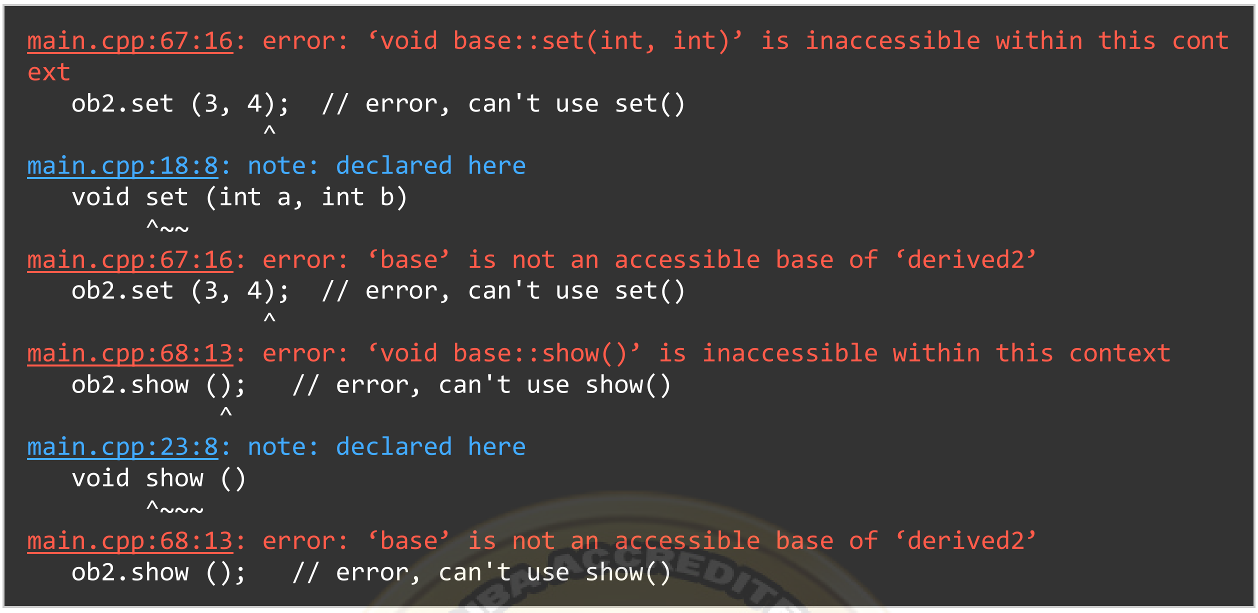
PROGRAM:

```
// This program won't compile.
#include <iostream>
using namespace std;

class base
{
protected:
  int i, j;
public:
  void set (int a, int b)
  {
    i = a;
    j = b;
  }
  void show ()
  {
    cout << i << " " << j << "\n";
  }
};

// Now, all elements of base are private in derived1.
class derived1:private base
{
  int k;
public:
// this is legal because i and j are private to derived1
  void setk ()
  {
    k = i * j;
  }                                    // OK
  void showk ()
  {
    cout << k << "\n";
  }
};

// Access to i, j, set(), and show() not inherited.
class derived2:public derived1
{
  int m;
public:
// illegal because i and j are private to derived1
  void setm ()
  {
   m = i - j;
  }                              // Error
  void showm ()
  {
   cout << m << "\n";
  }
};

int main ()
{
  derived1 ob1;
  derived2 ob2;
  ob1.set (1, 2);                // error, can't use set()
  ob1.show ();                   // error, can't use show()
  ob2.set (3, 4);                // error, can't use set()
  ob2.show ();                   // error, can't use show()
  return 0;
}
```

OUTPUT:



```
main.cpp: In member function ‘void derived2::setm()’:
main.cpp:53:9: error: ‘int base::i’ is protected within this context
     m = i - j;
         ^
main.cpp:16:7: note: declared protected here
   int i, j;
       ^
main.cpp:53:13: error: ‘int base::j’ is protected within this context
     m = i - j;
             ^
main.cpp:16:10: note: declared protected here
   int i, j;
          ^
main.cpp: In function ‘int main()’:
main.cpp:65:16: error: ‘void base::set(int, int)’ is inaccessible within this cont
ext
   ob1.set (1, 2);  // error, can't use set()
                ^
main.cpp:18:8: note: declared here
   void set (int a, int b)
        ^~~
main.cpp:65:16: error: ‘base’ is not an accessible base of ‘derived1’
   ob1.set (1, 2);  // error, can't use set()
                ^
main.cpp:66:13: error: ‘void base::show()’ is inaccessible within this context
   ob1.show ();   // error, can't use show()
             ^
main.cpp:23:8: note: declared here
   void show ()
        ^~~~
main.cpp:66:13: error: ‘base’ is not an accessible base of ‘derived1’
   ob1.show ();   // error, can't use show()
             ^
```



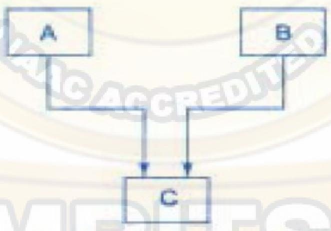
```
main.cpp:67:16: error: ‘void base::set(int, int)’ is inaccessible within this cont
ext
   ob2.set (3, 4);  // error, can't use set()
                ^
main.cpp:18:8: note: declared here
   void set (int a, int b)
        ^~~
main.cpp:67:16: error: ‘base’ is not an accessible base of ‘derived2’
   ob2.set (3, 4);  // error, can't use set()
                ^
main.cpp:68:13: error: ‘void base::show()’ is inaccessible within this context
   ob2.show ();   // error, can't use show()
             ^
main.cpp:23:8: note: declared here
   void show ()
        ^~~~
main.cpp:68:13: error: ‘base’ is not an accessible base of ‘derived2’
   ob2.show ();   // error, can't use show()
```

3. Multiple Inheritance

It is possible for a derived class to inherit two or more base classes. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new class.

General form:

class *derived-class-name : access base-class-name1, access base-class-name2*
{
// body of class
};



Multiple inheritance

PROGRAM:

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

class base1
{
protected:
  int x;
public:
  void showx ()
  {
    cout << x << "\n";
  }
};
```

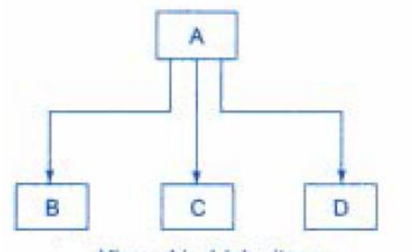
```
class base2
{
protected:
  int y;
public:
  void showy ()
  {
    cout << y << "\n";
  }
};

// Inherit multiple base classes.
class derived:public base1, public base2
{
public:
  void set (int i, int j)
  {
    x = i;
    y = j;
  }
};

int main ()
{
  derived ob;
  ob.set (10, 20);               // provided by derived
  ob.showx ();                   // from base1
  ob.showy ();                   // from base2
  return 0;
}
```

OUTPUT:

```
10
20
```

4. Hierarchical Inheritance

In this more than one class are derived from a single base class. This supports hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level.



Hierarchical inheritance

PROGRAM:

```
#include<iostream>
```

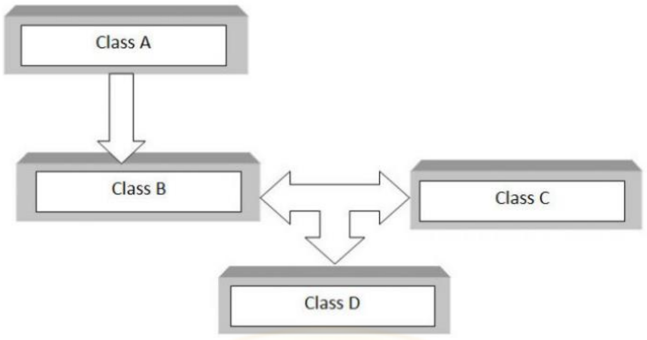
```
using namespace std;

class A                         //single base class
{
public:
  int x, y;
  void getdata ()
  {
    cout << "\nEnter value of x and y:\n";
    cin >> x >> y;
  }
};

class B:public A                //B is derived from class base
{
public:
  void product ()
  {
    cout << "\nProduct= " << x * y;
  }
};

class C:public A                //C is also derived from class base
{
public:
  void sum ()
  {
    cout << "\nSum= " << x + y;
  }
};

int main ()
{
  B obj1;                       //object of derived class B
  C obj2;                       //object of derived class C
  obj1.getdata ();
  obj1.product ();
  obj2.getdata ();
  obj2.sum ();
  return 0;
}                               //end of program
```

OUTPUT:

```
Enter value of x and y:
23 45
Product= 1035

Enter value of x and y:
34 56
Sum= 90
```

5. Hybrid Inheritance

It involves more than one form of any inheritance i.e. we apply two or more types of inheritance as one.



PROGRAM:

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

class A
{
public:
  int x;
};

class B:public A
{
public:
  B ()                                  //constructor to initialize x in base class A
  {
    x = 10;
  }
};

class C
{
public:
  int y;
    C ()                        //constructor to initialize y
  {
    y = 4;
  }
};

class D:public B, public C      //D is derived from class B and class C
{
public:
  void sum ()
  {
    cout << "Sum= " << x + y;
  }
};
```

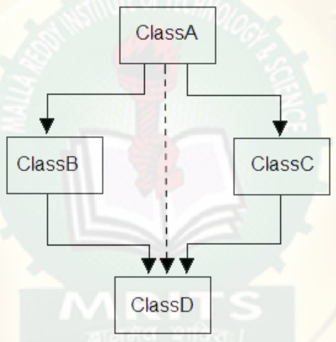
```
int
main ()
{
  D obj1;                    //object of derived class D
  obj1.sum ();
  return 0;
}                            //end of program
```

OUTPUT:

```
Sum= 14
```

6. Multipath Inheritance

It is a derivation of a class from other derived classes, which are derived from the same base class. This type of inheritance involves other inheritance like multiple, multilevel, hierarchical etc.



PROGRAM:

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

class person
{
    public:
    char name[100];
    int code;
    void input()
    {
        cout<<"\nEnter the name of the person : ";
        cin>>name;
        cout<<endl<<"Enter the code of the person : ";
        cin>>code;
    }
    void display()
    {
        cout<<endl<<"Name of the person : "<<name;
        cout<<endl<<"Code of the person : "<<code;
    }
};
```

```
class account:virtual public person
{
   public:
   float pay;
   void getpay()
   {
      cout<<endl<<"Enter the pay : ";
      cin>>pay;

   }
   void display()
   {
      cout<<endl<<"Pay : "<<pay;
   }
};

class admin:virtual public person
{
   public:
   int experience;
   void getexp()
   {
      cout<<endl<<"Enter the experience : ";
      cin>>experience;

   }
   void display()
   {
      cout<<endl<<"Experience : "<<experience;
   }
};

class master:public account,public admin
{
   public:
   char n[100];
   void gettotal()
   {
      cout<<endl<<"Enter the company name : ";
      cin>>n;
   }
   void display()
   {
      cout<<endl<<"Company name : "<<n;
   }
};

int main ()
{
  master m1;
```

```
    m1.input();
    m1.getpay();
    m1.getexp();
    m1.gettotal();
    cout<<"Displaying Information";
    m1.person::display();
    m1.account::display();
    m1.admin::display();
    m1.display();
  return 0;
}                                   //end of program
```

OUTPUT:

```
Enter the name of the person : asd
Enter the code of the person : 1234
Enter the pay : 20000
Enter the experience : 2
Enter the company name : cmr

Displaying Information
Name of the person : asd
Code of the person : 1234
Pay : 20000
Experience : 2
Company name : cmr
```

BASE AND DERIVED CLASS CONSTRUCTION, DESTRUCTORS

There are two important points relative to constructors and destructors when inheritance is involved.

1. When are base-class and derived-class constructors and destructors called?
2. How can parameters be passed to base-class constructors?

When Constructors and Destructors Are Executed

It is possible for a base class, a derived class, or both to contain constructors and/or destructors. It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence.

When an object of a derived class is created, the base class' constructor will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor.

Constructors are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first.

Destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed.

PROGRAM:

```
#include <iostream>
```

```
using namespace std;

class base
{
public:
  base ()
  {
    cout << "Constructing base\n";
  }
   ~base ()
  {
    cout << "Destructing base\n";
  }
};

class derived1:public base
{
public:
  derived1 ()
  {
    cout << "Constructing derived1\n";
  }
   ~derived1 ()
  {
    cout << "Destructing derived1\n";
  }
};

class derived2:public derived1
{
public:
  derived2 ()
  {
    cout << "Constructing derived2\n";
  }
   ~derived2 ()
  {
    cout << "Destructing derived2\n";
  }
};

int main ()
{
  derived2 ob;
// construct and destruct ob
  return 0;
}                                       //end of program
```

OUTPUT:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes.

PROGRAM:

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

class base1
{
public:
  base1 ()
  {
    cout << "Constructing base1\n";
  }
   ~base1 ()
  {
    cout << "Destructing base1\n";
  }
};

class base2
{
public:
  base2 ()
  {
    cout << "Constructing base2\n";
  }
   ~base2 ()
  {
    cout << "Destructing base2\n";
  }
};

class derived:public base1, public base2
{
public:
  derived ()
  {
    cout << "Constructing derived\n";
  }
   ~derived ()
  {
    cout << "Destructing derived\n";
  }
};
```

```
int main ()
{
  derived ob;
// construct and destruct ob
  return 0;
}                                      //end of program
```

OUTPUT:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

Constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list. Destructors are called in reverse order, right to left.

PROGRAM:

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

class base1
{
public:
  base1 ()
  {
    cout << "Constructing base1\n";
  }
   ~base1 ()
  {
    cout << "Destructing base1\n";
  }
};

class base2
{
public:
  base2 ()
  {
    cout << "Constructing base2\n";
  }
   ~base2 ()
  {
    cout << "Destructing base2\n";
  }
};

class derived: public base2, public base1
```

```
{
public:
  derived ()
  {
    cout << "Constructing derived\n";
  }
   ~derived ()
  {
    cout << "Destructing derived\n";
  }
};

int main ()
{
  derived ob;
// construct and destruct ob
  return 0;
}                                   //end of program
```

OUTPUT:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

Passing Parameters to Base-Class Constructors

When the derived class' constructor requires one or more parameters, you simply use the standard parameterized constructor syntax.

But to pass arguments to a constructor in a base class, we have to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors.

General form of expanded derived-class constructor declaration

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list),
                                // ...
                                baseN(arg-list)
{
        // body of derived constructor
}
```

base1 through *baseN* are the names of the base classes inherited by the derived class. A colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes.

PROGRAM:

```
#include <iostream>
```

```
using namespace std;

class base
{
protected:
  int i;
public:
    base (int x)
  {
    i = x;
    cout << "Constructing base\n";
  }
   ~base ()
  {
    cout << "Destructing base\n";
  }
};

class derived:public base
{
  int j;
public:
// derived uses x; y is passed along to base.
    derived (int x, int y):base (y)
  {
    j = x;
    cout << "Constructing derived\n";
  }
   ~derived ()
  {
    cout << "Destructing derived\n";
  }
  void show ()
  {
    cout << i << " " << j << "\n";
  }
};

int main ()
{
  derived ob (3, 4);
  ob.show ();                          // displays 4 3
  return 0;
}
```

OUTPUT:

```
Constructing base
Constructing derived
4 3
Destructing derived
```

Destructing base

The derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class.

PROGRAM:

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

class base1
{
protected:
  int i;
public:
    base1 (int x)
  {
    i = x;
    cout << "Constructing base1\n";
  }
   ~base1 ()
  {
    cout << "Destructing base1\n";
  }
};

class base2
{
protected:
  int k;
public:
    base2 (int x)
  {
    k = x;
    cout << "Constructing base2\n";
  }
   ~base2 ()
  {
    cout << "Destructing base1\n";
  }
};

class derived:public base1, public base2
{
  int j;
public:
    derived (int x, int y, int z):base1 (y), base2 (z)
  {
    j = x;
    cout << "Constructing derived\n";
  }
   ~derived ()
  {
    cout << "Destructing derived\n";
  }
  void show ()
  {
    cout << i << " " << j << " " << k << "\n";
  }
};

int main ()
{
  derived ob (3, 4, 5);
  ob.show ();                              // displays 4 3 5
  return 0;
}
```

OUTPUT:

```
Constructing base1
Constructing base2
Constructing derived
4 3 5
Destructing derived
Destructing base1
Destructing base1
```

Arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class requires it. In this situation, the arguments passed to the derived class are simply passed along to the base.

PROGRAM:

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

class base1
{
protected:
  int i;
public:
    base1 (int x)
  {
    i = x;
    cout << "Constructing base1\n";
  }
   ~base1 ()
  {
    cout << "Destructing base1\n";
  }
};

class base2
{
protected:
  int k;
public:
    base2 (int x)
  {
    k = x;
    cout << "Constructing base2\n";
  }
   ~base2 ()
  {
    cout << "Destructing base2\n";
  }
};

class derived:public base1, public base2
{
public:
/* Derived constructor uses no parameter, but still must be declared as taking them to pass
them along to base classes. */

  derived (int x, int y):base1 (x), base2 (y)
  {
    cout << "Constructing derived\n";
  }
   ~derived ()
  {
    cout << "Destructing derived\n";
  }
  void show ()
  {
    cout << i << " " << k << "\n";
  }
};

int main ()
{
  derived ob (3, 4);
  ob.show ();                          // displays 3 4
  return 0;
}
```

OUTPUT:

```
Constructing base1
Constructing base2
Constructing derived
3 4
```

```
Destructing derived
Destructing base2
Destructing base1
```

A derived class' constructor is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base
 {
        int j;
        public:
        // derived uses both x and y and then passes them to base.
        derived(int x, int y): base(x, y)
        { j = x*y; cout << "Constructing derived\n"; }
```

One final point to keep in mind when passing arguments to base-class constructors: The argument can consist of any expression valid at the time. This includes function calls and variables. This is in keeping with the fact that C++ allows dynamic initialization.

VIRTUAL BASE CLASS

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited.

PROGRAM:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;

class base
{
public:
  int i;
};

// derived1 inherits base.
class derived1:public base
{
public:
  int j;
};

// derived2 inherits base.
class derived2:public base
{
public:
  int k;
};

/* derived3 inherits both derived1 and derived2. This means that there are two copies of base
in derived3! */
```

```
class derived3:public derived1, public derived2
{
public:
  int sum;
};

int main ()
{
  derived3 ob;
  ob.i = 10;                          // this is ambiguous, which i???
  ob.j = 20;
  ob.k = 30;
// i ambiguous here, too
  ob.sum = ob.i + ob.j + ob.k;
// also ambiguous, which i?
  cout << ob.i << " ";
  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;
  return 0;
}
```

OUTPUT:

```
main.cpp: In function ‘int main()’:
main.cpp:40:6: error: request for member ‘i’ is ambiguous
   ob.i = 10;   // this is ambiguous, which i???
^
main.cpp:14:7: note: candidates are: int base::i
   int i;
       ^
main.cpp:14:7: note:                 int base::i
main.cpp:44:15: error: request for member ‘i’ is ambiguous
   ob.sum = ob.i + ob.j + ob.k;
               ^
main.cpp:14:7: note: candidates are: int base::i
   int i;
       ^
main.cpp:14:7: note:                 int base::i
main.cpp:46:14: error: request for member ‘i’ is ambiguous
   cout << ob.i << " ";
              ^
main.cpp:14:7: note: candidates are: int base::i
   int i;
       ^
main.cpp:14:7: note:                 int base::i
```

Both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like

ob.i = 10;

which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s!, the statement is inherently ambiguous.

Solutions:

There are two ways to remedy the preceding program.

1. Apply the scope resolution operator to i and manually select one i.

PROGRAM:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;

class base {
public:
int i;
};

// derived1 inherits base.
class derived1 : public base {
public:
int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
int k;
};

/* derived3 inherits both derived1 and derived2. This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};

int main()
{
derived3 ob;
ob.derived1::i = 10; // scope resolved, use derived1's i
ob.j = 20;
ob.k = 30;
// scope resolved
ob.sum = ob.derived1::i + ob.j + ob.k;
// also resolved here
cout << ob.derived1::i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

OUTPUT:

```
10 20 30 60
```

This solution raises a deeper issue: What if only one copy of **base** is actually required? Preventing two copies from being included in **derived.**

2. Virtual Base Class

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited.

PROGRAM:

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;

class base
{
public:
  int i;
};

// derived1 inherits base as virtual.
class derived1:virtual public base
{
public:
  int j;
};

// derived2 inherits base as virtual.
class derived2:virtual public base
{
public:
  int k;
};

/* derived3 inherits both derived1 and derived2. This time, there is only one copy of base
class. */
class derived3:public derived1, public derived2
{
public:
  int sum;
};

int main ()
{
  derived3 ob;
  ob.i = 10;                        // now unambiguous
  ob.j = 20;
  ob.k = 30;
// unambiguous
  ob.sum = ob.i + ob.j + ob.k;
// unambiguous
  cout << ob.i << " ";
  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;
  return 0;
}
```

OUTPUT:

```
10 20 30 60
```

Difference between a normal base class and a virtual base class

The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once. If **virtual** base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

VIRTUAL FUNCTIONS AND POLYMORPHISM

POLYMORPHISM

Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.

Polymorphism refers to the ability to associate multiple meanings to one function name.

Polymorphism is supported by C++ both at compile time and at run time. Compile time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions

STATIC AND DYNAMIC BINDING

Early Binding or Static Binding

Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.)

Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators.

The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

Late Binding or Dynamic Binding

The opposite of early binding is *late binding*. Late binding refers to function calls that are not resolved until run time. Virtual functions are used to achieve late binding. As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time.

The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.

VIRTUAL FUNCTIONS

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.

DYNAMIC BINDING THROUGH VIRTUAL FUNCTIONS

When accessed "normally," virtual functions behave just like any other type of class member function. Virtual functions behaviour when accessed via a pointer is what which makes them important and capable of supporting run-time polymorphism

A base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.

PROGRAM:
```
#include <iostream>
using namespace std;

class base
{
public:
  virtual void vfunc ()
  {
    cout << "This is base's vfunc().\n";
  }
};

class derived1:public base
{
public:
  void vfunc ()
  {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2:public base
{
public:
  void vfunc ()
  {
    cout << "This is derived2's vfunc().\n";
  }
};

int main ()
{
  base *p, b;
  derived1 d1;
  derived2 d2;
// point to base
  p = &b;
  p->vfunc ();                    // access base's vfunc()
// point to derived1
  p = &d1;
  p->vfunc ();                    // access derived1's vfunc()
// point to derived2
```

```
  p = &d2;
  p->vfunc ();                    // access derived2's vfunc()
  return 0;
}
```

OUTPUT:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

Rules for Virtual Functions:

When virtual functions are created for implementing late binding, observe some basic rules that satisfy the compiler requirements.

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer points to any type of the derived object, the reverse is not true. i.e. we cannot use a pointer to a derived class to access an object of the base class type.
9. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.

VIRTUAL FUNCTION CALL MECHANISM

1. Normal manner

You can call a virtual function in the "normal" manner by using an object's name and the dot operator, it is only when access is through a base-class pointer (or reference) that run-time polymorphism is achieved.

For example, assuming the preceding example, this is syntactically valid:

d2.vfunc(); // calls derived2's vfunc()

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc( )**.

2. Calling a Virtual Function through a Base Class Reference

Polymorphic nature of a virtual function is also available when called through a base-class reference. A reference is an implicit pointer. Thus, a base-class reference can be used to refer to an object of the base class or any object derived from that base. When a virtual function is called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call.

The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter.

PROGRAM:

```
/* Here, a base class reference is used to access a virtual function. */
#include <iostream>
using namespace std;

class base
{
public:
  virtual void vfunc ()
  {
    cout << "This is base's vfunc().\n";
  }
};

class derived1:public base
{
public:
  void vfunc ()
  {
    cout << "This is derived1's vfunc().\n";
  }
};

class derived2:public base
{
public:
  void vfunc ()
  {
    cout << "This is derived2's vfunc().\n";
  }
};
// Use a base class reference parameter.
void f (base & r)
{
  r.vfunc ();
}

int main ()
{
  base b;
  derived1 d1;
  derived2 d2;
  f (b);                    // pass a base object to f()
  f (d1);                             // pass a derived1 object to f()
  f (d2);                             // pass a derived2 object to f()
  return 0;
}
```

OUTPUT:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

PURE VIRTUAL FUNCTIONS
Situations leading to Pure Virtual Functions

When a virtual function is not redefined by a derived class, the version defined in the base class will be used. However, in many situations there can be no meaningful definition of a virtual function within a base class. For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created. Further, in some situations you will want to ensure that all derived classes override a virtual function. To handle these two cases, C++ supports the pure virtual function.

Definition: A *pure virtual function* is a virtual function that has no definition within the base class.

General form:

virtual *type func-name(parameter-list*) = 0;

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

PROGRAM:

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

class number
{
protected:
  int val;
public:
  void setval (int i)
  {
    val = i;
  }
// show() is a pure virtual function
  virtual void show () = 0;
};

class hextype:public number
{
public:
  void show ()
  {
    cout << hex << val << "\n";
  }
};
class dectype:public number
{
public:
  void show ()
```

```
 {
   cout << val << "\n";
 }
};

class octtype:public number
{
public:
 void show ()
 {
   cout << oct << val << "\n";
 }
};

int main ()
{
 dectype d;
 hextype h;
 octtype o;
 d.setval (20);
 d.show ();                    // displays 20 - decimal
 h.setval (20);
 h.show ();                    // displays 14 - hexadecimal
 o.setval (20);
 o.show ();                    // displays 24 - octal
 return 0;
}
```

OUTPUT:

```
20
14
24
```

ABSTRACT CLASSES

A class that contains at least one pure virtual function is said to be *abstract*. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created. Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes.

Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

PROGRAM:

```
// C++ Program to Illustrate Abstract Class
#include <iostream>
using namespace std;

class Abstract
{
  int i, j;
public:
    virtual void setData (int i = 0, int j = 0) = 0;
  virtual void printData () = 0;
};

class Derived:public Abstract
{
  int i, j;
public:
  Derived (int ii = 0, int jj = 0):i (ii), j (jj)
  {
    cout << "Creating object " << endl;
  }
  void setData (int ii = 0, int jj = 0)
  {
    i = ii;
    j = jj;
  }
  void printData ()
  {
    cout << "Derived::i = " << i << endl << "Derived::j = " << j << endl;
  }
};

int main ()
{
  // Cannot create an instance of Abstract Class
  // Abstract a;
  Derived d;

  cout << "Current data " << endl;
  d.printData ();
  d.setData (10, 20);
  cout << "New data " << endl;
  d.printData ();
}
```

OUTPUT:

```
Creating object
Current data
Derived::i = 0
Derived::j = 0
New data
Derived::i = 10
Derived::j = 20
```

VIRTUAL DESTRUCTORS

Inheritance also lends itself to *virtual methods*, where implementation is provided by any specific subclasses. However, once an inheritance hierarchy is created, with memory allocations occurring at each stage in the hierarchy, it is necessary to be very careful about how objects are destroyed so that any memory leaks are avoided. In order to achieve this, we make use of a *virtual destructor*.

In simple terms, a virtual destructor ensures that when derived subclasses go *out of scope* or are deleted the order of destruction of each class in a hierarchy is carried out correctly. If the destruction order of the class objects is incorrect, in can lead to what is known as a *memory leak*. This is when memory is allocated by the C++ program but is never deallocated upon program termination. This is undesirable behaviour as the operating system has no mechanism to regain the lost memory (because it does not have any references to its location!). Since memory is a finite resource, if this leak persists over continued program usage, eventually there will be no available RAM (random access memory) to carry out other programs.

For instance, consider a pointer to a base class (such as PayOff) being assigned to a derived class object address via a reference. If the object that the pointer is pointing to is deleted, and the destructor is not set to virtual, then the base class destructor will be called instead of the derived class destructor. This can lead to a memory leak. Consider the following code:

```
class Base
{
public:
 Base();
 ~Base();
};

class Derived : public Base {
private:
  double val;
public:
 Derived(const double& _val);
 ~Derived();
}

void do_something() {
 Base* p = new Derived;
 // Derived destructor not called!!
 delete p;
}
```

What is happening here? Firstly, we create a base class called Base and a subclass called Derived. The destructors are NOT set to virtual. In our do_something() function, a pointer p to a Base class is created and a reference to a new Derived class is assigned to it. This is legal as Derived *is a* Base.

However, when we delete p the compiler only knows to call Base's destructor as the pointer is pointing to a Base class. The destructor associated with Derived is not called and val is not deallocated.

A memory leak occurs!

Now consider the amended code below. The virtual keyword has been added to the destructors:

```
class Base {
public:
 Base();
 virtual ~Base();
};

class Derived : public Base {
private:
  double val;
public:
 Derived(const double& _val);
 virtual ~Derived();
}

void do_something() {
 Base* p = new Derived;
 // Derived destructor is called
 delete p;
}
```

What happens now? Once do_something() is called, delete is invoked on the pointer p. At code execution-time, the correct destructor is looked up in an object known as a *vtable*. Hence the destructor associated with Derived will be called prior to a further call to the destructor associated with Base. This is the behaviour we originally desired. val will be correctly deallocated.
No memory leak this time!

UNIT - IV
C++ I/O

C++ supports two complete I/O systems.

- Inherits from C.
- Object-oriented I/O system defined by C++

NOTE: C++ programs can also use the C++-style header #include<cstdio>

.

I/O using C functions

It includes,

- Console I/O
- Streams
- Files

Console I/O

This can be divided into Unformatted and Formatted Console I/O.

Unformatted Console I/O

a. Reading and Writing Characters

The simplest of the console I/O functions are getchar( ) and putchar().

getchar( ) : It reads a character from the keyboard. It waits until a key is pressed and then returns its value. The key pressed is also automatically echoed to the screen.

Prototype: int getchar(void);

putchar( ): It prints or writes a character to the screen at the current cursor position.

Prototype: int putchar(int c);

Program:

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

int main()
{
    char ch;
    cout<<"\n Enter a character in lower case: ";
    ch = getchar();
    cout<<"\nThe entered character is ";

    putchar(ch);
    cout<<"\nCharacter in UPPER CASE: ";
    putchar(ch - 32);
    return 0;
}
```

Output:

```
    Enter a character in lower case: t
    The entered character is t
    Character in UPPER CASE: T
```

b. Alternatives to getchar( )

getchar( ) is not useful in an interactive environment. Two of the most common alternative functions, getch( ) and getche( )

getch( ) : It waits for a keypress, after which it returns immediately. It does not echo the character to the screen.
Prototype: int getch(void);

getche( ) : It is the same as getch( ), but the key is echoed.
Prototype: int getche(void);

c. Reading and Writing Strings

gets( ) : It reads a string of characters entered at the keyboard and places them at the address pointed to by its argument. You may type characters at the keyboard until you press ENTER. The carriage return does not become part of the string; instead, a null terminator is placed at the end and gets( ) returns.
Prototype: char *gets(char *str);

Problem with gets( ) : It performs no boundary checks on the array that is receiving input. Thus, it is possible for the user to enter more characters than the array can hold. One alternative is the fgets( ) function.

puts( ): It writes its string argument to the screen followed by a newline.
Prototype: int puts(const char *str);

Program:
```
include <iostream>
#include<cstdio>
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    gets(str);
    cout << "You entered: " << str;

    char str1[] = "Happy New Year";
    char str2[] = "Happy Birthday";

    puts(str1);
    /*  Printed on new line since '/n' is added */
    puts(str2);

    return 0;
}
```

Output:
```
main.cpp:18:5: warning: ‘char* gets(char*)’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.cpp:18:13: warning: ‘char* gets(char*)’ is deprecated [-Wdeprecated-declarations]
/usr/include/stdio.h:638:14: note: declared here
main.cpp:(.text+0x31): warning: the `gets' function is dangerous and should not be used.
Enter a string: rt
You entered: rtHappy New Year
Happy Birthday
```

Function	Operation
getchar( )	Reads a character from the keyboard; waits for carriage return.
getche( )	Reads a character with echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
getch( )	Reads a character without echo; does not wait for carriage return; not defined by Standard C/C++, but a common extension.
putchar( )	Writes a character to the screen.
gets( )	Reads a string from the keyboard.
puts( )	Writes a string to the screen.

Formatted Console I/O

The functions printf( ) and scanf( ) perform formatted output and input. Both functions can operate on any of the built-in data types, including characters, strings, and numbers.

printf( ): It writes data to the console.
Prototype: int printf(const char *control_string, ...);

The control_string consists of two types of items. The first type is composed of characters that will be printed on the screen. The second type contains format specifiers that define the way the subsequent arguments are displayed. A format specifier begins with a percent sign and is followed by the format code.

Code	Format
%c	Character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer.
%%	Prints a % sign

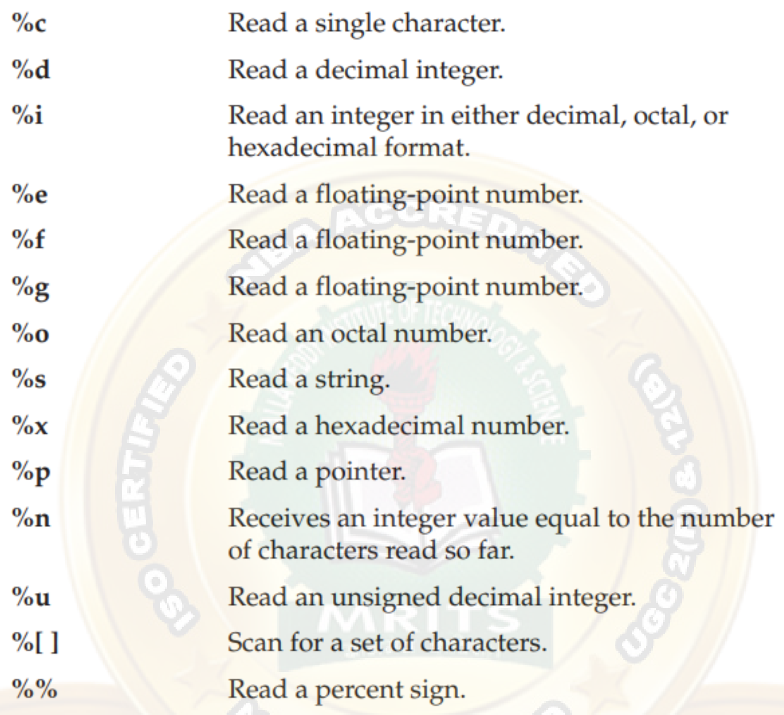
scanf( ): Its reads data from the keyboard.
Prototype: int scanf(const char *control_string, ...);

The control_string determines how values are read into the variables pointed to in the argument list. The control string consists of three classifications of characters:

- Format specifiers
- White-space characters
- Non-white-space characters

Format specifiers

The input format specifiers are preceded by a % sign and tell scanf( ) what type of data is to be read next.



%c	Read a single character.
%d	Read a decimal integer.
%i	Read an integer in either decimal, octal, or hexadecimal format.
%e	Read a floating-point number.
%f	Read a floating-point number.
%g	Read a floating-point number.
%o	Read an octal number.
%s	Read a string.
%x	Read a hexadecimal number.
%p	Read a pointer.
%n	Receives an integer value equal to the number of characters read so far.
%u	Read an unsigned decimal integer.
%[ ]	Scan for a set of characters.
%%	Read a percent sign.

White-space characters

A white-space character in the control string causes scanf( ) to skip over one or more leading white-space characters in the input stream. A white-space character is a space, a tab, vertical tab, form feed, or a newline.

Non-white-space characters

A non-white-space character in the control string causes scanf( ) to read and discard matching characters in the input stream. For example, "%d,%d" causes scanf( ) to read an integer, read and discard a comma, and then read another integer

Program:

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

int main()
{
    int f;
    printf("  ff");
    scanf("%d",f);
    printf(f);
    return 0;
}
```

Output: ff f

Streams

The C file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. The file system transforms each into a logical device called a stream. There are two types of streams: text and binary.

- **Text Streams**

A text stream is a sequence of characters. Standard C allows (but does not require) a text stream to be organized into lines terminated by a newline character.

Certain character translations may occur as required by the host environment. For example, a newline may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those on the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as those on the external device.

- **Binary Streams**

A binary stream is a sequence of bytes that have a one-to-one correspondence to those in the external device that is, no character translations occur. Also, the number of bytes written (or read) is the same as the number on the external device.

The Standard Streams

As it relates to the C file system, when a program starts execution, three streams are opened automatically. They are stdin (standard input), stdout (standard output), and stderr (standard error).

Using freopen( ) to Redirect the Standard Streams

You can redirect the standard streams by using the freopen( ) function. This function associates an existing stream with a new file. Thus, you can use it to associate a standard stream with a new file.

Prototype: FILE *freopen(const char *filename, const char *mode, FILE *stream);

filename is a pointer to the filename you wish associated with the stream pointed to by stream. The file is opened using the value of mode, which may have the same values as those used with fopen( ). freopen( ) returns stream if successful or NULL on failure.

Program

```
#include <cstdio>
#include <cstdlib>

int main()
{
   FILE* fp = fopen("test1.txt","w");
   fprintf(fp,"%s","This is written to test1.txt");

   if (freopen("test2.txt","w",fp))
   fprintf(fp,"%s","This is written to test2.txt");
   else
   {
   printf("freopen failed");
   exit(1);
}

   fclose(fp);
   return 0;
}
```

Output:

Files

In C/C++, a *file* may be anything from a disk file to a terminal or printer. Each stream that is associated with a file has a file control structure of type **FILE**.

➢ **File System Basics**

The C file system is composed of several interrelated functions. C++ programs may also use the C++-style header **<cstdio>**.

➢ **The File Pointer**

The file pointer is the common thread that unites the C I/O system. A *file pointer* is a pointer to a structure of type **FILE**. It points to information that defines various things about the file, including its name, status, and the current position of the file.

In order to read or write files, your program needs to use file pointers

Prototype : FILE *fp;

File Operations

- **fopen( )** : It opens a stream for use and links a file with that stream. Then it returns the file pointer associated with that file.

Prototype: FILE *fopen(const char **filename*, const char **mode*);

where *filename* is a pointer to a string of characters that make up a valid filename and may include a path specification.

The legal values for *mode are,*

Mode	**Meaning**
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Create a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append or create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append or create a binary file for read/write.

- **fclose( ):** It closes a stream that was opened by a call to **fopen( )**.

Prototype: int fclose(FILE **fp*);

where *fp* is the file pointer returned by the call to **fopen( )**. The function returns **EOF** if an error occurs.

Program:

```
#include <cstdio>
#include <cstring>
```

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

int main()
{
    int c;
    FILE *fp;
    fp = fopen("file.txt", "w+r");
    char str[20] = "Hello World!";
    if (fp)
    {
        for(int i=0; i<strlen(str); i++)
        putc(str[i],fp);
    }
    fclose(fp);
}
```

Output:

```
        Hello World!
```

- **putc( )** and **fputc( )**

These two equivalent functions writes characters to a file that was previously opened for writing using the **fopen( )** function.

Prototype : int putc(int *ch*, FILE **fp*);

where *fp* is the file pointer returned by **fopen( )** and *ch* is the character to be output. The file pointer tells **putc( )** which file to write to.

If a **putc( )** operation is successful, it returns the character written. Otherwise, it returns **EOF**.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    char str[] = "Testing putc() function";
    FILE *fp;

    fp = fopen("file.txt","w");

    if (fp)
    {
      for(int i=0; i<strlen(str); i++)
      {
      putc(str[i],fp);
      }

      for(int i=0; i<strlen(str); i++)
      {
      fputc(str[i],fp);
      }
    }
    else
      perror("File opening failed");
```

```
        fclose(fp);
    return 0;
}
```

Output:

```
Testing putc() functionTesting putc() function
```

- **getc( )** and **fgetc( )**
  These two equivalent functions reads characters from a file opened in read mode by **fopen( )**.

Prototype : int getc(FILE **fp*);

where *fp* is a file pointer of type **FILE** returned by **fopen( )**. The **getc( )** function returns an **EOF** when the end of the file has been reached. **getc( )** also returns **EOF** if an error occurs.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    int c;
    FILE *fp;

    fp = fopen("file.txt","r");

    if (fp)
    {
        while(feof(fp) == 0)
        {
        c = getc(fp);
        putchar(c);
        }
        while(feof(fp) == 0)
        {
        c = fgetc(fp);
        putchar(c);
        }
    }
    else
        perror("File opening failed");
        fclose(fp);
    return 0;
}
```

Output:

```
Testing putc() functionTesting putc() function
```

- **feof( )**: It determines when the end of the file has been encountered.

Prototype: int feof(FILE **fp*);

feof( ) returns true if the end of the file has been reached; otherwise, it returns 0.

- **fputs( ) and fgets( )**

These functions work just like **putc( )** and **getc( )**, but instead of reading or writing a single character, they read or write strings.

Prototypes:

int fputs(const char **str*, FILE **fp*);
char *fgets(char **str*, int *length*, FILE **fp*);

The **fputs( )** function writes the string pointed to by *str* to the specified stream. It returns **EOF** if an error occurs.

The **fgets( )** function reads a string from the specified stream until either a newline character is read or *length* −1 characters have been read.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    int count = 10;
    char str[10];
    FILE *fp;

    fp = fopen("file.txt","w+");
    fputs("An example file\n", fp);
    fputs("Filename is file.txt\n", fp);

    rewind(fp);

    while(feof(fp) == 0)
    {
        fgets(str,count,fp);
        cout << str << endl;
     }

    fclose(fp);
    return 0;
}
```

Output:

```
An exampl
e file

Filename
is file.t
xt

xt
```

- **rewind( )**

The **rewind( )** function resets the file position indicator to the beginning of the file specified as its argument. That is, it "rewinds" the file.

Prototype: void rewind(FILE **fp*);

where *fp* is a valid file pointer.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    int c;
    FILE *fp;
    fp = fopen("file.txt", "r+w");
    if (fp)
    {
      while ((c = getc(fp)) != EOF)
      putchar(c);

      rewind(fp);
      putchar('\n');

      while ((c = getc(fp)) != EOF)
      putchar(c);
    }
    fclose(fp);
    return 0;
}
```

Output:

```
welcome
welcome
```

- **ferror( )**

  The **ferror( )** function determines whether a file operation has produced an error.

Prototype: int ferror(FILE **fp*);

where *fp* is a valid file pointer. It returns true if an error has occurred during the last file operation; otherwise, it returns false.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
    int ch;
    FILE* fp;
    fp = fopen("file.txt","w");

    if(fp)
    {
      ch = getc(fp);
      if (ferror(fp))
      cout << "Can't read from file";
    }
    fclose (fp);
    return 0;
}
```

Output:

```
Can't read from file
```

- **remove( )**
  The **remove( )** function erases the specified file.

Prototype: int remove(const char **filename*);
It returns zero if successful; otherwise, it returns a nonzero value.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    char filename[] =  "file.txt";

    /*Deletes the file if exists */
    if (remove(filename) != 0)
    perror("File deletion failed");
    else
    cout << "File deleted successfully";

    return 0;
}
```

Output:

```
File deleted successfully
```

- **fflush( )**
  If you wish to flush the contents of an output stream, use the **fflush( )** function.

Prototype: int fflush(FILE **fp*);

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    int x;
    char buffer[1024];
    setvbuf(stdout, buffer, _IOFBF, 1024);
    printf("Enter an integer - ");
    fflush(stdout);
    scanf("%d",&x);
    printf("You entered %d", x);
    return(0);
}
```

Output:
```
Enter an integer - 89
You entered 89
```

- **fread( )** and **fwrite( )**.
  These functions allow the reading and writing of blocks of any type of data.

Prototypes:

size_t fread(void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);
size_t fwrite(const void **buffer*, size_t *num_bytes*, size_t *count*, FILE **fp*);

For **fread( )**, *buffer* is a pointer to a region of memory that will receive the data from the file. For **fwrite( )**, *buffer* is a pointer to the information that will be written to the file. The value of *count* determines how many items are read or written, with each item being *num_bytes* bytes in length. Finally, *fp* is a file pointer to a previously opened stream.

The **fread( )** function returns the number of items read. This value may be less than *count* if the end of the file is reached or an error occurs. The **fwrite( )** function returns the number of items written. This value will equal *count* unless an error occurs.

Program:
```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{
    int retVal;
    FILE *fp;
    char buffer[] = "Writing to a file using fwrite.";
    fp = fopen("data.txt","wb");

    retVal = fwrite(buffer,sizeof(buffer),1,fp);
    cout << "fwrite returned " << retVal;

    return 0;
}
```

Output:
```
fwrite returned 1
```

Program:
```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main()
{
 FILE *fp;
 char buffer[100];

 fp = fopen("data.txt","rb");
 while(!feof(fp))
 {
```

```
 fread(buffer,sizeof(buffer),1,fp);
 cout << buffer;
 }

 return 0;
}
```

Output:

```
Writing to a file using fwrite.
```

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;
int main()
{

    FILE *fp;
    double d = 12.23;
    int i = 101;
    long l = 123023L;
    if((fp=fopen("test", "wb+"))==NULL)
    {
       printf("Cannot open file.\n");
       exit(1);
    }
    fwrite(&d, sizeof(double), 1, fp);
    fwrite(&i, sizeof(int), 1, fp);
    fwrite(&l, sizeof(long), 1, fp);
    rewind(fp);
    fread(&d, sizeof(double), 1, fp);
    fread(&i, sizeof(int), 1, fp);
    fread(&l, sizeof(long), 1, fp);
    printf("%f %d %ld", d, i, l);
    fclose(fp);
    return 0;
}
```

Output:

```
12.230000 101 123023
```

- **fseek( )**

You can perform random-access read and write operations using the C I/O system with the help of **fseek( )**, which sets the file position indicator.

Prototype : int fseek(FILE **fp*, long int *numbytes*, int *origin*);

Here, *fp* is a file pointer returned by a call to **fopen( )**. *numbytes* is the number of bytes from *origin* that will become the new current position, and *origin* is one of the following macros:

Origin Macro	Name
Beginning of file	SEEK_SET
Current position	SEEK_CUR
End of file	SEEK_END

Program:

```
#include <cstdio>
```

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

int main(int argc, char *argv[])
{
      FILE * pFile;
      pFile = fopen ( "example.txt" , "wb" );
      fputs ( "This is an apple." , pFile );
      fseek ( pFile , 9 , SEEK_SET );
      fputs ( " sam" , pFile );
      fclose ( pFile );
      return 0;
}
```

Output:
After this code is successfully executed, the file example.txt contains:

```
            This is a sample.
```

- **fprintf( ) and fscanf( )**
  These functions behave exactly like **printf( )** and **scanf( )** except that they operate with files.

Prototypes:

int fprintf(FILE **fp*, const char **control_string*,. . .);
int fscanf(FILE **fp*, const char **control_string*,. . .);

where *fp* is a file pointer returned by a call to **fopen( )**. **fprintf( )** and **fscanf( )** direct their I/O operations to the file pointed to by *fp*.

Program:

```
#include <cstdio>
#include <cstring>
#include<iostream>
using namespace std;

int main(int argc, char *argv[])
{
     FILE *fp;
     char name[50];
     int age;

     fp = fopen("example.txt","w");
     fprintf(fp, "%s %d", "Tim", 9);
     fclose(fp);
     fp = fopen("example.txt","r");
     fscanf(fp, "%s %d", name, &age);
     fclose(fp);
     printf("Hello %s, You are %d years old\n", name, age);
     return 0;
}
```

Output:

```
Hello Tim, You are 9 years old
```

Name	Function
fopen( )	Opens a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same as **putc( )**.
getc( )	Reads a character from a file.
fgetc( )	Same as **getc( )**.
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.
fprintf( )	Is to a file what **printf( )** is to the console.
fscanf( )	Is to a file what **scanf( )** is to the console.
feof( )	Returns true if end-of-file is reached.
ferror( )	Returns true if an error has occurred.
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

Object-oriented I/O system defined by C++
C++ Streams

The C++ I/O system operates through streams. A *stream* is a logical device that either produces or consumes information.

A stream is linked to a physical device by the I/O system. All streams behave the same, the same I/O functions can operate

Stream classes' hierarchy

Standard C++ provides support for its I/O system in **<iostream>** header, which gives a set of class hierarchies is defined that supports I/O operations.
The C++ I/O system is built upon two related but different class hierarchies.

- **basic_streambuf**

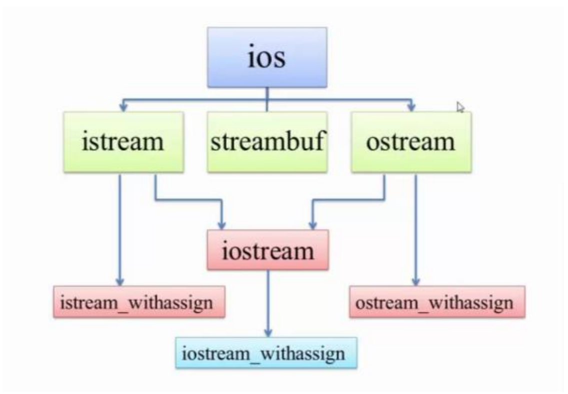
Low-level I/O class is called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly.

- **basic_ios**

The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O.

- **ios_base**

A base class for **basic_ios** is called **ios_base. basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively.



istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.
Class hierarchies for 8-bit characters and wide characters.

Class	Character-based Class	Wide-Character-based Class
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

C++'s Predefined Streams

When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

Streams **cin**, **cout**, and **cerr** correspond to C's **stdin**, **stdout**, and **stderr**. By default, the standard streams are used to communicate with the console.

Standard C++ also defines these four additional streams: **win**, **wout**, **werr**, and **wlog**. These are wide-character versions of the standard streams. Wide characters are of type **wchar_t** and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

Operator Overloading

Overloading << and >>

<< and the >> operators are overloaded in C++ to perform I/O. the << output operator is referred to as the *insertion operator* because it inserts characters into a stream. Likewise, the >> input operator is called the *extraction operator* because it extracts characters from a stream. The functions that overload the insertion and extraction operators are generally called *inserters* and *extractors*, respectively.

Creating Your Own Inserters

It is quite simple to create an inserter for a class that you create.

General form:

ostream &operator<<(ostream &*stream*, *class_type obj*)

```
{
    // body of inserter
    return stream;
}
```

The function returns a reference to a stream of type **ostream**. The first parameter to the function is a reference to the output stream. The second parameter is the object being inserted.

Within an inserter function, you may put any type of procedures or operations that you want. inserters cannot be members of the class for which they are defined seems to be a serious problem because they cannot access the private elements of a class. Solution is to Make the inserter a **friend** of the class

An inserter need not be limited to handling only text. An inserter can be used to output data in any form like CAD plotters, graphics images, dialog boxes etc.

Creating Your Own Extractors

Extractors are the complement of inserters.

General form

```
istream &operator>>(istream &stream, class_type &obj)
{
    // body of extractor
    return stream;
}
```

Extractors return a reference to a stream of type **istream**, which is an input stream. The first parameter must also be a reference to a stream of type **istream**. The second parameter must be a reference to an object of the class for which the extractor is overloaded. This is so the object can be modified by the input (extraction) operation.

Program:

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

class Box
{
    double height;
    double width;
    double vol ;

    public :
    friend istream & operator >> (istream &, Box &);
    friend ostream & operator << (ostream &, Box &);
};

istream & operator >> (istream &stream, Box &b)
{
    cout << "Enter Box Height: " ; stream >> b.height ;
    cout << "Enter Box Width : " ; stream >> b.width ;
    return (stream) ;
}
ostream & operator << (ostream &stream, Box &b)
{
    stream << endl << endl;
    stream << "Box Height : " << b.height << endl ;
    stream << "Box Width  : " << b.width << endl ;

    b.vol = b.height * b.width ;
    stream << "The Volume of Box : " << b.vol << endl;

    return(stream) ;
}

 int main()
 {
     Box b1;
     cin >> b1;
     cout << b1;
}
```

Output:

```
Enter Box Height: 1
Enter Box Width : 2

Box Height : 1
Box Width  : 2
The Volume of Box : 2
```

Creating Your Own Manipulator Functions

We can customize C++'s I/O system by creating your own manipulator functions. Custom manipulators are important for two main reasons.

- You can consolidate a sequence of several separate I/O operations into one manipulator.
- When you need to perform I/O operations on a nonstandard device. For example, you might use a manipulator to send control codes to a special type of printer or to an optical recognition system.

Types of manipulators

There are two basic types of manipulators:

- Those that operate on input streams
- Those that operate on output streams.

Apart from this , there is one more classification,

- Manipulators that take an argument

  The procedures necessary to create a parameterized manipulator vary widely from compiler to compiler, and even between two different versions of the same compiler. For this reason, you must consult the documentation to your compiler for instructions on creating parameterized manipulators
- Manipulators that don't.take an argument

  The creation of parameterless manipulators is straightforward and the same for all compilers.

General Form

```
ostream &manip-name(ostream &stream)
{
    // your code here
    return stream;
}
```

manip-name is the name of the manipulator. a reference to a stream of type **ostream** is returned. This is necessary if a manipulator is used as part of a larger I/O expression.

Using an output manipulator is particularly useful for sending special codes to a device. For example, a printer may be able to accept various codes that change the type size or font, or that position the print head in a special location. If these adjustments are going to be made frequently, they are perfect candidates for a manipulator.

General Form

```
istream &manip-name(istream &stream)
{
  // your code here
  return stream;
}
```

An input manipulator receives a reference to the stream for which it was invoked. This stream must be returned by the manipulator.

Program:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>
using namespace std;

// A simple output manipulator that sets the fill character to * and sets the field width to 10.
ostream &star_fill(ostream &stream)
{
      stream << setfill('*') << setw(10);
      return stream;
}

// A simple input manipulator that skips leading digits.
istream &skip_digits(istream &stream)
{
      char ch;/*w  w  w .  j  ava 2s.c  o  m*/
      do
      {
      ch = stream.get();
      } while(!stream.eof() && isdigit(ch));

      if(!stream.eof()) stream.unget();
      return stream;
}
int main()
{
      string str;
      // Demonstrate the custom output manipulator.
      cout << 512 << endl;
      cout << star_fill << 512 << endl;
      // Demonstrate the custom input manipulator.
      cout << "Enter some characters: ";
      cin >> skip_digits >> str;
      cout << "Contents of str: " << str;
       return 0;
}
```

Output:

```
512
*******512
Enter some characters: abc
```

```
Contents of str: abc
```

File streams and String streams

String streams

C++ provides a <sstream> header , which uses the same public interface to support I/O between a program and string object.

The string streams is based on **istringstream(subclass of istream)**, and **ostringstream(subclass of ostream) and bidirectional stringstream(subclass of iostream)**,

General Form:

typedef basic_istringstream<char>istringstream;
typedef basic_ostringstream<char>ostringstream;
Stream input can be used to validate input data,stream output can be used to format the output.

Ostringstream constructors

explicit ostringstream(ios::openmode mode=ios::out);//default with empty string
explicit ostringstream(const string &str, ios::openmode mode=ios::out);//with initial str
string str() const;//get contents
void str(const string &s)//set contents

Example:

```
ostringstream sout;
//write into string buffer
sout<<"apple"<<endl;
sout<<"orange"<<endl;
//get contents
cout<<sout.str()<<endl;
```

ostringstream is responsible for dynamic memory allocation and management.

istringstream constructors

explicit istringstream(ios::openmode mode=ios::in); //default with empty string
explicit istringstream(const string &str, ios::openmode mode=ios::in); //with initial str

Example:

```
istringstream sin("123    12.34    hello");
//read from buffer
int I;
double d;
string s;
sin>>i>>d>>s;
cout<<i<<","<<d<<","<<s<<endl;
```

stringstream constructors

explicit stringstream(ios::openmode mode = ios::in | ios::out);
explicit stringstream(const string &str,
ios::openmode mode = ios::in | ios::out);

Program:

```
// Demonstrate string streams.
#include <iostream>
#include <sstream>
using namespace std;
```

```
int main()
{
    stringstream s("This is initial string.");
    // get string
    string str = s.str();
    cout << str << endl;
    // output to string stream
    s << "Numbers: " << 10 << " " << 123.2;
    int i;
    double d;
    s >> str >> i >> d;
    cout << str << " " << i << " " << d;
    return 0;
}
```

Output:

```
This is initial string.
Numbers: 10 123.2
```

File streams
Formatted file streams

To perform file I/O, you must include the header **<fstream>** in your program. It defines several classes, including **ifstream**, **ofstream**, and **fstream**. These classes are derived from **istream**, **ostream**, and **iostream**, respectively. Remember, **istream**, **ostream**, and **iostream** are derived from **ios**, so **ifstream**, **ofstream**, and **fstream** also have access to all operations defined by **ios**

Opening and Closing a File
open()

In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream.
There are three types of streams:

Input

To create an input stream, you must declare the stream to be of class **ifstream**.

Output

To create an output stream, you must declare it as class **ofstream**.

Input/Output

Streams that will be performing both input and output operations must be declared as class **fstream**.

General form for creating streams

```
    ifstream in; // input
    ofstream out; // output
    fstream io; // input and output
```

Once you have created a stream, one way to associate it with a file is by using **open( )**. This function is a member of each of the three stream classes.

Prototype:

```
    void ifstream::open(const char *filename, ios::openmode mode = ios::in);
    void ofstream::open(const char *filename, ios::openmode mode = ios::out | ios::trunc);
    void fstream::open(const char *filename, ios::openmode mode = ios::in | ios::out);
```

filename is the name of the file; it can include a path specifier. The value of *mode* determines how the file is opened. It must be one or more of the following values

- **ios::app :** Including **ios::app** causes all output to that file to be appended to the end. This value can be used only with files capable of output.
- **ios::ate :** Including **ios::ate** causes a seek to the end of the file to occur when the file is opened.
- **ios::in :** The **ios::in** value specifies that the file is capable of input.
- **ios::out :** The **ios::out** value specifies that the file is capable of output.
- **ios::binary :** The **ios::binary** value causes a file to be opened in binary mode. By default, all files are opened in text mode.
- **ios::trunc :** The **ios::trunc** value causes the contents of a preexisting file by the same name to be destroyed, and the file is truncated to zero length.

Checking open() is successful or not

a. If **open( )** fails, the stream will evaluate to false when used in a Boolean expression. Therefore, before using a file, you should test to make sure that the open operation succeeded.

Example:

```
 if(!mystream) {
cout << "Cannot open file.\n";
// handle error
}
```

b. You can also check to see if you have successfully opened a file by using the **is_open( )** function, which is a member of **fstream**, **ifstream**, and **ofstream**.

Prototype:

bool is_open( );

It returns true if the stream is linked to an open file and false otherwise.

Example:

```
if(!mystream.is_open()) {
cout << "File is not open.\n";
// ...
```

close()

To close a file, use the member function **close( )**

Prototype: mystream.close();

The **close( )** function takes no parameters and returns no value.

Reading and Writing Text Files

It is very easy to read from or write to a text file. Simply use the << and >> operators the same way you do when performing console I/O, except that instead of using **cin** and **cout**, substitute a stream that is linked to a file.

Program:

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

int main()
{
    ifstream in("INVNTRY"); // input
    if(!in)
    {
        cout << "Cannot open INVENTORY file.\n";
        return 1;
    }
    char item[20];
    float cost;
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in >> item >> cost;
    cout << item << " " << cost << "\n";
    in.close();

    ofstream out;
    out.open("INVNTRY");// output, normal file
    if(!out)
    {
        cout << "Cannot open INVENTORY file.\n";
        return 1;
    }
    out << "Radios " << 39.95 << endl;
    out << "Toasters " << 19.95 << endl;
    out << "Mixers " << 24.80 << endl;
    out.close();
    return 0;
}
```

Output:

```
Radios 39.95
Toasters 19.95
Mixers 24.8
```

Unformatted and Binary I/O

There will be times when you need to store unformatted (raw) binary data, not text. When performing binary operations on a file , openshould use **ios::binary** mode specifier

- **get( )**
  get( ) will read a character

General form: istream &get(char &*ch*);

reads a single character and puts that value in *ch.* It returns a reference to the stream

Overloading of get()

The **get( )** function is overloaded in several different ways.

Prototypes:

istream &get(char *buf, streamsize num);

reads characters into the array pointed to by *buf* until either *num*-1

istream &get(char *buf, streamsize num, char delim);

reads characters into the array pointed to by *buf* until either *num*-1 characters have been read, the character specified by *delim* has been found, or the end of the file has been encountered.

int get( );

returns the next character from the stream

- **put( )**
  put( ) will write a character.

General form: ostream &put(char *ch*);
writes *ch* to the stream and returns a reference to the stream.

- **read( ) and write( )**
  Used to read and write blocks of binary data.

Prototypes:
istream &read(char **buf*, streamsize *num*);
reads *num* characters from the invoking stream and puts them in the buffer pointed to by *buf*.

ostream &write(const char **buf*, streamsize *num*);
writes *num* characters to the invoking stream from the buffer pointed to by *buf*.

- **getline( )**
  It also performs input. It is a member of each input stream class.

Prototypes:
istream &getline(char **buf*, streamsize *num*);
reads characters into the array pointed to by *buf* until either *num*-1

istream &getline(char **buf*, streamsize *num*, char *delim*);
reads characters into the array pointed to by *buf* until either *num*−1 characters have been read, the character specified by *delim* has been found

- **Detecting EOF**
  You can detect when the end of the file is reached by using the member function **eof( )**

Prototype: bool eof( );
It returns true when the end of the file has been reached; otherwise it returns false.

- **ignore( ) Function**
  You can use the **ignore( )** member function to read and discard characters from the input stream.

Prototype:
istream &ignore(streamsize *num*=1, int_type *delim*=EOF);
It reads and discards characters until either *num* characters have been ignored (1 by default) or the character specified by *delim* is encountered (**EOF** by default).

- **peek( )**
  You can obtain the next character in the input stream without removing it from that stream by using **peek( ).**

Prototype: int_type peek( );
It returns the next character in the stream or **EOF** if the end of the file is encountered.

- **putback( )**
  You can return the last character read from a stream to that stream by using **putback( ).**

Prototype : istream &putback(char *c*);
where *c* is the last character read.

- **flush( )**
  We can force the information to be physically written to disk before the buffer is full by calling **flush( )**.

Prototype: ostream &flush( );

Random Access

You perform random access by using the **seekg( )** and **seekp( ).**

- **seekg( )**
  The **seekg( )** function moves the associated file's current get pointer *offset* number

of characters from the specified *origin*, which must be one of these three values:

ios::beg	Beginning-of-file
ios::cur	Current location
ios::end	End-of-file

Prototype: istream &seekg(off_type *offset*, seekdir *origin*);

- **seekp( )**

The **seekp( )** function moves the associated file's current put pointer *offset* number of characters from the specified *origin*

Prototype: ostream &seekp(off_type *offset*, seekdir *origin*);

Obtaining the Current File Position

You can determine the current position of each file pointer by using these functions:

Prototypes: pos_type tellg( );
pos_type tellp( );

Here, **pos_type** is a type defined by **ios** that is capable of holding the largest value that either function can return.

You can use the values returned by **tellg( )** and **tellp( )** as arguments to the following forms of **seekg( )** and **seekp( )**, respectively.

istream &seekg(pos_type *pos);*
ostream &seekp(pos_type *pos*);

Error handling during files operations

We have been opening and using the files for reading and writing on the assumption that everything is fine with the files. This may not be true always true.

For instance, one of the following things may happen when dealing with the files,

1. A file which we are attempting to open for reading does not exists
2. The file name used for a new file may already exists
3. We may attempt an invalid operation such as reading past the EOF.
4. There may not be any space in the disk for storing more data.
5. We may use an invalid file name.
6. We may attempt to perform an operation when the file is not opened for that purpose.

We can handle these types of error situations in the following ways,

a. **I/O Status**

The C++ I/O system maintains status information about the outcome of each I/O operation. The current state of the I/O system is held in an object of type **iostate**, which is an enumeration defined by **ios** that includes the following members.

Name	Meaning
ios::goodbit	No error bits set
ios::eofbit	1 when end-of-file is encountered; 0 otherwise
ios::failbit	1 when a (possibly) nonfatal I/O error has occurred;0 otherwise
ios::badbit	1 when a fatal I/O error has occurred; 0 otherwise

There are two ways in which you can obtain I/O status information.

- Call the **rdstate( )** function.

Prototype: iostate rdstate( );

It returns the current status of the error flags.

- We can determine if an error has occurred is by using one or more of these functions:

bool bad( );	The **bad( )** function returns true if **badbit** is set.
bool eof( );	returns true when end of the file has reached
bool fail( );	The **fail( )** returns true if **failbit** is set.

bool good( );	The **good( )** function returns true if there are no errors. Otherwise, it returns false.

Clearing an Error

Once an error has occurred, it may need to be cleared before your program continues. To do this, use the **clear( )** function.

Prototype: void clear(iostate *flags*=ios::goodbit);

If *flags* is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set *flags* as you desire.

Formatted I/O.

The C++ I/O system allows you to format I/O operations. There are two related but conceptually different ways that you can format data.

1. Directly access members of the **ios** class.(flags and functions in ios class)
2. Special functions called *manipulators*

Formatting Using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted.

a. Flags

The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined.

- When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded.
- When the **left** flag is set, output is left justified.
- When **right** is set, output is right justified.
- When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character.
- **oct** flag causes output to be displayed in octal.
- Setting the **hex** flag causes output to be displayed in hexadecimal.
- To return output to decimal, set the **dec** flag.
- Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.
- By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase. When **uppercase** is set, these characters are displayed in uppercase.
- Setting **showpos** causes a leading plus sign to be displayed before positive values.
- Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.
- By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.
- When **unitbuf** is set, the buffer is flushed after each insertion operation.
- When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**.
- Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**.
- Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**.
- Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

Setting the Format Flags

To set a flag, use the **setf( )** function. This function is a member of **ios**.

Common form: fmtflags setf(fmtflags *flags*);

This function returns the previous settings of the format flags and turns on those flags specified by *flags*.

Example: stream.setf(ios::showpos);

stream is the stream you wish to affect.

NOTE: The format flags are defined within the **ios** class, you must access their values by using **ios** and the scope resolution operator.

Clearing Format Flags

The complement of **setf( )** is **unsetf( )**. This member function of **ios** is used to clear one or more format flags.

General form: void unsetf(fmtflags *flags*);

The flags specified by *flags* are cleared.

Overloaded Form of setf( )

There is an overloaded form of **setf( ) .**

General form: fmtflags setf(fmtflags *flags1*, fmtflags *flags2*);

In this version, only the flags specified by *flags2* are affected. the most common use of the two-parameter form of **setf( )** is when setting the number base, justification, and format flags.

Program:
```
#include <iostream>
using namespace std;

int main ()
{
    cout.setf (ios::uppercase | ios::scientific);
    cout << 100.12;                    // displays 1.001200E+02
    cout.unsetf (ios::uppercase);  // clear uppercase
    cout << " \n" << 100.12 << endl;       // displays 1.001200e+02
    //OVERLOADED FORM OF setf
    cout.setf (ios::showpoint | ios::showpos, ios::showpoint);
    cout << 100.0<<endl;                       // displays 100.000, not +100.000
    //TWO PARAMETER FORM  of setf
    cout.setf(ios::hex, ios::basefield);
    cout << 100; // this displays 64

    return 0;
}
```

Output:
```
1.001200E+02
1.001200e+02
1.000000e+02
64
```

Setting All Flags

The **flags( )** function has a second form that allows you to set all format flags associated with a stream.

Prototype: fmtflags flags(fmtflags *f*);

When you use this version, the bit pattern found in *f* is used to set the format flags associated with the stream. Thus, all format flags are affected. The function returns the previous settings.

Example: cout.flags(f);

Program:
```
#include <iostream>
using namespace std;

void showflags();
int main()
{
    // show default condition of format flags
    showflags();
    // showpos, showbase, oct, right are on, others off
    ios::fmtflags f = ios::showpos | ios::showbase | ios::oct | ios::right;
    cout.flags(f); // set all flags
    showflags();
    return 0;
}
// This function displays the status of the format flags.
void showflags()
{
    ios::fmtflags f;
    long i;
    f = cout.flags(); // get flag settings
    // check each flag
    for(i=0x4000; i; i = i >> 1)
    if(i & f) cout << "1 ";
    else cout << "0 ";
    cout << " \n";
}
```

Output:
```
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 1 0 1 0 1 1 0 0 0 0 0 0
```

b. Functions

There are three member functions defined by **ios.**

- **width( )**

By default, when a value is output, it occupies only as much space as the number of characters it takes to display it. However, you can specify a minimum field width by using the **width()** function.

Prototype; streamsize width(streamsize *w*);

Here, *w* becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output. If it isn't, the default field width is used. The **streamsize** type is defined as some form of integer by the compiler.

After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space, by default) to reach the field width. If the size of the value exceeds the minimum field width, the field will be overrun. No values are truncated.

- **precision( )**

When outputting floating-point values, you can determine the number of digits of precision by using the **precision( )** function.

Prototype: streamsize precision(streamsize *p*);

The precision is set to *p*, and the old value is returned. The default precision is 6. In some implementations, the precision must be set before each floating-point output. If it is not, then the default precision will be used.

- **fill( )**

By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill( )** function.

Prototype: char fill(char *ch*);

After a call to **fill( )**, *ch* becomes the new fill character, and the old one is returned.

Overloaded forms of width( ), precision( ), and fill( )

There are overloaded forms of **width( )**, **precision( )**, and **fill( )** that obtain but do not change the current setting. These forms are shown here:

```
char fill( );
streamsize width( );
streamsize precision( );
```

Program:

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


int main ()
{
    cout.precision (4);
    cout.width (10);
    cout << 10.12345 << "\n";   // displays 10.12
    cout.fill ('*');
    cout.width (10);
    cout << 10.12345 << "\n";   // displays *****10.12
  // field width applies to strings, too
    cout.width (10);
    cout << "Hi!" << "\n";         // displays *******Hi!
    cout.width (10);
    cout.setf (ios::left);  // left justify
    cout << 10.12345;              // displays 10.12*****
    return 0;
}
```

Output:

```
     10.12
*****10.12
*******Hi!
10.12*****
```

Using Manipulators to Format I/O

The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression. many of the I/O manipulators parallel member functions of the **ios** class.

Manipulator	Purpose	Input/Output
boolalpha	Turns on boolapha flag.	Input/Output
dec	Turns on dec flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on fixed flag.	Output
flush	Flush a stream.	Output
hex	Turns on hex flag.	Input/Output
internal	Turns on internal flag.	Output
left	Turns on left flag.	Output
nobooalpha	Turns off boolalpha flag.	Input/Output
noshowbase	Turns off showbase flag.	Output
noshowpoint	Turns off showpoint flag.	Output
noshowpos	Turns off showpos flag.	Output

noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

To access manipulators that take parameters (such as **setw( )**), you must include **<iomanip>** in your program.

Program:

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

int main ()
{
    cout << hex << 100 << endl;
    cout << setfill ('?') << setw (10) << 2343.0;
    return 0;
}
```

Output:

```
64
??????2343
```

Advantage

The main advantage of using manipulators instead of the **ios** member functions is that they often allow more compact code to be written. You can use the **setiosflags( )** manipulator to directly set the various format flags related to a stream.

Program:

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

int main ()
{
    cout << setiosflags (ios::showpos);
    cout << setiosflags (ios::showbase);
    cout << 123 << " " << hex << 123;
    return 0;
}
```

```
+123 0x7b
```

UNIT-V
EXCEPTION HANDILING

INDRODUCTION

Common types of Errors

The common types of errors are logic errors and syntactic errors.

Logic Errors: These occur due to poor understanding of the problem and solution procedure.
Examples: Assigning a value to the wrong variable, multiplying 2 numbers instead of adding them etc.

Syntactic Errors: These occur due to poor understanding of the language itself.
Examples: Spelling mistakes, missing out quotes or brackets or semicolon etc.

Apart from these two, one more type of errors is **Exception**.
Definition of Exception: Exceptions are run time errors or unusual conditions that a program may encounter while executing.
Examples: Division by zero, access to an array outside of its bounds, running out of memory or disk space.

Exception Handling

It is a C++ built in language feature that allows us to manage run time errors in an orderly fashion. Using exception handling, your program can automatically invoke an error-handling routine when an error occurs.

BENEFITS OF EXCEPTION HANDLING

1. **Automation:** it automates much of the error-handling code that previously had to be coded "by hand" in any large program.
2. **Separation of Error Handling code from Normal Code:** In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
3. **Functions can handle any exceptions they choose:** A function can throw many exceptions, but may choose to handle some of them.
4. **Grouping of Error Types:** In C++, both basic types and objects can be thrown as exception. We can group or categorize them according to types.
5. **Handles** the occurring of error and allows normal execution of the program.

EXCEPTION HANDLING FUNDAMENTALS
(THE TRY BLOCK, CATCHING AN EXCDEPTION, THROWING AN EXCCEPTION)

C++ exception handling is built upon three keywords: **try, catch, and throw**

- The program statements that you want to monitor for exceptions are contained in a **try block.**
- If an exception (i.e., an error) occurs within the try block, it is thrown using **throw**
- When an exception is thrown, it is caught by its corresponding **catch statement**, which processes the exception. There can be more than one catch statement associated with a try. Which catch statement is used is determined by the type of the exception.

General form of try and catch

```
 try {
          // try block
}
catch (type1 arg)
{
          // catch block
}
catch (type2 arg)
 {
          // catch block
 }
catch (type3 arg)
 {
          // catch block
 }
           . . .

catch (typeN arg)
{
          // catch block
 }
```

General form of the throw

```
          throw exception;
```

Program:

```
// A simple exception handling example.
#include <iostream>
using namespace std;
int main ()
{
  cout << "Start\n";
  try
  {                                    // start a try block
    cout << "Inside try block\n";
    throw 100;                         // throw an error
    cout << "This will not execute";
  }
  catch (int i)
  {                                    // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Abnormal Termination

The type of the exception must match the type specified in a catch statement. Usually, the code within a catch statement attempts to remedy an error by taking appropriate action. If the error can be fixed, execution will continue with the statements following the catch. However, often an error cannot be fixed i.e. throw an exception for which there is no applicable catch statement, an abnormal program termination may occur. Throwing an unhandled exception causes the standard library function **terminate ( )** to be invoked. By default, terminate ( ) calls **abort( )** to stop your program.

Program:

```
// This example will not work.
#include <iostream>
using namespace std;
int main ()
{
  cout << "Start\n";
  try
  {                                   // start a try block
    cout << "Inside try block\n";
    throw 100;                        // throw an error
    cout << "This will not execute";
  }
  catch (double i)
  {                                   // won't work for an int exception
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
terminate called after throwing an instance of 'int'
Aborted (core dumped)
```

Throwing an exception from outside the try block

An exception can be thrown from outside the try block as long as it is thrown by a function that is called from within try block.

Program:

```
/* Throwing an exception from a function outside the try block. */
#include <iostream>
using namespace std;
void
Xtest (int test)
{
  cout << "Inside Xtest, test is: " << test << "\n";
  if (test)
    throw test;
}

int main ()
{
  cout << "Start\n";
  try
  {                                    // start a try block
    cout << "Inside try block\n";
    Xtest (0);
    Xtest (1);
    Xtest (2);
  }
  catch (int i)
  {                                    // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

Localize a try/catch to a function

A try block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset.

Program:
```
#include <iostream>
using namespace std;
// Localize a try/catch to a function.
void
Xhandler (int test)
{
  try
  {
    if (test)
      throw test;
  }
  catch (int i)
  {
    cout << "Caught Exception #: " << i << '\n';
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (1);
  Xhandler (2);
  Xhandler (0);
  Xhandler (3);
  cout << "End";
  return 0;
}
```

Output:
```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End
```

Catch statements when no exception is thrown

The code associated with a catch statement will be executed only if it catches an exception. Otherwise, execution simply bypasses the catch altogether. When no exception is thrown, the catch statement does not execute.

Program:
```
#include <iostream>
using namespace std;
int main ()
{
  cout << "Start\n";
  try
  {                                    // start a try block
    cout << "Inside try block\n";
    cout << "Still inside try block\n";
  }
  catch (int i)
  {                                    // catch an error
    cout << "Caught an exception -- value is: ";
    cout << i << "\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Inside try block
Still inside try block
End
```

Using multiple catch Statements

There can be more than one catch associated with a try. However, each catch must catch a different type of exception. Which catch statement is used is determined by the type of the exception.

Program:

```
#include <iostream>
using namespace std;
// Different types of exceptions can be caught.
void
Xhandler (int test)
{
  try
  {
    if (test)
      throw test;
    else
      throw "Value is zero";
  }
  catch (int i)
  {
    cout << "Caught Exception #: " << i << '\n';
  }
  catch (const char *str)
  {
    cout << "Caught a string: ";
    cout << str << '\n';
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (1);
  Xhandler (2);
  Xhandler (0);
  Xhandler (3);
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

CATCHING ALL EXCEPTIONS

Using multiple catch statements we can write a separate catch statements for each type. But this is a complicated task. In these circumstances we want an exception handler to catch all exceptions instead of just a certain type.

General form:

```
catch(...)
{
// process all exceptions
}
```

Here, the ellipsis matches any type of data.

Program:

```
// This example catches all exceptions.
#include <iostream>
using namespace std;
void
Xhandler (int test)
{
  try
  {
    if (test == 0)
```

```
      throw test;             // throw int
    if (test == 1)
      throw 'a';              // throw char
    if (test == 2)
      throw 123.23;                  // throw double
  }
  catch ( ...)
  {                                  // catch all exceptions
    cout << "Caught One!\n";
  }
}

int main ()
{
  cout << "Start\n";
  Xhandler (0);
  Xhandler (1);
  Xhandler (2);
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught One!
Caught One!
Caught One!
End
```

One very good use for catch (...) is as the last catch of a cluster of catches which catch all exceptions that you don't want to handle explicitly. Also, by catching all exceptions, you prevent an unhandled exception from causing an abnormal program termination.

RETHROWING AN EXCEPTION

If you wish to rethrow an expression from within an exception handler, you may do so by calling throw, by itself, with no exception.

Reason: It allows multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a catch block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same catch statement. It will propagate outward to the next catch statement.

Program:

```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;
```

```
void Xhandler ()
{
  try
  {
    throw "hello";                    // throw a char *
  }
  catch (const char *)
  {                                   // catch a char *
    cout << "Caught char * inside Xhandler\n";
    throw;                            // rethrow char * out of function
  }
}

int main ()
{
  cout << "Start\n";
  try
  {
    Xhandler ();
  }
  catch (const char *)
  {
    cout << "Caught char * inside main\n";
  }
  cout << "End";
  return 0;
}
```

Output:

```
Start
Caught char * inside Xhandler
Caught char * inside main
End
```

EXCEPTION SPECICATION (RESTRICTING EXCEPTIONS)

You can restrict the type of exceptions that a function can throw outside of itself i.e. we are restricting a function to throw only certain specified exceptions .To accomplish these restrictions, we must add a throw clause to a function definition.

General form

ret-type func-name(*arg-list*) throw(*type-list*)
{
// ...
}

only those data types contained in the comma-separated *type-list* may be thrown by the function. If you don't want a function to be able to throw *any* exceptions, then use an empty list.

Attempting to throw an exception that is not supported by a function will cause the standard library function **unexpected ( )** to be called. By default, this causes **abort( )** to be called, which causes abnormal program termination.

Program:

```
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler (int test)
throw (int, char, double)
{
  if (test == 0)
    throw test;  // throw int
  if (test == 1)
    throw 'a';  // throw char
  if (test == 2)
    throw 123.23;    // throw double
}

int main ()
{
  cout << "start\n";
  try
  {
    Xhandler (0);        // also, try passing 1 and 2 to Xhandler()
  }
  catch (int i)
  {
    cout << "Caught an integer\n";
  }
  catch (char c)
  {
    cout << "Caught char\n";
  }
  catch (double d)
  {
    cout << "Caught double\n";
  }
  cout << "end";
  return 0;
}
```

Output:

```
start
Caught an integer
```

```
end
```

A function can be restricted only in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block *within* a function may throw any type of exception so long as it is caught *within* that function.The restriction applies only when throwing an exception outside of the function.

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
/* The following statements no longer work. Instead,
they will cause an abnormal program termination. */
if(test==0) throw test;
if(test==1) throw 'a';
if(test==2) throw 123.23;
}
```

STACK UNWINDING

Stack unwinding is a process of calling all destructors for all automatic objects constructed at run time when an exception is thrown. The objects are destroyed in the reverse order of their formation.

When an exception is thrown, the runtime mechanism first searches for an appropriate matching handler (catch) in the current scope. If no such handler exists, control is transferred from the current scope to a higher block in the calling chain or in outward manner. - Iteratively, it continues until an appropriate handler has been found. At this point, the stack has been unwound and all the local objects that were constructed on the path from a try block to a throw expression have been destroyed. - The run-time environment invokes destructors for all automatic objects constructed after execution entered the try block. This process of destroying automatic variables on the way to an exception handler is called stack unwinding.

Program:

```
#include <iostream>
#include <string>

using namespace std;

class MyClass
{
private:
  string name;
public:
  MyClass (string s):name (s)
  {
  }
   ~MyClass ()
  {
    cout << "Destroying " << name << endl;
  }
};

void fa ();
void fb ();
void fc ();
void fd ();

int main ()
{
  try
  {
    MyClass mainObj ("M");
    fa ();
    cout << "Mission accomplished!\n";
  }
  catch (const char *e)
  {
    cout << "exception: " << e << endl;
    cout << "Mission impossible!\n";
  }
  return 0;
}

void fa ()
{
  MyClass a ("A");
  fb ();
  cout << "return from fa()\n";
  return;
}

void fb ()
{
  MyClass b ("B");
  fc ();
  cout << "return from fb()\n";
  return;
}

void fc ()
{
  MyClass c ("C");
  fd ();
  cout << "return from fc()\n";
  return;

}

void fd ()
{
  MyClass d ("D");
  // throw "in fd(), something weird happened.";
  cout << "return from fd()\n";
  return;
}
```

Output:

```
return from fd()
Destroying D
return from fc()
Destroying C
return from fb()
Destroying B
return from fa()
Destroying A
Mission accomplished!
Destroying M
```

EXCEPTION OBJECT

The exception object holds the error information about the exception that had occurred. The information includes the type errors i.e. logic errors or run time error and state of the program when the error occurred.

An exception object is created as soon as exception occurs and it is passed to the corresponding catch block as a parameter. The catch block contains the code to catch the occurred exception.

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error.

General Form:

```
try
{
Throw exception object;
}
catch(Exception &exceptionobject)
{
```

```
…
}
```

When a throw expression is evaluated, an exception object is initialized from the value of the expression. The exception object which is thrown gets its type from the static type of the throw expression.

Inside a catch block, the name initialized with the caught exception object is initialized with this exception object

The exception object is available only in catch block. You cannot use the exception object outside the catch block.

Program:

```
#include <iostream>
#include <cstring>
using namespace std;
class MyException
{
public:
  char str_what[80];
  int what;
    MyException ()
  {
    *str_what = 0;
    what = 0;
  }
  MyException (char *s, int e)
  {
    strcpy (str_what, s);
    what = e;
  }
};

int main ()
{
  int i;
  try
  {
    cout << "Enter a positive number: ";
    cin >> i;
    if (i < 0)
      throw MyException ("Not Positive", i);
  }
  catch (MyException e)
  {                              // catch an error
    cout << e.str_what << ": ";
    cout << e.what << "\n";
  }
  return 0;
}
```

Output:

```
Enter a positive number: -1
Not Positive: -1
```