



Chapter 1

Basics and Functions

1. History:

C++ was written by [Bjarne Stroustrup](#) at Bell Labs during 1983-1985. C++ is an extension of C. Prior to 1983, Bjarne Stroustrup added features to C and formed what he called "**C with Classes**". He had combined the Simula's use of classes and object-oriented features with the power and efficiency of C. The term C++ was first used in 1983.

C++ was developed significantly after its first release.¹ In particular, "ARM C++" added exceptions and templates, and ISO C++ added RTTI, namespaces, and a standard library. C++ was designed for the UNIX system environment. With C++ programmers could improve the quality of code they produced and reusable code was easier to write.

C++ is an "object oriented" programming language created by Bjarne Stroustrup and released in 1985. It implements "data abstraction" using a concept called "classes", along with other features to allow object-oriented programming. Parts of the C++ program are easily reusable and extensible; existing code is easily modifiable without actually having to change the code. C++ adds a concept called "operator overloading" not seen in the earlier OOP languages and it makes the creation of libraries much cleaner.

C++ maintains aspects of the C programming language, yet has features which simplify memory management. Additionally, some of the features of C++ allow low-level access to memory but also contain high level features.

C++ could be considered a superset of C. C programs will run in C++ compilers. C uses structured programming concepts and techniques while C++ uses object oriented programming and classes which focus on data.

2. C++ Keywords:

*Keywords common to the
C and C++ programming
languages*

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

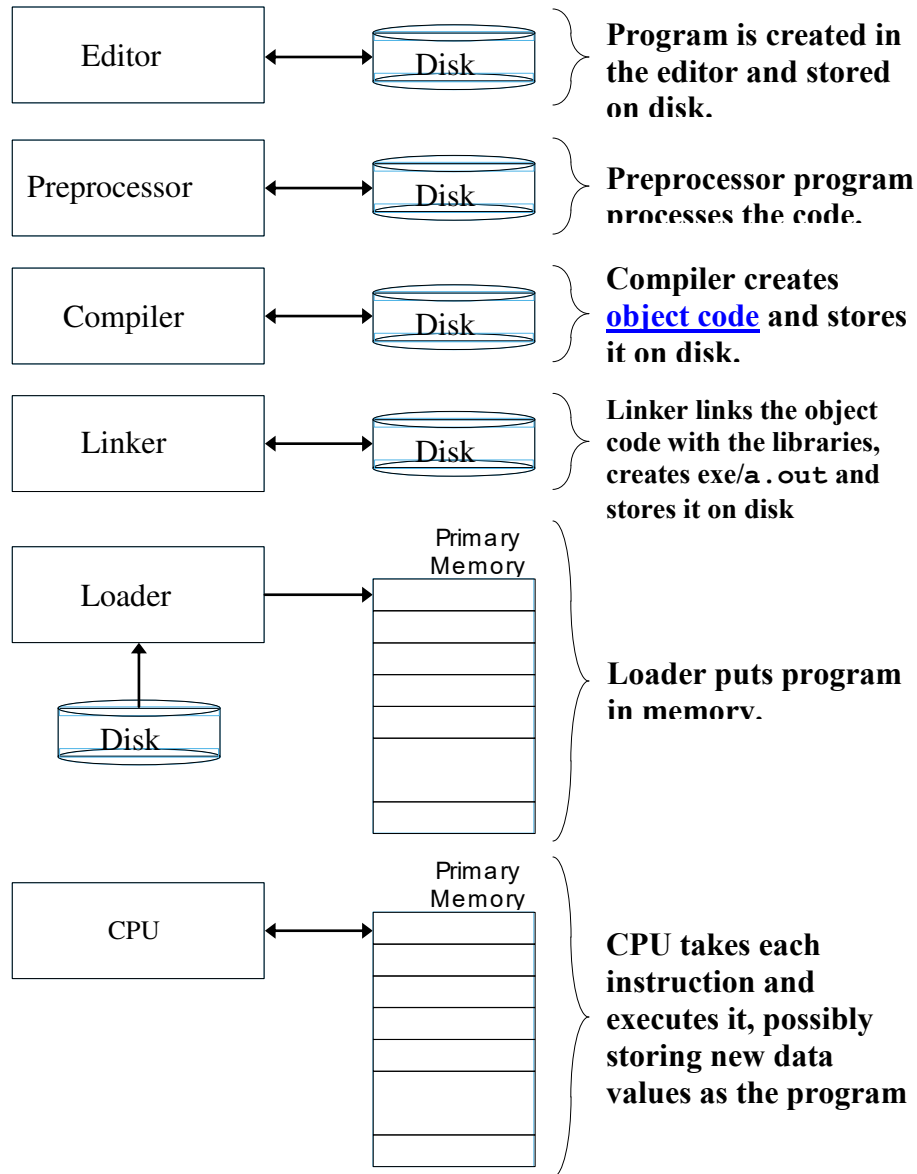
C++ only keywords

asm	bool	catch	class	const_cast
delete	dynamic_cast	explicit	false	friend
inline	mutable	namespace	new	operator
private	protected	public	reinterpret_cast	
static_cast	template	this	throw	true
try	typeid	typename	using	virtual
wchar_t				

3. Phase of Program Execution:

Phases of C++ Programs:

- Edit
- Preprocess
- Compile
- Link /build
- Load
- Execute



Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the **RESULT** variable is not the same as the **result** variable or the **Result** variable. These are three different variable identifiers.

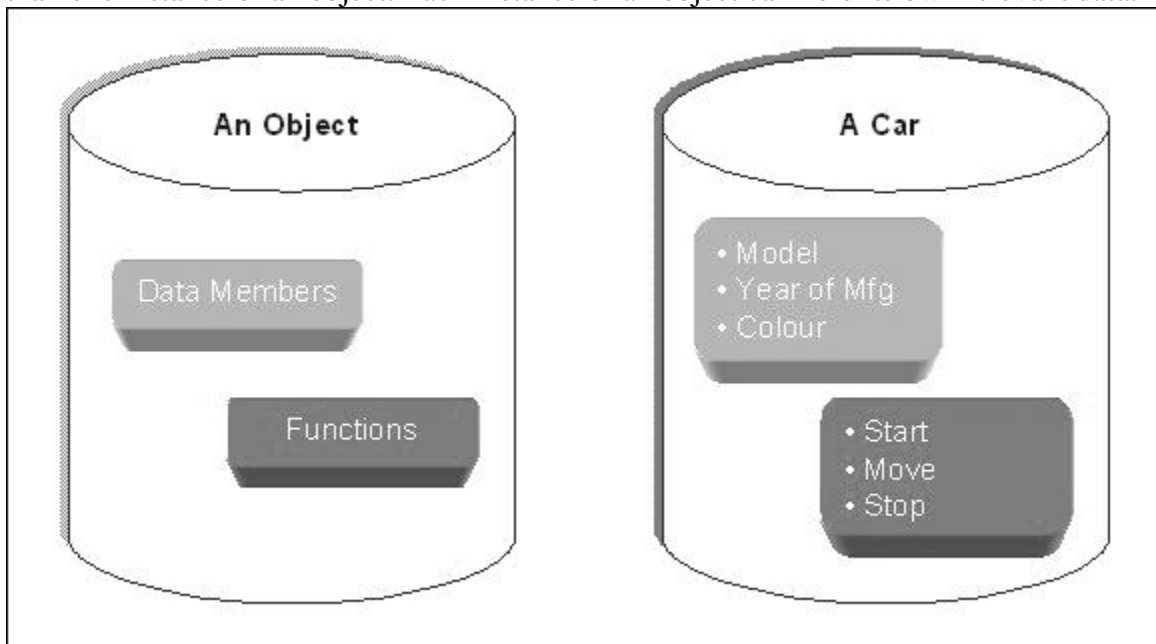
4. Basic Concepts of Object Oriented Programming:

- Objects
- Classes
- **Inheritance**
- Data Abstraction
- Data Encapsulation & hiding
- **Polymorphism**
- **Overloading**
- Reusability

In order to understand the basic concepts in C++, the programmer must have a command of the basic terminology in object-oriented programming. Below is a brief outline of the concepts of Object-oriented programming languages:

(a)Objects:

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.



An Object is a collection of data members and associated member functions also known as methods.

(b)Classes:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represent a set of individual objects. Characteristics of an object are represented in a class as **Properties**. The actions that can be performed by objects becomes functions of the class and is referred to as **Methods**.

For example consider we have a Class of *Cars* under which *Santro Xing*, *Alto* and *WaganR* represents individual Objects. In this context each *Car* Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form **Properties** of the *Car* class and the associated actions i.e., object functions like Start, Move, Stop form the **Methods** of *Car* Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

(c)Inheritance:

Inheritance is the process of forming a new class from an existing class or *base class*. The base class is also known as *parent class* or *super class*, The new class that is formed is called *derived class*. Derived class is also known as a *child class* or *sub class*. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

(d)Data Abstraction:

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

(e)Data Encapsulation:

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

(f)Polymorphism:

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways. There are two types of polymorphism in C++:

- static polymorphism
 - function overloading
 - operator overloading
- dynamic polymorphism
 - virtual functions or dynamic binding

(g)Overloading:

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an existing operator or function begins to operate on new data type, or class, it is understood to be overloaded.

(h)Reusability:

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance. This time saving feature optimizes code, helps in gaining secured applications and facilitates easier maintenance on the application.

The implementation of each of the above object-oriented programming features for C++ will be highlighted in later sections.

5. Data Types in C++

Following is the classification of data types in C:

1. Standard data type

a. Simple data type

- i. **bool**
- ii. char
- iii. int
- iv. float
- v. **wchar_t**

c. Pointer

b. Structured/composite data type

- i. array
- ii. struct
- iii. **class**
- iv. union

2. User Defined data type

- a. enum
- b. typedef

The *char* holds a character or a small integer.

It has two modifies:

- unsigned
- signed

The *int* holds a whole number.

It has two types of modifiers:

- Type 1:
 - signed
 - unsigned
- Type 2:
 - short
 - int
 - long

The *float* can hold a number with fraction parts. It has three modifiers:

- float
- double
- long double

Name	Description	Size	Range
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true (1) or false(0)
float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	10bytes	1.7e +/- 308 (20 ? digits)
wchar_t	Wide character.	2bytes	1 wide character

6. Operator in C++:

P.	Operator	Description	Associativity
1	::	Scoping operator (SRO)	none
2	() [] → . ++ --	Grouping operator Array access Member access from a pointer Member access from an object <u>Post-increment</u> <u>Post-decrement</u>	left to right
3	! ~ ++ -- - + * & (type) sizeof()	Logical negation Bitwise complement <u>Pre-increment</u> <u>Pre-decrement</u> Unary minus Unary plus Dereference Address of Cast to a given type Return size in bytes	(unary operators) <u>right to left</u>
4	->* .*	Member pointer selector Member object selector	left to right
5	* / %	Multiplication Division Modulus	left to right
6	+ -	Addition Subtraction	left to right
7	<< >>	Bitwise shift left Bitwise shift right	left to right
8	< <= > >=	Comparison less-than Comparison less-than-or-equal-to Comparison greater-than Comparison greater-than-or-equal-to	left to right
9	= = !=	Comparison equal-to Comparison not-equal-to	left to right
10	&	Bitwise AND	left to right
11	^	Bitwise exclusive OR	left to right
12		Bitwise inclusive (normal) OR	left to right
13	&&	Logical AND	left to right
14		Logical OR	left to right
15	? :	Ternary conditional (if-then-else)	<u>right to left</u>

16	= += - = *= /= %= &= ^= = <<= >>=	Assignment operator Increment and assign Decrement and assign Multiply and assign Divide and assign Modulo and assign Bitwise AND and assign Bitwise exclusive OR and assign Bitwise inclusive (normal) OR and assign Bitwise shift left and assign Bitwise shift right and assign	<u>right to left</u>
17	,	Sequential evaluation operator	left to right

```
a+ b* c
(a+ b) *c
a – b + c
a *= b += c
```

7. Escape Sequence:

<u>Escape Sequence</u>	<u>Description</u>
\'	Single quote
\"	Double quote
\\	Backslash
\nnn	Octal number (nnn)
\0	Null character (really just the octal number zero)
\a	Audible bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xnnn	Hexadecimal number (nnn)

8. Structure of A C++ Program and First Program in C++

```
// hello.cpp: my first program in C++

#include <iostream>
using namespace std;

int main ()
{
    cout << "Hello World!";
    return 0;
}
```

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream>

Lines beginning with a pound sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive `#include <iostream>` tells the preprocessor to include the `iostream` standard file. This specific file (`iostream`) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it - the instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them. Right after these parentheses we can find the body of the main function enclosed in braces {}. What is contained within these braces is what the function does when it is executed.

cout << "Hello World";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

comments:

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

```
// line comment  
/* block comment */
```


9. Another Program:

```
//program to calculate area of a circle
#include <iostream.h>

int main()
{
    float rad, area;
    const float PI = 3.14F;

    cout << endl << "Enter radius: ";
    cin >> rad;

    area = PI * rad * rad;

    cout << "Area is : " << area << endl;

    return 0;
}
```

Explain how to:

- edit
- compile
- build
- run and
- debug

the program in Visual Studio 6.0

Functions

1. Basics:

- Functions
 - Modularize a program
 - Software reusability
 - Call function multiple times
- Local variables
 - Known only in the function in which they are defined
 - All variables declared in function definitions are local variables
- Parameters
 - Local variables passed to function when called
 - Provide outside information
- Function prototype
 - Tells compiler argument type and return type of function
int cube(int);
 - Function takes an **int** and returns an **int**
 - Explained in more detail later
- Calling/invoking a function
cube(x);
 - Parentheses an operator used to call function
 - Pass argument x
 - Function gets its own copy of arguments
 - After finished, passes back result
- Format for function definition
return-value-type function-name(parameter-list)
{
declarations and statements
}
 - Parameter list
 - Comma separated list of arguments
 - Data type needed for each argument
 - If no arguments, use **void** or leave blank
 - Return-value-type
 - Data type of result returned (use **void** if nothing returned)
- Example function

```
int cube( int y )
{
    int result;
    result = y*y*y;
    return result;
}
```

- **return** keyword
 - Returns data, and control goes to function's caller
 - If no data to return, use **return**;
 - Function ends when reaches right brace
 - Control goes to caller
- Functions cannot be defined inside other functions

```
#include <iostream>

using std::cout;
using std::endl;

int cube( int y ); // function prototype

int main()
{
    int x;

    // loop 10 times, calculate cube of x and output results
    for ( x = 1; x <= 10; x++ )
        cout << cube( x ) << endl;

    return 0; // indicates successful termination

} // end main

// definition of function cube
int cube( int y )
{
    return y * y * y;
}
```

2. Inline Functions:

These are the functions, which are substituted at the place of call.

Drawback of normal function: Use of function increases the execution time. This increase is proportionally substantial when the function is very small. During execution when function is called, there is transfer of control from one area of memory to other area of memory. It requires a lot of accounting activities; hence some extra memory and more CPU time is needed.

When the function is sufficiently large, this increase in time is proportionally less

Inline functions are substituted at the place of call. It has two major advantages:

- (i) The time is saved because there is no transfer of control from one function to another. A normal function requires context switching, push and pop operations on system stack; it consumes a lot of CPU time.
- (ii) Modularity is maintained.

```
# include < iostream .h>

inline void add (int x, int y);
    //prototype

int main ()
{
    int a, b ;

    a = 10;
    b = 20;
    add(a, b);    //call to inline function
    return 0;
```

```
//inline function to add two variables
void add (int x, int y)
{
    int sum ;

    sum = x+ y ;
    cout << "sum is : "<< sum ;
}
```

The function add() is called in the main() function. The execution will not be transferred to the function. The function add() is substituted at the place of call at the **compile time.**

Important Points:

- a) To make a function inline, the keyword must appear in the declaration/prototype.
- b) **Inline is a request to the compiler, not a command.** It means that it is not necessary that if we have declared a function as inline, it will be treated as inline type. It means that if the compiler views that the function cannot be inline, it will discard the request and will treat the function as simple non-inline type. A function will not be treated as inline (even if it marked so by the programmer), in following cases:

- it contains loops.
- it contains switch statement.
- it contains multiple return statements.
- it is a recursive function

comparison of
inline function and
macro

- c) The member functions defined inside the class are considered as inline type, even if not marked by the programmer.

- d) Small functions should be declared as inline type.

- e) Virtual functions can never be inline.

disadv:

1. code size increases.
2. compilation time increases.

3. Reference Variables:

A reference variable is an alias (another name or synonym) for an existing variable.

```
int x;           // x is a variable
int& y = x;      // a reference variable
```

To create a reference, we simply put & (ampersand) sign as shown and equate it to an existing variable of same data type. Note that the = operator is not assignment operator here and the & is not used as address of operator.

x Location is
 same for both
y

```
//ref.cpp
#include <iostream.h>

int main( ) {
    int x;
    int& xx = x;    // xx is a reference of x

    x = 10;
    cout << endl << x << xx;    // both are 10
    xx++;
    cout << endl << x << xx;    // both are 11

    return 0;
}
```

Important Points:

- An identifier can have any number of references.
- A reference is a logical alternative name for an actual variable.
- A reference must be initialized at place of declaration; its **initialization** cannot be deferred.
- A reference once created and bound to a variable, cannot be rebounded to another variable.
- There is no concept of NULL reference.
- **A function returning a reference can appear on the left hand side of the assignment operator.**

Que: Comment about the following statements:

```
int& x = 10;
const int& y = 10;
```

4. Pass-by-Value and Pass-by-Reference Parameter Passing:

We can do pass-by-reference parameter passing with the help of references.

```
//byRef.cpp
#include <iostream.h>

void change(int& x, int y); //prototype

int main(){
    int a = 10,
        b = 20;

    change(a, b);    //function called
    cout << endl << a << b;    // a = 11, b = 20
    return 0;
}
```

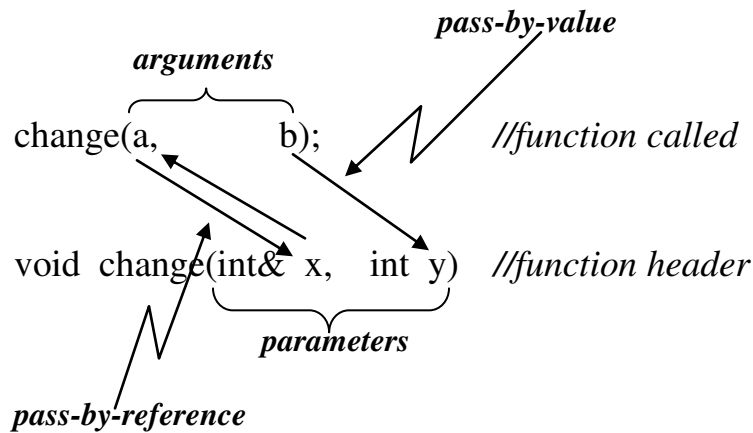
```
/*function to change two parameters,
   one is pass by reference, another is pass by
   value
*/
void change(int& x, int y)
{
    cout << endl << x << y;    // x = 10, y = 20
    x++;
    y++;
    cout << endl << x << y;    // x = 11, y = 21
}
```

Here, a and b are arguments; x and y are parameters. The values of arguments a and b are assigned to corresponding parameters x and y.

In the above example 'a' is pass by reference, and 'b' is pass by value. The value of a is assigned to x, and value of b is assigned to y.

```
x = a;
y = b;
```

x and y both are changed in the function. The change in x is reflected back to a; but change in y is not reflected back to b. It is because it is pass-by-reference and b is pass-by-value.



Advantages of Pass-by-Reference:

- it saves space, because no extra space is reserved for reference parameter.
- It saves time, because there is no data copying from calling function to called function.

Disadvantages of Pass-by-Reference:

- Pass-by-reference parameters increase the chances of errors in the program. Any wrong change in the pass-by-reference parameter is spread throughout the program, and it accumulates. During debugging, it becomes tough to identify the place of error.

According to the return behavior, parameters can be divided into three categories:

- The parameters are changing, but their change need not have any impact on corresponding arguments – in this case, “pass-by-value” is selected.
- The parameters are changing, and their changes are to be reflected to corresponding arguments – in this case, “pass-by-reference” is selected.
- The parameters are not changing – in this case, “pass-by-reference” is selected with the parameters as constant type. This scheme takes advantage of both types of parameters.

```
//threePara.cpp
#include <iostream.h>

void change(int, int&, const int&);    //prototype

int main() {
    int a = 10,
        b = 20,
        c = 30;
    cout << endl << a << b << c;    // a = 10, b = 20, c = 30
    change(a, b, c);                //function called
    cout << endl << a << b << c;    // a = 10, b = 21, c = 30
    return 0;
}
```

```
/*function to change two parameters,
   one is pass by reference, another is pass by value*/
void change(int x, int& y, const int& z) {
    cout << endl << x << y << z;    // x = 10, y = 20, z = 30
    x++;
    y++;
    // z++;
    //change in z not permitted because it is constant
    cout << endl << x << y << z;    // x = 11, y = 21, z = 30
}
```

5. Function Overloading: Same function name can be used for more than one function definitions. It is a type of static polymorphism. The advantage is that the user needs less function names to remember.

The overloading can be done in three different ways:

- a) Different number of arguments
- b) Different kind/type of arguments
- c) Different number and kind of arguments both

(a). Different Number of Arguments:

More than one function definitions with same function name, but with different number of arguments, are considered as different functions. The call to the function is resolved according to the number of arguments.

Ex:

```
//over1.cpp
#include <iostream.h>
// prototypes
void repchar( );
void repchar(char);
void repchar(char, int);

int main( ) {
    repchar( );           // no arguments, I function is called
    repchar( '$');        // one argument, II function is called
    repchar('*', 30);      // two arguments, III function is called
    return 0;
}

// I - function to display 40 '+' in one line
void repchar( ) {
    for (int i = 0; i < 40; i++)
        cout << '+';
    cout << endl;
}

// II - function to display a supplied characters 40 times
void repchar(char ch) {
    for (int i = 0; i < 40; i++)
        cout << ch;
    cout << endl;
}

// III - function to display the supplied character n number of times
void repchar(char ch, int n) {
    for (int i = 0; i < n; i++)
        cout << ch;
    cout << endl;
}
```

There are three functions `repchar()` with same name, but with different number of arguments. During the call to the function, number of arguments is matched with the number of parameters; when there is exact match, the function definition is called.

(b). Different Type of Arguments:

More than one function definitions with same function name and same number of arguments, but with different kind of arguments, are considered as different functions.

```
//over2.cpp
#include <iostream.h>

// the complex structure
struct complex {
    float real;
    float img;
};

// prototypes
void display(int);
void display(complex);

int main( ) {
    int x = 10;
    complex c;

    c.real = 5.0; c.img = 7.0;

    display(x);    //function to display integer is called
    display(c);    //function to display complex number is called

    return 0;
}

// 1st :function to display an integer
void display(int a) {
    cout << endl << "Integer is: " << a
}

// 2nd :function to display complex number
void display(complex b) {
    cout << endl << "Real Part: " << b.real
    << "Imaginary Part: " << b.img;
}
```

There are two functions with same name *display()*. The count of arguments is the same, but their type is different. When the argument is integer type, the 1st function is called. When the argument is complex type, 2nd function is called.

(c) Different Number and Kind of Arguments:

There can be overloading by using:

- different number, as well as
- different kind of arguments.

Demonstrate over3.cpp

NOTE: Function overloading is not possible based on return type of function. i.e. overloading resolution is done on the basis of only signature.

For example, there are two functions with same name and with the same parameter list but with different return type.

```
float sample(int x, float y);  
int sample(int x, float y);
```

It will be **an error** to declare such functions.

Resolution:

When an overloaded function is called, the decision of which definition is to be executed is called **overloading resolution**. Following are the criteria for overloading resolution:

- (I). Exact match or match using only trivial conversions:
e.g.; array name to pointer, function name to pointer-to-function.
- (II). Match using promotion:

Promotion in floating point type: Promotion in integral data-type:
bool → char → short int → int → long int
float → double float → long double float
- (III). Match using standard conversions:
Conversion in standard data types:
integral → float type
Conversion in hierarchy
derived * → base *
any data type * → void *
- (IV). Match using user defined conversions.
- (V). Match using the ellipses ... in a function declaration.

NOTE: If two matches are found at the same level, the call is rejected as ambiguous call.

7. Default Arguments:

Sometime, a pre-decided value (called default value) for a parameter is assigned. If the argument corresponding to the parameter is not supplied at the place of call, the parameter takes the default value. It is called default argument.

Ex1

```
//power.cpp  
#include <iostream.h>  
  
float power(float x, int n=2);    //prototype, default value of n is 2  
  
int main()  
{  
    float x;  
    int n;  
    x = 5.0; n = 3;  
    cout << power(x, n);          // 5.03 = 125.0; default value not used  
    cout << power(x);             // 5.02 = 25.0; default value used, n=2  
    return 0;  
}
```

```

/* Function to calculate power
   input: x and n
   output: x to the power n */
float power(float x, int n){
    float res = 1.0;
    for ( int i = 0; i < n; i++)    // loop to calculate power
        res *= x;
    return res;
}

```

Ex 2:

```

#include <iostream.h> //defaultRepChar.cpp

// prototype
void repchar(char ch = '+', int n = 40);

int main( ){
    repchar( );           // no arguments supplied, both default values are used
    repchar( '$');        // one argument supplied, for left-most parameter
    repchar('*', 30);      // two arguments supplied, no default assumed
    return 0;
}

// function to display the supplied character n number of times
// with default value of ch as '+' and n as 40
void repchar(char ch, int n){
    for (int i = 0; i < n; i++)
        cout << ch;
    cout << endl;
}

```

We see that the use defaults arguments sometimes reduce the number of functions to be designed.

Important Points:

- (I). Default arguments are type checked at the time of function declaration and evaluated at the time of call. **So, the default value is given in the prototype declaration.**
- (II). **Default value can only be provided to trailing arguments.** It is because the matching of arguments to parameters is **from left towards right**. It is logical because, otherwise the compiler cannot provide the exact correspondence between the parameters and missing argument values.

For example, the following is not possible:

float power(float x = 10, int n); // ERROR

Ex 3:

Let a prototype is:

```
void sample(int x, int y=10, int z=20);
```

Following are the calls, and their resolution:

- sample(5, 25, 12); // x = 5, y = 25, z = 12
- sample(5, 25); // x = 5, y = 25, z = 20
- sample(5); // x = 5, y = 10, z = 20

8. Random Number Generation:

- `rand()` function is in the header file `<stdlib.h>`

`i = rand();`

It generates unsigned integer between 0 and `RAND_MAX` (usually 32767). Scaling and shifting can be done by using modulus operator.

To generate numbers from 0 to `n-1`:

```
int x;  
x = rand() % n;
```

To generate numbers from `a` to `b`:

```
int x;  
x = a + rand() % b;
```

Following is the code segment to generate `n` random numbers for `a` to `b`

```
int main() {           //rand1.cpp  
    int a, b, n, x;  
  
    cout << "Enter n: ";  
    cin >> n;  
  
    for ( int i = 0; i < n; ++i) {  
        x = a + rand() % b;  
        cout << x < endl;  
    }  
}
```

But, it will generate the same sequence always. The sequence of numbers generated by `rand()` depends upon the seed value. The seed value can be specified by the [`srand\(\)` function](#).

```
int main() {  
    int a, b, n, x, seed;  
  
    cout << "Enter n: ";  
    cin >> n;  
  
    cout << "Enter seed: ";  
    cin >> seed;  
    srand(seed);  
  
    for ( int i = 0; i < n; ++i) {  
        x = a + rand() % b;  
        cout << x < endl;  
    }  
}
```

But, if you want to generate always a different sequence without getting the seed value from the user, the current system time can be used as the seed value. The system time always changes.

```

int main() {
    int a, b, n, x, seed;

    cout << "Enter n: ";
    cin >> n;

    srand( time(0) );

    for ( int i = 0; i < n; ++i) {
        x = a + rand( ) % b;
        cout << x < endl;
    }
}

```

The time(0) function returns the current system time in seconds. This function is available in <ctime.h> header file.

9. Unary scope resolution operator ::

The operator :: is used to:

- access the class members with class name.
- to access hidden outer scope identifiers.

```

int x = 10; //scoprRes.cpp
int main() {
    for ( int i = 0; i < 5; ++i) {
        int x = 20;
        cout << "inner x is – " << x << endl;
        cout << "outer x is – " << ::x << endl;
        ++x;
        ++(::x);
    }
}

```

Exercises:

R1.Design a program to add sum of digits of a number.

R2. Design a function:

int prime(int x);

Design the main also.

1. Write a function that takes the time as three integer arguments (hours, minutes and seconds) and returns the number of seconds since the last time the clock “struck 12.” Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock. (ex02_01.cpp).

2. Implement the following integer functions:

a) Function **celsius** returns the Celsius equivalent of a Fahrenheit temperature.

b) Function **fahrenheit** returns the Fahrenheit equivalent of a Celsius temperature.
c) Use these functions to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temperatures from 0 to 100 degrees, and the Celsius equivalents of all Fahrenheit temperatures from 32 to 212 degrees. Print the outputs in a neat tabular format that minimizes the number of lines of output while remaining readable. (ex02_02.cpp)

3. An integer is said to be a *perfect number* if the sum of its factors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number, because $6 = 1 + 2 + 3$. Write a function **perfect** that determines whether parameter **number** is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000. (ex02_03.cpp)

4. Write a program that simulates coin tossing. For each toss of the coin, the program should print **Heads** or **Tails**. Let the program toss the coin 100 times and count the number of times each side of the coin appears. Print the results. The program should call a separate function **flip** that takes no arguments and returns **0** for tails and **1** for heads. [Note: If the program realistically simulates the coin tossing, then each side of the coin should appear approximately half the time. (ex02_04.cpp)]

5. Use a single-subscripted array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, print it only if it is not a duplicate of a number already read. (ex02_05.cpp)

6. Write a program that simulates the rolling of two dice. The program should use **rand** to roll the first die and should use **rand** again to roll the second die. The sum of the two values should then be calculated. [Note: Each die can show an integer value from 1 to 6, so the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums.] Fig. shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000 times. Use a single-subscripted array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one sixth of all the rolls should be 7). (ex02_06.cpp)

7. A puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible to place eight queens on an empty chessboard so that no queen is “attacking” any other, i.e., no two queens are in the same row, the same column, or along the same diagonal? Run your program. (Hint: It is possible to assign a value to each square of the chessboard indicating how many squares of an empty chessboard are “eliminated” if a queen is placed in that square. Each of the corners would be assigned the value 22. Once these “elimination numbers” are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. (ex02_07.cpp)

Class and Object

1. Class:

A class is encapsulation of:

- Data
- The functions operating on that data.

Normally, the data part is not made available out of the class. It provides data hiding. Data hiding provides 'abstraction'. Data hiding minimizes the chances of accidental changes in the data. Abstraction provides simplicity by hiding the implementation details of the class from the user.

Hiding of details means "abstraction". Abstraction is of two types:

- Data Abstraction
- Instruction Abstraction

2. Object:

An object is an **instance** of a class. **Class is the template/type of object**. A class is a data type and an object is its variable. The object occupies space in main memory during run time, but the class does not.

```
//counter1.cpp
#include <iostream.h>

//counter class
class Counter {
private: // things accessible within
        //the class definition only
    int count;

public: // things accessible out of the class also
    //function to initialize
    void initialize(int c=0 )
    { count = c ; }

    //function to increment
    void increment( )
    { ++count ; }

    //function to decrement
    void decrement( )
    { --count ; }

    //getter function
    int get_count( )
    { return count ; }
}; //end of class
```

```
//main function
int main( ) {
    Counter c;           // object created, space allocated
                        //for c.count integer variable
    c.initialize( ) ;    // effectively c.count = 0

    //increment counters 3 times.
    c.increment( ) ;     // effectively c.count++
    c.increment( ) ;
    c.increment( ) ;

    //display
    cout << c.get_count( ) << endl;

    //decrement count
    c.decrement( ) ;    //effectively c.count- -

    //display
    cout << c.get_count( ) << endl;
    return 0 ;
}
```

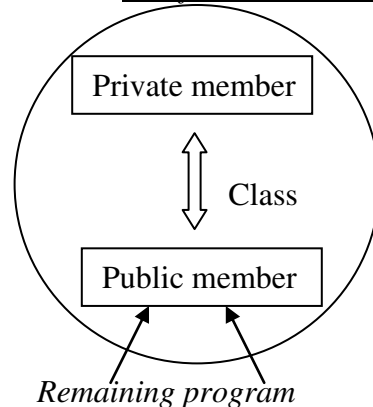
Notes :

- a) Conventionally the first letter of class name is kept capital.
- b) There are 2 access specifiers :
 - private
 - public

Public members are accessible within and out of the class. Private members are accessible within the class only. To take the advantages of object-oriented concept, the data is kept private and functions are kept public.

The private access specifier provides hiding & abstraction.

The public members acts as an interface between the class and remaining program.



- c) The default access specifier in a class is **private**.
- d) When we create an object of a class, it is called **instantiation**.
- e) The members of a class are accessed by using membership operator “.”. Private members are not accessible from outside.

< object name > . < member name >

membership operator

- f) A class normally contains the following functions:
 - a) function to initialize the member data.
 - b) *setter function* - this function sets the specified value to the member data.
 - c) *getter function* - this function returns the value of member data.
- g) We can create any no of objects of a class, for example:
Counter c1, c2 , c3 ;
- h) The space for the data members of each object is allocated separately in the data area
c1.count 4 byte
c2.count 4 byte
c3.count 4 byte
- i) Here *count* is called the **object data or instance variable**, because it is separate for each object.
- j) Space allocation for an object means - only the space for data members is allocated, not for local variables of the member functions.
- k) The object code for member functions of a class is stored **only once** in the code area. This single copy of the member functions is shared by the all objects of the class.
- l) The size of a class or its object is the sum of individually the size of member data only(**alignment matters**). The functions and their local variables **do not** contribute the size of the object. **The size of empty class or its object is always one byte.**
- m) An object follows all the rule of scope(visibility) and life time.
- n) Member functions are also called **methods or messages**.

- o) The member functions can be defined and inside as well as outside the class.
- p) *An object can be passed as parameter to the function. It can be return from the function as well. In both the cases, only member data contribute, not the member function. In this sense, an object behave like a structure.*
- q) The definition terminates by semicolon. It indicates that it is a data declaration.

Example:

Design a class for British Distance. It contains the member data feet & inches. The member functions are to initialize, to get the value and to display the value of member data.

```
// Distance1.cpp
#include <iostream.h>

class Distance {           //English Distance class
private:
    int feet;
    float inches;

public:
    void setdist(int ft = 0, float in = 0.0) //set Distance to args
    { feet = ft; inches = in; }

    void getdist( ) {           //get length from user
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }

    void showdist( )           //display distance
    { cout << feet << "\'-" << inches << "\" << endl; }
}; // end of class
//////////
int main( ){
    Distance d1, d2;           //define two lengths

    d1.setdist( );             //dist1 components are both 0
    d2.setdist(11, 6.25);      //set dist2

    //display lengths
    cout << "\ndistance1 = "; d1.showdist( );
    cout << "\ndistance2 = "; d2.showdist( );
    //get values from keyboard for dist1
    d1.getdist( );
    d1.showdist( );
    return 0;
}
```

Exercise: Design a class **Complex** . The member data are **real** and **img**. The member functions are: (Complex1.cpp)

- To set the values, use default values
- To display the complex number in (x + i y) format
- To read values of member data form keyboard

2. Constructor

A constructor is a member function of a class that is **automatically** called as soon as the **object is created**. It is used to:

- Initialize the member data of the object.
- To dynamically and automatically allocate memory.

Following are the main properties of the constructor:

- a) The constructor has the same name as that of class (**case sensitivity is there**).
- b) It does not have any return type.
- c) A constructor can be overloaded, *i.e.*, there can be more than one constructor in the class.
- d) Default arguments can be used in constructor.
- e) **It must be public type.**

Types of constructor

- a) Default constructor.
- b) Parameterized constructor
- c) Copy constructor.
- d) Conversion constructor.
- e) Explicit constructor.

Ex. Design a class Counter to provide following operations:

- setter function - function to set the value
- getter function - function to get the value
- Increment and decrement

```
//counter2.cpp
class Counter {
private:
    int count;
public:
    Counter()    //default constructor
    { count =0;}
    Counter(int c) //parametrized constructor
    {count =c;}

    //function to set value
    void set _count( int c)
    {count = c;}
    //function to get value
    int get _count ( )
    { return count ;}

    //function to increment
    void increment ( )
    { count ++}

    //function to decrement
    void decrement( )
    { count-- ; }
}; //end of class
```

```
//main function
int main( )
{
    Counter c1;
    //default constructor called  c1.count=0

    Counter c2 (100);
    //parameterized constructor called
    //c2 .count =100

    cout << c1.get _count( )
         << c2.get _count( ) << endl;

    c1.increment( );
    c1.increment( );
    c2.decrement( );

    cout << c1.get _count( )    // c1.count is 2
         << c2.get _count( )    // c2.count is 99
         << endl;
    return 0;
}
```

Note 1:

The elongated way of calling the constructor is

Counter c1 = Counter(); //call to default ctor

Counter c2 = Counter(100); //call to parameterized ctor

The abbreviated way is:

Counter c1, c2 (100);

A constructor is called default constructor, if:

- The constructor has no parameters, or
- All the parameters of constructor have default values

If there are parameters and any of them do not have default value, then it is called parameterized constructor.

The two constructors can be substituted by using default arguments:

```
Counter(int c = 0)      //parameterized constructor
{ count =c; }
```

NOTE: 1). The purpose of the constructor is to initialize the member data of the object. The constructor is called automatically immediately after the space is allocated for member data of the objects.

2) If there is no ctor in the class, the compiler inserts one with no parameter and with empty body. If the designer inserts a ctor in the class, then compiler inserts no ctor. So if the designer only give parameterized ctor, then there will be no default ctor available in the class.

3. Defining Member Functions:

There are two options for providing the definition of a member function:

- A member function can be defined inside the class.
- A member function can be declared (only prototype) inside the class and defined outside the class. In this case, following syntax is used:

<return type> <class name>::< function name> (<parameter list>)

The above distance class can also be written as:

```
// Distance1B.cpp
#include <iostream>
class Distance {                      //English Distance class
private
    int feet;
    float inches;
public:
    void setdist(int ft = 0, float in = 0.0) //set Distance to args
    { feet = ft; inches = in; }

    void getdist( );                    /**prototype inside the class**

    void showdist( )                    //display distance
    { cout << feet << "\'-" << inches << "\" << endl; }
}; // end of class

//****definition out of the class****
void Distance::getdist( ) {
    cout << "\nEnter feet: "; cin >> feet;
    cout << "Enter inches: "; cin >> inches;
}
```

The main function remains the same.

Normally, the function with small body should be defined inside the class, but the functions with larger body should be defined outside the class.

The member function defined inside the class is by default inline type. The member function declared inside & defined outside the class is by default non-inline type. Normally we should define the very small member functions inside the class, but larger functions out of the class with prototype inside.

4. Member Initializer List:

One of the most common tasks a constructor carries out is initializing data members. We write a constructor like this:

```
Counter( )
{ count = 0; }
```

We can write the constructor in the following way also:

```
Counter( ) : count(0)
{ }
```

The initialization takes place following the member function declaration but before the function body is executed. It is preceded by colon. The value is placed in parentheses following the member data. If multiple members are to be initialized, they are separated by commas, so a list is formed.

```
SomeClass( ) : m1(10), m2(20), m3(30) ← initializer list
{ }
```

The member initializer list can *only* be used in constructors, not in any other member functions.

The Counter class is redesigned as:

```
//counter3.cpp
class Counter {
private:
    int count;
public:
    Counter(int c = 0) : count(c)
    { }
    //function to set value
    void set_count( int c)
    { count = c; }
    //function to get value
    int get_count ( )
    { return count ; }

    //function to increment
    void increment ( )
    { count ++ }

    //function to decrement
    void decrement( )
    { count-- ; }
}; //end of class
```

```
//main function
int main( )
{
    Counter c1;
    //default constructor called c1.count=0

    Counter c2 (100);
    //parameterized constructor called
    //c2 .count =100

    cout << c1.get_count( )
         << c2.get_count( ) << endl;

    c1.increment( );
    c1.increment( );
    c2.decrement( );

    cout << c1.get_count( )    // c1.count is 2
         << c2.get_count( )    // c2.count is 99
         << endl;

    return 0;
}
```

Exercise: Redesign the Complex class including ctors with member initializer list. (Complex1b.cpp)

Constant member data can only be assigned values with the help of member initializer list.

Initializing const member data with the help of member initializer list:

There can be constant member data inside a class. The constant member data is by default static type, because there is need of a single copy of a const member. **The const member data of a class can only be initialized with the help of member initializer list, not with simple assignment in the constructor.**

Not providing member initializer list for a const member is a syntax error.

```
//counterIncr.cpp : counter with constructor
#include <iostream.h>
class Counter{
private:
    int count;
    const int incr;
public:
    Counter(int c = 0, int in=1): count(c), incr(in)
    {
    }
    //function to set value
    void set_count( int c)
    { count = c;}
    //function to get value
    int get_count( )
    { return count;}
    //function to increment
    void increment ( )
    { count += incr;}
    //function to decrement
    void decrement( )
    { count-= incr ; }
}; //end of class

//main function
int main()
{
    Counter c1;           //default constructor called  c1.count=0
    Counter c2(100, 10); //parameterized constructor called  c2 .count =100

    cout <<endl << c1.get_count( )
        <<endl << c2.get_count( ) << endl;

    c1.increment( );
    c1.increment( );
    c2.increment( );

    cout << endl << c1.get_count( )
        << endl << c2.get_count( ) << endl;

    return 0;
}
```

5. “const” Member Function:

A “const” member function is that member function that cannot modify the member data of the object for which it is called.

A function can be made constant by placing the keyword “const” after the declaration (header) but before the body of the function. If there is a separate function declaration and definition, the “const” keyword must be put at both the places.

```
//demonstrates const member functions
class SomeClass {
private:
    int alpha;
public:
    void nonFunc( )      //non-const member function
    { alpha = 99; }      //OK

    void conFunc( ) const //const member function
    { alpha = 99; }      //error: can't modify a member
};
```

The use of “const” should be preferred because if the member data is not being changed in the member function; it disallows accidental changes in the member data.

1. A const member function can not modify instance variable
2. *To call a non-const member function from the body of a const member function is an error.*
3. *Invoking a non-const member function on a const object is an error.*
4. **Constructors and destructors can never be const.**

Exercise 1: Design a class Cartesian containing two members x and y. The member functions are:

- Constructors using member initializer list.
- Set function
- Display function.

Exercise: time2.cpp
(it demonstrates the right methodology.

Use the concept of constness where ever necessary. (Cartesian.cpp)

Exercise 2: Design a class Date containing day, month, and year. The member functions should be: (date1.cpp). Use the concepts of member initializer list and constness.

- Default and parameterized ctor (1/1/2000 as default date)
- to print date
- to set date.

Exercise 3: Design a class Time with following declaration: (time1.cpp)

```
class Time {
private:
    int hour;           // 0 - 23 (24-hour clock format)
    int minute;         // 0 - 59
    int second;         // 0 - 59
public:
    Time( int = 0, int = 0, int = 0 ); // default constructor
    // set functions
    void setTime( int, int, int ); // set hour, minute, second
    void setHour( int ); // set hour
    void setMinute( int ); // set minute
    void setSecond( int ); // set second
    // get functions
    int getHour() const; // return hour
    int getMinute() const; // return minute
    int getSecond() const; // return second
    void printUniversal(); // output universal-time format
    void printStandard(); // output standard-time format
}; // end class Time
```

6. Objects as Parameters:

An object can be passed as parameter to a member or **non-member** function (also called **free** function). The behavior is same as is a structure is passed. **The important thing is that only member data is passed;** there is no significance of passing member functions and their local variables. The values of member data of argument object are assigned to corresponding member data to parameter object. The assignment is member-wise. The following program demonstrates the passing of objects.

Ex: Design a class Distance that has following member functions: (distance2.cpp)

- to set the distance.
- to display the distance.
- to add two distances.
- to subtract two distances

```
#include <iostream>    //distance2.cpp
class Distance {      //English Distance class
private:
    int feet;
    float inches;
public:
    Distance() : feet(0), inches(0.0)           //constructor (no args)
    { }
    Distance(int ft, float in) : feet(ft), inches(in) //constructor (two args)
    { }
    void getdist() {           //get length from user
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }

    void showdist() const //display distance; const member function
    { cout << feet << "\-" << inches << "\n"; }

    void add_dist(Distance, Distance);           //prototype declaration
    void sub_dist(Distance, Distance);           //prototype declaration
}; //end of class

// member function to add two distances
void Distance::add_dist(Distance d2, Distance d3){
    feet = d2.feet + d3.feet;           //add feet
    inches = d2.inches + d3.inches; //add the inches
    if (inches >= 12.0) { //if total exceeds 12.0, it is a carry then decrease inches

        inches -= 12.0;           //by 12.0 and
        feet++;           //increase feet by 1, add carry
    }
} // end of function

// member function to subtract two distances
void Distance::sub_dist(Distance d2, Distance d3) {
    feet = d2.feet - d3.feet;           //subtract feet
    inches = d2.inches - d3.inches; //subtract the inches
    if (inches < 0.0) { //if net is less than 0, it is a carry then increase inches
        inches += 12.0;           //by 12.0 and
        feet--;           //decrease feet by 1, subtract carry
    }
} // end of function
```

```

int main( ) {
    Distance d1, d3;           //define two lengths
    Distance d2(11, 6.25);     //define and initialize d2

    d3.getdist( );             //get d3 from user

    d1.add_dist(d2, d3);        //d1 = d2 + d3;
    cout << endl << d1 = "; d1.showdist( ); //display resultant length

    d1.sub_dist(d2, d3);        //d1 = d2 - d3
    cout << endl << d1 = "; d1.showdist( );
    cout << endl;
    return 0;
}

```

Notes:

The object with which a member function is called is known as the **current object**. The member data appearing alone in the **member function** is related with the object with which the member function is called, that is with the current object.

In add() and subtract() functions the feet and inches alone are associated with d1 object; because the functions are called with d1 object

7. Returning Objects from Member Functions:

The objects can be returned from a function or member function. An example will demonstrate the following:

```

//distance3.cpp
#include <iostream>

class Distance {
private:
    int feet;
    float inches;
public:
    //constructor (no args)
    Distance( ) : feet(0), inches(0.0)
    { }
    //constructor (two args)
    Distance(int ft, float in) : feet(ft), inches(in)
    { }
    void getdist( ) { //get length from user
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }
    void showdist( ) //display distance
    { cout << feet << "\-" << inches << "\" << endl; }

    Distance add_dist(Distance); //add
};

```

```

// function returns value of type Distance
//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2) {
    Distance temp; //temporary variable
    temp.inches = inches + d2.inches;
    if(temp.inches >= 12.0) //if total exceeds 12.0,
    { //then decrease inches
        temp.inches -= 12.0; //by 12.0 and
        temp.feet = 1; //increase feet
    } //by 1
    temp.feet += feet + d2.feet; //add the feet
    return temp;
} // end of function

```

```

int main() {
    Distance d1, d3; //define two lengths
    Distance d2(11, 6.25); //define, initialize dist2
    d1.getdist( ); //get dist1 from user
    d3 = d1.add_dist(d2); //d3 = d1 + d2
    cout << "\ndist1 = "; d1.showdist( );
    cout << "\ndist2 = "; d2.showdist( );
    cout << "\ndist3 = "; d3.showdist( );
    cout << endl;
    return 0;
}

```

Exercise: See main() of Complex2.cpp; and design the class.

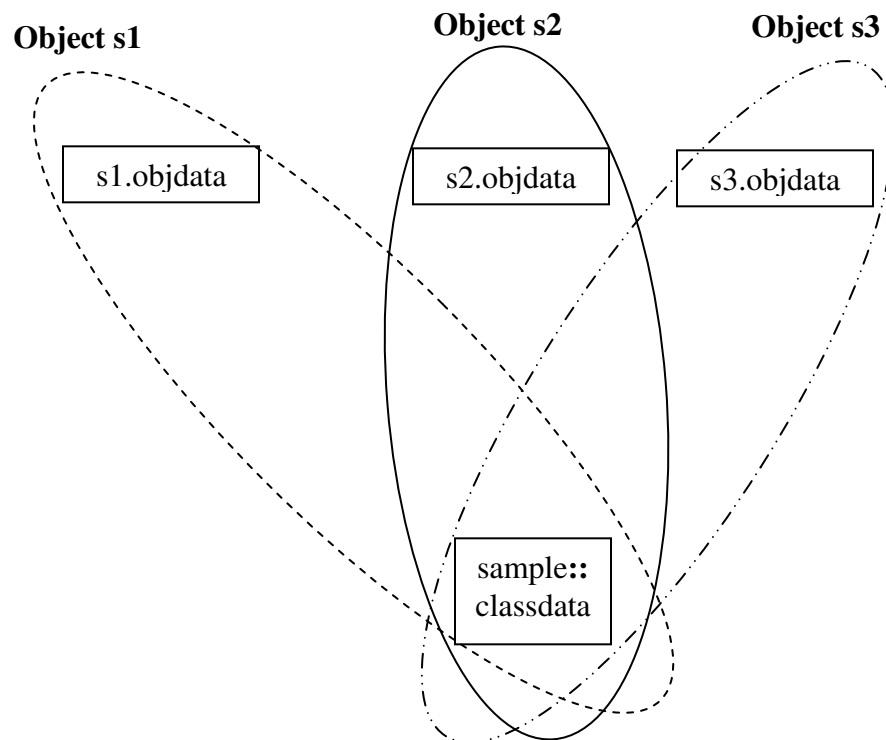
8. Static Member Data:

It is **class data**. It means that there is a single copy of this type of data for all objects. All the objects share this member data copy. Any changes done in static data member by calling member function with any object, are reflected in all the objects.

The simple non-static member data is called **object data**, because its copy is separate for all objects.

A dummy example will demonstrate the things:

```
class Sample {  
private:  
    static int classdata; // static class data  
    int objdata;         // object data  
public:  
    // member function  
    -----  
    -----  
}; // end of class  
  
//definition and initialization of static member  
int Sample::classdata = 0;  
  
int main() {  
    Sample s1, s2, s3;    //objects  
    -----  
    -----  
    return 0;  
}
```



The class or static member data is not associated with single object of the class, that is why it is referred to as:

<class name>::

~~Note: The const member data of a class are static by default. (WRONG)~~

Ex: Design a class that has a member function which returns the number of objects created so far (popular example of static member data).

```
#include <iostream.h> //static1.cpp
class Sample {
private:
    int x;
    static int count;    //private class data
public:
    // constructor
    Sample() {
        count++;    //if object created increment count
        x = 0;
    }

    //static int get_count( )
    static int get_count( ) {    // static function
        return count;
    }
};

//definition and initialization of static data
int Sample::count = 0;

int main( ) {
    cout << "Number of objects created: " << Sample::get_count( ) << endl;
    Sample s1, s2, s3;    //constructor called three times
    cout << "Number of objects created: " << s1.get_count( ) << endl;

    Sample s4, s5;    //constructor called two more times
    cout << "Number of objects created: " << Sample::get_count( ) << endl;

    cout << endl << "Size of object: " << sizeof(s1)
        << endl << "Size of class: " << sizeof(Sample) << endl;
    return 0;}
```

Note 1: The space for static member data is allocated even if no object is declared.

Note 2: Static member data do not contribute to the size of either class or object.

Note 3: We can not call a non-static member function inside a static member function. We can call a static member function inside a non-static member function.

There is one static member data count. It is incremented in constructor. A constructor is called each time when an object is instantiated / created. First time three objects are created, count will be incremented three times; the answer will be 3. Next time two more objects are created, count will be incremented two more times; the answer will be 5.

A **static member function** is that function which access only static member data. It means that no non-static or object member data is accessed inside the static member function. //static member functions can never be const type.

A static member function is a class function, it's not associated with any object of the class, that is why it is referred to as:

<class name>:: <function name> (<parameter list>)

Exercise: Design a class Singleton, it will permit only one object to instantiate. (singleton.cpp)

8. Friend of Class:

A friend of a class can access the private members of the class. There can be 3 types of friends:-

- Friend class
- Friend member function
- Friend non-member function

There are two class sample1 and sample2.

(I) **Friend Class**: If sample1 declares sample2 as its friend, then all members (private, public or protected) of sample2 can access the private members of sample1.

(II) **Friend member function**: A function *fr_mem_fun()* is a member function of class Sample2. The class Sample1 declares this function as its friend; then only function *fr_mem_fun()* member function of sample2 can access the all type (private, protected and public) member of sample1 class.

(III) **Friend non-member function**: Suppose function *fr_nonmem_fun()* is an external non-member function (That is not related to any class). The class Sample1 declares function *fr_nonmem_fun()* as its friend; then function *fr_nonmem_fun()* can access the private members of sample1.

This option can be used to bridge the gap between two classes. Function fr_nonmem_fun() can be the friend of Sample1 and Sample 2 classes. It can access the private members of both classes, so it can do the calculations in which there is a requirement of private data of both classes.

Note: The concept of friend violates the concept of data hiding. The need of too many friends reflects poor class design. So, the class hierarchy has to be redesigned. The concept of friend should only be used in unavoidable circumstances.

The following example will demonstrate the concept:

```
class Sample1{
    int x1, y1;
public:
    // member functions
    ---
    ---
    ---
    // declaration of friend
    friend class Sample2;
};

class Sample2
{
    int x2, y2;
public:
    // member functions
    /* all member functions can access the
    private data x1, y1 of sample1 Class
    */
    ---
    ---
};
```

```
class Sample1{
    int x1, y1;
public:
    // member functions
    ---
    // declaration of friend
    friend class Sample2:: fun();
}; // end of class Sample1

class Sample2 {
    int x2, y2;
public:
    // member functions
    ---
    ---
    // friend function of Sample1 class
    void fun( ) {
        // this function can access all the members
        //(private, protected and public) of the class
        //Sample1
        ---
    }
}; // end of class sample2
```

The class Sample1 is declaring class Sample2 as its friend. The statement about declaration of friend can be put any where in the class. It does not have any impact whether it is put in public domain or private domain.

Note1 : Transitivity is not possible in friendship. If class A is a friend of class B and class B is the friend of class C, then class A is **not** friend of class C (or vice – versa).

Note2: Class A is a friend of class B (A can access private data of B), it **does not** mean then class B is also friend of class A. The declaration of making class B as friend of class A must be explicit.

```
class Sample 1 {
private:
    int x1, y1;
public:
    // member functions
    ---
    ---
    // friend declaration
    friend void fun();
}; // end of class sample1

class Sample 2 {
private:
    int x2, y2;
public :
    // member functions
    ---
    ---
    // declaration of friend
    friend void fun();
}; // end of Sample2

// non-member friend function of both classes
void fun() {
    // private data of sample1 and
    // private data of sample2 can be accessed here
    ---
    ---
} // end of function //bridge function
```

9. Destructors:

A destructor is a member function that is called **automatically** when the scope of an object goes out that is **just before the space allocated for member data is being released.**

Following rules are followed in designing a destructor:

- Destructor has the same name as that of class.
- It is preceded with tilde character (~)
- It has no return type.
- It has no parameter.
- It cannot be overloaded. Hence, there can be only one destructor in a class.

```

//dtor1.cpp : objects using English measurements
#include <iostream.h>

class Distance {                                //English Distance class
private:
    int feet;
    float inches;

public:
    Distance(int ft = 0, float in = 0.0) {
        feet = ft; inches = in;
        show();
        cout << endl << "Ctor called" << endl;
    }
    void show( )                                //display distance
    { cout << feet << "\'-" << inches << '\"' << endl; }
    ~Distance() {
        show();
        cout << endl << "Dtor called" << endl;
    }
}; // end of class
/////////////////////////////////////////////////////////////////
int main( ) {
    Distance dist1(1, 1.0F),
        dist2(2, 2.0F); // Ctor called here first for dist1, then for dist2

    cout << endl << endl << "Show being called" << endl;
    dist1.show();
    dist2.show();

    return 0;                                // dtor called here first for dist2, then for dist1
}

```

Hence, the constructors are called in the order of declaration of object, and the destructors are called in the reverse order of declaration.

Exercises:

1.) Create a class called **Rational** for performing arithmetic with fractions. Write a driver program to test your class. Use integer variables to represent the **private** data of the class—the numerator and the denominator. Provide a constructor function that enables an object of this class to be initialized when it is declared. The constructor should contain default values in case no initializers are provided and should store the fraction in reduced form (i.e., the fraction would be stored in the object as 1 in the numerator and 2 in the denominator). Provide **public** member functions for each of the following:

- a) Addition of two **Rational** numbers. The result should be stored in reduced form.
- b) Subtraction of two **Rational** numbers. The result should be stored in reduced form.
- c) Multiplication of two **Rational** numbers. The result should be stored in reduced form.
- d) Division of two **Rational** numbers. The result should be stored in reduced form.
- e) Printing **Rational** numbers in the form **a/b** where **a** is the numerator and **b** is the denominator.
- f) Printing **Rational** numbers in double floating-point format.

2). Modify the **Time** class in example to include a **tick** member function that increments the time stored in a **Time** object by one second. The **Time** object should always remain in a consistent state. Write a program that tests the **tick** member function in a loop that prints the time in standard format during each iteration of the loop

to illustrate that the **tick** member function works correctly. Be sure to test the following cases:

Design polar class. polar.cpp

- a) Incrementing into the next minute.
- b) Incrementing into the next hour.
- c) Incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

3) Modify the **Date** class in example to perform error checking on the initializer values for data members **month**, **day** and **year**. Also, provide a member function **nextDay** to increment the day by one. The **Date** object should always remain in a consistent state. Write a program that tests the function **nextDay** in a loop that prints the date during each iteration to illustrate that the **nextDay** works correctly. Be sure to test the following cases: ([Date3.cpp](#))

- a) Incrementing into the next month.
- b) Incrementing into the next year

4) Create a class **HugeInteger** that uses a 40-element array of digits to store integers as large as 40-digits each. Provide member functions **input**, **output**, **add** and **subtract**. For comparing **HugeInteger** objects, provide functions **isEqualTo**, **isNotEqualTo**, **isGreaterThan**, **isLessThan**, **isGreaterThanOrEqualTo** and **isLessThanOrEqualTo**— each of these is a “predicate” function that simply returns **true** if the relationship holds between the two huge integers and returns **false** if the relationship does not hold. Also, provide a predicate function **isZero**. If you feel ambitious, provide member functions **multiply**, **divide** and **modulus**.

5) Create a class **TicTacToe** that will enable you to write a complete program to play the game of tic-tac-toe. The class contains as **private** data a 3-by-3 double-subscripted array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square; place a 2 wherever the second player moves. Each move must be to an empty square. After each move, determine if the game has been won or if the game is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three dimensional tic-tac-toe on a 4-by-4-by-4 board (Caution: This is an extremely challenging project that could take many weeks of effort!).

6) Design a Student class, containing members:

- roll number as integer (from 1 to 1000)
- name as string
- percentage of marks as float (0-100)

Give provision to have a count of students created so far ([hint – use static member](#))

Q: What is meaning of const class?

Pointer and Dynamic Allocation

1. Introduction & Definition:

A pointer is a variable that contains an address. There are two operators related with pointers:

- & (ampersand operators) – “address of”
- * (pointer or indirection) – “Value contained in”
- also called *dereferencing* or *indirection* operator

Each variable occupies space in main memory to store its value. The start address of these locations is stored in symbol table. Any access to the variable refers into the symbol table.

A pointer is also a data type like other variable. Space is also allocated for it. A pointer variable contains an address. A pointer may be of any standard data type, like:

- Pointer to character
- Pointer to integer
- Pointer to float
- Pointer to double float
- Pointer to structure
- Pointer to union
- Pointer to pointer

NOTE: We can declare an integer pointer as:

```
int * px;
```

When we say like this, we have declared *px* (not **px*); the type of *px* is “*int **” (pointer to integer). **The space will be allocated for *px* only, not for **px*.** *px* will hold the start address of a location where an integer is stored.

Ex 1. The example demonstrates the step-wise operations:

```
//ex1.c
int main() {
    1) int x ;           /* an integer */
    2) int * px ;        /* pointer to integer */

    3) x = 10 ;
    4) px = &x ;         /*assign address of x into pointer px */
    5) cout << x << *px << endl;

    6) ++x;
    7) ++(*px);
    8) cout << x << *px << endl;

    9) return 0;
}
```

1) `int x; /* allocate space for x */`

2) `int *px; /*allocate space for px */`

3) `x = 10;`

4) `px = &x; /*address of x goes to px */`

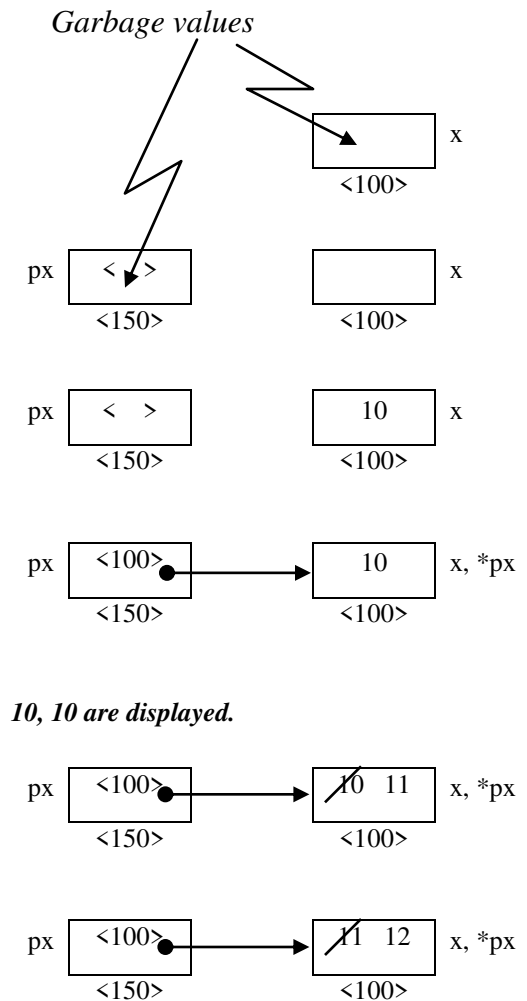
5) `cout << x << *px << endl;`

6) `x++;`

7) `(*px)++;`

8) `cout << x << *px << endl;`

9) `return 0;`



10, 10 are displayed.

12, 12 are displayed.

*/*space for x and px is released*/*

When `px` starts pointing `x`, the locations `x` and `*px` are same. Any changes done in any one will be reflected to the other.

Explain:

1. the difference between accessing the location by saying `x` and by `*px`.

2. the meaning of `*px++`.

Notes:

1. When we use pointer, there are two locations:

- Pointer location `px`, it contains address
- Pointed location `*px`, it contains data value

When we declare a pointer as:

`int *px;`

When we say like this, we have declared `px` (not `*px`); the type of `px` is “`int *`” (pointer to integer). **The space will be allocated for `px` only, not for `*px`.** `px` will hold the start address of a location where an integer is stored. The data will be stored at `*px`. **Before any operation, we must ensure that the space is reserved for `*px`.** ([Show ex3.cpp](#))

NOTE. SIZE OF POINTER:

There can be pointer of any standard data type like:

- Pointer to character
- Pointer to integer
- Pointer to double float
- Pointer to structure
- Pointer to array
- Pointer to pointer etc.

The size of pointer is independent of the size of the data type to which it points. It means that the size of pointer of a character variable and the size of pointer for a long double float variable are same.

It is because pointer is simply the start address. The size of address will be same for any type of data (**equal to 16 bits or 2 bytes/4 byte**). The address size is independent of data type; it only depends upon:

- the machine
- the memory model.

Ex:

```
//incrInPtr.cpp : increment in pointer and size of pointer
#include <stdio.h>
int main( ) {
    char *pc ;
    int *pi;
    double *pd;

    char c = 'a';
    int i = 20;
    double d = 3.14;

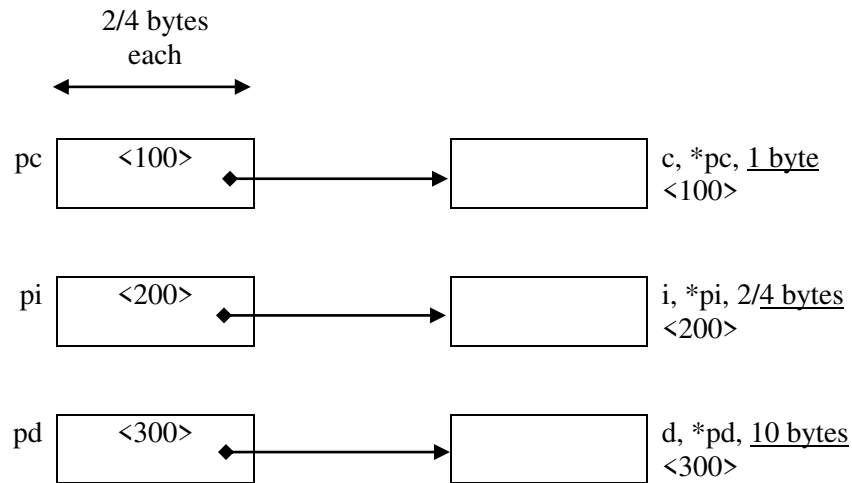
    clrscr();
    printf("\n %d %d %d", sizeof(pc), sizeof(pi), sizeof(pd) );
    printf("\n %d %d %d", sizeof(c), sizeof(i), sizeof(d) );

    pc = &c;
    pi = &i;
    pd = &d;

    printf("\n %u, %u, %u", pc, pi, pd);

    ++pc;
    ++pi;
    ++pd;

    printf("\n %u, %u, %u", pc, pi, pd);
    printf("\n%c, %d, %lf", *pc, *pi, *pd); //will display garbage
    return 0;}
```

In case of character pointer only one byte from the start location is to be accessed.
In case of integer pointer two bytes from the start location are to be accessed.
In case of long double float pointer ten bytes from the start location are to be accessed.

Note:

Due to the above said reasons we can not mix the pointer and the data to which it points.

It means that we can not use character pointer to point an integer.

When a pointer is incremented, it points to the next data location of same type.

so

increment in character pointer results in addition of 1

“ “ integer “ “ 4

“	“long double float	“	“	10
---	--------------------	---	---	----

In general, due to increment in a pointer of a data type:

new value of pointer = old value + size of data type to which pointer points.

2. Void Pointer:

Void is a pointer that can point to any data type. It means that address of any data type can be assigned to void pointer.

But following operations **can not** be done on a void pointer:

1. Dereferencing of the pointer.

```
void *p;  
int x;  
p = &x;           // possible  
*p = 10;          // not possible
```

2. Change in the value of pointer.

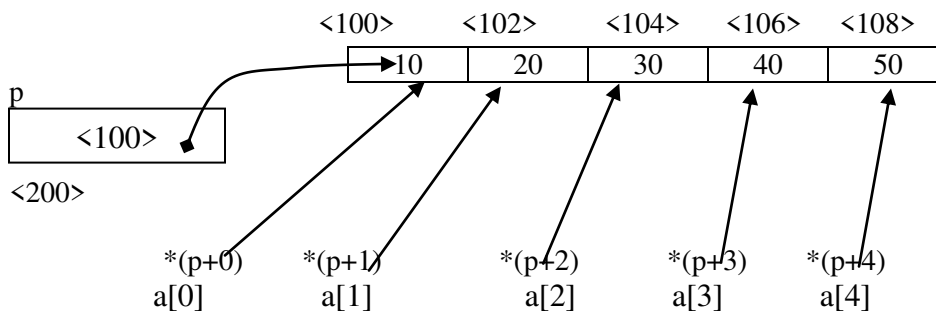
```
void *p;  
char x;  
p = &x;           // possible  
++p;              // not possible
```

[show example
readDispArray.cpp](#)

3. Pointer and Array:

The name of the array is the base or start address of the array. The subscript specifies the offset (distance). We can use a pointer to point elements of the array.

```
//arrPtr.cpp  
#include <stdio.h>  
  
int main( )  
{  
    int i;  
    int a[ ] = { 10, 20, 30, 40, 50};  
    int *p;  
  
    p = a; /*store base address of first element of array in to pointer */  
  
    /* now p can be used to access array elements */  
    for ( i = 0; i < 5; ++i )  
        cout << *(p+i) << i[a] << *(a+i) << p[i]; /*display current element*/  
    return 0;  
}
```



in general, $*(p+i)$ will give access to $a[i]$ elements

Note: The address of element $a[i]$ is represented as $\&a[i]$. The same can be written as $(p+i)$ also.

4. Dynamic Allocation:

When we say

```
int * p ;
```

We have declared a variable p which is $int *$ type; it means that the data type of p is pointer to integer. Here p is local automatic variable, it is statically allocated. There is no dynamic allocation for p . The space for p will be reserved as soon as the execution enters in the function. The space for ' p ' is released as soon as the execution comes out of the function. Since, ' p ' is local variable of a function, that is why its space is allocated on the stack.

Normally, allocation information is decided in compilation time. During execution the space is allocated for variable according pre-decided information. During execution, it can not be changed. It is static allocation.

Dynamic allocation means the allocation information is completely decided in execution time. It means that, the user can decide how many bytes are to be allocated.

The decision of allocation information of memory which is done by the system during execution time is called dynamic allocation. User can decide the number of bytes he wants to reserve. It provides flexibility and better utilization of memory.

Various ways of dynamic allocation:

discuss examples:

1. malloc.cpp
2. dyna1.cpp (given)

C Style	C++ Style
For allocation: malloc()	For allocation: New
For releasing: free()	For releasing: Delete

Here is the difference between the way used in C and the way used in C++:

	C Style	C++ Style
1. Type	malloc() and free() are functions	<i>new</i> and <i>delete</i> are operators.
2. Syntax – allocating space for a single unit.	int *p; p = (int *)malloc(sizeof(int)); < use *p > free(p);	int * p; p = new int; < use *p > <i>delete p;</i>
3. Syntax – allocating space for multiple units.	int *p; p = (int *)malloc(n*sizeof(int)); < use *p > free(p);	int * p; p = new int[n]; < use *p > <i>delete [] p;</i>
4. Allocation for an object of class Distance	Distance *p; p = (Distance *)malloc(sizeof(Distance)) <i>//no constructor is called</i>	Distance* p; p = new Distance; <i>//constructor is called</i>
5. Freeing memory for an object of class Distance	<i>free(p);</i> <i>//no destructor is called</i>	<i>delete p;</i> <i>//destructor is called</i>
6. When memory is not allocated	only NULL is returned	NULL is returned as well as an exception is also thrown

Keeping in view the above points, we shall use only *new* and *delete* here.

Example 1:

```
//dyna1.cpp
#include <iostream.h>
int main( ){
    I). int *px;

    /* allocate memory for one integer */
    II). px = new int;
    if ( px == NULL ) {                //Error
        cout << " Memory not allocated ";
        exit(1);
    }
    III). *px = 10;
    IV). (*px)++;
    V). cout << *px;
    VI). delete px;                /* release memory *px */

    VII). return 0;
}
```

Explain the operations with figure.

Memory leaks/holes: If the pointer of dynamically allocated memory is lost, that memory is not accessible. Such a memory can neither be accessed, nor can it be allocated to other variables. So, it is a lost memory. It is the most important point to worry. So, if a memory is allocated dynamically, it must be released also at proper time. (explain dyna1.c of C-language)

```
//dyna2.cpp
# include <iostream.h>

int main( ) {
    int *p, n, i;

    cout << "Enter the size: "; cin >> n;
    p = new int[n];                // allocated space for n integer *
    if (px == NULL ) {            // memory not allocated
        cout << "\n memory not allocated ";
        exit(1);                  // exit from program
    }
    // now p can be treated as an array of size n, read from keyboard
    for ( i = 0; i < n, ++i) {
        cout << "Enter the element: ";
        cin >> *(p + i);          // OR cin >> p[i];
    }
    /* display it */
    for ( i = 0; i < n; ++i)
        cout << *(p+i) << endl;    // OR  cout << p[i] << endl;

    delete[] p;                // memory for *px is released, not of px
    return 0;
}
```

5. Accessing object with pointers:

```
//distance1.cpp
#include <iostream.h>

class Distance {           //English Distance class
private:
    int feet;
    float inches;

public:
    Distance(int ft = 0, float in = 0.0){
        feet = ft; inches = in;
        show();
        cout << endl << "Ctor called" << endl;
    }

    void show( )           //display distance
    { cout << feet << "'-" << inches << "' " <<endl; }

    ~Distance(){
        show();
        cout << endl << "Dtor called" << endl;
    }

}; // end of class

////////////////////
int main( ){
    Distance d(1, 1.0F);    //ctor is called here
    Distance* dp;         //no ctor is called here
    dp = &d;
    d.show();               // membership operator . is used here
    dp->show();             // referential oepreator -> is used here

    cout << endl;
    return 0;               // dtor called here
}
```

6. Dynamic allocation for objects:

```
// distance2.cpp
#include <iostream.h>
#include <process.h>
#include <malloc.h>
class Distance
{
private:
    int feet;
    float inches;

public:
    Distance(int ft = 0, float in = 0.0): feet(ft), inches(in)
    {
    }

    void set() {
        cout << "Enter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }
}
```

```

void show( )                //display distance
{ cout << feet << "\'-" << inches << '\\"' << endl; }

~Distance( ) {
    cout << "Destructor call here for object: ";
    show( );
}

}; // end of class

////////////////////////////////////
int main( ){
Distance * d1,* d2, * d3;    //no ctor called here

d1 = new Distance( );        //default ctor called here
if (d1== NULL) {
    cout << "Memory not available" << endl;
    exit(1);
}

d2 = new Distance(10, 6.7f); //parameterized ctor called here
if (d2== NULL) {
    cout << "Memory not available" << endl;
    exit(1);
}

d3 = (Distance *)malloc(sizeof(Distance)); //ctor not called here
if (d3== NULL) {
    cout << "Memory not available" << endl;
    exit(1);
}

cout << endl <<endl << "Show being called" <<endl;
d1->show();
d2->show();
d3->show();

d3->set();
d3->show();

delete d1;    //dtor called here
delete d2;    //dtor called here
free(d3);     // no dtor called here

return 0;
}

```

Show the member data and
main() of student.cpp;
the give exercise to design the
complete class.

Exercise: Draw the stepwise figure for the following example:

```

//dyna3.cpp
#include <iostream.h>

class Sample{
    int* px;
public :
    // default constructor
    Sample(int x = 0)
    {
        cout << "Constructor executing, space being allocated "<< endl ;
        px = new int;           // allocate space for *px
        *px = x;
    }
}

```

First demonstrate its simpler version:

```

void main() {
    Sample s(10);
    s.display();
    s.set(50);
    s.display();
} dyna4.cpp

```

```

// setter function
void set (int x = 0)
{ *px = x ; }

//display function
void display ()
{ cout << *px << endl; }

// destructor
~Sample( ) {
    cout << endl << "Destructor executing, space being released "<< endl;
    delete px; //released member occupied by *px
}
}; // end of class

int main( ) {
Sample* sp; //space allocated for pointer only, not for object
            // constructor not called here

cout << endl << "CONSTRUCTOR CALLED AFTER IT" << endl;
sp = new Sample(100); // space allocated for object, i.e. for its member data
                     // constructor called here

sp->display( ); //display 100 i.e. *px
sp->set(500);
sp->display( );

delete sp; //space for object released, destructor called here

cout << endl << "DESTRUCTOR HAS BEEN CALLED BEFORE IT" << endl;

return 0; //scope of sp goes out
}

```

[show multiDistance1.cpp first](#)

7. Array of Objects:

```

// multiDistance2.cpp: accessing member functions by pointer
#include <iostream.h>
#include <malloc.h>
class Distance { //English Distance class
private:
    int feet;
    float inches;

public:
    Distance(int ft=0, float in = 0.0):feet(ft), inches(in)
    { cout << "Ctor called" << endl; }

    void getdist( ) { //get length from user
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }

    void showdist( ) //display distance
    { cout << feet << "\'-" << inches << '\\"' << endl; }

    ~Distance() {
        cout << "Dtor called" << endl;
        showdist();
    }
};

```

```

int main( ) {
    Distance* p;
    int n, i;

    n = 3;
    //allocate dynamically
    p = new Distance[n];           //c'tor called n times in the order

    //loop to read the data
    for (i = 0; i < n; ++i) {
        cout << endl << "Enter data for object" << i+1 << endl;
        p[i].getdist();           // or (p+i)->getdist();
    }

    //loop to display the data
    for (i = 0; i < n; ++i) {
        cout << endl << "You Enter data for object" << i+1 << endl;
        (p+i)->showdist(); //or p[i].showdist();
    }

    delete []p; //dynamically release the memory
                // dtor called n time in the reverse order

    return 0;
}

```

**Differentiate between
delete [] p; and
delete p;**

8. Array of pointer to objects:

```

// person.cpp : array of pointers to objects
#include <iostream.h>

class Person {                               //class of persons
    char name[40];                           //person's name
    int age;                                 //person's age
public:
    void setData( ) {                       //set the name and age
        cout << "Enter name: "; cin >> name;
        cout << "Enter Age: "; cin >> age;
    }

    void printData( ) {                     //display the name and age
        cout << "\nName: " << name
            << ", Age: " << age;
    }
};
//-----
int main( ){
    Person* persPtr[10];                    //array of pointers to persons
    int n = 0;                              //number of persons in array
    char choice;

    do {
        persPtr[n] = new Person;            //put persons in array
        persPtr[n]->setData( );             //make new object
        n++;                                //set person's name and age
                                           //count new person

        cout << endl;
        cout << "Enter another (y/n)? ";    //enter another person
        cin >> choice;
    } while( choice == 'y' );                //quit on 'n'
}

```



```

//loop to number of employees
for(int j=0; j<n; j++){
    cout << "\n\nPerson number " << j+1;
    persPtr[j]->printData( );
}

//loop to dynamically deallocate the space
for (j = 0; j < n; ++j)
    delete persPtr[j];

cout << endl;
return 0;
} //end main

```

Exercise: Modify the above class so that it maintains the number of Person object currently in existence. (personCount.cpp)

9. Pointer and const:

```

//constness.cpp :This program will not compile
#include <iostream.h>

int main( ){
    int x0 = 0; x1 = 10, x2 = 20, x3 = 30;

    int* p0;                //p0 variable, *p0 variable
    const int* p1;          //p1 variable, *p1 constant
    int* const p2;          //p2 constant, *p2 variable
    const int* const p3;    //p3 constant, *p3 constant

    p0 = &x0;
    p1 = &x1;
    p2 = &x2;
    p3 = &x3;

    //BEHAVIOR OF P0
    (*p0)++;                //value changes from 0 to 1
    p0++;                  //p1 changes, now not pointing to x

    //BEHAVIOR OF P1
    (*p1)++;                //ERROR: change not permitted
    p1++;                  //p1 changes, now not pointing to x

    //BEHAVIOR OF P2
    (*p2)++;                //contents of x changes to 21,
                           //target location of pointer p2 not constant
    p2++;                  //ERROR: p2 can not change, it is a const pointer

    //BEHAVIOR OF P3
    (*p3)++;                //ERROR: target location constant
    p3++;                  //ERROR: p3 can not change, it is a const pointer

    cout << endl << *p0 << *p1 << *p2 << *p3 << endl;
    return 0;
}

```

9. Self Referential Classes

If a class contains the data member as, pointer to object of **same class**, then it is called a self-referential-class.

Ex.

```
class node {
    int    info;
    node* next; //pointer to object of same type
public :
    |
    |
    |
};
```

10. Mutable Data Members:

Mutable data member is that member which can **always be changed**; even if the object is “const” type. It is just opposite to “const” data that can never be changed.

The following example will clear the concept:

```
//mutable.cpp : program to demonstrate the mutable behaviour
#include <iostream.h>

class Sample {
private:
    int x;                                //non-mutable member
    mutable int y;                        //mutable member
public:
    Sample(int xx = 0, int yy = 0)
    { x = xx; y = yy; }

    //function to set value of only x
    void setx(int xx =0)
    { x = xx; }

    //function to set value of only y
    //value of y being changed, even if member function is constant
    void sety(int yy =0) const
    { y = yy; }

    //fuction to display x and y,
    //this has to be const type, if used with const objects
    void display() const
    {      cout << endl << "x: " << x  << " y: " << y << endl; }
};

int main(){
    const Sample s(10, 20);           //A const object

    cout << endl << "Value before change: ";
    s.display();

    s.setx(100);                      //ERROR: x can not be changed,
                                     //because object is constant
    s.sety(200);                      //y still can be changed,
                                     //because y is mutable
    cout << endl << "Value after change: ";
    s.display();
    return 0;
}
```

An example: Draw the stepwise figure of the following and tell the drawback.
(shallow copying)

```
//copyCtorNeed.cpp
#include <iostream.h>
class Sample {
private:
    int* px;
public:
    //default constructor
    Sample(int x = 0) {
        px = new int ;
        *px = x;
    }

    //setter function
    void set(int x = 0)
    { *px = x; }

    //display function
    void display( )
    { cout << *px << endl; }

    //distructor
    ~Sample( )
    { delete px; }
};
//-----
int main( ) {
    Sample s1(50);

    cout << endl << endl << "Before if: ";
    cout << endl << "s1: "; s1.display( );

    if (1) { //begin a new scope
        Sample s2 = s1; //no ctor called here for s2
                       //s1 is assigned memberwise to s2

        cout << endl << endl << "Inside if: ";
        cout << endl << "s2: "; s2.display( );

        s2.set(100);
        cout << endl << "s1: "; s1.display( );
        cout << endl << "s2: "; s2.display( );
        //scope of s2 goes out
        //destructor of s2 called, space for *px of s2 released.
    }

    cout << endl << endl << "Out of if: ";
    cout << endl << "s1 : "; s1.display( );
    cout << endl;

    return 0; //scope of s1 goes out
    //destructor of s1 called, space for *px of s1 released.
}
```

11. Copy Constructor: (deep and shallow copying)

It is a constructor with a single parameter which is the reference of the existing object of same class. It is used for initializing a new object with the existing object i.e. to create clone/replica of existing object.

The copy constructor has following essential properties:

- Copy constructor has a single parameter.
- The parameter is the reference of object of same class.
- The parameter must be passed by reference and “const” by convention.

Normally, the initialization of an object to another object of same class can be done with the help of assignment operator. In case of objects, the assignment is member-wise. But this usual process fails in case of dynamic allocation. In that case copy constructor is used.

Copy constructor “initializes” a new object with an existing object. Initialization is a two step process:

- Memory allocation.
- Member to member assignment.

The following example will clear the concept:

```
//copyCtor.cpp
#include <iostream.h>

class Sample
{
private:
    int* px;
public :
    //default constructor
    Sample(int x = 0)
    {
        px = new int ;
        *px = x;
    }

    //copy constructor
    Sample(Sample& subj)
    {
        px = new int; //allocates memory for new object
        *px = *(subj.px); //assign from existing object.
    }

    //setter function
    void set(int x = 0)
    { *px = x; }

    //display function
    void display( )
    { cout << *px; }

    //distructor
    ~Sample( )
    { delete px; }
};

/////
int main( )
{
    Sample s1(50);
    cout << endl << endl << "Before if: ";
    cout << endl << "s1: "; s1.display( );
}
```

Why is it necessary to have reference of existing object as parameter in a copy constructor?

```

if (1) {
    //begin a new scope
    Sample s2(s1);    //copy constructor called
    // there will be separate data space for *px of s2
    cout << endl << endl << "Inside if: ";
    cout << endl << "s2: "; s2.display( );

    s2.set(100);
    cout << endl << "s1: "; s1.display( );
    cout << endl << "s2: "; s2.display( );

    //scope of s2 goes out
    //destructor of s2 called, space for *px of s2 released.
}

cout << endl << endl << "Out of if: ";
cout << endl << "s1 : "; s1.display( );
cout << endl;

return 0;    //scope of s1 goes out
//destructor of s1 called, space for *px of s1 released.
}

```

12. The “this” Pointer:

The “this” is the default pointer of that object with which the member function is called. In other words, the “this” pointer contains the address of current object, **inside the member function**. Current object is the object with which the member function is called.

The this pointer exists only inside the member function.

An example will demonstrate the behavior of this pointer:

```

// the this pointer referring to data
#include <iostream> //this.cpp

class What {
    int alpha;
public:
    void tester( ) {
        this->alpha = 11;    //same as alpha = 11;
        cout << alpha;    //same as cout << alpha;
    }
};

//main function
int main( )
{
    What w;
    w.tester( );    //this will be the pointer of w
    cout << endl;
    return 0;
}

```

one more example on this:

The following class has a function that displays the base address of the current object.

```

//reveal.cpp: the demonstration of this pointer
#include <iostream.h>

class Where {
    char chararray[10]; //occupies 10 bytes
public:
    void reveal( )      // displays the address of current object
    { cout << "\nMy object's address is " << this; }
};

//main function
int main( )
{
    Where w1, w2, w3;      //make three objects

    //see where they are, i.e. there addresses
    w1.reveal( );          //this is the pointer of w1
    w2.reveal( );          //this is the pointer of w2
    w3.reveal( );          //this is the pointer of w3
    cout << endl;
    return 0;
}

```

Note: There is no significance of this pointer with static members. It will be a syntax error to use this pointer with static members. (why?)

The self assignment of object may generate serious errors if dynamic allocation is used. It can be prevented with the use of **this**.

Using this to enable cascaded function call:

Multiple member functions can be called in a single statement, if the member functions return the dereferenced location by this pointer. It is called **cascaded call**. The following example will clear the idea.

```

// cascaded call //time.cpp
class Time {
private:
    int hour;      // 0 - 23 (24-hour clock format)
    int minute;    // 0 - 59
    int second;    // 0 - 59

public:
    Time( int = 0, int = 0, int = 0 ); // default constructor

    // set functions
    Time &setTime( int, int, int ); // set hour, minute, second
    Time &setHour( int );          // set hour
    Time &setMinute( int );        // set minute
    Time &setSecond( int );        // set second

    // get functions (normally declared const)
    int getHour() const;          // return hour
    int getMinute() const;        // return minute
    int getSecond() const;        // return second

    // print functions (normally declared const)
    void printUniversal() const;  // print universal time
    void printStandard() const;   // print standard time
}; // end class Time

```

```

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec ) {
    setTime( hr, min, sec );
}

// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s ){
    setHour( h );
    setMinute( m );
    setSecond( s );

    return *this; // enables cascading
}

// set hour value
Time &Time::setHour( int h ){
    hour = ( h >= 0 && h < 24 ) ? h : 0;

    return *this; // enables cascading
}

// set minute value
Time &Time::setMinute( int m ){
    minute = ( m >= 0 && m < 60 ) ? m : 0;

    return *this; // enables cascading
}

// set second value
Time &Time::setSecond( int s ){
    second = ( s >= 0 && s < 60 ) ? s : 0;

    return *this; // enables cascading
}

// get hour value
int Time::getHour() const {
    return hour;
}

// get minute value
int Time::getMinute() const {
    return minute;
}

// get second value
int Time::getSecond() const {
    return second;
}

// print Time in universal format
void Time::printUniversal() const {
    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
}

// print Time in standard format
void Time::printStandard() const{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
         << ":" << setfill( '0' ) << setw( 2 ) << minute
         << ":" << setw( 2 ) << second
         << ( hour < 12 ? " AM" : " PM" );
}

```

```

int main(){
    Time t;

    // cascaded function calls
    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );

    // output time in universal and standard formats
    cout << "Universal time: ";
    t.printUniversal();

    cout << "\nStandard time: ";
    t.printStandard();

    cout << "\n\nNew standard time: ";

    // cascaded function calls
    t.setTime( 20, 20, 20 ).printStandard();

    cout << endl;
    return 0;
} // end main

```

13. Proxy Classes:

Sometimes it is desirable to hide the implementation of a class to prevent access to propriety information like private data and program logic. It can be done with the help of a **proxy class** that knows only the public interface to class to be hidden. The user of class can use the full functionality of the class, but cannot see the implementation details. Following are the steps used in the example program:

1. Create the class containing the propriety information (the information to be hidden), say ***Implementation***.
2. Create a **proxy class** called ***Interface***, that essentially contains the following:
 - A private member data, ***ptr***, that is pointer to the ***Implementation*** class.
 - A parameterized constructor that dynamically instantiates an object of ***Implementation*** class – the constructor of ***Implementation*** calls is called automatically or it can be called explicitly.
 - A destructor that release the dynamically allocated space in the constructor.
 - Member function corresponding to each member function of the ***Implementation*** class.

```

//the class of which the implemenatin details are to be hidden
class Implementation {
private:
    int value;
public:
    Implementation( int v ) : value( v ) // initialize value with v
    { /* empty body*/ }

    void setValue( int v ) {
        value = v;
    }

    int getValue() const {
        return value;
    }
}; // end class Implementation

```



```

//The proxy class
class Implementation;      // forward class declaration

class Interface {
private:
    // requires previous forward declaration
    Implementation *ptr;

public:
    Interface( int );
    void setValue( int ); // same public interface as
    int getValue() const; // class Implementation
    ~Interface();
}; // end class Interface

```

```

// the definitions of the functions of proxy class
// constructor
Interface::Interface( int v ): ptr ( new Implementation( v ) ) // initialize ptr
{ /*empty body */ } // end Interface constructor

// call Implementation's setValue function
void Interface::setValue( int v ) {
    ptr->setValue( v );
}

// call Implementation's getValue function
int Interface::getValue() const {
    return ptr->getValue();
}

// destructor
Interface::~~Interface()
    delete ptr;
}

```

```

//main function
int main(){
    Interface i( 5 );

    cout << "Interface contains: " << i.getValue()
          << " before setValue" << endl;

    i.setValue( 10 );

    cout << "Interface contains: " << i.getValue()
          << " after setValue" << endl;

    return 0;
} // end main

```

Exercise:

1. Create a **SavingsAccount** class. Use a **static** data member to contain the **annualInterestRate** for each of the savers. Each member of the class contains a **private** data member **savingsBalance** indicating the amount the saver currently has on deposit. Provide a **calculateMonthlyInterest** member function that calculates the monthly interest by multiplying the **balance** by **annualInterestRate** divided by 12; this interest should be added to **savingsBalance**. Provide a **static** member function **modifyInterestRate** that sets the **static annualInterestRate** to a new value. Write a driver program to test class **SavingsAccount**. Instantiate two different **savingsAccount** objects, **saver1** and **saver2**, with balances of \$2000.00 and \$3000.00, respectively. Set **annualInterestRate** to 3%, then calculate the monthly interest and print the new balances for each of the savers. Then set the **annualInterestRate** to 4% and calculate the next month's interest and print the new balances for each of the savers.

2. It would be perfectly reasonable for the **Time** class of the example to represent the time internally as the number of seconds since midnight rather than the three integer values **hour**, **minute** and **second**. Clients could use the same public methods and get the same results. Modify the **Time** class to implement the **Time** as the number of seconds since midnight and show that there is no visible change in functionality to the clients of the class.

Operator Overloading

In C++ we can assign new task to existing operators; its called operator overloading. *e.g.* '+' can be assigned:

- for adding two complex numbers
- for concatenating two strings

The objective of operator overloading is to provide the facility to the user of the class to write expressions in the *most natural form*.

1. Rules of Operator Overloading

- I). The overloaded operator must match the syntax of the language. It means that unary operator can be overloaded only as unary operator; can not be overloaded as binary operators (and vice-versa). *e.g.* The operator "/" can not be overloaded as unary operators.
- II). We cannot invent new operator for overloading, it means that only existing operators can be overloaded. *e.g.* "\$" cannot be overloaded.
- III). We cannot overload the way an operator works with the intrinsic C++ data type. It means that operator overloading only works for user defined class. *e.g.* we cannot overload numeric addition operation by some other operator.
- IV). Overloading cannot change the precedence and associativity of the operator.
- V). Following operators cannot be overloaded:
 - a) . class membership operator.
 - b) .* pointer to member operator.
 - c) → * pointer to member operator.
 - d) :: scope resolution operator.
 - e) ?: ternary or conditional operator.

Guideline : we should overload an operator in the most intuitive way. It means that the new purpose of the operator should be similar to its basic purpose.

There can be two types of overloading:

- overloading of unary operators
- overloading of binary operators

An operator overloading function can be designed in two ways:

- Operator overloading member function
- Operator overloading nonmember (free) function.

Normally, overloading function is designed as member function, because it strengthens the concept of class & encapsulation. But sometimes, an overloading function cannot be designed as member function; in this case nonmember free function is designed.

2. Overloading Unary Operator

An example will clear the concept:

Ex. Design a counter class that contains the function to overload the pre-increment operator.

```
//counter1.cpp :Overload ++ in the simplest manner
#include <iostream.h>

class Counter {
private:
    int count;
public :
    //constructor
    Counter(int c = 0) : count(c)
    { }

    //setter function
    void set(int c = 0)
    {count = c;}

    //display function
    void display( )
    {cout << "count = " << count << endl; }

    //increment : overloaded ++
    void operator++( )
    {++count; }

    //decrement : overloaded --
    void operator--( )
    { --count ; }
}; //end of class

//main function
int main( ){
    Counter c1(100), c2;

    --c1;
    c1.operator--();

    ++c2;
    c2.operator++();

    c1.display( );
    c2.display( );
    return 0;
}
```

The expressions

- -c1; **OR** ++c2;

are in fact function calls. The compiler converts these expressions into elongated forms:

c1.operator--();

c2.operator++();

Even, the programmer can write the elongated forms also.

The above overloaded operator behave only in prefix manner. We cannot write

c1- -; or

c2++;

Note: The unary operators can be overloaded in both ways.

- prefix - type
- postfix – type

The above example has a drawback. We cannot do the operation like,
c2 = ++c1;
it is because the overloaded operator function doesn't return anything.

Ex: Modify the above class to support the assignment operation:

```
//counterWithAssign.cpp
//Counter class with ++ overloaded and assignment support
#include <iostream.h>
class Counter{
private:
    int count;
public :
    //constructor
    Counter(int c = 0) : count(c)
    {
    }

    //setter function
    void set(int c = 0)
    {count = c;}

    //display function
    void display( )
    {cout << endl << "count = " << count; }

    //function to increment with support to assignment
    Counter operator++( ) {
        Counter temp;
        temp.count = ++count;
        return temp;
    }

    //function to decrement with support to assignment
    Counter operator--( ) {
        Counter temp;
        temp.count = --count;
        return temp;
    }
}; //end of counter class

int main( ){
    Counter c1(100), c2;

    ++c1;                // enlarged c1.operator ++( )
    c2 = ++c1;           // enlarged c2 = c1.operator ++( )
    c1.display( );
    c2.display( );

    c2 = --c1;           // enlarged c2 = c1.operator --( )
    c1.display( );
    c2.display( );
    return 0;
}
```

3. Nameless Temporary Objects:

When we type casts an object to an object of another class, a temporary object of target class is created. This temporary object has no name. This concept can be used to reduce the code length sometimes.

In the counter class, the functions support to assignment operation can be redesigned.

```

// CounterNameLessTemp.cpp
//Counter class with ++ overloaded uses unnamed temporary object
#include <iostream.h>

class Counter{
private:
    int count;                //count
public:
    Counter( ) : count(0)     //constructor no args
    { }

    Counter(int c) : count(c) //constructor, one arg
    { }

    int get_count()           //return count
    { return count; }

    Counter operator ++ ( ){   //increment count
        ++count;

        //increment count, then return an unnamed temporary object
        // initialized to this count; constructor necessary
        return Counter(count);
    }
};

/////////////////////////////////////////////////////////////////
int main( ){
    Counter c1, c2;           //c1=0, c2=0

    cout << "\nc1=" << c1.get_count( );    //display
    cout << "\nc2=" << c2.get_count( );

    ++c1;                     //c1=1
    c2 = ++c1;                 //c1=2, c2=2

    cout << "\nc1=" << c1.get_count( );    //display again
    cout << "\nc2=" << c2.get_count( ) << endl;
    return 0;
}

```

The overloaded operator function essentially requires the constructor with one argument.
In fact, the expression:

return Counter(count);

in the overloaded operator function calls the one argument constructor.

Counter(int c) : count(c) *//constructor, one arg*
{ }

4. Postfix Notation in Unary Operators:

The overloading function in normal way only supports prefix type expressions. For example, increment of Counter type object is to be written:

+ +c1;

If we want the postfix support also

c1++;

then, we will have to prepare another function with a dummy unnamed “int” argument.

```

/* CounterPostfix.cpp: overloaded ++ operator in both prefix and
postfix without using the concept of nameless temporary object */
#include <iostream.h>
class Counter{
private:
    int count;          //count
public:
    Counter() : count(0) //constructor no args
    { }
    Counter(int c) : count(c) //constructor, one arg
    { }
    int get_count( ) const //return count
    { return count; }

    //increment count (prefix)
    Counter operator ++( ) {
        Counter temp;
        temp.count = ++count;
        return temp;
    }

    //increment count (postfix)
    Counter operator ++(int) {
        Counter temp;
        temp.count = count;
        ++count;
        return temp;
    }
};
//////////
int main( ){
    Counter c1, c2;                      //c1=0, c2=0

    cout << endl << "After initialization: ";
    cout << "\nc1=" << c1.get_count( ); //display
    cout << "\nc2=" << c2.get_count( );

    ++c1;                                //c1=1
    c2 = ++c1;                           //c1=2, c2=2 (prefix)

    cout << endl << endl << "After pre-increment: ";
    cout << "\nc1=" << c1.get_count( ); //display
    cout << "\nc2=" << c2.get_count( );

    c1++;                                //c1 = 3
    c2 = c1++;                           //c1=4, c2=3 (postfix)

    cout << endl << endl << "After post-increment: ";
    cout << "\nc1=" << c1.get_count(); //display again
    cout << "\nc2=" << c2.get_count() << endl;
    return 0;
}

```

5. Non-Member Operator Overloading Function:

The overloading functions can also be made as non-member free functions. Such functions access private members of the class, so these should be declared as friend of the class. But, remember that overloading function should be designed as member function as far as possible, because it strengthens the concept of class & encapsulation.

Ex: The increment operator overloading function can be made out of the class as non-member function.

```

// counterNonMember.cpp : non member operator overloading function
#include <iostream.h>
class Counter{
private:
    int count;

public:
    Counter( ) : count(0)           //constructor no args
    { }
    Counter(int c) : count(c)       //constructor, one arg
    { }
    int get_count( )                //return count
    { return count; }

    //friend declarations
    friend Counter operator ++(Counter &c);
    friend Counter operator++(Counter &c, int );

};

//-----
// non member operator overloading function for pre-increment
// parameter c is call be reference to reflect the change
Counter operator++(Counter &c ) { //increment count
    ++c.count;                      //increment count, then return
    return c;                       //return the object
}

//-----
// non member operator overloading function for post-increment
// parameter c is call be reference to reflect the change
Counter operator++(Counter &c, int ) { //increment count
    Counter temp;
    temp.count = c.count;           //assign count of parameter to count of temp
    ++c.count;                      //now increment count of parameter object
    return temp;                   //return temp object
}

//-----
int main( ){
    Counter c1, c2;                 //c1=0, c2=0

    cout << endl << "After initialization:";
    cout << "\nc1=" << c1.get_count( ); //display
    cout << "\nc2=" << c2.get_count( );

    ++c1;                          //c1=1
    c2 = ++c1;                     //c1=2, c2=2

    cout << endl << endl << "After increament:";
    cout << "\nc1=" << c1.get_count( ); //display again
    cout << "\nc2=" << c2.get_count( ) << endl;

    ++c1;                          //c1=3
    c2 = c1++;                     //c1=4, c2=3

    cout << endl << endl << "After increament:";
    cout << "\nc1=" << c1.get_count( ); //d
    cout << "\nc2=" << c2.get_count( ) << endl;

    return 0;
}

```

Demonstrate
counterNonMem1.cpp
first

Demonstrate the
example again by
removing reference

6. Overloading Binary Operator:

Like unary operator, binary operator can also be overloaded in both ways.

- As member function
- As non-member function

(a) In case of member function, the prototype is:

return_type class-name::operator @(operand2);

It will be used as:

operand1 @ operand2

The equivalent enlarged call is:

operand1.operator@operand2;

So, for the overloaded operator:

- the first argument is the object with which the member function is called.
- the second argument is the object which is appearing as the parameter to the function.

(b) In case of non-member free function, the prototype is:

return_type operator @(operand1, operand2);

It will be used as:

operand1 @ operand2

The equivalent enlarged call is:

operator@(operand1, operand2);

So, for the overloaded operator:

- the first argument is the first object which is appearing as the parameter to the function.
- the second argument is the second object which is appearing as the parameter to the function.

Ex Design a complex class with following functions:

- constructor
- setter function
- display function
- function to overload + operator for adding 2 complex numbers.

```
//complex1.cpp
class complex{
    float real, img;
public :
    //constructor
    complex(float r=0.0, float i=0.0)
    { real = r; img = i; }

    //setter function
    void set(float r=0.0, float i=0.0)
    { real = r; img = i; }

    //display function
    void display(void)
    { count << "real:" << real
        << "imaginary:" << img<< endl;}

    //prototype of + operator overloaded
    complex operator+(complex c2);
};
```

```
//definition outside the class
complex Complex::operator+(complex c2){
    Complex temp;
    temp.real = real + c2.real; //add real part
    temp.img = img + c2.img; //add imaginary part
    return temp;
    //return resultant object
}

//-----
int main( )
{
    Complex c1(4, 5.2), c2, c3;

    c2.set(7, 4.9);
    c3 = c1 + c2;
    //the enlarged call: c3 = c1.operator+(c2);

    c3.display( );
    return 0;
}
```

Exercise: (1) Design the above with non-member function (complexNonMember.cpp)

(2) Design a class to support following main function.

```
int main( ){    //complexWithAssign.cpp
    Complex  c1(1, 1.1F), c2, c3;

    c2.set(2, 2.2F);

    c3 = c1 += c2;    //the enlarged call:  c3= c1.operator+=(c2);

    c1.display( );
    c3.display();

    return 0;
}
```

Ex: Design a complex class with following functions:

- constructor
- setter function
- display function
- function to overload + operator for adding 2 complex numbers.
- function to overload * operator to multiply 2 complex nos.
- functions for multiplication of a complex and a float number; it should be supported in commutative way, i.e.,
(*complex * float*) and (*float * complex*)

```
/complex2.cpp
#include <iostream.h>
class Complex{
private:
    float  real,  img;
public :

    Complex(float r=0.0, float i=0.0) { //constructor
    {  real = r;  img = i;  }

    void  set(float r=0.0, float i=0.0) //setter  function
    {real = r;  img = i;  }

    void  display(void)          //display  function
    {  cout <<  " real: " << real << "  imaginary: " << img << endl;}

    // +  operator  overloaded for addition of two complex numbers
    Complex  operator+(Complex c2);

    // * operator  overloaded for multiplication of two complex numbers
    Complex  operator*(Complex c2);

    // * operator  overloaded for multiplication of complex and float
    Complex  operator*(float x);

    //nonmember friend function for multiplication of float & Complex
    friend Complex operator*(float x, Complex c); //remove
};
```

```

/member function for adding two complex numbers
Complex Complex::operator+(Complex c2){
    Complex temp;
    temp.real = real + c2.real;    //add real part
    temp.img = img + c2.img;       //add imaginary part
    return temp;                   //return resultant object
}

//member function to multiply two numbers
Complex Complex::operator*(Complex c2){
    Complex temp;
    temp.real = real * c2.real - img * c2.img;
    temp.img = real * c2.img + img * c2.real;
    return temp;
}

//Member function to perform:    complex * float
Complex Complex::operator*(float x){
    Complex temp;
    temp.real = x * real;
    temp.img = x * img;
    return temp;
}

//-----
//Non-member free function to perform operation:    float * complex
Complex operator*(float x, Complex c){
    return (c*x); //it is a call to member function:    c.operator*(x)
} //no need to design a separate detailed body

//-----
int main( )
{
    Complex c1(5, 4.0F),    c2,    c3, c4, c5, c6;

    c2.set (4, 7.2F);

    c3 = c1 + c2;    //enlarged call:    c3 = c1.operator+(c2);
    cout << endl << "c1 + c2 = ";
    c3.display( );

    c4 = c1 * c2;    //enlarged call:    c4 = c1.operator*(c2);
    cout << endl << "c1 * c2 = ";
    c4.display( );

    float x1 = 2.0,    x2 = 10.0;
    c5 = c1 * x1;    //enlarged call:    c5 = c1.operator*(x);
    cout << endl << "c1 * x1 = ";
    c5.display();

    c6 = x2 * c1;    //enlarged call:    c5 = operator*(x, c1);
                    //non-member function called this time
    cout << endl << "x2 * c1 = ";
    c6.display();
    return 0;
}

```

Note:

The multiplication member function:

```
// * operator overloaded for multiplication of complex and float
```

```
Complex operator*(float x);
```

can only do the following operation:

```
(Complex * float)
```

This member function can not support the commutative operation:

```
(float * Complex)
```

But, in our example the second operation is supported due to non-member free function (a friend of the class):

```
//non-member friend function for multiplication: (float* Complex)
```

```
friend Complex operator+(float x, Complex c);
```

This function can never be a member function of the class, because the first argument is the intrinsic data type “float”. So, here the non-member operator overloading function is necessary.

Exercises:

Design a number class for the following main function:

```
void main( ) {
    Number x, y, r1, r2, r3, r4;

    x.set(20.0);
    y.set(10.0);

    r1 = x+y;
    r2 = x-y;
    r3 = x*y;
    r4 = x/y;

    r1.display();
    r2.display();
    r3.display();
    r4.display();
}
```

(1) Design a class Date that includes following data members :-

dd, mm, yy

(date.cpp & date1.cpp)

It has the following member function :-

- Constructor, that initializes the date to 1/1/2000 by default.
- Setter function; Display function.
- **next() – it sets the current date to the next date (consider about month and year change). It will act the basis for remaining functions. Design prev() also.**
- Overloaded – operator to subtract 2 dates. This function returns number of days between dates.
- Overloaded + operator to add a date in days or days in date, that is, it should be commutative. The function returns the new date.
- Overload += and -= operator
- Overload >, <, == operators to compare two dates.
- Design a check() function also; if a date is invalid, it is automatically set to 1-1-2000. The class permits the date for year range 1900 to 2100 .

Hint: Take static array inside the class to store days in a month as:

```
static int day-in-month[12]={31, 28, 31, 30, ...};
```

(2) Design a class Distance that includes following data members :-

feet, inches

It has the following member function :-

- Constructor, that initializes the distance to 0, 0 by default. Give a check so that the inches part is always less than 12.0.
- Setter function. Give a check so that the inches part is always less than 12.0.
- Display function.
- Overloaded – operator to subtract 2 distances.
- Overloaded + operator to add two distances
- Overload += in the following manner: d3 += d2 += d1;
- Overload > and < operators to compare two distances.

[Distance.cpp](#)

Ex. Design a class ‘safe array’, when we specify an index which is out of range, it should produce error message. It contains following data members - An array of integer of size MAX, Current size – which is less than the maximum size.

Design following member functions:

1. Constructor -It specifies the current size of 10 (by default). it should also initializes all current element by a specified value which is zero by default.

2. A function to set a specified element by the specified value.

3. A function to return the value of the specified element.

The two function must check the array bounds.

```

//safeArray1.cpp : program for demonstrating an array with the size constraints
#include <iostream.h>
#include <process.h>
const int MAX = 10;

class SafeArray{
private:
    int a[MAX];
    int size;          // size <= MAX

public :
    //constructor
    SafeArray(int s = 10, int val = 0)  {
        if (s > MAX)                    {
            cout << endl << "ERROR: Impossible current size\n";
            exit(1);                    //exit with error code
        }
        size = s;
        //initialize all elements.
        for (int i=0; i<size; i++)
            a[i] = val;
    }

    //function to set an element
    void setel(int i, int val){
        if ( (i < 0) || (i >= size) ) { //out of range error
            cout <<endl << "ERROR: out of bounds\n";
            exit(1);                    //exit with error code
        }
        a[i] = val;                    //otherwise assign the element
    }

    //function to return an element
    int getel( int i ) {
        if ( ( i< 0) || (i >= size) ){ //out of range error
            cout << endl << "ERROR: out of bound\n";
            exit(1);                    //exit with error code
        }
        return a[i];                    //otherwise return element
    }
}; // end of class

//-----
//main function
int main( ){
    int i;
    SafeArray sa(5, 0);                //size = 5

    for (i = 0; i < 5; i++)             //assign element
        sa.setel (i, 100*i)

    for (i = 0; i < 5; i++)             //display elements
        cout << sa.getel(i) << endl;

    //sa.setel (10, 700);                //ERROR
    //cout << endl << sa.getel(-5);      //ERROR

    return 0;
}

```

7. A function returning reference:

A function that returns the reference of a variable, can appear at the left hand side of assignment operator (as l-value). But, due to life time aspects, the variable whose reference is being returned may only be:

- either global variable
- or static variable

Such a function can appear on the left hand side of assignment operator.

```
//program demonstrates the return of reference from a function
// it means simply the item being returned is being accessed
// such function call can appear as lvalue
#include <iostream.h> //returnRef.cpp

int x;           // a global variable
int& setx(void); //prototype of function returning reference

int main() {
    x = 10;
    int y;

    y = setx();
    cout << endl << "1: Value of x is: " << y << endl;

    setx() = 20;
    y = setx();
    cout << endl << "2: Value of x is: " << y << endl;
    return 0;
}

//definition of the function returning reference
int& setx(void) {
    return x;
}
```

8. Overloading Subscript Operator:

The subscript operator [] is used to access array elements and string elements.

Let array is an object of a class. We want to assign value x to ith element. It should look like as:

array[i] = x;

The elongated call is:

array.operator[](i) = x;

The function call is appearing at the left hand side of the assignment operator. It is only possible when the function returns the reference of an element.

```
//safeArray2.cpp : program for demonstrating an array with the size
constraints
#include <iostream.h>
#include <process.h>
const int MAX = 10;

class SafeArray{
private:
    int a[MAX];
    int size;           // size <= MAX
public :
    //constructor
    SafeArray(int s = 10, int val = 0);
    int& operator[]( int i );
};
```

```

//ctor
SafeArray::SafeArray(int s = 10, int val = 0){
    if (s > MAX) {
        cout << endl << "ERROR: Impossible current size\n";
        exit(1);          //exit with error code
    }
    size = s;
    //initialize all elements.
    for (int i=0; i<size; i++)
        a[i] = val;
}

//function to overload subscript [ ] operator
int& SafeArray::operator[](int i){
    if ( ( i< 0) || (i >= size) ) { //out of range error
        cout << endl << "ERROR: out of bound\n";
        exit(1);          //exit with error code
    }
    return a[i];          //otherwise return element
}

//main function
int main( ) {
    int i;
    SafeArray sa(5, 0);    //size = 5

    for (i = 0; i < 5; i++) //assign element
        sa[i] = i *100;

    for (i = 0; i < 5; i++){ //display elements
        int x = sa[i];
        cout << x << endl;
    }

    //sa[10] = 700;          //ERROR
    //cout << endl << sa[-5]; //ERROR
    return 0;
}

```

Exercise: Design the copy constructor also for the SafeArray class.

Exercise: Use dynamic allocation for safe array class (SafeArray2Dyna.cpp- show main())

Exercise:

(1) Design a class MyString that has data members:

- size
- character pointer

The member functions are:

- constructor,
- Copy constructor to assign existing string to another.
- setter function
- display function
- function to access, set or get a specified character by using [] operator.
- overload + operator to concat two strings.
- overload > and < operators to compare two strings (lexographically, i.e., dictionary order)
- Destructor that releases the dynamically allocated space.

**show main of
string-TEST.cpp**

(2)

Develop class **Polynomial**. The internal representation of a **Polynomial** is an array of terms. Each term contains a coefficient and an exponent. The term

$$2x^4$$

has a coefficient of 2 and an exponent of 4. Develop a full class containing proper constructor and destructor functions as well as *set* and *get* functions. The class should also provide the following overloaded operator capabilities:

- a) Overload the addition operator (+) to add two **Polynomials**.
- b) Overload the subtraction operator (-) to subtract two **Polynomials**.
- c) Overload the assignment operator (=) to assign one **Polynomial** to another.
- d) Overload the multiplication operator (*) to multiply two **Polynomials**.
- e) Overload the addition assignment operator (+=), the subtraction assignment operator (-=), and the multiplication assignment operator (*=).

(3)

A machine with 32-bit integers can represent integers in the range of approximately -2 billion to +2 billion. This fixed-size restriction is rarely troublesome. But there are applications in which we would like to be able to use a much wider range of integers. This is what C++ was built to do, namely create powerful new data types. Design the class **HugeInteger** to support the following operations:

- a) Describe precisely how it operates.
- b) What restrictions does the class have?
- c) Overload the * multiplication operator.
- d) Overload the / division operator.
- e) Overload all the relational and equality operators.

Chapter 6

Conversion

1. Conversions: Sometimes there is a need to convert data of one type to the data of another type. There can be following type of conversions:

- i. Conversion between basic types
- ii. Conversion between object of one class and the object of another class.
- iii. Conversion between basic and object of a class type
 - a) from basic to object type
 - b) from object to basic type

(I) Conversion between basic types: There are many basic data type provided in C++ language, like char, integer, float, and their variants. There is conversion between variables of some data types automatically; like:

char → short int → int → long int
float → double float → long double

i.e., “char” is converted to “short int”, “short int” is converted to “int” etc. The above conversion is called promotion or widening conversion, because the data type occupying less number of bytes is being converted to the data type occupying more number of bytes.

The reverse conversion is also possible, but there is always information loss. It is called narrowing conversion. For example, a float variable may be converted to int, but the fractional part is lost. Such conversions are used only when explicitly required. Normally, type casting is used to explicitly emphasize this conversion.

```
char a = 10;  
int b, c;  
float x = 45.657;
```

```
b = a;           // widening conversion – no data loss  
c = x;           // narrowing conversion – data loss is there
```

The second one, i.e., narrowing conversion should preferably be written as:

```
c = (int)x;      //type casting: c-style
```

(II) Conversion between object of one class and the object of another class:

Sometimes, there is a need to convert object of a class into object of another class.

Ex. We have two classes FTemp and CTemp. 2 objects are:-

- CTemp cobj;
- FTemp fobj;

Following assignment operations may be needed.

```
cobj = fobj // from Fahrenheit to Centigrade temperature  
fobj = cobj // from Centigrade to Fahrenheit temperature
```

There are two types of member functions to implement this conversion:-

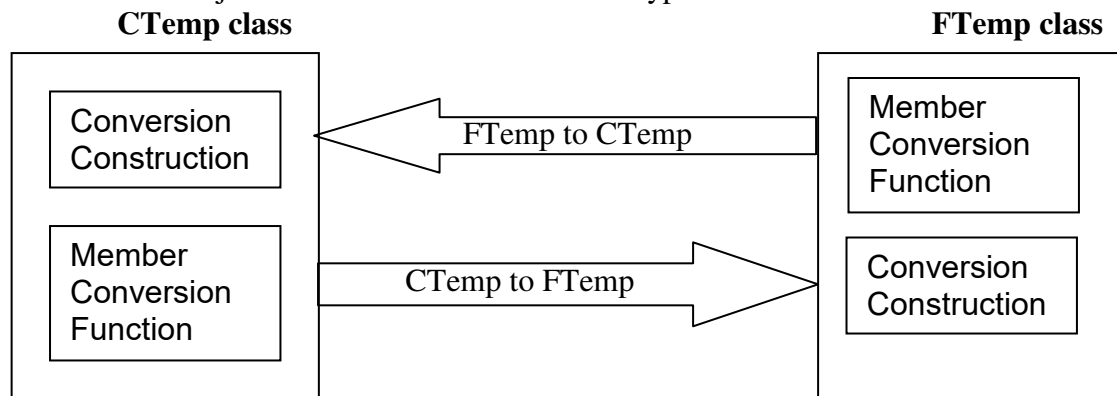
- a) Conversion constructor.
- b) Member conversion function/conversion operator

(a) Conversion Constructor: A constructor inside a class is conversion constructor if:

- It's parameterized
- The parameter is the object of another class.
- The body of the constructor has the code to convert the parameter object to object of the current class.

The conversion construction inside 'CTemp' class converts object of 'FTemp' into 'CTemp' object.

(b) Member Conversion Function: It converts the object of the class in which it's defined to the object of the different class or data type.



The purpose of Conversion Construction in CTemp class is same as the purpose of Member Conversion Function in FTemp class. So, only any one of them can exist. The purpose of Conversion Construction in FTemp class is same as the purpose of Member Conversion Function in CTemp class. So, only any one of them can exist.

So there are two options:

- Conversion constructors and member conversion function in CTemp class.
- Conversion constructors and member conversion function in FTemp class.

Example: Designed the FTemp & CTemp classes in such a way that both type of conversions are supported by CTemp class. The CTemp class contains:

- Conversion constructor
- Member conversion function both

Do not use the concept of friend.

```

//tempConv1.cpp
#include <iostream.h>

// class FTemp
class FTemp
{
private:
    float ft;
public:
    FTemp (float f = 0.0)
    { ft = f;}

    void set_FTemp (float f = 0.0)
    { ft = f;}

    float get_FTemp (void)
    { return ft;}
}; // end of class

//class CTemp
class CTemp
{
private:
    float ct;
public:

    // default constructor
    CTemp (float c = 0.0)
    { ct = c;}

```

// conversion constructor; FTemp to CTemp
//make it explicit later on

```

CTemp(FTemp fObj){
    float x;
    x = fObj.get_FTemp();    // x = fobj.ft
    ct = 5 * ( x - 32 ) / 9;
}

```

// member conversion function : CTemp to FTemp
operator FTemp(); // definition outside

```

// setter_function
void set_CTemp (float c = 0.0)
{ct = c;}

```

```

// getter_function
float get_CTemp()
{ return ct; }
}; // end of class c temp

```

// member conversion function : CTemp to FTemp

```

CTemp::operator FTemp( )
{
    float x;
    x = (9* ct / 5) + 32;
    FTemp fObj;
    fObj.set_FTemp(x);
    return fObj;
}

```

```

//main function
int main( ) {
    CTemp cObj;
    FTemp fObj(100);

    // implicit conversion
    // conversion constructors called
    cObj = (CTemp) fObj;
    cout << endl << cObj.get_CTemp( )
         << endl << fObj.get_FTemp( ) << flush;

    cObj.set_CTemp(50);

    // explicit conversion
    // member conversion function called
    fObj = (FTemp) cObj;
    cout << endl << cObj.get_CTemp( )
         << endl << fObj.get_FTemp( )
         << endl;
    return 0;
}

```

Note 1. : Member conversion function is also called **conversion operator**.

Note 2. : Following things are to be remembered in preparing member conversion function:-

- (a). The functions name is the reserve word operator followed by target class name.
- (b). It has no return type, but
- (c). The function returns the object of target class.
- (d) It has no parameter

Note 3: The conversion function can also be used to do conversions between the object of a class and variable of a basic data type.

If we precede the conversion constructor header by explicit keyword, it will explicit constructor. It will necessitate to type cast the conversion explicitly. It is always suggested to make the conversion constructor as explicit.

Exercise: Now design the CTemp class first, and then FTemp. Design both the conversion function inside the FTemp class. (temp_conv2.cpp)

III) Conversion between object of one class and Basic Data Type:

In the case both the conversion functions has to be put in the class.

Ex: Design the class Bdistance. The metric distance is taken as float type.

There will be only one class, both the function has to be designed in that.

- Conversion constructor (Bdistance \leftarrow meter in float)
- Member conversion function (meter in float \leftarrow Bdistance)

```

//convFeetMeter.cpp
#include <iostream.h>

const float convFactor = 3.25F;      //feet in one meter
const float inchesInFeet =12.0F;

class BDistance
{
private:
    int feet;
    float inches;

public :

```

```

//default ctor
BDistance ()
{feet = 0; inches = 0.0;}

//parameterized
BDistance (int ft, float inc)
{feet = ft; inches = inc;}

// conversion constructor: float to distance
explicit BDistance(float meter) {
    float x;
    x = meter * convFactor;    //convert into feet
    feet = int(x);              // get integer part of x

    // get fractional part of x,
    //and convert it into inches
    inches = (x - feet) * inchesInFeet;
}

// member conversion function. BDistance to float
operator float(void) {
    float meter;
    meter = (feet + inches / inchesInFeet) / convFactor;
    return meter;
}

// setter function
void set(int ft = 0, float inch = 0.0F)
{ feet = ft; inches = inch;}

// display function
void display (void) {
    cout << endl << "feet :" << feet
         << " inches :" << inches << endl;
}
}; //end of class

int main( )
{
    BDistance d;
    float mt = 1.1F;

    // conversion constructor called
    d = (BDistance)mt;           // float to distance
    d.display( );
    d.set(20, 10.7F);            // set value of d
    mt = float(d);               // member conversion function called

    cout << endl <<"meter :" << mt << endl;
    return 0;
}

```

The context in which conversions occur:

The object conversion from one class to another occurs in following cases:

- Assignment
- Function Argument Passing
- Initialization
- Return Values
- Statement Expressions

Exercises

1): Design two classes Time24 (hh, mm, ss) and Time12 (hr, min, sec and meridian) design default constructor, setter & display function for both the classes. Give provision to convert the object of the classes into each other. Both the classes has a tick() function, that adds one second to the existing time. (time12to24.cpp)

2): Design two classes for storing coordinate system: Cartesian and Polar. Give necessary functions in both classes. Give provision to convert the object of the classes into each other. (polarCatres.cpp)

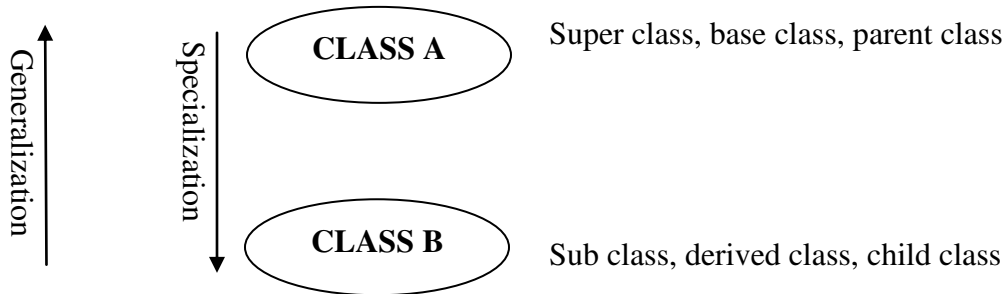
3). Design two classes of date:

- Date1 with dd, mm and yy
- Date2 with dd and yy. dd stores days elapsed from 1st January of the year.

There should be provision of conversion from one class to another.
(convDate.cpp)

Inheritance

1. **Definition:** Inheritance provides the mechanism to use the attributes and properties of a class by another class.



Following are the advantages of inheritance:

- I). **Reusability:** A class once developed, can be reused to develop more classes. It saves effort and time, both.
- II). **Decreased Cost:** Inheritance is based on commonality concept. The common member data and member functions of more than one classes can be put into a common base class. The characteristics of base class are inherited by all the derived classes. So repetition is reduced. It again reduces effort and time both; and hence cost is reduced.
- III). **Simplicity in Understanding:** A complex system of real world has 2 types of hierarchy.
 - CLASS (type) HIERARCHY -Inheritance
 - OBJECT(instance) HIERARCHY -ContainershipThe hierarchy simplified the representation because it highlights on:
 - Similarity.
 - Differences
- IV). The base class remains unaffected.

2. Why Inheritance or Uses of Inheritance:

Following are the prominent uses of inheritance:

- I). The base class might be used by other part of the program and by other programs, and we want its original behavior to remain intact for those objects that already use it. By deriving a new class, the original base class remains unaffected.
- II). The source code of the base class may not be available to us. To use the class, we only need its declaration in header file. Though we cannot access the code, but we can use it by inheritance.
- III). The base class might define a component in a class library that supports a wide community of users. So, we must not modify the base class even if we can.
- IV). The base class may be an abstract base class, which is a class designed to be a base class only. A class hierarchy can contain general-purpose classes that do nothing on their own. Their purpose is to define the behavior of a generic data structure to which derived classes and implementation details.
- V). We might be building a class hierarchy to derive the benefits of the object-oriented approach.

Note: Inheritance provides “economy of expression”. It means economy of everything - analysis, design, coding, testing & debugging, modification & maintenance. It reduces the total cost of system.

Note : The concept of data hiding and inheritance are opposite to each-other. Data hiding is achieved by encapsulation. Encapsulation gives a boundary around the properties to be hidden. Inheritance tries to break this boundary and looks the hidden characteristics.

Gady Booch says – “There is a fair war between inheritance and data hiding”

3. Abstraction

It is to focus on commonalities among objects in system. There are two schemes:

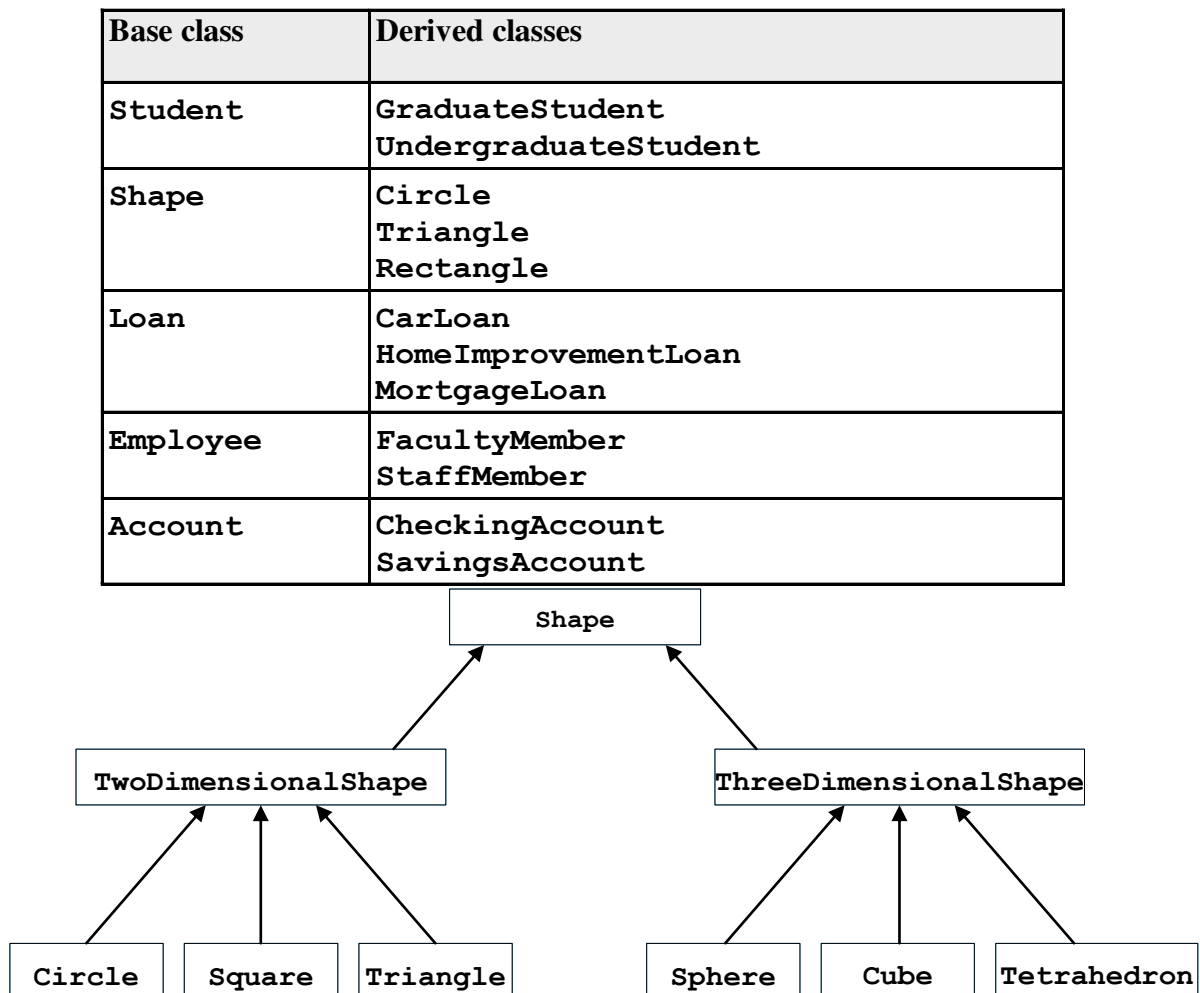
- **is-a** hierarchy: class or type hierarchy -Inheritance
 - Derived class object treated as base class object

Example: Car is a vehicle .Vehicle properties/behaviors are also car properties/behaviors

- **has-a** hierarchy: object hierarchy -Containership / Composition
 - Object contains one or more objects of other classes as members

Example: Car has a steering wheel.

4. Inheritance examples:



5. Type of Inheritance:

There are two classification schemes:

Classification A: There are three types of inheritance:

- a) **Public Inheritance** – it is the most common to use.
- b) **Private Inheritance** – used for terminating inheritance chain.
- c) **Protected Inheritance.**

Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
Public	public in derived class. Can be accessed directly by any non- static member functions, friend functions and non-member functions.	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	private in derived class. Can be accessed directly by all non- static member functions and friend functions.
Protected	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	protected in derived class. Can be accessed directly by all non- static member functions and friend functions.	private in derived class. Can be accessed directly by all non- static member functions and friend functions.
Private	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non- static member functions and friend functions through public or protected member functions of the base class.

Classification B: There are three types of inheritance:

- a) Multilevel Inheritance
- b) Multiple Inheritance
- c) Hybrid Inheritance

Multilevel Inheritance:

Class A derives class B, class B derives class C and so on. It is called multi-level inheritance. For multi-level inheritance to be meaningful:

- Data members should be protected.
- Public inheritance should be used.

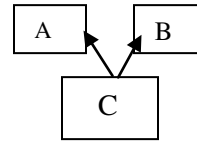
Ex.

```
class A
{
    :
};
class B :    public A
{
    :
};
class C :    public B
{
    :
};
```

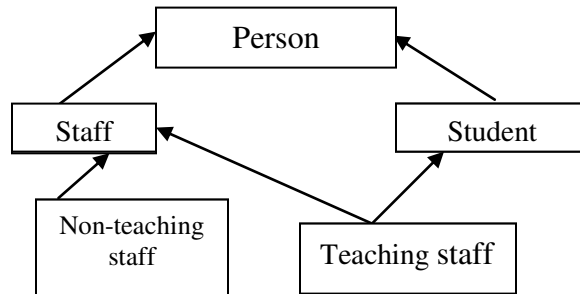
Multiple Inheritance:

A class can be derived by more than one class, it's called multiple inheritance.

```
class A {  
:  
};  
class B {  
:  
};  
class C : public A, public B  
{  
:  
};
```



Hybrid Inheritance:



6. Access Specifiers:

<i>Access specifier</i>	Access from own class	Access from derived class	Access in remaining program
Private	√	X	X
Protected	√	√	X
Public	√	√	√

7. An Example:

```
//Counter1.cpp  
#include <iostream.h>  
class Counter { //class only supporting ++  
protected:  
    int count ;  
public:  
    Counter(int c = 0 ): count(c)  
    {   }  
  
    void set ( int c=0)  
    { count =c;}  
  
    void display( )  
    { cout << " count = " << count << endl;}  
  
    Counter operator++( )  
    {  
        Counter temp;  
        temp.count = ++count;  
        return temp;  
    }  
}; //end of class
```

```

//DCounter class derived from Counter
class DCounter : public Counter { //class also supporting --
public:
    DCounter() //implicit call to default ctor of base class
    {}
    //decrement function
    DCounter operator--( ) {
        DCounter temp;
        temp.count = --count ;
        return temp ;
    }
}; //DCounter class
int main( ){
    Counter c(100);
    DCounter dc;
    ++c; c.display( );
    ++dc; dc.display( );
    --dc; dc.display();
    return 0;
}

```

8. Calling of Constructors and Destructors in Inheritance:

When the object of the derived class is instantiated, following is the sequence:

- The space for member data of the object is allocated.
- The constructor of derived class is called (**but body is not executed**)
- The constructor of base class is called and its body is also executed.
- The body of derived class constructor is executed.
- The execution return to the function.
- Scope of derived class object goes out, destructor of derived is called and executed.
- Destructor of base is executed.
- Space for the object is released.

The constructor of base class can be called explicitly from the constructor of derived class, by using member initializer list. If there is no explicit call, the default constructor of the base is called automatically (in this case there must be default constructor in the base class, otherwise it will be a syntax error.)

In the following example the Counter class is the same, but DCounter is redesigned.

```

//Counter2.cpp : DCounter class derived from Counter
class DCounter : public Counter {
public:
    DCounter(int c = 0) : Counter(c)
        //explicit call to the parameterized constructor of base class
    {
    }
    //decrement function
    DCounter operator--( ) {
        DCounter temp;
        temp.count = --count ;
        return temp ;
    }
}; //DCounter class
int main( ){
    Counter c(100);
    DCounter dc(500);

    ++c; c.display( );
    ++dc; dc.display( );
    --dc; dc.display();

    return 0;
}

```

9. Overriding of a Base Class Function in Derived Class:

(Show overriding1.cpp)

The derived class may have a function with same name as that in the base class. When the function is called with the object of base class, the base class function is called. When the function is called with the object of derived class, the derived class function is called.

s1.display(); *//call to display() of base class*

s2.display(); *//call to display() of derived class*

Ex: Design a class Distance with necessary function. Design another class DistSign that also stores sign with the distance value. All the functions in the derived class also deal with sign.

```
//overriding2.cpp
#include <iostream.h>
enum posneg { pos, neg };               //for sign in DistSign
class Distance {                        //English Distance class
protected:                            //NOTE: can't be private
    int feet;
    float inches;
public:
    Distance( ) : feet(0), inches(0.0)   //no-arg constructor
    { }

    Distance(int ft, float in) : feet(ft), inches(in)
    { }

    //get length from user
    void getDist( ) {
        cout << "\nEnter feet: "; cin >> feet;
        cout << "Enter inches: "; cin >> inches;
    }

    void showDist( ) const               //display distance
    { cout << feet << "\'-" << inches << '\\"'; }
};

//-----
class DistSign : public Distance {   //adds sign to Distance
    posneg sign;                        //sign is pos or neg
public:
    DistSign( ) : Distance( )   // no-arg constructor; call base constructor
    { sign = pos; }                      //set the sign to +

    //2- or 3-arg constructor : call base constructor
    DistSign(int ft, float in, posneg sg = pos) : Distance(ft, in)
    { sign = sg; }                       //set the sign

    void getdist( ) {                    //get length from user
        Distance::getdist( );           //call base getdist( )
        char ch;                        //get sign from user
        cout << "Enter sign (+ or -): "; cin >> ch;
        sign = (ch== '+' ) ? pos : neg;
    }

    void showdist( ) const               //display distance
    {
        cout << ( (sign==pos) ? "(+)" : "(-)" );   //show sign
        Distance::showdist();           //ft and in
    }
};
```

```

int main( ){
    DistSign alpha;                //no-arg constructor
    alpha.getdist( );              //get alpha from user

    DistSign beta(11, 6.25);        //2-arg constructor

    DistSign gamma(100, 5.5, neg);  //3-arg constructor

    //display all distances
    cout << "\nalpha = ";  alpha.showdist( );
    cout << "\nbeta = ";   beta.showdist( );
    cout << "\ngamma = ";  gamma.showdist( );

    cout << endl;
    return 0;
}

```

Note :

(1)When we declare an object of the derived class, the compiler executes the constructor of base class first, then the constructor of the derived class. The parameter list for the derived class constructor can differ from that of the base class. We must tell the compiler, which value to use as argument to the constructor function for the base class. When a base class has more than one constructors the compiler decides, which one is to be called based on the type of argument in the derived class constructor's parameter initialization list for the base constructor.

(2) In the derived class the set() of base class is called with class name and scope resolution operator.

Sample1: :set (xx);

If it were set (xx);

It would be treated as recursion. It would have been a call without termination, because there is no termination condition.

10. Exercise: Design a class employee to have name and number. The class has member functions to get and set the data. Derive classes manager, scientist and labour from it. Design a suitable main function.

```

//employee.cpp : models employee database using inheritance
#include <iostream.h>
const int LEN = 80;                //maximum length of names

class employee                      //employee class
{
protected:
    char name[LEN];                 //employee name
    unsigned long number;          //employee number
public:
    void getdata( ) {
        cout << "\n   Enter last name: "; cin >> name;
        cout << "   Enter number: ";      cin >> number;
    }
    void putdata( ) const {
        cout << "\n   Name: " << name;
        cout << "\n   Number: " << number;
    }
};

```

```

class manager : public employee {           //management class
private:
    char title[LEN];                        //"vice-president" etc.
    double dues;                           //golf club dues
public:
    void getdata( ){
        employee::getdata();
        cout << "    Enter title: "; cin >> title;
        cout << "    Enter golf club dues: "; cin >> dues;
    }
    void putdata( ) const {
        employee::putdata();
        cout << "\n    Title: " << title;
        cout << "\n    Golf club dues: " << dues;
    }
};

class scientist : public employee {         //scientist class
private:
    int pubs;                              //number of publications
public:
    void getdata( ) {
        employee::getdata( );
        cout << "    Enter number of pubs: "; cin >> pubs;
    }
    void putdata( ) const {
        employee::putdata( );
        cout << "\n    Number of publications: " << pubs;
    }
};

class laborer : public employee            //laborer class
{ };

//-----
int main( ) {
    manager  m1, m2;
    scientist s1;
    laborer  l1;

    cout << endl;           //get data for several employees
    cout << "\nEnter data for manager 1";
    m1.getdata();
    cout << "\nEnter data for manager 2";
    m2.getdata( );
    cout << "\nEnter data for scientist 1";
    s1.getdata( );
    cout << "\nEnter data for laborer 1";
    l1.getdata( );
    //display data for several employees
    cout << "\nData on manager 1";
    m1.putdata( );

    cout << "\nData on manager 2";
    m2.putdata( );

    cout << "\nData on scientist 1";
    s1.putdata( );
    cout << "\nData on laborer 1";
    l1.putdata( );
    cout << endl;
    return 0;
}

```

**Exercise: Design a
queue class. Derive
dequeue from this**

11. Ambiguities in Inheritance:

- I). Multiple occurrence of same function
- II). Diamond shape ambiguity

There are two types of ambiguity:

I) Multiple Occurrence of Same Function:

There are two classes A and B, both have same function f(). A class C is derived from both the classes. When an object of c is created and function f() is called with this object, it will be an ambiguity. A compiler will be unable to decide which f() is called, f() of A or f() of B.

```
class A {
    :
public:
    void f();
    :
};           //end of class A

class B {
    :
public :
    void f();
    :
};           //end of class B

class C : public A, public B
{
    :
};           //end of class C

int main() {
    C cobj;
    :
    cobj.f(); //error : ambiguous call.
    :
}
```

Resolving Ambiguity:

```
int main() {
    C cobj;
    :
    cobj.A::f();           //f() of A is called
    cobj.B::f();           //f() of B is called
    :
}
```

II) Diamond Shape Ambiguity (Hybrid Inheritance Ambiguity):

There is a class A. It derives two classes B and C. Class A contains a function f(). A class D is derived from B and C both. When an object of D is created and the function f() is called with the object of D, it will be an ambiguous call,

because the compiler doesn't know, which function f() to call, through class B or through class C.

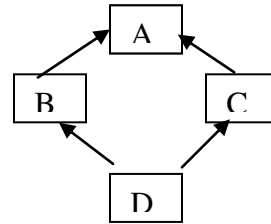
```
class A {
    :
    public
    void f();
    :
};    //end of class A

class B : public A {
    :
    :
}    //end of class B

class C : public A
{
    :
};    //end of class B

class D : public B, public C {
    :
    :
}

int main() {
    D dobj;
    :
    :
    dobj.f();    //ERROR
}
```



Question: How can it be resolve?

12. Containership or Composition:

There are two classes A and B. Class B contains object of class A as a data member. It is called containership. Containership is also called "has a" relationship or "is-part-of" relationship.

Ex.

```
class A {
    :
};
class B {
    protected :
        A aobj ; //containership
    public:
        :
        :
}
```

Ex: There is a class student, that stores name of school or university from which he is enrolled and name of highest degree he has obtained so far. It has the function to get and display the members. Design a class Employee with name and employee number. Derive Manger, Scientist and Laborer classes. The manager class has extra attributes title and dues. The Scientist class has extra attribute number of publications. The Laborer class has nothing extra. The classes have necessary functions for set and display the information. A manager or a scientist is employee; and he may be student of a university also. Use containership.


```

// containership with employees and degrees
#include <iostream>
#include <string>
class student {           //educational background
private:
    string school;        //name of school or university
    string degree;        //highest degree earned
public:
    void getedu() {
        cout << " Enter name of school or university: ";
        cin >> school;
        cout << " Enter highest degree earned \n";
        cout << " (Highschool, Bachelor's, Master's, PhD): ";
        cin >> degree;
    }
    void putedu( ) const {
        cout << "\n School or university: " << school;
        cout << "\n Highest degree earned: " << degree;
    }
}; //end of class
class employee {
private:
    string name;           //employee name
    unsigned long number;  //employee number
public:
    void getdata( ) {
        cout << "\n Enter last name: "; cin >> name;
        cout << " Enter number: ";   cin >> number;
    }
    void putdata( ) const {
        cout << "\n Name: " << name;
        cout << "\n Number: " << number;
    }
}; //-----
class manager {           //management
private:
    string title;          //"vice-president" etc.
    double dues;           //golf club dues
    employee emp;          //object of class employee
    student stu;           //object of class student
public:
    void getdata( ) {
        emp.getdata( );
        cout << " Enter title: ";      cin >> title;
        cout << " Enter golf club dues: "; cin >> dues;
        stu.getedu( );
    }
    void putdata( ) const {
        emp.putdata( );
        cout << "\n Title: " << title;
        cout << "\n Golf club dues: " << dues;
        stu.putedu( );
    }
};

```

```

class laborer {           //laborer
private:
    employee emp;        //object of class employee
public:
    void getdata( )
        { emp.getdata( ); }
    void putdata( ) const
        { emp.putdata( ); }
};

int main( ) {
    manager m1;
    scientist s1, s2;
    laborer l1;

    cout << endl;
    cout << "\nEnter data for manager 1";           //get data for
    m1.getdata();                                   //several employees

    cout << "\nEnter data for scientist 1";
    s1.getdata( );

    cout << "\nEnter data for scientist 2";
    s2.getdata( );

    cout << "\nEnter data for laborer 1";
    l1.getdata( );

    cout << "\nData on manager 1";                 //display data for
    m1.putdata( );                                   //several employees

    cout << "\nData on scientist 1";
    s1.putdata( );

    cout << "\nData on scientist 2";
    s2.putdata( );

    cout << "\nData on laborer 1";
    l1.putdata( );
    cout << endl;
    return 0;
}

```

Q1.cpp
q2.cpp
q3.cpp
q4.cpp

Explain the same example by using multiple inheritance, and tell the difference.
(multiInheritance.cpp)

Que: What is abstract base class? What is a concrete class?

1. An Example:

There are three classes A, B and C. All the 3 classes have a member function with same name show(). The body of show() is different in all the three classes.

```
//staticBinding.cpp
#include <iostream.h>

//class A
class A {
public:
    void show (void)
    { cout << endl << "function of A called"; }
};

//class B
class B {
public:
    void show(void)
    { cout << endl << "function of B called"; }
};

//class C
class C {
public:
    void show (void)
    { cout<<endl<< "function of C called"; }
```

```
int main() {
A* ap; B* bp; C* cp;    //pointer of objects
A ao; B bo; C co;      //objects

// linking pointers to objects
ap = &ao;
bp = &bo;
cp = &co;

//accessing member function
ap->show();    //show() of class A called
bp->show();    //show() of class B called
cp->show();    //show() of class C called

return 0;
}
```

2. Binding – static & dynamic:

Let there are more than one functions of same prototype (may be member functions of different classes). Let there are three different classes A, B and C. All the three contains a function with same prototype:

```
void show( );
```

There is a call in the any function:

```
show( );
```

Now, which of the three will be called, is called binding.

Binding means which definition will be bounded with the call.

Any activity that is fixed before execution, is called static or early activity. Any activity that is fixed during execution, is called dynamic or late activity.

There can be two types of binding:

1. Static binding, early binding or link time binding.
2. Dynamic binding, late binding or run time binding.

If binding of the function definition with the call to the function is fixed before execution, it is called **static binding** or early binding.

If binding of the function definition with the call to the function is fixed during execution, it is called **dynamic binding** or late binding.

The default binding that we have studied so far is all early binding.

Note:

- | | | |
|------------------|---|---------------------------------------|
| Before execution | - | Only type information is available |
| During execution | - | Type and contents both are available. |

Hence, static binding is on the basis of type. Dynamic binding is on the basis of contents.

Note : Normally, a pointer of a class can only point to the object of the same class; it can't point the objects of any other class. But, this restriction is relaxed in case of inheritance. **The pointer of base class can point the objects of derived classes. But, the pointer of base class can access only those members that are members of the base class.**

Following is the example of static binding:

```
//staticBinding2.cpp
class Base {
public:
    void show(void)
    { cout << endl << "in base "; }
};

class der1 : public base {    //derived class -1
public:
    void show(void)
    { cout << endl << "in derived - 1"; }
};

class der2 : public base {    //derived class -2
public:
    void show(void)
    { cout << endl << "in derived -2"; }
};
```

```
int main() {
    base* bp; //base class pointer
    base bo; der1 do1; der2 do2;    //objects

    bp = &bo;    //pointer pointing to objects of base class
    bp->show();    //show() of base called

    bp = &do1;    //pointer pointing to object of der1 class
    bp->show();    //but, show() of base called

    bp = &do2;    //pointer pointing to object of der2 class
    bp->show();    //but, show() of base called

    return 0;
}
```

The reference of base class can refer to an object of derived class.

If the binding of the call statement with the function is according to the type, it is called early binding or static binding. The type is decided before execution. The linking of the call to the occurrence of the function is done before execution. It is actuality (no virtuality); the binding is done according to what is visible (the type).

“Virtual” means existing in appearance, but not in actual. When virtual function is used, an object that appears to be calling a function of its class may in reality be calling a function of different class.

Following is the example of dynamic binding:

```
//dynamicBinding1.cpp
#include <iostream.h>

class Base {
public:
    virtual void show(void)
    { cout << endl << "in base "; }
};

//derived 1 class
class der1 : public base {
public:
    void show(void)
    { cout << endl << "in derived - 1"; }
};

//derived2 class
class der2 : public base {
public:
    void show(void)
    { cout << endl << "in derived -2"; }
};
```

```
int main() {
    base* bp;    //base class pointer
    base bo; der1 do1; der2 do2;    //objects

    bp = &bo;    //pointer pointing to objects of base class
    bp->show();    //show() of base called

    bp = &do1;    //pointer pointing to object of der1 class
    bp->show();    //show() of der1 called

    bp = &do2;    //pointer pointing to object of der2 class
    bp->show();    //show() of der2 called

    return 0;
}
```

The virtuality is that, the type of pointer ‘bp’ is base pointer, but the statement,
bp → show();
 is calling the function of some other class (according to contents of the pointer at execution time).

3. Pure Virtual Function:

A function is a pure virtual function if, it has following properties:

- The function is virtual, i.e., header is preceded by keyword **virtual**.
- It has no body.
- The header is equal to zero.

Ex.

```
virtual void show() = 0;
```

Not assignment, only syntax.

A class containing a pure virtual function is called **abstract-class**. It means that the compiler will not permit to create the object of this class. It means that, it **cannot be instantiated**. A pure abstract class can have only following permitted operations :-

- It can derive another class.
- Pointers and **references** of this class can be created.

A pure virtual function must be redefined or overridden in the derived class.

If a derived class has the detailed definition of pure abstract function of base class, only then its objects can be created.

If the pure virtual function is not defined inside the derived class, then this function will be inherited as pure abstract function from base class and the derived class will also be treated as **abstract class**. Hence, its objects cannot be created.

Note: The concept of pure virtual function facilitates us to defined a class completely even if its behavior (i.e. the operation of member functions) are not known.

Demonstrate pureVirtual1.cpp

```
#include <iostream.h>

class Base {           //base class
public:
    virtual void show( ) = 0; //pure virtual
    function
};

class Derv1 : public Base { //derived class 1
public:
    void show( )
    { cout << "Derv1\n"; }
};

class Derv2 : public Base { //derived class 2
public:
    void show( )
    { cout << "Derv2\n"; }
};
```

```
// pureVirtual1.cpp : pure virtual function
int main( )
{
    Base* bp;    //base class pointer
    Der1 do1; Der2 do2;    //objects

    bp = &do1;    //pointer pointing to object of Der1 class
    bp->show( ); //show( ) of Der1 called

    bp = &do2;    //pointer pointing to object of Der2 class
    bp->show( ); // show( ) of Der2 called
    return 0;
}
```

```
// pureVirtual2.cpp : pure virtual function
int main( ){
// Base bad;    //can't make object from abstract class
    Base* arr[2];    //array of pointers to base class
    Derv1 dv1;    //object of derived class 1
    Derv2 dv2;    //object of derived class 2

    arr[0] = &dv1;    //put address of dv1 in array
    arr[1] = &dv2;    //put address of dv2 in array

    arr[0]->show( );    //execute show() in both objects
    arr[1]->show( );
    return 0;
}
```

NOTE: Virtual functions can never be inline.

Ex: Design a class `person`. Each person has name & age. A member functions are `readinfo()`, `display()` and `isoutstanding()`. Student has marks, and teacher has no-of-research-papers for grading of outstanding performance. Override all the 3 functions, in both classes.

```
// virtPersons.cpp : virtual functions with Person class
#include <iostream.h>
class Person {                                //Person class
protected:
    char name[40];
public:
    void getName( )
    { cout << "    Enter name: "; cin >> name; }

    void putName( )
    { cout << "Name is: " << name << endl; }

    virtual void getData() = 0;                //pure virtual function
    virtual bool isOutstanding() = 0;         //pure virtual function
};
//////////
class Student : public Person {                //Student class
private:
    float gpa;                                //grade point average
public:
    void getData( ) {                          //get Student data from user
        Person::getName( );
        cout << "    Enter Student's GPA: "; cin >> gpa;
    }
    bool isOutstanding( )
    { return (gpa > 3.5) ? true : false; }
};
//////////
class Professor : public Person {              //Professor class
private:
    int numPubs;                              //number of papers published
public:
    void getData() {                            //get Professor data from user
        Person::getName( );
        cout << "    Enter number of Professor's publications: ";
        cin >> numPubs;
    }

    bool isOutstanding()
    { return (numPubs > 100) ? true : false; }
};
int main( ){
    Person* persPtr[5];                        //array of pointers to Persons
    int n = 0;                                //number of Persons on list
    char choice;

    do {                                        //loop to read the data
        cout << "Enter Student or Professor (s/p): ";
        cin >> choice;
        if(choice=='s')                      //put new Student
            persPtr[n] = new Student;        //    in array
        else                                  //put new Professor
            persPtr[n] = new Professor;       //    in array

        persPtr[n++]->getData();              //get data for Person

        cout << "    Enter another (y/n)? "; //do another Person?
        cin >> choice;
    } while( choice=='y' );                  //cycle until not 'y'
```

```

// loop to display the data
for(int j=0; j<n; j++) {           //print names of all Persons,
    persPtr[j]->putName( );        //say if outstanding
    if( persPtr[j]->isOutstanding( ) )
        cout << "    This Person is outstanding\n";
    cout << endl << endl;
}

//dynamically deallocate the memory
for ( j = 0; j < n; ++j)
    delete persPtr[j];

return 0;
} //end main( )

```

4. Virtual Destructor:

An abstract/base class destructor must be always virtual type. It is because this destructor will be completely defined in a derived class; it is sure that there will be no object of base class, so the base class destructor will not be called directly. The derived class may contain extra members. The memory for these extra members is released with the help of derived class destructor.

The memory for base class members is released by the destructor of base class, with the help of virtual base class destructor and dynamic binding we can release the memory contained by the derived class member's, even if the pointer is of base class. If the base class destructor is non-virtual type, due to static binding it will be called. Hence the memory for the extra derived class member will remain allocated.

```

//virtDetor.cpp: program to demonstrate virtual destructor
#include <iostream.h>
class Base {
protected :
    int* p1;
public :
    Base(int x = 0) {           //constructor
        cout << endl << "base constructor called" << endl;
        p1 = new int;
        *p1 = x;}
    virtual void display( )
    {cout << endl << *p1 ;}

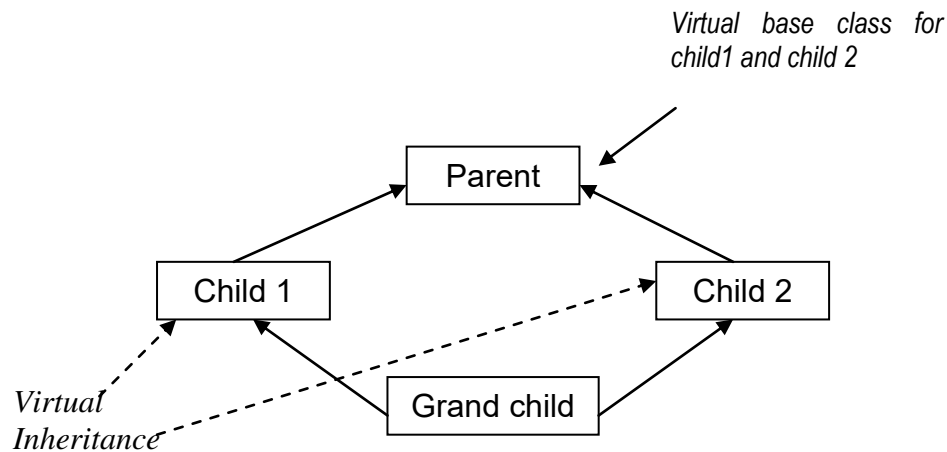
    virtual ~Base( ){           //virtual distructor
        delete p1 ;           //Base class destroyed
        cout << endl << "Base destroyed";
    }
};

// derived class
class Derv : public Base {
protected:
    int* p2;
public :
    //constructor of derived class
    Derv(int x1 = 0, int x2 = 0) : Base(x1) {
        cout << endl << "derived constructor called" << endl;
        //Base::Base(x1);      //call to Base constructor
        p2 = new int;          //allocated for *p2
        *p2 = x2;
    }
    void display( ) {
        Base::display( ); //call to display of Base class
        cout << endl << *p2 ;
    }
    //destructor
    ~Derv( ){
        //Base::~~Base( ); //call to destructor of Base class; *p1 released
        delete p2;          //derived destroyed
        cout << endl << "derived destroyed"; }
};

```

5. Virtual Base Class:

Virtual base class is used to avoid **diamond shape ambiguity.** / hybrid inheritance ambiguity



The classes child1 and child2 are derived by using virtual inheritance.

Ex:

```
// virtbase.cpp : virtual base classes

class Parent {
protected:
    int basedata;
};

class Child1 : virtual public Parent //shares copy of Parent
{
};

class Child2 : virtual public Parent //shares copy of Parent
{
};

class Grandchild : public Child1, public Child2 {
public:
    int getdata()
    { return basedata; } // OK: only one copy of Parent
};
```

Due to virtual base class only one copy of base data arrives to derived class - grandchild.

With the help of virtual inheritance the grand child access only a single copy of base data.

Polymorphism

- 1. Static
 - a. Function Overloading.
 - b. Operator Overloading.
- 2. Dynamic
 - a. Late Binding. or Virtual function.

Note 2: With virtual functions and polymorphism, it becomes possible to design and implement systems that are more easily extensible. Programs can be written to process objects of type that may not exist when the program is under development.

Note 3: Polymorphic programming with virtual functions can eliminate the need for switch logic.

1. See the **Shape** class hierarchy in the previous chapter. Implement the hierarchy so that class **Shape** is an abstract base class containing the interface to the hierarchy. Derive **TwoDimensionalShape** and **ThreeDimensionalShape** from class **Shape**—these classes should also be abstract. Use a **virtual print** function to output the type and dimensions of each class. Also include **virtual area** and **volume** functions so these calculations can be performed for objects of each concrete class in the hierarchy. Write a driver program that tests the **Shape** class hierarchy.

Exception Handling

1. Exception Handling:

Exceptions are those *rare* situations which occur during the execution of the program. There are two types of exceptions.

- Exceptions due to external agencies, these are called **interrupts**.
- Exceptions due to instructions in the program itself, called **'TRAPS'**

Here, we are concerned with exceptions of type two. Normally, these exceptions are errors(but not always). Following may be some exceptions:

- Divided by zero
- square root of negative numbers.
- file not found.
- Memory from heap not allocated.
- Stack underflow or overflow.

*In traditional programming, exceptions are handled in **static manner** i.e. the designer fixes the action and no flexibility left for the user. Normally, designer displays an error message and terminates the process. Exception handling modifies this approach. The handling of error is delayed. The user is informed about the exception; he is free to take the action to handle these exception.*

This approach provides operational flexibility and generality. Hence, its suitable for designing general purpose library. Hence, exception handling supports us in designing fault-tolerant programs.

2. Implementation:

There are three basic operations:

- a) Developer identifies the exception in the code.
- b) Developer throws an exception object towards the user.
- c) **User catches this object and decides the action to be taken.**

Corresponding to these operations, there are three basic constructs:

- try block - user
- throw statement - designer
- catch block. - user

"throw" is in developer's code, "try" and "catch" are in user's code.

Try Block:

The statements and function that can generate errors are enclosed within the try-block.

```
try{
    // statement generating
    // and detecting exceptions
    :
    :
}
```

The code executing outside the try-block can't detect or handle exceptions. The try-block may call other functions that are able to detect exceptions.

Catch Exception Handler:

A catch exception handlers with a parameter list follows a try-block. There may be one or multiple catch blocks. In case of multiple catch blocks, the argument type or number must be different.

```

try
{   //statements generating & detecting exceptions
    :
    :
    :
}
catch (exception class1 object1)
{   //exception handling code }

catch (exception class2 object2)
{   //exception handling code }

```

The parameters in catch parameter list need not necessarily have to be named. If parameter is named, it declares an object with that name and the exception detecting code can pass a value in that parameter. If the parameter is unnamed, the exception detecting code can jump to catch exception handler by simply match the type.

Throw Statement

To detect an exception and to jump to an exception handler, the call function issued a “throw” statement with a data type or a class that matches the parameter list of proper catch handler.

Ex. Demonstrate exception handling

```

//sample.cpp
//program to demonstrate
//exception handling
#include <iostream.h>
#include <string.h>

//exception class
class anError {
private:
    char message[50];
public:
    void set(char *s)
    {strcpy(message, s);}

    void display()
    {cout << message << endl;}
};

class AnyException
{   };

//-----
class Sample {
    int a, b, c;

public :

    Sample()          //c'tor
    { a= 0; b = 0; c=0;}

    void set() {
        cout << endl
        << "Enter values of a and b: ";
        cin >> a >> b;
    }
}

```

```

/**member function
void div(void)
{
    if ( b == 0 )      //exception detected
    {
        //create an object of
        //anError class
        anError eobj;

        eobj.set("Denominator is zero - can
        not perform division");

        throw eobj;    //throwing exception
    }

    else
        c = a/b;
    } //end of div function

/**display function
void show()
{
    cout << a << " " << b
        << " " << c << endl;
}

void otherException()
{
    throw (AnyException());
}

};

```

```

//main function
int main( ){
    Sample  sobj;

myLab:
    try                                //try  block
    {
        sobj.set();
        sobj.div( );                //this may create error
        cout << "The values are now: ";
        sobj.show();
        sobj.otherException();
    }                                //end of try

    catch(anError eobj)  {
        //display error message or do corrective measures
        eobj.display();
        cout << "Give the proper input again.\n";
        goto myLab;
    }//end of catch

    catch(...) {
        cout << "Unknown exception generated.\n";
    }

    return 0;
}

```

One more example of exception handling:

```

// DivByZero.cpp : A simple exception-handling example that
//checks for divide-by-zero exceptions.
#include <iostream>
#include <exception>

using std::cout;
using std::cin;
using std::endl;
using std::exception;

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public exception {
public:
    // constructor specifies default error message
    DivideByZeroException::DivideByZeroException()
        : exception( "attempted to divide by zero" ) {}

}; // end class DivideByZeroException

// perform division and throw DivideByZeroException object if
// divide-by-zero exception occurs
double quotient( int numerator, int denominator )
{
    // throw DivideByZeroException if trying to divide by zero
    if ( denominator == 0 )
        throw DivideByZeroException(); // terminate function

    // return division result
    return static_cast< double >( numerator ) / denominator;

} // end function quotient

```

```

int main(){
    int number1;    //user-specified numerator
    int number2;    //user-specified denominator
    double result;  // result of division
    cout << "Enter two integers (end-of-file to end): ";

    // enable user to enter two integers to divide
    while ( cin >> number1 >> number2 ) {
        // try block contains code that might throw exception
        // and code that should not execute if an exception occurs
        try {
            result = quotient( number1, number2 );
            cout << "The quotient is: " << result << endl;
        } // end try

        // exception handler handles a divide-by-zero exception
        catch ( DivideByZeroException &divideByZeroException ) {
            cout << "Exception occurred: "
                << divideByZeroException.what() << endl;
        } // end catch

        cout << "\nEnter two integers (end-of-file to end): ";
    } // end while

    return 0; // terminate normally
} // end main

```

Ex. Design the class stack. In case of PUSH & POP operation, identify stack overflow and underflow, throw exceptions in these cases, in place of displaying the error message.

```

//stack.cpp :exception haldling
#include <iostream.h>

//exception dummy classes
class error1
{
};
class error2
{
};

const int MAX = 5;
class MyStack {
private :
    int top;
    int ele[MAX];
public :
    //constructor
    MyStack( )
    { top=-1; }

    //initialize
    void initialize( )
    { top = -1; }

    //push and pop function
    void push( int x );
    int pop( );

    //status function
    int full( )
    { return top == MAX -1;}
    int empty( )
    { return top == -1;}
};

```

```

void MyStack::push (int x) {
    if (top == MAX-1) //error identified
        throw (error1()); //exception thrown
    else
        ele[++top] = x;
}

int MyStack::pop( ){
    int x;
    if (top == -1)// error identified
        throw (error2()); // exception thrown
    else
        x = ele[top--];
    return x;
}

//-----main
int main( ) {
    MyStack s;
    try {
        s.push (75);
        cout << endl << s.pop() << endl;
        cout << endl << s.pop() << endl;
    }
    catch (error1)
    {cout << endl << "stack overflow";}
    catch (error2)
    {cout << endl << "stack underflow"; }
    catch(...) {
        cout << "Unknown Exception" << endl;
        throw();
    }
    return 0;
}

```

Example : Design a class template dynamic array. It has a pointer of type T. It has a constructor that allocates memory of default size 10. It has member function to assign and get an element (overload subscript operator). Design a member function that increases the size of array by n (default value 5). Design a destructor also. Use exception handling if memory is not allocated from heap with new operator. Now, design a stack template class by using the dynamic array. Design all five regular function :

- constructor
- initialize()
- push()
- pop()
- is_full()
- is_empty()

Refer at the last for exercise.

Use exception handling for overflow and underflow cases. If these happens, increase the size of array, and then push the element.

Catch All Exception Handler:

A catch handler with **ellipse (...)** for a parameter list catches all uncaught exceptions.

```
catch(...)
{ //error handler }
```

catch all handler appear at last of catch ladder.

Note : A catch handler can throw the exception again. A single throw exception with no arguments will re-throw the original exception.

3. Misc Points on Exception Handling:

(A) Exception objects:

- Base class **exception** (<exception>)
- Constructor can take a string (to describe exception)
- Member function **what()** returns that string

(B) Rethrowing exceptions:

It is used when exception handler cannot process exception, but can still rethrow if handler did some processing. To rethrow use statement

"throw;"

with no arguments.

example (rethrow.cpp)

(c) **The throw list (not in C++):**

```
int someFunction( double value ) throw ( ExceptionA, ExceptionB, ExceptionC )
{
    // function body
}
```

It can only throw **ExceptionA**, **ExceptionB**, and **ExceptionC** and the objects of their derived classes. If it throws other type, function **unexpected()** is called and, terminates program.

If there is no throw list, function can throw any exception. If throw list is empty, the function cannot throw any exceptions.

(d) Stack Unwinding:

If exception thrown but not caught, execution goes to enclosing **try** block. The current function is terminated. Function call stack is unwound. It looks for **try/catch** that can handle exception; if none found, unwinds again. If exception never caught, calls **terminate**.

(stackUnwind.cpp)

(e) Dynamic allocation and exception:

When **new** fails to get memory,

- it **throw bad_alloc** exception, or
- returns NULL

Result depends on compiler. The exception approach is better

(example new.cpp)

The exception generated by new can also be handled in the following manner:

- Register function to call when **new** fails with the help of **set_new_handler**
- The header used is **<new>**
- Takes function pointer to function that
 - Takes no arguments
 - Returns **void**
- Once registered, function called instead of throwing exception

(register.cpp)

Exercise: Design a class SafeArray.

Data members:

int *p; int size;

Member Functions:

- C'tor SafeArray(int n =5); it allocates space for n members dynamically.
- Function to access the ith element of the array. Bound checking is required. Overload [] operator for accessing the element. Both operations must be supported:

```
a[i] = x;  
x = a[i]
```
- Function int * increase(int incr); to increase the size by incr, previous values are maintained.
- D'tor to de allocate the memory that was allocated in C'tor.
- Use exception handling to handle the cases when memory is not allocated and when array index is out of bound.

1. Templates:

Templates are used for **generic programming**. With the help of templates, we can effectively **pass data type as parameters**. Templates are used to generate general purpose function's or classes. **Templates are preferred when logic is known but data type is not known**. There are two types of templates:

- Function Templates
- Class Templates

2. Function Templates:

A function template defines a parametric non-member function which enables a program to call the same function with different data types. The compiler determines which types are used and generate the appropriate code from the template.

Template reduces the size of sources code. But, the size of objects code remains the same, because the compiler generates different instances of the function.

Ex: Design a function template to add two numbers, numbers can be int, float, etc.

```
//funTemplate1.cpp
#include <iostream.h>

template < class T >
T add( T a , T b) {
    T c ;
    c = a + b ;
    return c ;
}

// main function
int main() {
    cout << endl << add ( 5, 7 ) ;           // T is int.
    cout << endl << add ( 4.9, 3.7);         // T is double float.
    cout << endl << add ( 'a', 'p');         // is character.

    return 0 ;
}
```

Here, T is the identifier for the data type to be passed as parameter. It can be used as data type. The variables can be declared with its help. T is an identifier. It follows all the rules of making identifier, but conventionally it's taken as T.

The first line:

```
template < class T >
```

is called the **template specifier**.

The compiler generates 3 different function for this template:

- one function for T as int.
- one other function for T as float
- one function for T as character.

Hence is no reduction in the object code size.

Note: We must call a function template only when the body of the function contains valid operation for the argument.

for example:

```
// funTemplate2.cpp
int main() {
    char* s1 = "RGPV"
        * s2 = "BHOPAL";
    cout << endl << add(s1, s2);

    return 0;
}
```

The '+' operation inside the function template is meaningless here, because the addition of two addresses is an invalid operation.

Solution: We should make a class MyString. The '+' operator should be overloaded to concatenate two string. The string s1 and s2 should be made as objects of MyString class.

Ex: Design a function template that returns the smallest of three arguments.

```
//smallestTemplate.cpp
//The function template to find smallest of three numbers
#include <iostream.h>

// Template definition.
template<class T>
T min3(T arg1, T arg2, T arg3) {
    T min;

    min = arg1;
    if (arg2<min) min = arg2;
    if (arg3<min) min = arg3;

    return min;
}

// The main( ) function.
int main() {
    cout << min3(10, 20, 30) << endl;
    cout << min3(100.60, 10.872, 5.897) << endl;
    cout << min3('C', 'A', 'Z') << endl;

    // cout << min3("Ram", "Shyam", "Mohan") << endl;

    return 0;
}
```

Que: What will be the behavior of the template for the following main() function.

```
// The main( ) function.
int main() {
    cout << min3("Ram", "Shyam", "Mohan") << endl;
    return 0;
}
```

Ex : Design a function template that is supplied with the array pointer, an element of array type elements and size of array. The function template returns the position of element in the array, if it is there; otherwise -1. (template for sequential search)

```

//findTemplate.cpp
//template used for function that finds number in array
#include <iostream.h>

// function returns index number of item, or -1 if not found
template <class T> int find(T* a, T value, int size) {
    //repeat the loop
    for(int j=0; j<size; j++)
        if( a[j] == value ) //SUCCESS
            return j;
    return -1; //FAILURE
}
//-----
char chrArr[] = {1, 3, 5, 9, 11, 13}; //array
char ch = 5; //value to find

int intArr[] = {1, 3, 5, 9, 11, 13}; //array
int in = 6; //value to find

long lonArr[] = {1L, 3L, 5L, 9L, 11L, 13L}; //array
long lo = 11L; //value to find

double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0}; //array
double db = 4.0; //value to find
//-----
int main( ) {
    cout << "\n 5 in chrArray: index=" << find(chrArr, ch, 6);
    cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
    cout << "\n11 in lonArray: index=" << find(lonArr, lo, 6);
    cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
    return 0;}

```

3. Function Template Specialization:

Some times we need an operation in different way as compared to the general way for a special data type. In this case for general purpose we make a template function for general type and we make a special function for the special data type.

Ex : Design a function template that returns the smallest of three arguments. The arguments may be string type also.

```

//minSpecTemplate.cpp:
//The function template specialization
#include <iostream.h>
#include <string.h>

// Template definition - for general arguments
template<class T> T min3(T arg1, T arg2, T arg3) {
    T min;
    min = arg1;
    if (arg2<min) min = arg2;
    if (arg3<min) min = arg3;

    return min;
}

```

```

//The main( ) function.
int main( )
{
    cout << min3(10, 20, 30) << endl;

    cout << min3(100.60, 10.872, 5.897) << endl;

    cout << min3('C', 'A', 'Z') << endl;

    cout << min3("Ram", "Shyam", "Mohan") << endl;
    return 0;
}

```

```

// Specialized function - only for comparing strings
char* min3(char* arg1, char* arg2, char* arg3) {
    char* min;

    min = arg1;
    if ( strcmp(arg2, min) < 0 ) min = arg2;
    if ( strcmp(arg3, min) < 0 ) min = arg3;
    return min;
}

```

const removed from
all arguments

Note :

There can be many parameter to a function template:

```
template < class T1, class T2, class T3 >
T3 sample( T1 a, T2 b);
```

4. Class Template:

A class template enables us to generate generic data type . A class template provides parameter passing of data type inside a class. Following are the advantages and uses :-

- A class template provides a generic class for developing library.
- A class can be designed completely, even if there is incomplete information about types of data.
- Program size is reduced , because we need to develop only a single class for different data-types.

Ex : Design a class-template sample that has member data x , its data type is not yet decided, the member functions are:

- constructor
- setter function and getter function

```
//SampleClassTemp.cpp
//The first class template

#include <iostream.h>

template<class T>
class Sample
{
protected :
    T x ;
public :
    //constructor
    Sample(T xx =0)
    { x = xx ;}

    // setter function
    void set( T xx =0)
    { x = xx ;}

    // getter function
    T get (void)
    { return x ;}
};
```

```
// main function
int main() {
Sample <int>      s1(5);      // T is int
Sample <float> s2 (3.14F);    // T is float type
Sample <char> s3 ('a');      // T is char type
cout << endl<< s1.get( )
    << endl<< s2.get( )
    << endl<< s3.get( )
    << endl;

    return 0;
}
```

Here, T is any identifier. It acts like data type. Its type is decided by the user at the time at the creation of objects. T can be used any-where to declare a variable or constant. We can use any identifier in place of T, But, the convention is to take templates parameter as T.

There can be more than one parameterized data type:

Template < class T1, class T2>

Ex : Design class template for stack. Design push and pop member function out of the class.

```
//stack.cpp
# include <iostream.h>
#include <process.h>

const int MAX =20;

template <class T>
class Stack
{
protected:
    int top ;
    T ele [MAX];
public :
    //constructor
    Stack( )
    { top = -1 ;}
```

```
// initialize function
void initialize(void)
{ top = -1 ;}

// full and empty function.
int full( )
{ return (top == MAX-1) ;}

int empty( )
{ return (top== -1);}

//push function
void push (T x);

// pop function
T pop (void);
};
```

```

//definition of push
template <class T>
void Stack <T>::push(T x) {
    if (top == MAX-1 ) {
        cout << endl << "stack overflow " << flush;
        exit(1);
    }
    ele[++top] = x;
}

// definition of pop
template <class T>
T Stack <T>::pop () {
    T x ;
    if (top == -1) {
        cout << endl << "stack underflow " << flush;
        exit(1);
    }
    x = ele[top--];

    return x ;
}
////////////////////
int main( ) {
    Stack <float> s;           // T is float

    s.push(3.14F);
    s.push(7.5F);
    s.push(10.0F);

    //pop and display till not empty
    while ( !(s.empty() ) )
        cout << endl << s.pop( ) << flush;

    return 0;
}

```

Exercise: Design a template class SafeArray.

Data members:

T *p; int size;

Member Functions:

- C'tor SafeArray(int n =5); it allocates space for n members dynamically.
- Function to access the ith element of the array. Bound checking is required. Overload [] operator for accessing the element. Both operations must be supported:

```

a[i] = x;
x = a[i]

```

- Function **void** increase (int n); to increase the size by n, previous values are maintained.
- D'tor to de allocate the memory that was allocated in C'tor.
- Use exception handling to handle the cases when memory is not allocated.

5. Class Template Specialization:

Normally, class template is made when all data types works in a similar manner. The template reflects the general working. But, sometimes a class behaves in a different manner for a special data-type. It needs template specialization.

Ex.: There is a structure 'Date' and a class 'Sample'. The class has a member of type T, constructor, setter function, display function. Design a program so that sample class is able to create objects by taking T as int, float, Date.

```
//ClassTempSpec.cpp
//The first class template
#include <iostream.h>
```

```
//*****Date structure
struct Datex {
    int dd, mm, yy;
};
```

```
//*****Generic class
template<class T>
class Sample {
protected :
    T x ;
public :
    //constructor
    Sample(T xx =0)
    { x = xx ;}

    // setter function
    void set( T xx =0)
    { x = xx ;}

    // getter function
    T get (void)
    { return x ;}
};
```

```
//*****specialized class
```

```
template<Datex>
```

```
class Sample {
protected :
```

```
    Datex x ;
```

```
public :
```

```
    //constructor
```

```
    Sample(int d=1, int m = 1, int y =2000)
    { dd=d; mm = m; yy = y;}
```

```
    // setter function
```

```
    void set(int d, int m, int y)
    { dd=d; mm = m; yy = y;}
```

```
    // getter function
```

```
    void display() {
        cout << dd << "-" << mm << "-" << yy << endl;
    }
```

```
};
```

```
// -----main function
```

```
int main() {
```

```
    Sample <int> si(5);           // T is int
```

```
    cout << endl << si.get( );
```

```
    Sample <Datex> sd;           //specialized class used
```

```
    sd.display();
```

```
    return 0;
```

```
}
```

6. Template Member Function Specialization:

Sometimes, only some member function of class template behave in special manner for a data type. In this case its advisable to specialized only those function, not the complete class.

Ex : There is a structure 'Date' and a class 'sample'. The class has a member of type T, constructor, setter function, display function. Design a program so that sample class is able to create objects by taking T as int, float, Date.

The example is done with the help of function specialization.

```
//SampleMemTempSpec.cpp
//The first class template
#include <iostream.h>
//****Date structure
struct Datex {
    int dd, mm, yy;
};
//*****Generic class
template<class T>
class Sample
{
protected :
    T x ;
public :
    //constructor
    Sample()
    { }
    Sample(T xx)
    { x = xx ;}
```

```
    // setter function
    void set( T xx)
    { x = xx ;}

    // getter function
    void display (void);
};
```

```
template<class T>
void Sample<T>::display (void)
{ cout << x << endl;}
```

```
//specilaised member function foe Datex type
template <Datex>
void Sample<Datex>::display(void) {
    cout << endl << dd << "-" << mm << "-"
        << yy << endl;
};
```

```
// main function
int main() {
    Sample <int>    si(5);    // T is int
    si.display();

    Sample<Date>    sd;
    sd.display();           //specialized function used

    return 0;
}
```


7. Default Class Template Argument:

We can assign default data type to a class template argument. It is similar to default argument value in functions. Only trailing template arguments can be given the default type.

During object creation, if no data type is specified the default data type is selected.

Example :

default data type



```
Template <class T1, class T2 = int>
class sample
{
    :
    :
}
//main function
int main()
{
    sample <char, float> s1;    // T1 is char, T2 is float.
    sample <char>s2;           //T1 is char, T2 is int.

}
```

Note:

There are two types of parameters in class template:

(a) Nontype parameters

- Default arguments
- Treated as consts

template< class T, int elements >

Stack< double, 100 > mostRecentSalesFigures;

It declares object of type **Stack< double, 100>**

(b) Type parameter

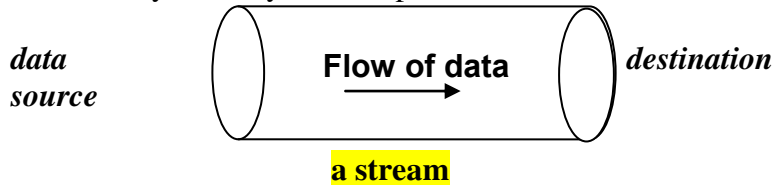
- Default type

template< class T = string >

1. Introduction & Basic Idea:

A stream is a general name given to a flow of data, it is a series of bytes, i.e., heading from a device to another. In case of file streams the two devices are connected by the stream:

- main memory of computer
- secondary memory of computer

**Input Stream:**

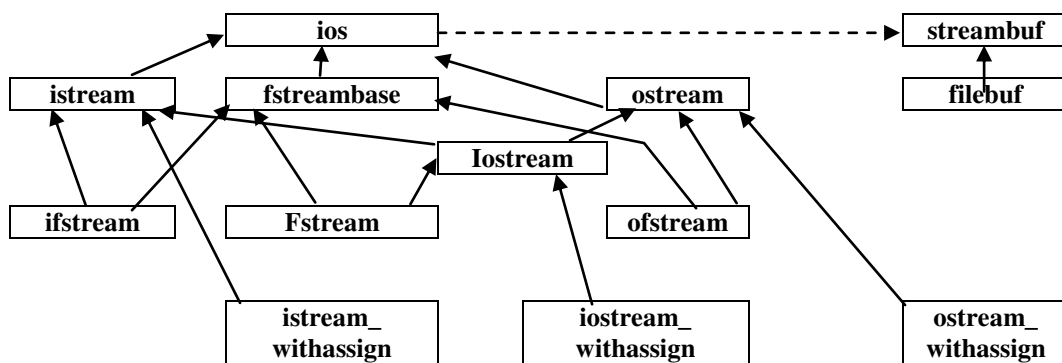
source: file on secondary storage or any input device
 destination: main memory

Output Stream:

source: main memory
 destination: file on secondary storage or any output device

Following are the advantages of concept of stream:

- The stream based I/O operations are simple.
- The difference between the working of various devices is abstracted or hidden from the user. So there is simplicity.
- We can overload existing operators on streams, thus making the operations easier.

2. The Stream Class Hierarchy:**Misc:**

1. The extraction operator (`>>`): The extraction is a operator overloading member function of `istream` class.
2. The insertion operator (`<<`): It is a operator overloading member function of `ostream` class.
3. The `cin` is a pre-defined object of `istream` class.
The `cout` is pre-defined object of `ostream` class.
4. `ios` is the base class that contains many constants and functions that are common to both input and output operations.

This class also contains following things:

- Formatting flags.
- Error status flags.
- File operation modes.
- Formatting flags are the set of enumerated definitions.

5. The *ios* class contains a pointer to *streambuff* class, that contains actual memory buffer into which data is read or written. Similarly, it also contains low level routines to handle the data.
6. The two *_withassign* classes contains overloaded assignment operator (=) additionally.
7. The *istream* class contains functions for reading operations. It also contains overloaded extraction operator function.
8. The *ostream* class contains functions for writing operation and it also contains the overloaded insertion operator.

3. File I/O Operations:

Following are the sequence to do file I/O operations:

1. Create an object of *ifstream*, *ofstream* or *fstream* according to the need. It will also be treated as stream name.
2. File name that is to be associated with this stream can be passed into the constructor. With *istream* object the file will be opened in input (read) mode. With *ostream* object the file will be opened in output (write) mode. With *fstream* object, the mode is read and write by default; it can be specified also.
3. Read or write operations are performed on the file through stream name.
4. The destructor of the class closes the stream automatically.

Ex: Design a program to create a file and write into it.

```
//opstream1.cpp: program to demonstate the output stream file
#include<iostream.h>
#include<fstream.h>
#include <process.h>

int main() {
    char ch='A';
    int x = 100;
    char str[5] = "RGPV";

    //create an o/p stream, associate it with file
    // file is automatically opened in write mode
    ofstream outfile( "outdata.txt" );
    if (outfile.good() != 1 ) {
        cout << "File not opened";
        exit(1);
    }

    outfile << ch << endl << x << endl << str;
    cout << endl << "file written" << endl;

    return 0;    //destructor called, file closed automatically
}
```

4. Error Status Bits (FLAG)

	NAME	PERPOSE
1.	goodbit	set when there is no error
2.	eofbit	set when eof reached
3.	failbit	set when user operation fails due to premature end of file.
4.	badbit	set when some invalid operation occur e.g. no buffer is associated with the stream.
5.	hardfail	unrecoverable error, set bit.

These bits are defined in ios class. These are simply enumerated data types.

Following are the functions for accessing the status bits:

	<u>NAME</u>	<u>PERPOSE</u>
1.	int = eof()	returns true if eof file is reached.
2.	int = fail()	returns true if fail bit, bad bit or hard flag is set.
3.	int = bad()	returns true if bad bit or hard flag is set.
4.	int = good()	it returns true, if everything is o.k.
5.	int = clear()	it clear all the bits to zero.
6.	int=clear(bit name)	it clears the specify bit to zero.

These are the members functions of *ios* class. These can be called by specifying object name and membership operator.

Examples: Design a program to read a file.

```
//inputstream.cpp:program to demonstate the output stream file
#include<iostream.h>
#include<fstream.h>
#include <process.h>
int main(){
    char ch; int x; char str[5];

    //create an o/p stream, associate it with file
    ifstream infile( "outdata.txt" );
    if (infile.bad() == 0 ) {
        cout << "File not opened";
        exit(1);
    }

    infile >> ch >> x >> str;
    cout << ch << endl << x << endl << str;

    return 0;    //destructor called, file closed automatically
}
```

Design a program to copy a file.

```
//inoutStream.cpp: program to demonstrate i/o stream
#include <iostream.h>
#include <fstream.h>
#include <process.h>
int main( ) {
    char ch;
    //create an input stream object
    //associate it with the source file.
    ifstream infile("outdata.txt");
    if ( infile.good() != 1 ) {
        cout << "File not opened";
        exit(1);
    }
    //create an output stream object
    //associate it with destination file.
    ofstream outfile( "destination.txt" );
    if (outfile.good() != 1 ) {
        cout << "File not opened";
        exit(1);
    }
    //read first character
    infile >> ch;
    //repeat loop till eof is reached
    while (!infile.eof()) {
        outfile << ch;
        infile >> ch; //get character from input file
    }
    cout <<endl << "copy done";
    return 0;}
}
```


5. Command Line Argument

Command line arguments are those arguments that are specified at Command line.

C:>

Let there is a program copy.c. We can create its exe file by F9, the copy.exe. The we can goto the command prompt and type as:

C:> copy sum.c sm.c



These all are called command line arguments. The main function in the sum.c will be:

```
void main(int argc, char * argv[ ]) {  
    :  
}
```

argc → It is argument count

argv → It is argument vector.

It is array of string each element in array is a string specified at commend line.

For the above example **argc will be 3.**

argv[0] = "copy"

argv[1] = "sum.c"

argv[2] = "sm.c"

Another example:

```
//hello.cpp : to demonstrate command line arguments  
#include <stdio.h>  
int main(int argc, char *argv[ ]) {  
    int i;  
  
    //loop to get command line parameters  
    for (i = 1; i < argc; ++i)  
        cout << "Hello " << argv[i] << endl;  
  
    return 0;  
}
```

Prepare a file hello.exe and type on the command line.

C:> hello ram shyam mohan

argc = 4

argv[0] = "hello"

argv[1] = "ram"

argv[2] = "shyam"

argv[3] = "mohan"

- The argument names argc and argv are simply identifiers we can take any other name also but it is a convention to take them as argc and argv.
- The argument supplied from the command line are in the form of string if any numeric processing is to be done on them, these must be converted into numbers.

Ex. Modify the file copy program so that the source and destination file name are given from command like: **(inoutStream2.cpp)**

copy.cpp is the file name. make the exe file copy.exe

then execute the following command.

c:\>Mycopy <source-file-name> <destination-file-name>

argc = 3

argv[0] = "Mycopy"

argv[1] = "outdata.txt"

argv[2] = "destination.txt"

Note :

in place of:

```
while (!infile.eof())
{ ... }
```

we can also write

```
while(infile)
{ ... }
```

On *eof* 0 (zero) will return.

6. The open() function:

We can open a file for various operations with the help of open function.

object name.open (file name, modes);

↑ ↑ ↑
object of fstream *membership* *modes may be more than one mode*
class *operator* *joined by | (pipeline character)*

The file may be closed explicitly, but due to destructor, it is automatically closed. Open function is a member function of fstream class.

VARIOUS MODES TO OPEN A FILE : The following are the modes in which a file can be open. These modes are defined in *ios* class. So these are by using *ios ::*

MODE	Result
1. in	for input operation. It is default mode for <i>ifstream</i> object
2. out	open for output operation. Its default mode for <i>ofstream</i> object.
3. ate	start reading or writing at <i>eof</i> .
4. app	start writing at <i>eof</i> .
5. trunc	truncates the file to zero length, if its exists.
6. nocreate	ERROR when opening, if file does not already exist.
7. noreplace	ERROR when opening a file for output if file already exist.
8. binary/text	open a file in binary/text mode.

Ex. Open a file in output mode with a precaution that no existing file is replaced.

```
fstream myfile;
myfile.open( "outfile.txt ", ios::out | ios::noreplace);

if (myfile.good() !=1) {
    cout<<"file is existing";
    exit (1);
}
```

Ex: open a file in input and output modes; do not open, if file does not already exist.

```
fstream myfile;
myfile.open( "file.txt", ios::in | ios::out | ios::nocreat);
if (myfile.fail() ) {
    cout<<endl<<"file does not exist";
    exist (1);
}
```

The *ofstream*, *ifstream* and *fstream* classes have destructors. These destructors contain statements for closing the file. Destructor is called as soon as the scope of the object goes out. So file is closed automatically. Hence there is no need of closing a file explicitly.

7. Type of I/O:

There are five types of IO operations possible on files:

- a) Formatted I/O
- b) Character I/O**
- c) String based I/O
- d) Binary I/O
- e) Object I/O**

(a) Formatted I/O: It is simply reading and writing of variable of any type to a file. Simply << and >> are used. Following is the example:

```
//formatOutput.cpp: writes formatted output to a file, using <<
#include <fstream.h>                //for file I/O
#include <iostream.h>
#include <string.h>

int main( ) {
    char ch = 'x';
    int j = 77;
    double d = 6.02;
    char str1[5] = "RGPV";           //strings without
    char str2[6] = "India";          //embedded spaces

    ofstream outfile("fdata.txt");    //create ofstream object

    outfile << ch << j                //insert (write) data
        << ' '                        //needs space between numbers
        << d << str1
        << ' '                        //needs spaces between strings
        << str2;
    cout << "File written\n";
    return 0;
}
```

The following example demonstrates formatted output:

```
//formatIn.cpp: reads formatted output from a file, using >>
#include <fstream.h>                //for file I/O
#include <iostream.h>
#include <string.h>

int main() {
    char ch; int j; double d;
    char str1[5]; char str2[6];

    ifstream infile("fdata.txt");    //create ifstream object
    //extract (read) data from it
    infile >> ch >> j >> d >> str1 >> str2;
    cout << ch << endl                //display the data
        << j << endl
        << d << endl
        << str1 << endl
        << str2 << endl;
    cout << endl << "file read successfully";
    return 0;
}
```

(b) Character I/O: A single character can be read and written to the disk file. Refer example on pages 2 & 3.

(c) String based I/O:

A complete string can be read or written from and to the file.

```

//stringOut.cpp: file output with strings
#include <fstream.h>
int main() {
    ofstream outfile("TEST.TXT"); //create file for output

    //send text to file
    outfile << "I fear thee, ancient Mariner!\n";
    outfile << "I fear thy skinny hand\n";
    outfile << "And thou art long, and lank, and brown,\n";
    outfile << "As is the ribbed sea sand.\n";
    return 0;
}

```

```

//stringIn.cpp : file input with strings
#include <fstream.h> //for file functions
#include <iostream.h>
int main( ){
    const int MAX = 80; //size of buffer
    char buffer[MAX]; //character buffer

    ifstream infile("TEST.TXT"); //create file for input

    while( !infile.eof() ) { //until end-of-file
        infile.getline(buffer, MAX); //read a line of text
        cout << buffer << endl; //display it
    }
    return 0;
}

```

(d) Binary I/O: File can be open in binary mode.

```

//binaryIO.cpp : binary input and output with integers
#include <fstream.h> //for file streams
#include <iostream.h>

const int MAX = 100; //size of buffer
int buff[MAX]; //buffer for integers

int main( ) {
    for(int j=0; j<MAX; j++) //fill buffer with data
        buff[j] = j; // (0, 1, 2, ...)

    ofstream os("edata.txt", ios::binary); //create output stream
    os.write( reinterpret_cast<char*>(buff), MAX*sizeof(int) );
    os.close(); //must close it

    for(j=0; j<MAX; j++) //erase buffer
        buff[j] = 0;

    //display data
    for(j=0; j<MAX; j++)
        cout << buff[j] << "\t";

    ifstream is("edata.txt", ios::binary); //create input stream
    is.read( reinterpret_cast<char*>(buff), MAX*sizeof(int) );

    //check data
    for(j=0; j<MAX; j++)
        if ( buff[j] != j ) {
            cerr << "Data is incorrect\n";
            return 1;
        }
    cout << "Data is correct\n";
    //display data
    for(j=0; j<MAX; j++)
        cout << buff[j] << "\t";
    return 0;
}

```

(vi) **Object Based I/O:** A complete record/object can be read from or written to the file. In the following example, we create a file and write a person type object into it.

```
// objectOut.cpp :saves person object to disk
#include <fstream.h>           //for file streams
#include <iostream.h>
class person {                //class of persons
protected:
    char name[80];            //person's name
    short age;                //person's age
public:
    void getData(){           //get person's data
        cout << "Enter name: "; cin >> name;
        cout << "Enter age: ";   cin >> age;
    }
};

int main( ){
    person pers;              //create a person
    pers.getData();           //get data for person

    //create ofstream object
    ofstream outfile("PERSON.DAT", ios::binary);

    //write to it
    outfile.write(reinterpret_cast<char*>(&pers), sizeof(pers));
    return 0;
}
```

Read the person type objects from the file just prepared.

```
// objectIn.cpp : reads person object from disk
#include <fstream.h>           //for file streams
#include <iostream.h>

class person {                //class of persons
protected:
    char name[80];            //person's name
    short age;                //person's age
public:
    void showData() {         //display person's data
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

int main( ) {
    person pers;              //create person variable

    //create stream
    ifstream infile("PERSON.DAT", ios::binary);

    //read stream
    infile.read( reinterpret_cast<char*>(&pers), sizeof(pers) );

    pers.showData();          //display person
    return 0;
}
```

8. File Pointer:

There are two types of pointers for files:

- get pointers
- put pointers

The *get pointer* is used in case of reading a file.

The *put pointer* is used in case of writing a file.

There are two types of functions:

- seek function
- tell functions

The *seek() functions* are used to randomly position pointer in a file.

The *tell() functions* are used to return the position of pointers.

There are four combinations:

	functions	purpose
1.	seekg()	It is used to randomly position a get or input pointer.
2.	seekp()	It is used to randomly position a put or output pointer.
3.	tellg()	It is used to get the position of get or input pointer.
4.	tellp()	It is used to get the position of put or output pointer.

seekg() and tellg() can only be used with ifstream type objects.

seekp() and tellp() can only be used with ofstream type objects.

The syntax of *seek()* functions is:

object-name.seekg(int byte offset, ios::flag);
object-name.seekp(int byte offset, ios::flag);

The object is the stream type object.

Where the *flag* may be:

- beg- position from the beginning
- cur - position from the current
- end- position from the end

the offset is the number of bytes by which the pointer is to be moved.

```
infile.seekg(10, ios::beg);  
infile.seekg(-10, ios::end);  
infile.seekg(10, ios::cur);  
infile.seekg(-10, ios::cur);
```

The syntax of tell() functions is:

object-name.tellg();
object-name.tellp();

Ex: Design a program to encrypt a file. The file_name will be supplied from the command line. Design a program to de-crypt the file encrypted. The file name will be supplied from the command line.

Ex: Design a program to write and read person object to and from a file.

```
//objSeveralIO.cpp  
// reads and writes several objects to disk  
  
#include <fstream.h>                                //for file streams  
#include <iostream.h>  
  
class person                                         //class of persons  
{  
    protected:  
        char name[80];                             //person's name  
        int age;                                    //person's age  
    public:  
        void getData()                             //get person's data  
        {  
            cout << "\n    Enter name: "; cin >> name;  
            cout << "    Enter age: "; cin >> age;  
        }  
        void showData()                             //display person's data  
        {  
            cout << "\n    Name: " << name;  
            cout << "\n    Age: " << age;  
        }  
};
```

```

int main(){
    char ch;
    person pers;                //create person object
    fstream file;               //create input/output file

    //open for append
    file.open("GROUP.DAT", ios::app | ios::out | ios::in | ios::binary );

    //loop to enter data from user to file
    do {
        cout << "\nEnter person's data:";
        pers.getData();          //get one person's data

        //write to file
        file.write( reinterpret_cast<char*>(&pers), sizeof(pers) );
        cout << "Enter another person (y/n)? ";
        cin >> ch;
    } while(ch=='y');            //quit on 'n'

    file.seekg(0);               //reset to start of file

    //read first person
    file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );

    //loop to read from file
    while( !file.eof() ) {       //quit on EOF
        cout << "\nPerson:";     //display person
        pers.showData();         //read another person
        file.read( reinterpret_cast<char*>(&pers), sizeof(pers) );
    }
    cout << endl;
    return 0;
}

```

Desing a function:
 void readOnePersonData(int n);
 program : readOnePerson.cpp

9. Overloading of Insertion and Extraction Operator:

We can do input and output operation on simple data type as:

```

int x;
cin >> x;           //cin.operator>>(x);
cout<<x;            //cout.operator<<(x);

```

But, the insertion and extraction operators are not applicable directly for object based input and output operations. To support the following operations we will have to overload the operators.

Let *Person* is a class

```

Person p;
cin >> p;    //cin.operator>>(p);    operator >>(cin, p);
cout << p;    //cout.operator<<(p);    operator<<(cout, p);

```

Ex: Design a *person* class that supports the overloaded operator operations of insertion and extraction operators. The person class contains two members, name & age.

Sol: To overload extraction and insertion operator, we will have to design a non member friend function of person-class, which has following two parameters.

- First parameter is: the object of istream & ostream class.
- Second parameter is: object of person-class.


```

//overloadPers.cpp
//overloading of << and >> operators on standard objects cin and cout
#include <iostream.h>
#include <fstream.h>

class Person {
protected :
char name [20];
int age;
public :
//non-member friend function
friend istream& operator>>(istream& io, Person& p);
friend ostream& operator<<(ostream& oo, const Person& p);
};

// >>operator function
istream& operator>>(istream& io, Person& p){
cout << endl << "enter name :";
cin >> p.name;
cout << endl << "enter age:";
cin >> p.age;
return io; //for cascading
}

//<< operator function
ostream& operator<<(ostream& oo, const Person& p){
cout << endl << "name :"<< p.name
<< " \t "<< "Age: "<< p.age;
return oo; //for cascading
}

//main function
int main( ){
Person per;

cin >> per; //read Person data
cout << per; //display Person data

return 0;
}

```

Ex: Design the Distance class to support following main() behavior.

```

//overloadDist.cpp
int main(){
    Distance dist1, dist2; //define Distances
    Distance dist3(11, 6.25); //define, initialize dist3

    cout << "\nEnter two Distance values:";
    cin >> dist1 >> dist2; //get values from user

    //display distances
    cout << "\ndist1 = " << dist1 << "\ndist2 = " << dist2;
    cout << "\ndist3 = " << dist3 << endl;
    return 0;
}

```

Ex: Overload the insertion and extraction operators to:

- write a Distance object to a file
- read a Distance object from a file.

Design the main function also.

```

//overFile.cpp: overloaded << and >> operators can work with files
#include <fstream.h>
#include <iostream.h>
class Distance { //English Distance class
private:
    int feet;
    float inches;

public:
    Distance() : feet(0), inches(0.0) //constructor (no args)
    { }
    Distance(int ft, float in) : feet(ft), inches(in)
    { }
    void getDist(void){
        cout << endl << "enter feet:"; cin >> feet;
        cout << endl << "enter inches:"; cin >> inches;
    }

    friend istream& operator >> (istream& s, Distance& d);
    friend ostream& operator << (ostream& s, Distance& d);
};

//get distance from file or keyboard with overloaded >> operator
istream& operator >> (istream& s, Distance& d){
    char dummy; //for ('), (-), and (")
    s >> d.feet >> dummy >> dummy >> d.inches >> dummy;
    return s;
}

//send Distance object to file or to screen with overloaded<< operator
ostream& operator << (ostream& s, Distance& d) {
    s << d.feet << "\'-" << d.inches << '\\"';
    return s;
}
//-----
int main( ){
    char ch;
    Distance dist1;
    ofstream ofile; //create and open
    ofile.open("DIST.DAT"); //output stream

    do {
        cout << "\nEnter Distance: ";
        dist1.getDist(); //get distance from user
        ofile << dist1; //write it to output str
        cout << "Do another (y/n)? ";
        cin >> ch;
    } while(ch != 'n');

    ofile.close(); //close output stream

    //now open file for read operation
    ifstream ifile; //create and open
    ifile.open("DIST.DAT"); //input stream

    cout << "\nContents of disk file are:\n";
    while(true)
    {
        ifile >> dist1; //read dist from stream
        if( ifile.eof() ) //quit on EOF
            break;
        cout << "Distance = " << dist1 <<endl;
    }
    return 0;
}

```

10. Binary v/s Text Files:

There are three major differences:

(1) Newline character: In binary mode file, the end of line is represented by a single character which has ASCII value 10. But in the text mode file, the end of line is marked by two characters:

- Carriage returned CR with ASCII value 13 and
- Line feed character LF with ASCII Value 10

The text mode follow CR, LF combination which is compatible with DOS. The binary mode follows '\n' alone which is compatible with 'C++' format.

2. End of file :- In the text mode file, a special character EOF (^Z) with ASCII value 26 is stored at the end of the file. There is n such character in the binary mode file; the OS keeps track of the size of file, and it returns the appropriate signal.

3.Storing of numbers: Numbers in text format are stored as string of characters; but in binary format these are stored the same way as they are stored in computers main memory.

Exercise:

1) You are the owner of a hardware store and need to keep an inventory that can tell you what different tools you have, how many of each you have on hand and the cost of each one. Write a program that initializes the random-access file "**hardware.dat**" to one hundred empty records, lets you input the data concerning each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update *any* information in the file. The tool identification number should be the record number. Use the following information to start your file: (folder hardware)

Record	# Tool name	Quantity	Cost
3	Electric sander	7	57.98
17	Hammer	76	11.99
24	Jig saw	21	11.00
39	Lawn mower	3	9.50
56	Power saw	18	99.99
68	Screwdriver	106	6.99
77	Sledge hammer	11	21.50
83	Wrench	34	7.50

2) Design a class Date that has dd, mm and yy. The member functions are parameterized Ctor, set() [overload >> operator], display() [overload << operator] and check(). Throw exception if the date is not proper. The ctor and set function assigns the data.

Design a class Student, that have the following member data:

rollno, name, date-of-birth, (date of birth is Date type) and marks.

The member functions are:

Ctor, set(), display(), readFromKb().

Overload << and >> operators to write on a stream and to read from the stream.

Design a function:

void displays(int roll);

It displays the record of the student with roll number supplied. (use random accessing)

Design a suitable main().

cascading

```
cin >> x >> y;
```

```
(cin.operator>>(x) ).operator>>(y)
```

cascading

```
cin >> p1 >> p2;
```

```
operator>>( operator>>(cin, p1) ), p2)
```

Exercise 2:

DAF

```
void getOnePersonData(char * filename, int n);
```

The function reads the data of only nth student, and displays it. Use the Person class that is designed already.

(demo main of getOnePersonData.cpp)

Exercise 3:

```
int isContained(char *filename, char *str);
```

(show main of isContained.cpp)

1. Namespaces:

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is:

```
namespace identifier
{
    entities
}
```

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
namespace myNamespace
{
    int a, b;
}
```

In this case, the variables a and b are normal variables declared within a namespace called myNamespace. In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::. For example, to access the previous variables from outside myNamespace we can write:

```
myNamespace::a
myNamespace::b
```

The functionality of namespaces is especially useful in the case that there is a possibility that a global object or function uses the same identifier as another one, causing redefinition errors. For example:

<pre>// namespaces #include <iostream> using namespace std; namespace first { int var = 5; }</pre>	<pre>5 3.1416</pre>		
<pre>namespace second { double var = 3.1416; } int main () { cout << first::var << endl; cout << second::var << endl; return 0; }</pre>	<table border="1"> <tr> <td data-bbox="925 1444 1291 1816"> <pre>int main () { using namespace first; using namespace second; cout << var << endl; cout << var << endl; return 0; }</pre> </td><td data-bbox="1291 1444 1490 1816"> <p><u>ERROR</u></p> </td></tr> </table>	<pre>int main () { using namespace first; using namespace second; cout << var << endl; cout << var << endl; return 0; }</pre>	<p><u>ERROR</u></p>
<pre>int main () { using namespace first; using namespace second; cout << var << endl; cout << var << endl; return 0; }</pre>	<p><u>ERROR</u></p>		

In this case, there are two global variables with the same name: var. One is defined within the namespace first and the other one in second. No redefinition errors happen thanks to namespaces.

using

The keyword using is used to introduce a name from a namespace into the current declarative region. For example:

<pre>#include <iostream> using namespace std; namespace first{ int x = 5; int y = 10; } namespace second{ double x = 3.1416; double y = 2.7183; } int main () { using first::x; using second::y; cout << x << endl; cout << y << endl; cout << first::y << endl; cout << second::x << endl; return 0; }</pre>	<pre>5 2.7183 10 3.1416</pre>
---	-------------------------------

Notice how in this code, x (without any name qualifier) refers to first::x whereas y refers to second::y, exactly as our using declarations have specified. We still have access to first::y and second::x using their fully qualified names.

The keyword using can also be used as a directive to introduce an entire namespace:

<pre>// using #include <iostream> using namespace std; namespace first{ int x = 5; int y = 10; } namespace second{ double x = 3.1416; double y = 2.7183; } int main () { using namespace first; cout << x << endl; cout << y << endl; cout << second::x << endl;</pre>	<pre>5 10 3.1416 2.7183</pre>
--	-------------------------------

```
cout << second::y << endl;
return 0;
}
```

In this case, since we have declared that we were using namespace first, all direct uses of x and y without name qualifiers were referring to their declarations in namespace first. using and using namespace have validity only in the same block in which they are stated or in the entire code if they are used directly in the global scope. For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

<pre>// using namespace example #include <iostream> using namespace std; namespace first { int x = 5; } namespace second { double x = 3.1416; } int main () { { using namespace first; cout << x << endl; } { using namespace second; cout << x << endl; } return 0; }</pre>	<pre>5 3.1416</pre>
---	---------------------

Namespace alias

We can declare alternate names for existing namespaces according to the following format:

```
namespace new_name = current_name;
```

Namespace std

All the files in the C++ standard library declare all of its entities within the std namespace. That is why we have generally included the using namespace std; statement in all programs that used any entity defined in iostream.

2. Casting:

Converting an expression of a given type into another type is known as *type-casting*. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example:

```
short a=2000;
int b;
b=a;
```

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This can be avoided with an explicit conversion. Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions. For example:

```
class A { };
class B { public: B (A a) { } };

A a;
B b=a;
```

Here, a implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
int a=2000;
short b;
b = (int) a; // c-like cast notation
b = int (a); // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. **However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors.** For example, the following code is syntactically correct:

```
// class type-casting
#include <iostream>
using namespace std;

class CDummy {
    float i, j;
};

class CAddition {
    int x,y;
public:
    CAddition (int a, int b) { x=a; y=b; }
    int result() { return x+y; }
};

int main () {
    CDummy d;
    CAddition * padd;
    padd = (CAddition*) &d;
    cout << padd->result( );
    return 0;
```



```
} 
```

The program declares a pointer to CAddition, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

```
padd = (CAddition*) &d;
```

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. **The subsequent call to member result will produce either a run-time error or a unexpected result.**

In order to control these types of conversions between classes, we have four specific casting operators: `dynamic_cast`, `reinterpret_cast`, `static_cast` and `const_cast`. Their format is to follow the new type enclosed between angle-brackets (<>) and immediately after, the expression to be converted between parentheses.

```
dynamic_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
static_cast <new_type> (expression)
const_cast <new_type> (expression)
```

The traditional type-casting equivalents to these expressions would be:

```
(new_type) expression
new_type (expression)
```

but each one with its own special characteristics:

dynamic_cast

dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class. Therefore, `dynamic_cast` is always successful when we cast a class to one of its base classes:

```
class CBase { };
class CDerived: public CBase { };
```

```
CBase b; CBase* pb;
CDerived d; CDerived* pd;
```

```
pb = dynamic_cast<CBase*>(&d); // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b); // wrong: base-to-derived
```

The second conversion in this piece of code would produce a compilation error since base-to-derived conversions are not allowed with `dynamic_cast` unless the base class is polymorphic. When a class is polymorphic, `dynamic_cast` performs a special checking during runtime to ensure that the expression yields a valid complete object of the requested class:

<pre>// dynamic_cast #include <iostream> #include <exception> using namespace std; class CBase { virtual void dummy() {} }; class CDerived: public CBase { int a; }; int main () { try { CBase * pba = new CDerived; CBase * pbb = new CBase;</pre>	Null pointer on second type-cast
---	-------------------------------------

```

CDerived * pd;

pd = dynamic_cast<CDerived*>(pba);
if (pd==0) cout << "Null pointer on first type-cast" << endl;

pd = dynamic_cast<CDerived*>(pbb);
if (pd==0) cout << "Null pointer on second type-cast" << endl;

} catch (exception& e) {cout << "Exception: " << e.what();}
return 0;
}

```

Compatibility note: `dynamic_cast` requires the [Run-Time Type Information \(RTTI\)](#) to keep track of dynamic types. Some compilers support this feature as an option which is disabled by default. This must be enabled for runtime type checking using `dynamic_cast` to work properly.

The code tries to perform two dynamic casts from pointer objects of type `CBase*` (`pba` and `pbb`) to a pointer object of type `CDerived*`, but only the first one is successful. Notice their respective initializations:

```

CBase * pba = new CDerived;
CBase * pbb = new CBase;

```

Even though both are pointers of type `CBase*`, `pba` points to an object of type `CDerived`, while `pbb` points to an object of type `CBase`. Thus, when their respective type-castings are performed using `dynamic_cast`, `pba` is pointing to a full object of class `CDerived`, whereas `pbb` is pointing to an object of class `CBase`, which is an incomplete object of class `CDerived`.

When `dynamic_cast` cannot cast a pointer because it is not a complete object of the required class -as in the second conversion in the previous example- it returns a null pointer to indicate the failure. [If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead.](#)

`dynamic_cast` can also cast null pointers even between pointers to unrelated classes, and can also cast pointers of any type to void pointers (`void*`).

static_cast

[static_cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived.](#) This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of `dynamic_cast` is avoided.

```

class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);

```

This would be valid, although `b` would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

`static_cast` can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

reinterpret_cast

reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by `reinterpret_cast` but not by `static_cast` have no specific uses in C++ are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

const_cast

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter:

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

void print (char * str)
{
    cout << str << endl;
}

int main () {
    const char * c = "sample text";
    print ( const_cast<char *> (c) );
    return 0;
}
```

sample text

typeid

`typeid` allows to check the type of an expression:

```
typeid (expression)
```

This operator returns a reference to a constant object of type `type_info` that is defined in the standard header file `<typeinfo>`. This returned value can be compared with another

one using operators == and != or can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member.

<pre>// typeid #include <iostream> #include <typeinfo> using namespace std; int main () { int * a, b; a=0; b=0; if (typeid(a) != typeid(b)) { cout << "a and b are of different types:\n"; cout << "a is: " << typeid(a).name() << "\n"; cout << "b is: " << typeid(b).name() << "\n"; } return 0; }</pre>	<p>a and b are of different types: a is: int * b is: int</p>
---	--

When typeid is applied to classes typeid uses the RTTI to keep track of the type of dynamic objects. When typeid is applied to an expression whose type is a polymorphic class, the result is the type of the most derived complete object:

<pre>// typeid, polymorphic class #include <iostream> #include <typeinfo> #include <exception> using namespace std; class CBase { virtual void f(){} }; class CDerived : public CBase {}; int main () { try { CBase* a = new CBase; CBase* b = new CDerived; cout << "a is: " << typeid(a).name() << "\n"; cout << "b is: " << typeid(b).name() << "\n"; cout << "*a is: " << typeid(*a).name() << "\n"; cout << "*b is: " << typeid(*b).name() << "\n"; } catch (exception& e) { cout << "Exception: " << e.what() << endl; } return 0; }</pre>	<p>a is: class CBase * b is: class CBase * *a is: class CBase *b is: class CDerived</p>
--	---

Notice how the type that typeid considers for pointers is the pointer type itself (both a and b are of type class CBase *). However, when typeid is applied to objects (like *a and *b) typeid yields their dynamic type (i.e. the type of their most derived complete object). If the type typeid evaluates is a pointer preceded by the dereference operator (*), and this pointer has a null value, typeid throws a bad_typeid exception.

3. Manipulators:

Manipulators are the most common way to control output formatting.

#include <iomanip>

I/O *manipulators* that take parameters are in the **<iomanip>** include file.

Default Floating-point Format

Unless you use I/O manipulators (or their equivalent), the default format for each floating-point number depends on its value.

- No decimal point: 1.0000 prints as 1
- No trailing zeros: 1.5000 prints as 1.5
- Scientific notation for large/small numbers: 1234567890.0 prints as 1.23457e+09

I/O Manipulators

The following output manipulators control the format of the output stream. Include **<iomanip>** if you use any manipulators that have parameters. The Range column tells how long the manipulator will take effect: *now* inserts something at that point, *next* affects only the next data element, and *all* affects all subsequent data elements for the output stream.

Manip.	Rng	Description
<i>General output</i>		
endl	now	Write a newline ('\n') and flush buffer.
setw(n)	next	Sets minimum field width on output. This sets the minimum size of the field - a larger number will use more columns. Applies only to the next element inserted in the output. Use left and right to justify the data appropriately in the field. Output is right justified by default. Equivalent to <code>cout.width(n)</code> ; To print a column of right justified numbers in a seven column field: <code>cout << setw(7) << n << endl;</code>
width(n)	next	Same as setw(n) .
left	next	Left justifies output in field width. Only useful after setw(n) .
right	next	Right justifies output in field width. Since this is the default, it is only used to override the effects of left . Only useful after setw(n).
setfill(ch)	all	Only useful after setw . If a value does not entirely fill a field, the character <i>ch</i> will be used to fill in the other characters. Default value is blank. Same effects as cout.fill(ch) For example, to print a number in a 4 character field with leading zeros (eg, 0007): <code>cout << setw(4) << setfill('0') << n << endl;</code>
<i>Floating point output</i>		
setprecision(n)	all	Sets the number of digits printed to the right of the decimal point. This applies to all subsequent floating point numbers written to that output stream. However, this won't make floating-point "integers" print with a decimal point. It's necessary to use fixed for that effect. Equivalent to <code>cout.precision(n)</code> ;

fixed	all	Used fixed point notation for floating-point numbers. Opposite of scientific . If no precision has already been specified, it will set the precision to 6.
scientific	all	Formats floating-point numbers in scientific notation. Opposite of fixed .
<i>bool output</i>		
boolalpha noboolalpha	all	Uses alphabetic representation (true and false) for bool values. Turned off with noboolalpha .
<i>Input</i>		
skipws noskipws	all	For most input values (eg, integers and floating-point numbers), skipping initial whitespace (eg, blanks) is very useful. However, when reading characters, it is often desired to read the whitespace characters as well as the non-spacing characters. The these I/O manipulators can be used to turn whitespace skipping off and on. Eg, cin >> noskipws; turns whitespace skipping off for all subsequent cin input.
ws	now	Reads and ignores whitespace at the current position.
<i>Other</i>		
		showpoint, noshowpoint, uppercase, nouppercase, dec, oct, hex, setbase(8 10 16), showbase, noshowbase, ends, showpos, noshowpos, internal, flush, unitbuf, nunitbuf, setiosflags(f), resetiosflags(f)

Example

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    const float tenth = 0.1;
    const float one = 1.0;
    const float big = 1234567890.0;

    cout << "A. " << tenth << ", " << one << ", " << big << endl;
    cout << "B. " << fixed << tenth << ", " << one << ", " << big << endl;
    cout << "C. " << scientific << tenth << ", " << one << ", " << big << endl;
    cout << "D. " << fixed << setprecision(3) << tenth
        << ", " << one << ", " << big << endl;
    cout << "E. " << setprecision(20) << tenth << endl;
    cout << "F. " << setw(8) << setfill('*') << 34 << 45 << endl;
    cout << "G. " << setw(8) << setfill('*') << 34 << setw(8) << 45 << endl;

    return 0;
}
```

produces this output:

- A. 0.1, 1, 1.23457e+009
- B. 0.100000, 1.000000, 1234567936.000000
- C. 1.000000e-001, 1.000000e+000, 1.234568e+009
- D. 0.100, 1.000, 1234567936.000
- E. 0.1000000014901161
- F. *****3445
- G. *****34*****45

Lines F and G show the scope of **setw()** and **setfill()**.

Pointer to functions

Variable argument list

functors