Object Oriented Programming [ETCS-210]

Dr. A K Yadav
Amity School of Engineering and Technology
(affiliated to GGSIPU, Delhi)
akyadav1@amity.edu
+91 9911375598

April 19, 2019

Features of OOP: I

- Class
- Object
- Encapsulation: Encapsulate data and code into a single unit to protect from external usages. Apply using class.
- Data Abstraction: Crating new data type for real world application with their operations. class is used to create ADT.
- Inheritance: Inherit the property of existing class without touching the existing class.
- Polymorphism: One operator or function can be used for more than one purpose. Example operator and function overloading.
- Information Hiding:
- Message Passing: Invoking an operation with object. Example Operator overloading.
- Extensibility: Existing software can be extended in functionality by using Abstract classes and inheritance.

Features of OOP: II

- Persistence: Object or data persist or live during different executions. Not supported but can be used via files.
- Delegation: alternative to class inheritance, object composition, receiving object delegates operation to its delegate.
- Genericity: same function can be used for different data types.

Operator: cin, cout, new, delete

- cin: cin>>varname, scanf().
- cout: cout<<"..."<<varname, printf().
- new: used in dynamic memory allocation for efficient use of memory. syntax: DataType *=new DataType[size]. void *malloc (sizeof(DataType)*size).
- delete: ensures safe and efficient use of memory. very useful in local dynamic memory allocation. Memory freed up automatically after program termination. syntax delete pointer, free((DataType*)pointer).

void and wild pointer

- void pointer is a generic pointer and very helpful in programming, it reduces the ptr declaration.
- A pointer becomes wild when it is pointing to a unallocated memory. ptr becomes wild under 3 cases:
  - int *ptr; cout<<*ptr;: Pointer declared but not initialized.
  - int *ptr=new int; ptr++, ptr=new int, cout<<*ptr;: Pointer modified :-garbage memory
  - int *ptr=new int; delete ptr, cout<<*ptr;: Pointer destroyed :-dangling reference
  - Side effect of this is : garbage memory (hidden money but not know where) and dangling reference

Memory Type

- Stack Memory
- Heap Memory

main function

- int main () body
- int main (int argc, char *argv[]) body
  - argc - Non-negative value representing the number of arguments passed to the program from the environment in which the program is run.
  - argv - Pointer to the first element of an array of pointers to null-terminated multibyte strings that represent the arguments passed to the program from the execution environment (argv[0] through argv[argc-1]). The value of argv[argc] is guaranteed to be a null pointer.
  - body - The body of the main function
- void main()-don't use it is a non standard function.

Storage Classes I

Storage classes except extern are used for defining variables, but extern is used for declaration.
-Declaration informs informs compiler about existence of the data or function
-But definition allocates memory as per datatype.
-Multiple declaration but one definition is allowed.

- Auto: by default all are auto.
  -auto DataType varname;
  -scope and life is local only within the {}
  -uses stack memory
  -Default value is garbage

Storage Classes II

- Register: register DataType varname;
  -scope and life is local only within the {},
  -uses cpu registers to store data for fast processing.
  -is not a command but request to compiler.
  -limited only two or three can be used else will be treated as auto.
  -uses register memory
  -memory is not referred.
  -Default value is garbage
- Static : static DataType varname;
  -use shared memory within calls.
  -scope is local only within the block maximum up to a file but life is full program execution.
  -Declaration may be local or global but have no effect.
  -memory from global heap.
  -static var defined outside of all fun in a file like global is called file static var. -Default value is 0.

Storage Classes III

- External :extern DataType varname;
  -use shared memory scope is global with in all file.
  -multiple declaration but one definition.
  -memory from global heap.
  -Default value is 0.

Structure of program

Comments

-Single Line
-Multi Line

Constants

-define

-enum

-const

Scope Resolution

Variable Definition

-at the point of use
-in the beginning of fun

Reference Variable

-as powerful as pointer
-as simple as simple variable
-declaration at the point of definition
-datatype & *varname* = *existing_varname*;

Inline Function

Function Overloading

Default Arguments

Class and Object

-Class is only a blue print so don't have any space.
-Object is an instance of a class so having some space and physical existence.
-Size of a object is the sum of size of its non static data members.
-Function defined inside the class is in-line by default.
-Syntex
class ClassName
{
Visibility specifiers(by default private to provide security)
Data Members
Member Methods
};

Continued...

- Accessing of objects
- Empty Class or stubs
- Passing object as arguments
- Returning objects from functions
- Friend function and Friend class
- Constants parameters and Member Function.
- Structures and Classes
- Static Data members and member functions.

Constructor and Destructor I

Constructor/Destructor: It is a special member function having some unique properties such as:

- Name of the Constructor/Destructor is same as name of the class.
- Name of the Destructor is preceded by ~
- They have no return type even void.
- They are called automatically without their explicit call.
- Constructor are called automatically whenever they are created.
- Destructor are called automatically whenever they are out of scope.
- Constructor can be overloaded and they are of 3 types.
- Destructor can not be overloaded and only one type.
- Destructor don't have any argument even void.
- Constructor can not be virtual.
- Destructor can be virtual.

- Why Destructor are not allowed to pass arguments?
- Why Destructor are not allowed to overload?
- What is the order of calling of Constructor and Destructor?
- Constructor with default agr.
- Nameless Objects.
- Write a program to count the total objects created and total live object.

Constant Data Member and Constant Objects

- Constant Data Member
- Constant Objects
- Static Data Members with Constructor and Destructor
- Nested Class
- Local Class

Dynamic Objects

- Pointers to Objects: $ClassName * ptr$, $ptr = \&object, ptr = newClassName$
- Accessing Members of Objects: $ptr- > memberName$, $*ptr.membername$
- Creating and Deleting Dynamic Objects
- Reference to Dynamic Objects, no delete for reference
- Live Objects: Created dynamically with Constructor initialization
- Array of Objects; $ClassName$ $objectName[size], objectName[i].member$
- Array of pointers to Objects: $ClassName$ *o[10]; o[i]=new $ClassName[size\_of\_i]$
- *this* pointer
- Self Referential Classes

Operator Overloading

- Type of Operator
- Need of Operator Overloading
- Syntex of Operator Overloading

Operator Overloading(continued...) I

Table: C++ overloadable Operators

Operator Category	***Operators***
Arithmetic Operator	$+, -, *, /, \%$
Bitwise Operator	$\&, \vert, \ ,$
Logical Operator	$\&\&, \vert\vert, !$
Relational Operator	$>, <, ==, != , <=, >=, \vert =,$
Assignment Operator	$=$
Arithmetic Assignment Operator	$+ =, - =, * =, / =,$
Shift Operator	$<<, >>, <<=, >>=$
Unary Operator	$++, --, +, -$
Subscripting Operator	[]
Function Call Operator	()
Dereferencing Operator	$->$
Memory Allocate and Free	*new*, *delete*

Operator Overloading(continued...)

Non overloadable Operator

- Member access Operator (.)
- :: Scope Operator (::)
- ?: Conditional Operator (?:)
- Pointer to member Operator (*)
- sizeof Operator (sizeof())

Operator Overloading(continued...)

- You cann't change the precedence.
- You cann't change numbers of operand.
- You cann't change syntax.
- You cann't define new operators.
- You can ignore operand.
- You cann't have default arguments except function call operator.
- Assignment $=$, Subscripting [], Function Call (), Dereferencing $->$ operator can only be overloaded by member function.

Operator Overloading(continued...)

- new : void * operator new ($size\_t$). new UT
- new[] : void * operator new []($size\_t$). new UT[10]
- delete : void operator delete ($void * p$). delete p
- delete[] : void operator delete[]($void * p$). delete[] p

Pointers to Object Members

- Declaration: DataType $ClassName :: *ptr$
- Assignment: $ptr = \&ClassName :: Member$ //member data type must be DataType
- Use: $ObjectName.*ptr$ or $PointerToObject->*ptr$
- Declaration: ReturnType $(ClassName :: *fptr)(arg)$
- Assignment: $fptr = \&ClassName :: MemberMethod$ //member method prototype must match
- Use: $(ObjectName.*fptr)(arg)$ or $(PointerToObject->*fptr)(arg)$

Inheritance

Type of Inheritance

- Single Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Multipath Inheritance
- Hybrid Inheritance

Inheritance(continued...) I

- Privete members can't be inherited, so can be accessed only through class members of their own.
- Protected members can be accessed by members of its own class and their derived class.
- The visibility mode can be private, protected and public.
- Default visibility mode is private.
- Members used in in-line function must be declared before in-line function definition.
- Constructors of base class and derived class invoked automatically when derived class objects created.
- Normally Base class constructors are used to initialize the base class data members and derived class constructors are used to initialize the derived class data members.
- Base class members are override by the derived class members if the name/prototype is same.

Inheritance(continued...) II

- Overridden Base class members can be accessed using scope resolution operator
  DerivedOjectName.*BaseClassName* :: *Member*.

Inheritance(continued...) I

- No constructor in base class and derived class
- Constructor only in base class
- Constructor only derived class
- Constructor in both base class and derived class
- Multiple constructor in base class and single constructor in derived class
- Constructor in base class and derived class without no-argument constructor
- Explicit invocation in the absence of no-argument constructor
- Constructor in multiple inheritance
- Constructor in multilevel inheritance
- Constructor in virtual class

```
#include<iostream.h>
#include<alloc.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>

class B1
{
private:
int i;
public:
B1()
{
i=0;
}
B1(int i)
{
this->i=i;
}
```

```
void input()
{
cout<<"\nEnter value of i=";
cin>>i;
}
void display()
{
cout<<"\ni="<<i;
}
};

class B2
{
private:
int j;
public:
B2()
{
j=0;
```

```
}
B2(int j)
{
this->j=j;
}
void input()
{
cout<<"\nEnter value of j=";
cin>>j;
}
void display()
{
cout<<"\nj="<<j;
}
};

class D:public B1, public B2
{
private:
```

```
int k;
public:
D()
{
k=0;
}
D(int i, int j, int k):B1(i), B2(j)
{
this->k=k;
}
void input()
{
B1::input();
B2::input();
cout<<"\nEnter value of k=";
cin>>k;
}
void display()
{
```

```
B1::display();
B2::display();
cout<<"\nk="<<k;
}
};

int main()
{
clrscr();
D o1, o2(10,20,30);
o1.display();
o2.display();
o1.input();
o1.display();
getchar();
return 0;
}
```

```
#include<iostream.h>
#include<alloc.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>

class B1
{
private:
int i;
public:
B1()
{
i=0;
}
B1(int i)
{
this->i=i;
}
```

```
void input()
{
cout<<"\nEnter value of i=";
cin>>i;
}
void display()
{
cout<<"\ni="<<i;
}
};

class B2
{
private:
int j;
public:
B2()
{
j=0;
```

```
}
B2(int j)
{
this->j=j;
}
void input()
{
cout<<"\nEnter value of j=";
cin>>j;
}
void display()
{
cout<<"\nj="<<j;
}
};

class D
{
private:
```

```
int k;
B1 o1;
B2 o2;
public:
D()
{
k=0;
}
D(int i, int j, int k):o1(i), o2(j)
{
this->k=k;
}
void input()
{
o1.input();
o2.input();
cout<<"\nEnter value of k=";
cin>>k;
}
```

```
void display()
{
o1.display();
o2.display();
cout<<"\nk="<<k;
}
};

int main()
{
clrscr();
D o1, o2(10,20,30);
o1.display();
o2.display();
o1.input();
o1.display();
getchar();
return 0;
}
```

Polymorphism in OOP I

Two types of Object Oriented Polymorphism:

- Method Polymorphism
  - Operator Overloading:
    It provides new meanings for existing operators to enable them to work with user defined data types.
  - Inheritance Polymorphism:
    Use of same function name for methods in different classes in inheritance. Base class methods inherited by all derived classes. This inherited method can be either expanded, replaced or used as it is by the derived class.
  - Run Time Polymorphism:
    This is inheritance polymorphism or operator overloading where the type(s) of objects using the methods or operators are not known to the until run time.

Polymorphism in OOP II

- Polymorphism by Parameter
  - In-class Method Overloading:
    This gives the ability to define a number of methods with the same name in a single class but differentiated by the parameter lists. The method name is same but implementation is different in all methods.
  - Genericity/Parametric Polymorphism:
    This refers to the creation of methods or classes which are generic. In this the implementation of the method or class is same for different data types. The different data types are passed as parameters to the method or class.

Polymorphism and Binding

Two type of Polymorphism in C++:

- Compile Time
  - Function Overloading
  - Operator Overloading
- Run Time
  - Virtual Functions

Two type of Binding:

- Early, Compile Time or Static binding
  - Function Overloading
  - Operator Overloading
- Late, Run Time or Dynamic binding
  - Virtual Functions

Pointer of Base class can be used for Derived class for overridden members

```
//virtual function
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>

class B
{       public:
int i;
B(){ i=1; }
void get(){cout<<"\nB"<<endl;}
};
class D:public B
{
public:
int i;
```

```
D(){ i=1; }
void get(){cout<<"\nD"<<endl;}
};

int main()
{       clrscr();
B b,*ptr;
D d;
ptr=&b;
//ptr=new B;
ptr->i=10;
ptr->get();
cout<<"\nB::i=10="<<b.i;
ptr=&d;
//ptr=new D;
ptr->i=100;
ptr->get();
cout<<"\nD::i=100="<<d.i;
getch();
```

```
return 0;
}
```

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

class B
{       public:
int i;
B(){ i=1; }
virtual void get(){cout<<"\nB"<<endl;}
};
class D:public B
{
public:
int i;
D(){ i=1; }
void get(){cout<<"\nD"<<endl;}
};
```

```
int main()
{       clrscr();
B b,*ptr;
D d;
ptr=&b;
//ptr=new B;
ptr->i=10;
ptr->get();
cout<<"\nB::i=10="<<b.i;
ptr=&d;
//ptr=new D;
((D *)ptr)->i=100;
ptr->get();
cout<<"\nD::i=100="<<d.i;
getch();
return 0;
}
```

```
#include<iostream.h>
#include<alloc.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>

class B1
{
private:
int i;
public:
B1()
{
i=1;
}
B1(int i)
{
this->i=i;
}
```

```
void input()
{
cout<<"\nEnter value of i=";
cin>>i;
}
virtual void display()
{
cout<<"\ni="<<i;
}
void fun1(){cout<<"\nB1::fun1()\ni="<<i;}
};

class B2:public B1
{
private:
int j;
public:
B2()
{
```

```
j=2;
}
B2(int i, int j):B1(i)
{
this->j=j;
}
void input()
{
cout<<"\nEnter value of j=";
cin>>j;
}
void display()
{
cout<<"\nj="<<j;
}
};

class D:public B2
{
```

```
private:
int k;
public:
D()
{
k=3;
}
D(int i, int j, int k):B2(i,j)
{
this->k=k;
}
void input()
{
B1::input();
B2::input();
cout<<"\nEnter value of k=";
cin>>k;
}
void display()
```

```
{
B1::display();
B2::display();
cout<<"\nk="<<k;
}
void fun2(){cout<<"\nD::fun2()\nk="<<k;}
};

int main()
{
clrscr();
B1 *b1ptr, o1;
B2 *b2ptr, o2;
D *dptr, o3;
b2ptr=&o3;
b2ptr->display();
dptr=(D*)&o1;
dptr->display();
dptr->fun1();
```

```
dptr->fun2();
getchar();
return 0;
}
```

Virtual Function

- Virtual Function
- Pure Virtual Function
- Virtual Destructor
- Abstract Class
- Pure Abstract Class
- Concrete Class
- Why are Constructor not allowed to be virtual?
- Why are Destructor allowed to be virtual?

Virtual function Trade-offs

- Time consuming
- Greater Flexibility

Rules for Virtual function

- They can't be static.
- They can be friend to another class.
- They can be accessed using object pointers.
- Base pointer can be used for derived object but not vice versa.
- Prototype must be same in base class and derived class.
- Virtual constructors are not allowed but virtual destructor are allowed.
- Virtual operator overloading is allowed.
- They should be declared as public.

When to use Inheritance?

- Specialization
- Common Interfaces
- Extension
- Similar Behaviour

Benefits of Inheritance

- Increase Reliability and Decrease Maintenance cost
- Code Sharing
- Saving in Testing and Development time
- Development of Tools and Module library

Cost of Inheritance

- May be Slower
- Improper or overuse may increase complexity

Generic Programming

- Function Template
  - Syntax of Function Template
  - Invocation of Function Template
  - Overriding of Function Template
  - Overloading of Function Template
  - Recursive call of Function Template
  - Mixing of arguments in Function Template
  - User Defined Template Arguments
- Class Template
  - Syntax of Class Template
  - Invocation of Class Template
  - Mixing of arguments in Class Template
  - Member Functions defined inside the class
  - Member Functions defined outside the class
  - Inheritance of Class Template
  - Class Template Containership
  - Operator Overloading in Class Template

Invalid Declarations

- No argument
- Unused arguments
- Partial arguments used

Template Function Overloading and Overriding

- Overloading
  - Look for exact match
  - Look for Function Template, no trivial type conversion
  - Look for overloading resolution
- Overriding : if exact match found then override the Function Template

```
//temp1.cpp
void swap(int &x, int &y)
{
int t;
t=x;
x=y;
y=t;
}

void swap(char &x, char &y)
{
char t;
t=x;
x=y;
y=t;
}

void swap(float &x, float &y)
{
```

```
float t;
t=x;
x=y;
y=t;
}
class A{...};
void swap(A &x, A &y)
{
A t;
t=x;
x=y;
y=t;
}
```

Syntax of Function Template:

```
template <class T, class U, ...>
Return_Type Function_Name (arguments_list)
//At least one argument must be of T,U,...
{ Function_Body }

template <class T>
void swap(T &x, T &y)
{
T t;
t=x;
x=y;
y=t;
}
```

```
//Syntax of Class Template
template<class T1, class T2, ..., class Tn, int size>
class Class_Name
{
T1 var1;
T2 var2;
.
.
.
Tn varn[size];
T1 fun(T2 o,...){T3 j;...}
...
};
//Invocation of Class Template
Class_Name<int,float,A,20>o1;
```

```
//Mixing of arguments in Class Template
//Member Functions defined inside the class
//Member Functions defined outside the class

template <class T1, class T2,..., class Tn>
Ti Class_Name<T1,T2,...,Tn>fun(T1 i, T2 j,..., Tn n)
{
Ti k;
}//all Ti to be used
```

```
//temp2.cpp
//Member Functions defined inside the class
//Member Functions defined outside the class

#include<iostream.h>
#include<string.h>
#include<stdio.h>
#include<conio.h>

template <class T>
class B
{
T i;
public:
B()
{
i=0;
}
B(T j)
```

```
{
i=j;
}
void input()
{
cin>>i;
}
void display()const;
};
template<class T>
void B<T>::display()const
{
cout<<i;
}

int main()
{
    clrscr();
    B<int>o1;
```

```
    o1.display();
    o1.input();
    o1.display();
    B<float>o2(29.5);
    o2.display();
    getchar();
    return 0;
}
```

```
//Inheritence in Template Class
//temp3.cpp
#include<iostream.h>
#include<alloc.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>

template<class T>
class B1
{
private:
T i;
public:
B1()
{
i=0;
}
B1(T i)
```

```
{
this->i=i;
}
void input()
{
cout<<"\nEnter value of i=";
cin>>i;
}
void display()
{
cout<<"\ni="<<i;
}
};

template<class T>
class B2
{
private:
T j;
```

```
public:
B2()
{
j=0;
}
B2(T j)
{
this->j=j;
}
void input()
{
cout<<"\nEnter value of j=";
cin>>j;
}
void display()
{
cout<<"\nj="<<j;
}
};
```

```
template<class TB1,class TB2,class TD>
class D:public B1<TB1>, public B2<TB2>
{
private:
TD k;
public:
D()
{
k=0;
}
D(TB1 i, TB2 j, TD k):B1<TB1>(i), B2<TB2>(j)
{
this->k=k;
}
void input()
{
B1<TB1>::input();
B2<TB2>::input();
```

```
cout<<"\nEnter value of k=";
cin>>k;
}
void display()
{
B1<TB1>::display();
B2<TB2>::display();
cout<<"\nk="<<k;
}
};

int main()
{
clrscr();
char a='a';
D<int,float,char> o1, o2(10,20,a);
o1.display();
o2.display();
o1.input();
```

```
o1.display();
getchar();
return 0;
}
```

```
//temp4.cpp
//Template Class Container
#include<iostream.h>
#include<alloc.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>
template <class T>
class B1
{
private:
T i;
public:
B1()
{
i=0;
}
B1(T i)
{
```

```
this->i=i;
}
void input()
{
cout<<"\nEnter value of i=";
cin>>i;
}
void display()
{
cout<<"\ni="<<i;
}
};
template<class T>
class B2
{
private:
T j;
public:
B2()
```

```
{
j=0;
}
B2(T j)
{
this->j=j;
}
void input()
{
cout<<"\nEnter value of j=";
cin>>j;
}
void display()
{
cout<<"\nj="<<j;
}
};
template<class TB1, class TB2, class TD>
class D
```

```
{
private:
TD k;
B1<TB1> o1;
B2<TB2> o2;
public:
D()
{
k=0;
}
D(TB1 i, TB2 j, TD k):o1(i), o2(j)
{
this->k=k;
}
void input()
{
o1.input();
o2.input();
cout<<"\nEnter value of k=";
```

```
cin>>k;
}
void display()
{
o1.display();
o2.display();
cout<<"\nk="<<k;
}
};

int main()
{
clrscr();
D<int,int,int> o1, o2(10,20,30);
o1.display();
o2.display();
o1.input();
o1.display();
getchar();
```

```
return 0;
}
```

```
//temp5.cpp
//Operator Overloading in Template
#include<iostream.h>
#include<string.h>
#include<stdio.h>
#include<conio.h>

class B
{
int j;
public:
B(){j=100;}
void operator=(int k)
{
cout<<"In B operator=(int k)\n";
j=k;
}
friend istream & operator>>(istream & read, B o);
friend ostream & operator<<(ostream & write, B o);
```

```
};
istream & operator>>(istream & read, B o)
{
cout<<"Enter member of objects\n";
read>>o.j;
return read;
}
ostream & operator<<(ostream & write, B o)
{
write<<"Value of o.j is "<<o.j<<endl;
return write;
}

template<class T>
class A
{
T i;
public:
A(){i=0;}
```

```
A(T j){i=j;}
void input()
{
cout<<"Entet value of i=";
cin>>i;
}
void display()const
{
cout<<"Value of i is "<<i<<endl;
}
}


int main()
{
clrscr();
B o;
A<int> o1(10);
o1.input();
```

```
o1.display();
A<float> o2(3.5);
o2.input();
o2.display();
A<B> o3(o);
o3.display();
A<B> o4;
o4.display();
getchar();
return 0;
}
```

Stream Handing with Console I

get(): defined in istream
void get(char &)
int get(void)
read one character from input device
including white spaces

put(): defined in ostream
write a single character to output device
getline()
write()
width()
precision()
fill()
setf()
unset()

flags
ios::left ios::adjustfield
ios::right ios::adjustfield

```
ios::internal ios::adjustfield

ios::dec ios::basefield
ios::oct ios::basefield
ios::hex ios::basefield

ios::scientific ios::floatfield
ios::fixed ios::floatfield

ios::showbase/ for output
ios::showpos/ for output
ios::showpoint/ for output
ios::uppercase/ for output
ios::skipws/ for input
ios::unitbuf/Flush after insertion use a buffer of size 1 i
ios::stdio/ Flush stdout and stderr after insertion (<<)

manipulators/ iomapip.h
dec-Sets the conversion base to 10
```

```
hex-Sets the conversion base to 16
oct-Sets the conversion base to 8
ws-Extracts white space characters from an input stream
//Characters inthe stream will be extracted
//until a non-white-space characters is found.
endl-insert a new line in output
ends-Output a NULL character
flush-Flues the output stream cout.flush()
setw(int )
setprecision(int )
setfill(int )
setbase(int 0/8/10/16 )//0 for base 10 for output
//10 for decimal input and output
setiosflags(long )//setiosflags(ios::showpoint|ios::fixed|
resetiosflags(long )
```

File Handling I

- Hierarchy of File Stream Classes
- Step involved:
  - Name of the file on disk
  - Open file to get file pointer
  - Process file for read/write
  - Checking for errors while processing
  - Close file after complete use
- Opening and Closing of Files: Two ways of opening/closeing
  - Using Constructor/destructor function
    ifstream(const char *path, int mode=ios::in, int prot=filebuf::openprot)
    ofstream(const char *path, int mode=ios::out, int prot=filebuf::openprot)
    fstream(const char *path, int mode=ios::in—ios::out, int prot=filebuf::openprot)
  - Using open/close function:
    fobject_name.open("filename")
    fobject_name.close()

File Handling II

- Length of file name and maximum number of files opened at a time is OS dependants.
- Testing for Errors: !fileObject returns non zero if successful or zero if unsuccessful. if(!file) true if file not opened, while (file) true if no EOF
- File opening Modes:

File Handling III

Mode Value	***Effect of the Mode***
ios::in	Open for reading
ios::out	Open for writing
ios::ate	Seek(go) to the end of file at opening time
ios::app	All writes occur at end of file
ios::trunc	Truncate the file if already exists
ios::nocreate	Open existing file without creating new file
ios::noreplace	Open new file but don't replace existing file
ios::binary	Open file as a binary file

Table: File Opening Modes

```
//file1.cpp
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>
int main()
{
clrscr();
int x1=100,x2=200;
char *s="Amity";
ostream &print=cout;
istream &input=cin;
{
print<<s;
print<<x1<<x2;

ofstream fprint("abc.txt");
fprint<<s;
```

```
fprint<<x1<<x2;
//fprint<<s<<'\t';
//fprint<<x1<<'\t'<<x2<<endl;
}
//fprint.close();
print<<"\nwait and check file abc.txt ...";
getchar();

{
s=new char[10];
print<<"Enter s: ";
input>>s;
print<<"Enter x1 and x2: ";
input>>x1>>x2;

ifstream finput("abc.txt");
finput>>s;
finput>>x1>>x2;
```

```
print<<"s: "<<s<<endl<<"x1: "<<x1<<endl<<"x2: "<<x2<<endl;
}
getchar();
return 0;
}
```

File Pointer and their Manipulations I

- get pointer (ifstream):- specifies a location from where the current reading operation is initiated.
- put pointer (ofstream):- specifies a location from where the current writing operation is initiated.
- Default Actions:
  Read-only Mode:- get pointer in the beginning of file for reading.
  Write-only Mode:- put pointer in the beginning of file for writing.
  Append Mode:- Existing contents remains unchanged and put pointer in the end of the file file for writing.

File Pointer and their Manipulations II

- Functions for Manipulation of File Pointers:

Function	***Action***
seekg()	Moves get file pointer to a specific location
seekp()	Moves put file pointer to a specific location
tellg()	Returns the current position of get pointer
tellp()	Returns the current position of put pointer

Table: File Pointer Control Functions

- Prototype of seekg()
  istream & seekg(long offset, $seek\_dir = ios :: beg$)
  ostream & seekp(long offset, $seek\_dir = ios :: beg$)

File Pointer and their Manipulations III

- File Seek Positions

Origin Value	***Seeks from***
ios::beg	Seek from beginning of the file
ios::cur	Seek from current of the file
ios::end	Seek from end of the file

Table: File Seek Positions

File Pointer and their Manipulations IV

- File Seek Calls and their Actions
-seekg() sets the get pointer and seekp() set the put pointer

Seek Call	***Action Performed***
fin.seekg(0, ios::beg)	Go to the beginning of the file
fin.seekg(0, ios::cur)	Stay at the current file position
fin.seekg(0, ios::end)	Go to the end of the file
fin.seekg(n, ios::beg)	Move forward to (n+1) bytes from beginning of file
fin.seekg(n, ios::cur)	Move forward to n bytes from current position
fin.seekg(-n, ios::cur)	Move backward to n bytes from current position
fin.seekg(-n, ios::end)	Move backward to n bytes from end

Table: File Seek Calls and their Actions

```
//file2.cpp
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>


int main(int c, char *str[])
{
clrscr();
ifstream inf;
inf.open("abc.cpp");
for(int i=1;i<c;i++) cout<<str[i]<<endl;
inf>>c;
getchar();
return 0;

}
```

```
//file3.cpp
#include<iostream.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<conio.h>

int main()
{
clrscr();
fstream in("input.txt",ios::in|ios::out|ios::trunc),
out("output.txt",ios::out|ios::in|ios::trunc);
char ch;
in.seekp(0,ios::end);
out.seekp(0,ios::end);
cout<<in.tellg()<<endl<<out.tellp()<<endl;
char str[]="ashok";
// cout<<str<<endl;
cout<<"wait and check input,txt and output.txt\n";
```

```
getchar();
for(int i=0;i<sizeof(str);i++) in.put(str[i]);
cout<<in.tellp()<<endl;
in.seekg(0);
out.seekp(0);
while(in)
{
in.get(ch);
out.put(ch);
//cout<<ch;
}
cout<<"wait and check input,txt and output.txt\n";
out.seekg(0,ios::beg);
out>>str;
cout<<str<<endl;
    getchar();
    return 0;
}
```

Exception Handling I

An unexpected condition in a program during execution.

- Two Types of Error: Compile Time and Run Time
- Compile Time: Two Type Syntactic and Logical Error. Syntactic errors can be detected by compiler during compilation. Logical Errors can not be detected by compiler.
- Run Time: This are depending upon the environment and input at the time of execution of program. They terminate the program abnormally. They are of two types:
  - Synchronous: The exception because of input data or technique that is not suitable to handle the that class of data e.g. divide by zero, array out of bound, file permission etc.
  - Asynchronous: These exception are external to the program and are not in control of the programmer e.g. Keyboard interrupts, hardware failure or malfunctioning, disk failure, memory corruption etc.
  - C++ allow only handling of synchronous exception.

Exception Handling II

- Exception Handling Model: Three blocks to handle the exception (synchronous) in C++
  - try Block
  - throw Block
  - catch Block
- Re-throw: A throw in catch block is a re-throw.
- List of Exceptions or Restricting of throw
  ReturnType FunctionName(Argument List) throw(List of allowed throw)
- Catch All Exception
  catch(...)
- Handling uncaught/restricted Exception
  - terminate()//by default terminate is set to call abort()
  - set_terminate()//set_terminate(abort)
  - unexpected()//by default unexpected is set to call terminate()
  - set_unexpected()//set_unexpected(terminate)
  - all these function are declared in exception.h void myfun()

Exception Handling III

- Exceptions in Operator Overloading
- Exceptions in Inheritance Overloading
- Exceptions in Template

```
//structure of Exception Handling
...
try{// Detector
...
if(condition)
 throw object;//informer//throw either should be in try blo
 ...
 catch(object)//Receiver
 {
 ...//handler
 }
 ...
```

```
//EXCEPT1.cpp
int main(int c, char *str[])
{
int a,b;
cout<<"Enter Values of a and b \n";
cin>>a>>b;
try
{
if(b)
{
cout<<"Result(a/b) = "<<a/b;
}
else
{
throw (b);
}
}
catch (int i)
{
```

```
cout<<"Exception caught: b = "<<b<<endl;
}
catch (int i)
{
}
getchar();
return 0;
}
```

```
//Exceptions in Operator Overloading
//EXCEPT2.cpp

const int MaxSize=20;
class Vector
{
int *arr;
int size;
public:
class SIZE{};
class RANGE{};
Vector(int s)
{
if(s<=0||s>MaxSize)
throw SIZE();
arr=new int[size=s];
}
~Vector(){delete arr;}
int & operator[](int i)
```

```
{
if (i<0||i>size) throw RANGE();
return arr[i];
}
int GetSize(){return size;}
};
int main()
{
    int size,index;
    cout<<"Enter Size of Vector <MaxSize=20>:";
    cin>>size;
    try
{
Vector V(size);
cout<<"Enter Index of Vector <0 to "<<size-1<<">:";
    cin>>index;
V[index]=20;
cout<<"V["<<index<<"]="<<V[index]<<endl;
}
```

```
catch(Vector::SIZE)
{
        cout<<"Size must be between 1 to "<<MaxSize<<endl;
}
catch(Vector::RANGE)
{
        cout<<"Index must be between 0 to "<<size-1<<endl;
}
return 0;
}
```

```
//Exceptions in Inheritance Overloading
////EXCEPT3.cpp
class Wrong_f_age{};
class Wrong_s_age{};
class Father
{
protected:
int f_age;
public:
Father(int i)
{
if(i<21) throw Wrong_f_age();
f_age=i;
}
virtual void display()
{
cout<<"Father age is "<<f_age<<endl;
}
};
```

```
class Son: public Father
{
int s_age;
public:
Son(int f, int s):Father(f)
{
if(f<(s+21)) throw Wrong_s_age();
s_age=s;
}
virtual void display()
{
cout<<"Father age is "<<f_age<<" son age is "<<s_age<<endl;
}
};
int main()
{
Father *bp;
int f_age, s_age;
try
```

```
{
    cout<<"Enter Father age <greater than or equal to 21>:"
    cin>>f_age;
cout<<"Enter Son age :";
    cin>>s_age;
bp=new Son(f_age,s_age);
bp->display();
}
catch(Wrong_f_age)
{
        cout<<"Father age must be greater than or equal to
}
catch(Wrong_s_age)
{
        cout<<"Father age must be greater than son age by a
}
}
```

```
//Exceptions in Template
//EXCEPT4.cpp
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<exception.h>

template<class T>
T div(T a, T b) throw(int, float)
{
try
{
if(b==0)
throw b;
}
catch (int i)
{
```

```
throw;
}
catch (float f)
{
throw;
}
catch(...)
{
throw;
}
}

int main()
{
try
{
div(2,3);
}
catch (int i)
```

```
{
cout<<"Exception caught:= "<<i<<endl;
}
catch (float f)
{
cout<<"Exception caught:= "<<f<<endl;
}
catch(...)
{
cout<<"Exception caught:= "<<endl;
}

getchar();
return 0;
}
```

Test II OOPs, CSE 4^{th} *Sem*, Dated: 09^{th} *Apr*, 2019

Even numbered students will attempt even numbered questions and odd numbered students will attempt odd numbered questions.

Q1. Write a program to show the actual uses of assignment operator overloading for the same class objects.

Q2. Why do we need delete operator overloading even it is already overloaded for user defined data types? Explain with program.

Q3. What is the restricting of throws? What happened when we throw a restricted throw?

Q4. Write a program to show the application of virtual destructor. Why constructor are not allowed to be virtual?

```
//Understanding of copy constructor and
//assignment operator overloading
//COPYASSI.cpp

class cl2{};
class cl
{
    public:
        cl(){cout<<"cl()"<<endl;}
        cl(cl & o){cout<<"cl(cl & o)"<<endl;}
        cl(cl2 o){cout<<"cl(cl2 & o)"<<endl;}
        //cl(cl o1, cl o2)
        {cout<<"cl(cl o1, cl o2)"<<endl;}
        //cl operator=(cl o)
        {cout<<"cl operator=(cl o)"<<endl; return *this;}
};

int main()
{
```

```
    //clrscr();
    //case I
    //cl o1,o2(o1),o3(o1,o2);
    //o2=o1;

    //case II
    //No copy constructor but assignment
//cl o1,o2=o1;
//o2=o1;

//case III
//With copy constructor and with/without assignment
//cl o1,o2=o1;
//o2=o1;

    //case IV
    //With copy constructor
    //No assignment overloading
cl o1,o2=o1;
```

```
cl2 o3;
o2=o3;

getchar();
return 0;
}
```

Standard Template Library (STL) I

- Developed by Alexander Stepanov and Meng Lee of Hewlett Packard
- STL is part of the Standard C++ class library
- Made of template classes
- Can be applied nearly any type of data including user defined data types.
- Three most important components: containers, algorithms, and iterators.
  - Container: A way of storing data in memory e.g. stacks, linked lists, vector etc. Implemented by using template classes .
  - Algorithms: Functions that are applied to containers to process their data e.g sort, copy, search, and merge and implemented using by template functions.
  - Iterators: Just like a generalized pointers used to point to elements in a container. We can increment an iterator as a pointer.

Standard Template Library (STL) II

Term	***Iterator***	***Access Allowed***
RandIter	Random Access	Store and retrieve values. Elements may be accessed randomly
BiIter	Bidirectional	Store and retrieve values. Forward and backward moving.
ForIter	Forward	Store and retrieve values. Forward moving only.
InIter	Input	Retrieve, but not store values. Forward moving only.
OutIter	Output	Store, but not retrieve values. Forward moving only

Table: Five types of iterators

Container Classes I

Table: Containers Classes Defined by the STL

Container	***Description***	***Header File***
bitset	A set of bits.	< *bitset* >
deque	A double-ended queue.	< *deque* >
list	A linear list.	< *list* >
map	Stores key/value pairs in which each key is associated with only one value.	< *map* >
multimap	Stores key/value pairs in which one key may be associated with two or more values.	< *map* >
multiset	A set in which each element is not necessarily unique.	< *set* >

Container Classes II

priority _queue	A priority queue.	< *queue* >
queue	A queue.	< *queue* >
set	A set in which each element is unique.	< *set* >
stack	A stack.	< *stack* >
vector	A dynamic array.	< *vector* >

Container Classes III

Most common typedef names

typedef	***Type***
size_type	Some type of integer
reference	A reference to an element
const_reference	A const reference to an element
iterator	An iterator
const_iterator	A const iterator
reverse_iterator	A reverse iterator
const_reverse _iterator	A const reverse iterator
value_type	The type of a value stored in a container
allocator_type	The type of the allocator
key_type	The type of a key

Container Classes IV

key_compare	The type of a function that compares two keys
value_compare	The type of a function that compares two values

Vector I

Some Commonly Used Member Functions Defined by vector

Member Function	***Description***
reference back( ), const_reference back( ) const;	Returns a reference to the last element in the vector.
iterator begin( ), const_iterator begin( ) const	Returns an iterator to the first element in the vector.
void clear( )	Removes all elements from the vector.
bool empty( ) const	Returns true if the invoking vector is empty and false otherwise.
iterator end( ), const_iterator end( ) const	Returns an iterator to the end of the vector.

Vector II

iterator erase(iterator i)	Removes the element pointed to by i. Returns an iterator to the element after the one removed.
iterator erase(iterator start, iterator end)	Removes the elements in the range start to end. Returns an iterator to the element after the last element removed.
reference front( ), const_reference front( ) const	Returns a reference to the first element in the vector.
iterator insert(iterator i, const T &val)	Inserts val immediately before the element specified by i. An iterator to the element is returned.
void insert(iterator i, size_type num, const T & val)	Inserts num copies of val immediately before the element specified by i.

Vector III

template <class InIter> void insert(iterator i, InIter start, InIter end)	Inserts the sequence defined by start and end immediately before the element specified by i.
reference operator[ ](size_type i) const, const_reference operator[ ](size_type i) const	Returns a reference to the element specified by i.
void pop_back( )	Removes the last element in the vector.
void push_back(const T &val)	Adds an element with the value specified by val to the end of the vector.
size_type size( ) const	Returns the number of elements currently in the vector.

```
// Demonstrate a vector.
//STL1.CPP
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
    vector<char> v(10); // create a vector of length 10
    unsigned int i;
    // display original size of v
    cout << "Size = " << v.size() << endl;
    // assign the elements of the vector some values
    for(i=0; i<10; i++) v[i] = i + 'a';
    // display contents of vector
    cout << "Current Contents:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << "\n\n";
```

```
    cout << "Expanding vector\n";
    /* put more values onto the end of the vector,
    it will grow as needed */
    for(i=0; i<10; i++) v.push_back(i + 10 + 'a');
    // display current size of v
    cout << "Size now = " << v.size() << endl;
    // display contents of vector
    cout << "Current contents:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << "\n\n";
    // change contents of vector
    for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
    cout << "Modified Contents:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

```
// Access the elements of a vector through an iterator.
//STL2.CPP
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

int main()
{
vector<char> v(10); // create a vector of length 10
vector<char>::iterator p; // create an iterator
int i;
// assign elements in vector a value
p = v.begin();
i = 0;
while(p != v.end())
{
*p = i + 'a';
p++;
```

```
i++;
}
// display contents of vector
cout << "Original contents:\n";
p = v.begin();
while(p != v.end())
{
cout << *p << " ";
p++;
}
cout << "\n\n";
// change contents of vector
p = v.begin();
while(p != v.end())
{
*p = toupper(*p);
p++;
}
// display contents of vector
```

```
cout << "Modified Contents:\n";
p = v.begin();
while(p != v.end())
{
cout << *p << " ";
p++;
}
cout << endl;
return 0;
}
```

```
// Demonstrate insert and erase.
//STL3.CPP
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<char> v(10);
    vector<char> v2;
    char str[] = "<Vector>";
    unsigned int i;
    // initialize v
    for(i=0; i<10; i++) v[i] = i + 'a';
    // copy characters in str into v2
    for(i=0; str[i]; i++) v2.push_back(str[i]);
    // display original contents of vector
    cout << "Original contents of v:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << "\n\n";
```

```
vector<char>::iterator p = v.begin();
p += 2; // point to 3rd element
// insert 10 X's into v
v.insert(p, 10, 'X');
// display contents after insertion
cout << "Size after inserting X's = " << v.size() << en
cout << "Contents after insert:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";
// remove those elements
p = v.begin();
p += 2; // point to 3rd element
v.erase(p, p+10); // remove next 10 elements
// display contents after deletion
cout << "Size after erase = " << v.size() << endl;
cout << "Contents after erase:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";
// Insert v2 into v
```

```
    v.insert(p, v2.begin(), v2.end());
    cout << "Size after v2's insertion = ";
    cout << v.size() << endl;
    cout << "Contents after insert:\n";
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;
    return 0;
}
```

```
// Store a class object in a vector.
//STL4.CPP
#include <iostream>
#include <vector>
#include <cstdlib>
using namespace std;

class DailyTemp
{
int temp;
public:
DailyTemp() { temp = 0; }
DailyTemp(int x) { temp = x; }
DailyTemp &operator=(int x)
{
        temp = x; return *this;
    }
    double get_temp() { return temp; }
};
```

```
bool operator<(DailyTemp a, DailyTemp b)
{
    return a.get_temp() < b.get_temp();
}
bool operator==(DailyTemp a, DailyTemp b)
{
    return a.get_temp() == b.get_temp();
}

int main()
{
    vector<DailyTemp> v;
    unsigned int i;
    for(i=0; i<7; i++)
    v.push_back(DailyTemp(60 + rand()%30));
    cout << "Fahrenheit temperatures:\n";
    for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";
    cout << endl;
```

```
    // convert from Fahrenheit to Centigrade
    for(i=0; i<v.size(); i++)
    v[i] = (int)(v[i].get_temp()-32) * 5/9 ;
    cout << "Centigrade temperatures:\n";
    for(i=0; i<v.size(); i++)
    cout << v[i].get_temp() << " ";
    return 0;
}
```

```
// List basics.
//STL5.CPP
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> lst; // create an empty list
    int i;
    for(i=0; i<10; i++) lst.push_back(i);
    cout << "Size = " << lst.size() << endl;
    cout << "Contents: ";
    list<int>::iterator p = lst.begin();
    while(p != lst.end())
    {
        cout << *p << " ";
        p++;
    }
```

```
    cout << "\n\n";
    // change contents of list
    p = lst.begin();
    while(p != lst.end())
    {
        *p = *p + 100;
        p++;
    }
    cout << "Contents modified: ";
    p = lst.begin();
    while(p != lst.end())
    {
        cout << *p << " ";
        p++;
    }
    return 0;
}
```

```
// Understanding end().
//STL6.CPP
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> lst; // create an empty list
    int i;
    for(i=0; i<10; i++) lst.push_back(i);
    cout << "List printed forwards:\n";
    list<int>::iterator p = lst.begin();
    while(p != lst.end())
    {
        cout << *p << " ";
        p++;
    }
    cout << "\n\n";
```

```
    cout << "List printed backwards:\n";
    p = lst.end();
    while(p != lst.begin())
    {
        p--; // decrement pointer before using
        cout << *p << " ";
    }
    return 0;
}
```

```
/* Demonstrating the difference between
push_back() and push_front(). */
//STL.CPP
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> lst1, lst2;
    int i;
    for(i=0; i<10; i++) lst1.push_back(i);
    for(i=0; i<10; i++) lst2.push_front(i);
    list<int>::iterator p;
    cout << "Contents of lst1:\n";
    p = lst1.begin();
    while(p != lst1.end())
    {
        cout << *p << " ";
```

```
        p++;
    }
    cout << "\n\n";
    cout << "Contents of lst2:\n";
    p = lst2.begin();
    while(p != lst2.end())
    {
        cout << *p << " ";
        p++;
    }
    return 0;
}
```

```
// Sort a list.
//STL8.CPP
#include <iostream>
#include <list>
#include <cstdlib>
using namespace std;

int main()
{
    list<int> lst;
    int i;
    // create a list of random integers
    for(i=0; i<10; i++)
    lst.push_back(rand());
    cout << "Original contents:\n";
    list<int>::iterator p = lst.begin();
    while(p != lst.end())
    {
        cout << *p << " ";
```

```
        p++;
    }
    cout << endl << endl;
    // sort the list
    lst.sort();
    cout << "Sorted contents:\n";
    p = lst.begin();
    while(p != lst.end())
    {
        cout << *p << " ";
        p++;
    }
    return 0;
}
```

```
// Merge two lists.
//STL9.CPP
#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<int> lst1, lst2;
    int i;
    for(i=0; i<10; i+=2) lst1.push_back(i);
    for(i=1; i<11; i+=2) lst2.push_back(i);
    cout << "Contents of lst1:\n";
    list<int>::iterator p = lst1.begin();
    while(p != lst1.end())
    {
        cout << *p << " ";
        p++;
    }
```

```
cout << endl << endl;
cout << "Contents of lst2:\n";
p = lst2.begin();
while(p != lst2.end())
{
    cout << *p << " ";
    p++;
}
cout << endl << endl;
// now, merge the two lists
lst1.merge(lst2);
if(lst2.empty())
cout << "lst2 is now empty\n";
cout << "Contents of lst1 after merge:\n";
p = lst1.begin();
while(p != lst1.end())
{
    cout << *p << " ";
    p++;
```

```
    }
    return 0;
}
```

```
// Store class objects in a list.
//STL10.CPP
#include <iostream>
#include <list>
#include <cstring>
using namespace std;

class myclass
{
    int a, b;
    int sum;
    public:
    myclass() { a = b = 0; }
    myclass(int i, int j)
    {
        a = i;
        b = j;
        sum = a + b;
    }
```

```
    int getsum() { return sum; }
    friend bool operator<(const myclass &o1,const myclass &
    friend bool operator>(const myclass &o1,const myclass &
    friend bool operator==(const myclass &o1,const myclass
    friend bool operator!=(const myclass &o1,const myclass
};
bool operator<(const myclass &o1, const myclass &o2)
{
    return o1.sum < o2.sum;
}
bool operator>(const myclass &o1, const myclass &o2)
{
    return o1.sum > o2.sum;
}
bool operator==(const myclass &o1, const myclass &o2)
{
    return o1.sum == o2.sum;
}
bool operator!=(const myclass &o1, const myclass &o2)
```

```
{
    return o1.sum != o2.sum;
}

int main()
{
    int i;
    // create first list
    list<myclass> lst1;
    for(i=0; i<10; i++) lst1.push_back(myclass(i, i));
    cout<< "First list: ";
    list<myclass>::iterator p = lst1.begin();
    while(p != lst1.end())
    {
        cout << p->getsum() << " ";
        p++;
    }
    cout << endl;
    // create a second list
```

```
list<myclass> lst2;
for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));
cout << "Second list: ";
p = lst2.begin();
while(p != lst2.end())
{
    cout << p->getsum() << " ";
    p++;
}
cout << endl;
// now, merget lst1 and lst2
lst1.merge(lst2);
// display merged list
cout << "Merged list: ";
p = lst1.begin();
while(p != lst1.end())
{
    cout << p->getsum() << " ";
    p++;
```

```
    }
    return 0;
}
```

```
// A simple map demonstration.
//STL11.CPP
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<char, int> m;
    int i;
    // put pairs into map
    for(i=0; i<26; i++)
    {
        m.insert(pair<char, int>('A'+i, 65+i));
    }
    char ch;
    cout << "Enter key: ";
    cin >> ch;
    map<char, int>::iterator p;
```

```
    // find value given key
    p = m.find(ch);
    if(p != m.end())
        cout << "Its ASCII value is " << p->second;
    else
        cout << "Key not in map.\n";
    return 0;
}
```

```
// Use a map to create a phone directory.
//STL12.CPP
#include <iostream>
#include <map>
#include <cstring>
using namespace std;

class name
{
    char str[40];
    public:
    name() { strcpy(str, ""); }
    name(char *s) { strcpy(str, s); }
    char *get() { return str; }
};
    // Must define less than relative to name objects.
bool operator<(name a, name b)
{
    return strcmp(a.get(), b.get()) < 0;
```

```
}
class phoneNum
{
    char str[80];
    public:
    phoneNum() { strcmp(str, ""); }
    phoneNum(char *s) { strcpy(str, s); }
    char *get() { return str; }
};

int main()
{
    map<name, phoneNum> directory;
    // put names and numbers into map
    directory.insert(pair<name, phoneNum>(name("ABC"),
    phoneNum("555-4533")));
    directory.insert(pair<name, phoneNum>(name("XYZ"),
    phoneNum("555-9678")));
    directory.insert(pair<name, phoneNum>(name("AAA"),
```

```
    phoneNum("555-8195")));
    directory.insert(pair<name, phoneNum>(name("BBB"),
    phoneNum("555-0809")));
    // given a name, find number
    char str[80];
    cout << "Enter name: ";
    cin >> str;
    map<name, phoneNum>::iterator p;
    p = directory.find(name(str));
    if(p != directory.end())
        cout << "Phone number: " << p->second.get();
    else
        cout << "Name not in directory.\n";
    return 0;
}
```


ज्योतिर्वृणीत तमसो विजानन

Thank you

- Please send your feedback or any queries to akyadav1@amity.edu or call me on +91 9911375598