C++ Compilation Module:
Preprocessor – Input: header and implementation file, output: Intermediate source file
Compiler – Input: Intermediate source file, output : current.obj,
Linker – Input : current.obj, *.lib, *.obj output: current.exe,
Loader – Input: *.dll and current.exe, output: running process
C++ Computational Module: I/O Model : cin,command line,ifstream, output:cout,ofstream
Program model, memory model: static memory:public global functions,private global functions and local static data, heap memory : allocated heap and free heap, stack memory
The C++ compilation model encourages us to define separate packages for each program activity which are then built into an executable whole. We manage the compilation sequence with preprocessor directives and can build the project in parts for incremental testing.
->In Project #2: we define several packages each of which focuses on a single activity like xml document, parsing, xml element, and xml writer. Each package that depends on the services of another package must include the header file of the used package, e.g., #include "service.h". For any standard libraries that are used by the package there must be a directive: #include <libfile>. For example XML Document we include xml element, xml parser
If you have elected to build some of the program's services as Dynamic Link Libraries, your project must contain the Lib file that describes the DLL exports.

C++ Memory Model : Static Memory: Objects are created in static memory by preceding their declarations with the keyword static". Static objects have the life-time of the program. A static object's constructor when static memory is initialized at the beginning of the program. Their destructors are called when the thread of execution leaves main. **Stack Memory**: Objects are created in stack memory by declaring them as function parameters or declaring them locally in a function. Stack-based objects have a lifetime defined by the thread of execution residing within the scope in which the object is declared. Their constructors are called at the point of declaration and their destructors are called when the thread of execution leaves the scope in which they are declared. **Heap Memory**: Objects are created on the heap by declaring them with the "new" keyword and destroying them with "delete". Heap-based objects live from execution of the statement using "new" to the statement using "delete". The new statement causes the allocated object's constructor to be called and delete invokes the object's destructor.
In Project 2, we are using count and tabsize as static members of Abstract XML elements which are used while displaying and writing values to file which have global access with life time of application. Stack memory in several places in different functions like whenever we want to process the input string and construct abstract syntax tree. Heap Memory is used in makeElement while creating derived objects of Abstract syntax tree. In Project 1, we are using static functions from Path, Directory and File classes to get the file system information. Stack memory is used in several places for processing of command line,Data Store object is build completely in stack memory passing its reference to different functions

Software Development Design Goals/Principles : Simple(Cyclomatic Complexity, small functions), Understandable(Prologue, Public Interface), Maintainable(Parser is example of open closed principle), Selectable(Packages,easy to build, easy to run,), Reliable(no surprises,thoroughly tested), Robust(will not crash), Flexible, Extendable, re-usable(prologue, understandable and no application specific code)
Design Attributes: abstraction,Modularity,Encapsulation,Hierarchy,Cohesion Coupling,Locality of reference,Size and complexity,Use of objects Performance
OOD Design Objectives : Develop High Quality Software, Reuse software Components, manage change, deal with complexity, support sw design in application domain
int x = 2;int *pInt = &x; *pInt = 3; || int &rInt = x; rInt = 2;
-Encapsulation : Classes should provide a simple public interface that does not allow binding to its implementation. That is, client code should only be able to access the class's public interface member functions and should not be able to directly access its member data and private member functions. We say that such classes are encapsulated.
- All Public functions should have few arguments, should not return pointers, partition functions to several parts, data should never be made public – write code that is robust against change. Encapsulation Requires that functions, modules and classes ,Have clearly defined external interfaces, Hide implementation details

Functions Interfaces :
Parameter : 1. Value parameters and const references are input only
Non-const reference parameters and pointers are both input and output
2. Returned items – Return by value has no after-affects inside the function.
Return by reference can change the state of an object to which the function belongs.
3. Global data – remote coupling: 4. Static Data
Software Quality Factors : Correctness, Robustness, Re-usability, compatibility, performance
Object Model : Abstraction, Modularity, Encapsulation, Hierarchy
Data Abstractions: -> A good abstraction is built around a **model based view** of an object which describes the behavior expected of the object by clients. Models are often based on a **metaphor** which helps a client understand and relate to an object's behavior, e.g. the window, matrix, dictionary, ...
-> Designers can create new types using classes, objects can be declared and destroyed, operators can be overloaded, overloading, c++ provides new() and delete() . class declarations may be used to create many objects, determined either at compile time or run time. many instances, that is objects, can be declared. an object of a class can be provided with virtually all of the capabilities of the built in types, e.g., int, char, float, etc.
-> Abstraction emphasizes the client's view of the System while suppressing the implementation view
ADT or User Defined Data Type : declaration of multiple objects at either compile or run time
declaration and initialization of arrays of objects -> objects can take care of themselves, e.g., acquire and release system resources. -> objects can participate in mixed type expressions, implicitly calling promotion constructors or cast operators as needed -> objects can be assigned and passed by value to functions -> objects can use the same operator symbolism as built in types
Hierarchy : Hierarchy: Form **aggregations** by class compositions, e.g., one class uses objects of another as data elements. Supports "part-of" semantic relationship between contained and containing objects.
Define subclasses of objects through **inheritance** from base class. Supports an "is-a" semantic relationship between derived classes and base class.
New instances can be created at run time

```
str(int n = 10);              // void and size ctor
2. str(const str& s);          // copy ctor
3. str(str&& s);              // move ctor
explicit str(const char* s);   // promotion ctor
~str();                       // dtor
6.str& operator=(const str& s); // copy ssignment
7. str& operator=(str&& s); // move assignment
char& operator[](int n);      // index operator
char operator[](int n) const;  // index operator
str& operator+=(char ch);      // append char
str& operator+=(const str& s);  // append str s
str operator+(const str& s);    // concatenate strs
operator const char* ();        // cast operator
int size() const;              // return number of chars
void flush();                  // clear string contents
str& str::operator=(str&& s)

->istream& operator>>(istream& in, str& s) {
  char ch; s.flush(); in >> ch; while((ch != '\n') &&
in.good()) { s += ch; in.get(ch); } return in;}
ifstream in("test.dat");
str extract;  while(in.good()) { in >> extract;
→ostream& operator<<(ostream& out, const str& s) { int i; for(i=0; i<s.size(); i++)
  out << s[i]; return out;}
→template<class T> class basic
template <class T>
basic<T>::basic(const char *inMsg, const T t) :
param(t)
```

```
2. str::str(const str& s) :
array(new char[s.max]),
max(s.max), len(s.len){
  for (int i = 0; i <= len; i++)
    array[i] = s.array[i];};
3.str::str(str&& s) :
array(s.array), max(s.max),
len(s.len){ s.array = nullptr; };
4. str::str(const char* s) :
len(static_cast<int>(strlen(s)))
{ max = len+1;
  array = new char[len+1];
  for(int i=0; i<=len; i++)
    array[i] = s[i];}
7. str& str::operator=(str&&s)
{ if (this == &s) return *this;
  max = s.max; len = s.len;
  delete [] array; array =
s.array; s.array = nullptr;
  return *this;}
6. str& str::operator=(const
str& s) { if (this == &s) return
*this; if (max >= s.len + 1) {
  len = s.len;   int i;
  for (i = 0; i <= len; i++)
    array[i] = s.array[i];
  return *this}
str extract;
  while(in.good()) {
    in >> extract; }
```

OOD Design Strategies:
Encapsulate the implementation of an abstraction with a class.
Layer implementation using composition. Extend an abstraction through inheritance.
Loosely couple interacting objects using polymorphism. Encapsulation and composition are essentially bottoms up design tools. Using inheritance and polymorphism are not. They are something new and powerful, unique to OOD.
KISS PRINCIPLE: Keep It Small and Simple Don't solve problems that don't yet exist. Solve the specific problem, not the general case but don't make it needlessly inflexible either Keep the door open for extension through composition and inheritance Use polymorphism to encapsulate "need to know" in specific derived classes, allowing clients to be blissfully ignorant, knowing only the base class protocol.Design function code so that it: fits on a single page has cyclomatic complexity well below 10 Keep a package small enough that its structure chart fits on a single page

Static cast: can cast pointer to derived, pointer to base. No checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type class Base {}; class Derived: public Base {}; Base * a = new Base; Derived * b = static_cast<Derived*>(a);
double b = static_cast<double> (10) /20; converts a foreign type to a type defined by a class with a promotion constructor converts an instance of a type that provides a cast operator to a foreign type
Dynamic cast: dynamic_cast can only be used with pointers and references to classes (or with void*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.

```
class Base { virtual void dummy() {} }; class Derived: public Base { int a; };
  Base * pba = new Derived;  Base * pbb = new Base; Derived * pd;
  pd = dynamic_cast<Derived*>(pba);  // Passes     if (pd==0)
cout << "Null pointer on first type-cast.\n";     pd = dynamic_cast<Derived*>(pbb); // Fails    if
(pd==0) cout << "Null pointer on second type-cast.\n";
```

Composition Example:

```
class composed {  public:
   composed(void);
   composed(const composed& b);
   composed(int x);
   virtual ~composed(void);
   composed& operator= (const
composed& b);   private : int data };

class composer { public:
   composer(void);
   composer(const composer &a);
   composer(int x1, int x2);
   virtual ~composer(void);
   composer& operator=(const
composer& a);  private:
 composed in1; composed in2;};

1-> composer::composer(const
composer &a) : in1(a.in1), in2(a.in2)
[composer outer1 = outer3]
2-> composer::composer(int size1, int
size2) : in1(size1), in2(size2)
[composer outer2(6,4)]
3-> composer::composer(int oneSize) :
in1(oneSize), in2(oneSize)
[composer outer(2)]
```

Without Initialization list:
Void constructor is called and then copied
composer::composer(const composer &a) { in1 = a.in1; in2 = a.in2;}

with Init List: copied directly

```
composed::composed(const
composed& b) : data(b.data)
composed::composed(int x) : data(x)
composer::composer(const composer
&a) : in1(a.in1), in2(a.in2)
composer::composer(const int size1, int
size2) : in1(size1), in2(size2)
composer& cmposer::operator=
(const composer &a) {
  if(this==&a) return *this;
  in1 = a.in1;  in2 = a.in2;  return *this;}
```

Inheritance:

```
class base {
  public:  base(void);    base(const base
&);   virtual ~base(void);
   base& operator= (const base& b);
  private:    int data; };

class derived : public base { public:
derived(void); derived(const derived
&);  derived(const base &);
   ~derived(void);
derived& operator= (const derived&
b);  private:int moredata;};

derived::derived(const derived &d) {
  moredata = d.moredata;}
```

Base object not assigned
Base object not copied

```
derived& derived::operator= (const
derived& b) {
  if(this == &b) return *this;
  moredata = b.moredata; return *this;}
```

Base Assigned:

```
derived& derived::operator= (const
derived& d) {
  if(this == &d) return *this;
  ((base&) *this) = d; moredata =
d.moredata; return *this;
} derived::derived(const derived &d) :
base(d), moredata(d.moredata)
```

Pointer to char array : char str[] = {'a','e','i','o','u'};
char *str = s; str[0];
p[5] = 's';
Pointer to String array std::string *ar = new std::string[20];

```
string fruit[] = {"Apple", "Orange",
"Banana"};
int size = sizeof(fruit)/sizeof(string);

std::array<int,5> myarray = { 2, 16, 77,
34, 50 };
 for ( auto it = myarray.begin(); it !=
myarray.end(); ++it )
   std::cout << ' ' << *it;
```

Insert to Set

```
std::set<int> myset;myset.insert(10);
std::set<int>::iterator it;
 for (it = myset.begin(); it !=
myset.end(); it++) { cout << *it; }
it = find(myset.begin(),myset.end(),3);
 if(it != myset.end())   myset.erase(it);
```

Insert to map:

```
Map.insert(std::pair<int, int>(0, 42))
function[0] = 42;
map<int, int> address_book;
for ( auto address: address_book )
{  address.first   address.second;
}
```

Pass Address of Vector to a function
void do_something(int el, std::vector<int> &arr)
Read and Write to File

```
#include <iostream>
#include <fstream>
int main() {
ofstream myfile("example.txt");
if (myfile.is_open()){
myfile << "This is sandesh.\n";
myfile.flush();myfile.close();
}
else cout << "Unable to open file";
string line;
ifstream myfile1("example.txt");
if (myfile1.is_open()){
/*while (getline(myfile1, line)){
cout << line << '\n';
}*/
char ch;   myfile1 >> ch;  string str;
while ((ch != '\n') &&
myfile1.good())//reading character by
character { str += ch;
myfile1.get(ch); }
std::cout << str;
myfile1.close(); }
else cout << "Unable to open file for
reading"; return 0; }
```

Ways to Call Constructor(Equivalent Syntax):

X x1; X x3[2]; X(); X* ptr = new X; ptr = new X();
ptr = new X[2];
Copy Constructor : X x4 = x1; X x5(x1); X x6[2] = {x1,x4}; ptr = new X(x1);
Copy Assignment: x5=x1; x5.operator=(x1);

Template:

Templates are patterns used to generate code, instances of which, differ only in the symbolic use of a type name. Templates allow us to write one function for an unspecified type, and let the compiler fill in the details for specific type instances.
Instantiation happens at application compile time, when the compiler sees the concrete type associated with the template parameter. No code can be generated before that time, as the size of T instances is not known. The consequence of this is that you must put all template function and class method bodies in the header file, included with application code, so the compiler sees the definition at the time the parameter is associated with a concrete type.

```
template<class T> T max(const T& t1, const T& t2)
{    return ((t2 > t1) ? t2 : t1); }
int x = max(2,1);
typedef char* pChar;
template<class T> T max(const T& t1, const T& t2)
{ return ((t2 > t1) ? t2 : t1);}
template<> const pChar max(const pChar& s1,
const pChar& s2){
   return ((strcmp(s1,s2)>0) ? S1 : s2);}
template <typename T> class stack {
public:   stack();  void push(const T& t);  T pop(void);};
stack<string> myStack;
```

Template Specialization:

```
template <class T> widget   - No specialization
template <class T*> widget – Partial
template<class U> Widget<U,double> - Partial
template <> widget<int> - Full
```

Template Container Types: Homogeneuos, Heter
array<char> myHomogeneousArray;
array<myBaseClass*> myHeteroge neousArray;
Template Semantics:
Value semantics, Reference semantics
Template Functional Pointer

```
template <void(*f)()>
class templ1 {  string str;  public:    void show() { f(); }};
```

user defined template:

```
template <class T>
class templ {  T t;  public:void show() { t.show(); }};
class implem1{public:
 void show() { cout << "\n  this is implementation #1";
}};
templ<implem1> tl;   tl.show();
```

Template Traits:

```
template <typename T, int Size> class Array {
public: typedef T value_type;
 typedef T* iterator;
 Array() : pArray(new T[Size]) {}
 Array(T* pT) : pArray(new T[Size])
 {    for(int i=0; i<Size; ++i)
    *(pArray + i) = *(pT + i); }
 T& operator[](int n)
 {if(n<0 || Size<=n) throw std::exception("index out of
range");   return *(pArray+n); }
 T operator[](int n) const
 {if(n<0 || Size<=n) throw std::exception("index out of
range"); return *(pArray+n); }
 iterator begin() { return pArray; }
 iterator end() { return (pArray + Size); } private:
 Array(const Array<T,Size>&);  // can implement later
 Array<T,Size>& operator=(const Array<T,Size>&) //
ditto
 T* pArray;};
template <typename Cont>
typename Cont::value_type sum(Cont& c){
Cont::iterator iter;
 Cont::value_type sum_ = Cont::value_type();
 for(iter = c.begin(); iter != c.end(); ++iter)sum_ +=
*iter;
 return sum_;}
int main(){double temp1[] = { 0, 0.5, 1.0, 1.5, 2.0 };
Array<double, 5> arr1(temp1);  display(arr1);
 std::cout << "\n  sum = " << sum(arr1) << "\n";
 int temp2[] = { 1, 2, 3, 4, 5 };
 Array<int,5> arr2(temp2);
 display(arr2);
 std::cout << "\n  sum = " << sum(arr2) << "\n\n";}
```

Templates and function pointers

```
//----< declare template class taking function pointer
parameter >
template<void(*f)()>
class templ1 {string str;
public:void show() { f(); }  };
//----< declare template class taking pointer to C
string >----
template<char **s>
class templ2 {public: void show() { cout << "\n  " <<
(*s); }};
tmpl1<fun1>tl; tl.show();
```

Policy Example:

```
struct rowDisplayPolicy
{       static std:: string seperator() {
           return ", ";  };
struct columnDisplayPolicy{
     static std:: string seperator() {return "\n  ";}};
template <typename T , int Size, typename
DisplayPolicy >class Array
{public: Array() : pArray( new T[Size]) {}
 Array( T* pT) : pArray( new T[Size]){
 for ( int i = 0; i<Size; ++i)*(pArray + i) = *( pT + i);}
~Array() { delete[] pArray;}
 T& operator[]( int n) {  if ( n<0  || Size <= n)
   throw std::Exception( "index out of range");
   return *(pArray + n); }       T operator[]( int n) const {
if ( n<0 || Size <= n) throw std::Exception( "index out
of range");  return *(pArray + n); } void display() const
{ for ( int i = 0; i<Size - 1; ++i)cout << *(pArray + i)
<<DisplayPolicy::seperator(); std::cout << *(pArray +
Size - 1) << "\n";    }private:
 Array( const Array< T, Size, DisplayPolicy>&);  //
make public impl. Later
 Array< T, Size, DisplayPolicy>& operator=(const Array
<T , Size, DisplayPolicy>&); // ditto
      T* pArray;};int main()
{    double temp1[] = { 0, 0.5, 1.0, 1.5, 2.0 };
Array< double, 5, rowDisplayPolicy> arr1(temp1);
arr1.display(); }
```

Policy
A policy is a class designed to tailor behavior of a template class in some narrow specific way: Locking policy for class that may be used in multi-threaded program: template<typename T, typename LockPolicy> class queue;Allows designer to use Lock or noLock policy. Enqueuing policy for a thread class:template <typename queue> class thread; Allows designer to optionally add queue and queue operations as part of thread class's functionality.HashTable policy for hashing table addresses: template <typename key, typename value, typename Hash> class HashTable;
Allows designer to provide hashing tailored for application long after HashTable class was designed.

Traits types are introduced by typedefs: Traits provide common type aliases used by all containers so functions that operate on the containers can be written to apply to every one of them without modification. std::string value_type std::string& reference_type std::string* pointer_type , HashIterator<double,string,HashDouble> iterator Traits allow a template parameterized class to be used in a function that is not aware of the parameter types. The function simply uses the "standard" name for the type provided by the class's traits

1. **Overriding**: Providing, in a derived class, a declaration and definition of a virtual base class function, using exactly the same function signature and the same or covariant return type.
a. A covariant return type is a pointer or reference of the derived type, when the base virtual function returns a pointer or reference of the base type.
Write Virtual Pointer table:
2. **Overloading**: Providing, in the same class, or in the same global scope, a function definition that uses the function identifier of another existing function with a different sequence of formal parameter types.
a. Note that you cannot overload on return type, because a client is not compelled to use the return type, so the compiler cannot figure out which function to bind to.

Evils: Dark Corners

1. **The compiler will generate certain member functions:**
a. If, and only if, no constructors are declared the compiler will generate, if needed, a
default constructor that does default construction of each of the class bases and member data. b. If no copy constructor, assignment operator, or destructor is declared in a class the compiler will generate one, if needed, which does copy, assignment, or destruction of each of the class bases and member data. This is only correct if each of the bases and data members has correct copy, assignment, or destruction semantics. C. So, for each class you design you should decide to let the compiler generate these functions if correct. Otherwise you must either implement them or make them inaccessible by making private or declare =delete on each signature.
2. **Each constructor may be initialized with list:**
a. When a constructor is called, as its first action, it calls a constructor for each of its bases and data members. For non-default constructors you should always specify how each of the bases and data members is to be constructed using an initialization list. If the compiler chooses which constructor to call it may not choose correctly.
b. Constant and reference members must be initialized with an initialization list since they cannot be reset.
3. Overloading non-virtual base class functions in derived classes (Hiding 1)
a. Overloads work only within a single scope, not across both base and derived class scopes.
b. The result may be hiding of base class member functions that are inherited by the derived class.
Class base { Void process(int x); void process();
} class derived : public base {void process(int x,int y); }
4. **overloading virtual functions (Hiding**
a. If a derived class redefines a base class virtual function, which is a correct procedure, that will hide the base class overloads that are inherited.

```
class base { public: virtual void
process(int i); void process(double d); }
 class derived:public base { {//hides
base proc
public:
  virtual ~base() {
  virtual void process(int x);
  virtual base& id(); };
```

5. **Avoid Redefining, in derived classes, non-virtual base class functions**
a. Non-virtual member functions do not have vtable entries and so the function called is the type of the pointer or reference, not the type of object attached to the pointer or reference.
b. So it is possible for a base class function to be called on a derived class object, with possibly disastrous results.
6. **Avoid Using default parameters in virtual functions**
a. Parameters don't have vtable entries, so they are bound based on the type of pointer or reference to an object, not of the object type.
b. This results in a derived class using base class defaults even though the derived class defined different values for the defaulted parameters.

```
class base{ public: virtual ~base() {}
  virtual void defaults(int i=12);};
class derived : public base{
public:   virtual void defaults(int i=10);};
```

7. **Provide a virtual destructor** if your class may be used as the base class for a derivation.
a. If you don't do that, and a client creates an instance of a class derived from your base class on the heap, bound to a base pointer, then when the client calls delete on that pointer, the destructor called is based on the type of pointer not the type of object, so the base destructor only will be called.
8. Multiple virtual Inheritance of implementation:
1. Virtual multiple inheritance merges multiple copies of an inherited base into one unique
shared base type. This type is initialized by the most derived class in the inheritance
chain. That means that one or more based of the most derived may not be initialized
as they specified in their constructors. So correct code can break correct code under
multiple virtual inheritance.
2. The best solution is to design your classes so that virtual inheritance is not needed.
3. Note that in virtual inheritance the keyword virtual qualifies classes not functions.

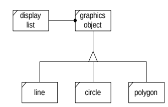
```
vector<Blob*> Blobs;
 Blobs.push_back(new Blob("Charley"));
 // push_back will copy base pointers
which act polymorphically.
 // No more slicing!
 Blobs.push_back(new
verboseBlob("Joe","Frank"));
 show(Blobs);
sort(Blobs.begin(),Blobs.end(),lessForBl
obs);
 show(Blobs);
 cout << endl;
 vector<Blob*>::iterator it;
 for(it=Blobs.begin(); it!=Blobs.end();
++it
```

Composition :

composition is a strong ownership relation. A composed part P becomes an integral part of the composer C. P is constructed at the same time as C and shares its lifetime
->Containing class has no special privileges regarding access to contained objects internals.
It has access only to public interfaces of contained objects, just like any other client.
-> Since the contained object lies in the private or protected parts of the containing object, clients can not call its functions and do not see its functionality, except as manifested by the containing object's behaviors - that is - the contained object helps to implement the outer object's functionality.Compositions are special associations which model a "part-of" or "contained" semantic relationship
Aggregation: Aggregations are special associations which model a weak "part-of" or "contained" semantic
relationship. An **aggregation** is a specific type of composition where no ownership between the complex object and the subobjects is implied. When an aggregate is destroyed, the subobjects are not destroyed. Because these subclass objects live outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed, but the subclass objects themselves will still exist.
Using: Using relationships provide access via references to instances of types R that are not Owned by the using class U. A using class does not own its used instance. That must be Provided by some other code and passed as an argument to a member function of the using class.

Inheritance :->

Inheritance enables the derivation of a new class with almost all the existing methods and data members of its base class.
->Derived class functions have access to protected data and functions of the base class.
->The derived class "is a" base class object with addi-tional capabilities, creating a new specialized object.
Inheritance has two main functions: 1) to support substitution of any one of a set of derived class instances, depending on the application context, and
2) to provide in one place code and resources that are shared across all derived instances



Presumably a client of the display list creates graphic objects based on user inputs and attaches them to the list. The display list and its clients do not need to know about the types of each of the objects.
They simply need to know how to send messages defined by the graphics object base class.
The base class graphicsObject provides a protocol for clients like the display list to use, e.g., draw(), erase(), move(), ... Clients do not need to know any of the details that distinguish one of the derived class objects from another.
Public Inheritance : Public derivation makes all of the base class functionality available to derived class objects. This has two very important consequences:
->New capabilities occur when the derived class adds new member functions or new state members which give the derived object richer state and functional behaviors.
->Specialized capabilities occur when the derived class modifies a base class virtual function.
Private Inheritance : Private derivation hides all of the base class interface from clients. By default none of the base class member functions are accessible to derived class clients.
Protected Inheritance : Protected inheritance is just like private inheritance from client's perspective. Clients have no access to base class functionality except through the derived class interface.
Protected inheritance is just like public inheritance from the derived class's point of view. Derived class member functions have access to all the base class public and protected members. The derived class members can use a derived class object anywhere a base class object is expected.
Overriding is supported by means of the virtual function pointer table that belongs to each class containing at least one virtual function.
Inheritance has two main functions: 1) to support substitution of any one of a set of derived class instances, depending on the application context, and 2) to provide in one place code and resources that are shared across all derived instances
Any base class guarantees that code which uses a pointer or reference to the base will compile and operate with a pointer or reference to any class that publically derives from the base. A base class that provides non-virtual member functions intends to provide exactly that code to each derived instance without need to define the common operations in more than one place
Interface: Is a C++ class with all pure virtual functions, an empty virtual destructor, and no data members. Its purpose is to establish a contract for services that can be implemented by any concrete derived class
Abstract class : Has at least one pure virtual function which prevents clients from creating instances
. Abstract classes provide common code and sometimes common data, shared by every concrete derived class. Often an abstract class derives from an interface and defines some non-virtual functions to be shared. It may create instances of a common data type, shown as SharedResource in the Figure, above. If that type is qualified as static the instance is shared with all concrete derived classes. If not, then each concrete derived class gets a copy of the same type.
Any pure virtual functions in its base must be defined by a concrete derived class.
Otherwise the inherited function remains pure virtual and the derived class is also abstract.
Concrete class
Must have definitions for all functions. It either inherits the definitions from a base or defines itself. The inherited definitions may be provided by any base class, e.g., the interface or the abstract base, shown in the diagram.
When concrete derived classes are allocated on the heap it is crucial that they have a virtual destructor or the correct sequence of derived and base destructors will not get called on destruction. If the top-most base class has a destructor explicitly declared to be virtual, then all the destructors in the hierarchy will also be virtual, whether they are explicitly declared to be or not.

Polymorphism:

polymorphism is the provision of a single interface to entities of different types.
This powerful mechanism is implemented in C++ using virtual functions.
Each derived class redefines the base class virtual draw() and hide() member functions in ways appropriate for its class, using exactly the same signature as in the base class.
An invocation of a virtual function through a base class pointer or reference to an object will call the function definition provided by the class of the object referred to.
This process is called polymorphic dispatching. We say that the display list object dispatches the virtual function draw()
Polymorphism places the responsibility for choosing the implementation to call with the object, not with the caller.
Allowing different objects (which must be type compatible) to respond to a common message with behaviors suited to the object is a powerful design mechanism.
It allows the caller to be ignorant of all the details associated with the differences between objects an simply focus on their base protocol.
-> Consider the display list example from the next page. Objects on the list may be any of the types derived from graphicsObject. The display list is said to contain a heterogeneous collection of objects since any one of the graphicsObject types can occur on the list in any order.
->The list manager needs to be able to apply one of several specific operations, like draw() or hide(), to every member of the list. However, draw() and hide() processing will be different for each object.
->**In above diagram** : If, however, a display list object has a list of pointers to base class graphicsObjects, the list can point to any derived object, line, circle, ... and an invocation: will call the draw function of the object pointed to, e.g. line, circle, ... , polygon.

Default Functions Generated :Default constructor

T() Create an instance with default initialization (defined by the body of this function) for creation of single instances and arrays of instances. The compiler generates this only if no constructors are declared in your class and your code attempts an unparmaterized construction.
copy constructor T(const T& t) Create copy for pass and return by value. Creates instance and copies state from the source. The compiler always generates this if not declared in your class, no move operations are declared, and your code attempts a copy, as in pass or return by value. Its action is to call a copy construction on each of the bases and data members of your class.
move constructor T(const T&& t)
Create copy by moving contents of source, used for return by value and create from temporary. Often this entails making the target point to state created by the source instance, on the heap, and setting the souce pointer to nullptr.
The compiler will generate this if it's not declared and implied by code that uses the class and move assignment and copy construction are not declared. Its action is to do memberwise move construction on bases and data members that are movable and copy construction on those that are not.
co**py assignment operator** T& operator=(const T& t)Copy state of source into existing object. The compiler will always generate this if it's not declared and implied by code that uses the class. Its action is to do memberwise assignment on all the bases and members of the class.
move assignment operator (C++11) T& operator=(const T&& t)
Move state of source into existing object. State handled in a fashion similar to move construction.The compiler will generate this if it's not declared and no copy or move constructors are declared. Its action is to do memberwise move assignment on all the bases and data members of the class.
destructor [virtual] ~T()
The compiler will generate this if no destructor is declared. Its action is to do member wise destruction on all bases and data members of the class.
Address Operator

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 Double d = 2345122.90; typedef char byte;

```
byte* byteArray = reinterpret_cast<char*>(&d); for(int i=0; i<sizeof(d); i++) cout << byteArray[i]
```

const cast : manipulates the constness of the object pointed by a pointer, either to be set or to be removed. `const std::string s = "hello"; string &sRef = const_cast<string&>(s);`

Liskov Substitution Principle:
Functions that use pointers or references statically typed to some base class must be able to use objects of classes derived from the base through those pointers or references without any knowledge specialized to the derived classes.

Substitution Failures
The base class does not make its destructor virtual,Derived classes redefine non-virtual member functions of the base,Virtual functions are overloaded or given default parameters
Clients use dynamic_cast to access derived class extensions to base class protocol through base class pointers or references.
To ensure a design supports the Liskov Substitution Principle:
derived objects must not expect users to obey pre-conditions stronger than expected for the base class their pre-conditions must be no stronger. derived objects must satisfy all of the post-conditions satisfied by the base class their post-conditions must be no weaker
base classes must provide virtual functions including a virtual destructor.
Deriving a square from a rectangle implies that one of the state variables, height or width, is redundant. Clients of rectangle need to know they are working with square if they take advantage of square's property - height = width.
The behavior of a square - change its height and you change its width - does not apply to rectangles and so square objects are not rectangle objects.

Project 2: Without knowing the implementation of tagged element, proc element or document element we are accessing the functions of these classes using base class shared pointer AbstractXMLElement by not violating any of the substituion principles

```
using sPtr = std::shared_ptr < AbstractXmlElement > ; sPtr pDocElement_;
std::shared_ptr < AbstractXmlElement > sp= makeTaggedElement(str); std::shared_ptr < AbstractXmlElement > sp3 = makeTaggedElement("SSP"); sp->addChild(makeTextElement("Sai-Sandesh-Pavan")); std::shared_ptr < AbstractXmlElement > sp1 = makeXmlDeclarElement(); sp1->addAttrib("Hello", "SSP Creations"); pDocElement_ = makeDocElement(); pDocElement_->addChild(sp1); pDocElement_->addChild(sp);pDocElement_->addChild(sp3);
```

Open Closed Principle :: Explain the graphic editor example for this
well designed code can be extended without modification and new features are added by adding new code rather than changing already working code
Open:A component is open if it is available for extension: -> add data members and operations through inheritance. ->Create a new policy template argument for a class that accepts policies.
Closed:A component is closed if it is available for use by other components but may not, itself, be changed, e.g., by putting it under configuration management, allowing read only access.

When open closed Interface occurs :
Latent errors force change. We can't fix incorrect operation by extension. The component itself must be fixed. Performance failures force change. When performance needs are not met we are forced to change the implementation to perform better, usually by changing a computational algorithm or data structure.

Abstract Interfaces to Fix open Closed Principle Issue

->When we program to abstract interfaces: changes to derived classes which implement the interface will not break any client code, and may not even require recompilation of some clients. What we can't do is change the interface definition. Any change here may force changes on most or all of its clients.
-> Make all member data private, e.g., no public, no protected data.
->No global data - ever -> No use of RTTI -> Use polymorphism and/or templates to provide extensions

Project 2 : We have an abstract class called abstract xml element, whenever we find a new XML property, we can design a class which inherits abstract xml element class without modifying the existing interface which is abstract class. Initially we did not have docElement and declaration element, since we have a interface which is open for extension we are able configure these new classes by deriving from the base class which is abstract xml element

```
class AbstractXmlElement{}
  class XmlDeclarElement : public AbstractXmlElement{
virtual bool addAttrib(const std::string& name, const std::string& value);
   virtual bool removeAttrib(const std::string& name); virtual std::string value() { return ""; }
   virtual std::string toString();}//derived frm interface
```

Dependency Inversion Principle: Use Example of IVector

High level components should not depend upon low level components. Instead, both should depend on abstractions.

Abstractions should not depend upon details. Details should depend upon the abstractions.
We all can agree that complex systems need to be structured into layers. But if that is not done carefully the top levels tend to depend on the lower levels.
In our project 2, consider the scenario where client is XML document, it depends on the abstractXmlElement for implementation details and makeTextElement(),makeTaggedElement for creating new objects. The XML element classes such as DocElemenet,TextElement element etc uses the interface declaration of abstract xml element for their implementation.In this project, the client XMLDocument is not depending on implementation details of XMLElements, It interacts only with interface(Abstract XML element) created by XMLElement and makefactory provides object of interface to XMLDocument.

```
class AbstractXmlElement{virtual std::string value() = 0;}
  class TextElement : public AbstractXmlElement{ std::string TextElement::toString(){
   std::string spacer(tabSize * ++count, ' '); std::string xml = "\n" + spacer + text_;
   --count; return xml;}}}//derived frm interface
std::shared_ptr<AbstractXmlElement> XmlProcessing::makeTextElement(const std::string& text){ std::shared_ptr<AbstractXmlElement> ptr(new TextElement(text)); return ptr;} std::shared_ptr < AbstractXmlElement > sp= makeTextElement(str);
```

Interface Segregation Principle :

->Clients should not be forced to depend upon interfaces they do not use.
this applies to clients of the public interface of a class, it also applies to derived classes
->We create interfaces to satisfy the needs of clients. When a component has several different clients it is tempting to provide a large interface that satisfies the needs of all clients.
->It is much better design to have the component support multiple interfaces, one appropriate for each client. ->Otherwise, if we have to change an interface we affect even those clients that do not use the features we change.

Project 2 Example: Consider CommentElement and TextElement derived clases, which don't have any child or attributes, keeping under abstract xml element might not make perfect sense.
we can create split the abstract syntax tree into a hierarchy of three interface,
the main interface provides the shared pointer, another interface provides addchildren and addabributes which are essential for TaggedElement, DocElement. The final interface is for Comment and Text Element which only provides value() and toString().

How Polymorphism is used in Project 2

In project 2, we have a abstract class abstractxmlElement which acts as protocol class for all the XMLElement classes where in we override the virtual and pure virtual functions of protocol classes by polymorphism. By using shared pointer, we are accessing the overriden functions of derived classes, wherein we need to assign the corresponding derived class to the shared pointer. using sPtr = std::shared_ptr <AbstractXmlElement>;
sPtr pDocElement_; std::shared_ptr < AbstractXmlElement > sp= makeTaggedElement(str);

Design Rules: Keep Principle, Door for Composition and Inheritnce, Separate Interface from Implementation, Decompose into smaller tasks, small is beautiful, User interface should be consitent, Data Type is important, Minimize dependencies, Handling pointers

Shared Pointer : std::shared_ptr<T> sptr1(new T); std::shared_ptr<T> sptr2(sptr1);
std::shared_ptr is a smart reference counted pointer that represents shared ownership of a resource created on the heap. When a shared pointer goes out of scope it decrements its reference count. Only if the count goes to zero is the resource deleted. You must be careful not to create more than one std::shared_ptr from a raw pointer. After the first is created you must create the remainder from an already defined std::shared_ptr sharing the same resource. using sPtr = std::shared_ptr <AbstractXmlElement>; sPtr pDocElement_;std::shared_ptr < AbstractXmlElement > sp= makeTaggedElement(str); sp1->addAttrib("Hi", "Sa");

Project 2 : Abstract Syntax Tree

```
class AbstractXmlElement { public: virtual bool addChild(std::shared_ptr<AbstractXmlElement> pChild); virtual bool removeChild(std::shared_ptr<AbstractXmlElement> pChild); virtual bool addAttrib(const std::string& name, const std::string& value);
   virtual bool removeAttrib(const std::string& name); virtual std::string value() = 0;
   virtual std::string toString() = 0; virtual ~AbstractXmlElement();
 protected: static size_t count; static size_t tabSize; };
class TaggedElement : public AbstractXmlElement { public: TaggedElement(const std::string& tag) : tag_(tag) {}; TaggedElement(const TaggedElement& te) = delete; virtual ~TaggedElement() {} TaggedElement& operator=(const TaggedElement& te) = delete; virtual bool addChild(std::shared_ptr<AbstractXmlElement> pChild); virtual bool removeChild(std::shared_ptr<AbstractXmlElement> pChild); virtual bool removeChild(const std::string& value); virtual bool addAttrib(const std::string& name, const std::string& value); virtual bool removeAttrib(const std::string& name); virtual std::string value(); virtual std::string toString(); private: std::string tag_; std::vector<std::shared_ptr<AbstractXmlElement>> children_; std::vector<std::pair<std::string, std::string>> attribs_;};
std::string TaggedElement::toString() { std::string spacer(tabSize*++count, ' '); std::string xml = "\n" + spacer + "<" + tag_; for (auto at : attribs_) { xml += " "; xml += at.first; xml += "=\""; xml += at.second; xml += "\""; } xml += ">"; for (auto pChild : children_) xml += pChild->toString(); xml += "\n" + spacer + "</" + tag_ + ">"; --count; return xml;}
std::shared_ptr < AbstractXmlElement > XmlDocument::makeTree(std::string str){
std::shared_ptr < AbstractXmlElement > sp= makeTaggedElement(str);
std::shared_ptr < AbstractXmlElement > sp3 = makeTaggedElement("S");
std::shared_ptr < AbstractXmlElement > sp9 = makeTextElement("F");
sp->addChild(makeTextElement("Hi")); std::shared_ptr < AbstractXmlElement > sp1 = makeXmlDeclarElement();sp1->addAttrib("Hi", "Sa");sp1->addChild(makeTextElement("Hieee"));
pDocElement_ = makeDocElement();pDocElement_->addChild(sp1);
pDocElement_->addChild(sp);pDocElement_->addChild(sp3);return pDocElement_;}
```

Liskov Substitution Principle

```
class quadrilateral {
public :
quadrilateral() :height(0), width(0){};
//quadrilateral(int h, int w) :height(h), width(w){};
virtual int getHeight(){ return 0; };
virtual int getWidth(){ return 0; };
virtual void setHeight(int){};
virtual void setWidth(int){};
virtual int area(){ return height*width; }
protected: int height;int width;};
class square :public quadrilateral {
private:void setHW(int value){
height = value; width = value;}
public : square() : quadrilateral() {}
//square(int h, int w) :quadrilateral(h, w){};
virtual int getHeight(){ return height; }
virtual int getWidth(){ return width; }
virtual void setHeight(int val){ setHW(val); } virtual void setWidth(int val){ setHW(val); }};
class rectangle :public quadrilateral {
public:
rectangle() : quadrilateral() {}
//rectangle(int h, int w) :quadrilateral(h, w){};
virtual int getHeight(){ return height; }
virtual int getWidth(){ return width; }
virtual void setHeight(int h){ height = h; } virtual void setWidth(int w){ width=w; }};
```

RTTI: C++ mechanism that exposes information about an object's data type at runtime.

```
Derived* pd = new Derived;
Base* pb = pd;
Derived d;
cout << typeid(pb).name() << endl;
//prints "class Base *"
cout << typeid(*pb).name() << endl;
//prints "class Derived"
cout << typeid(pd).name() << endl;
//prints "class Derived *"
cout << typeid(*pd).name() << endl;
//prints "class Derived"
cout << typeid(pb).before(typeid(pd)) << endl;
```

Mid Term Questions:
Members Not Inherited:
Constructors, assignment operator, and destructor are not inherited, but will be generated if the class does not declare them and are needed by code that uses the class. It would not make sense for these to be inherited since derived classes usually declare instances of data members and so inheriting these methods from the base class would not provide correct semantics as the derived data would not be initialized, assigned, and destroyed.

Abstract Class vs Inheritance
An abstract class has at least one pure virtual method. All methods of an interface are pure virtual. Furthermore, an interface has no data members and no constructors. It may provide a virtual destructor to insure that all derived destructors are virtual.

Compile Time/Run Time
You will use dynamic binding through virtual functions to change behavior at run time. The set of classes are the derived classes with a common base that defines a virtual function for the code that depends on the client's context, e.g., the application specific details.
You may define a template class to achieve changes in behavior at compile time. The set of classes are the template class and a class for each template parameter that defines one of the target behaviors.
These methods provide the flexibility to meet changing application requirements without changing existing code, but only adding new code, e.g., a derived class or a template parameter, so they directly support the Open Closed Principle.

Bind Application specific code to unkown library:
A C++ library can bind to application specific code at compile time by providing template functions and classes that accept template parameters that define application specific processing. The HashTable class provides template parameters for keys, values, and hashing functions, all of which depend on the specifics of an application that uses the HashTable.
A C++ library can bind to application specific code at run time by providing a base "hook" class for applications to derive from. An example of this is the navig class that provides the defProc hook for applications to use for derived classes that implement what the application needs when a new directory or file is discovered.

Model is Re-usable:
ObjectFactory,Interface&DDL package

Constantness:

What const implies is determined by where you find it: str(const str& s); is a contract that the argument s will not be changed. The compiler attempts to enforce the contract.
char operator[](int n) const; implies the state of the object on which the operator is applied will not change. Again, the compiler attempts to enforce the contract. Thus:
const str cs = "a constant string";
cs[3] = 'a'; will fail to compile

Polymorphism Example :

```
#include <iostream>
using namespace std;
class Polygon { protected:
int width, height;
public: Polygon (int a, int b) : width(a), height(b) {}
virtual int area (void) =0;
void printarea()
 { cout << this->area() << '\n'; }};
class Rectangle: public Polygon {
 public: Rectangle(int a,int b) : Polygon(a,b) {}
int area(){ return width*height; }};
class Triangle: public Polygon {
 public: Triangle(int a,int b) : Polygon(a,b) {}int area()
{ return width*height/2; }};
int main () {
Polygon * ppoly1 = new Rectangle (4,5);
 Polygon * ppoly2 = new Triangle (4,5);
 ppoly1->printarea();
 ppoly2->printarea();
 delete ppoly1; delete ppoly2;}
```

CallBack Patterns:

```
std::string testFunction(size_t lineNumber, const std::string& msg){
 std::ostringstream out;
 out << "\n testFunction invoked from line number " << lineNumber << " - " << msg;
 return out.str();}
```

Function Pointer:

```
std::string(*pt) (size_t, const std::string&) = testFunction;
```

Functor:

```
class Functor {
public:
 Functor(size_t lineNumber, const std::string& msg) : ln_(lineNumber), msg_(msg) {}
 std::string operator()(int i) { std::cout << i; return testFunction(ln_, msg_); }
private: size_t ln_; std::string msg_;};
Functor functor(__LINE__ + 1, "via functor");
 std::cout << functor(5);
```

via std:function std::function<std::string(size_t, const std::string&)> f = testFunction;

```
 std::cout << f(__LINE__, "via std::function<R(A...)>");
```

Example 1: class FunctorClass{

```
public:explicit FunctorClass(int& evenCount)
     : m_evenCount(evenCount) { }
   void operator()(int n) const {
     if (n % 2 == 0) {++m_evenCount;} }
int main() {vector<int> v;
  for (int i = 1; i < 10; ++i) {
     v.push_back(i);} int evenCount = 0;
 for_each(v.begin(), v.end(), FunctorClass(evenCount));}
```

Example 2: class ShorterThan {

```
 public: explicit ShorterThan(size_t maxLength) : length(maxLength) {}
 bool operator() (const string& str) const {
 return str.length() < length; }
 private: const size_t length; };
ShorterThan st(length);
 count_if(myVector.begin(), myVector.end(), st);
```

Lambda:

```
f = [](size_t size, const std::string& msg) ->std::string { return testFunction(size, msg); };
 std::cout << f(__LINE__, "via lambda");

int lineNo = __LINE__ + 2;
 std::string mutableMsg = "first message";
 std::cout << [&mutableMsg, lineNo]()->std::string { // capture mutableMsg by reference, lineNo by value
   return mutableMsg = testFunction(lineNo, mutableMsg) + " with some more text";
 }();
int lineNo = __LINE__ + 2;
 std::string mutableMsg = "first message";
 std::cout << [&mutableMsg, lineNo]()->std::string { // capture mutableMsg by reference, lineNo by value
   return mutableMsg = testFunction(lineNo, mutableMsg) + " with some more text";
 }();
 std::cout << mutableMsg;

 lineNo = __LINE__ + 2;
 mutableMsg = "second message";
 std::cout << [mutableMsg, lineNo]()->std::string {
// capture both arguments by value
   // mutableMsg += " with some more text\n" // this is an error, mutableMsg treated as const
   return testFunction(lineNo, mutableMsg) + " with some more text"; }();

class testClass {
public:
 using F = std::function<std::string(size_t, const std::string&)>;
 std::string mf1(F f, size_t line, const std::string& msg) { return f(line, msg); };

 using G = std::function<std::string()>;
 std::string mf2(G g){ return g(); }  // note: no arguments, will use capture
};
```

Lambda Example:check if a vector has even nm

```
 std::vector<int> vs{ 0,1,2,3 };
 std::vector<int> vs1;
 bool first = true;
 std::for_each(begin(vs), end(vs),
  [&vs1](const int i){
  if (i/2 == 0) { vs1.push_back(str); } } );
```

Exceptions : Example:

```
class betterUserException : public std::exception {
 public:
  betterUserException() : std::exception("betterUserException") { }
  virtual const char* what() const { return exception::what(); }
  void saveNumber(int num) { _number = num; }
  int savedNumber() { return _number; }
 private:  int _number;
};
```

Exception Matching

If you throw a literal string, say: "big trouble in River City" then it can be caught with the catch handler: catch(char *msg) { … }.
An exception handler that accepts a reference or pointer to a base class object will match a derived class object or pointer to a derived class object, respectively, as well as the base type specified.
If a derived class object is passed to a handler by value it will be sliced to a base class object.
If, however, a derived object is passed by reference or pointer, no slicing occurs, and polymorphic calls within the handler are honored. A catch handler with an ellipsis, catch(…) { … }, will catch any exception thrown in its context, not caught earlier.
->terminate function can be overriden

Exception Specifications:

Void f() throw E1,E2,E3
Void f() throw() -> no exceptions thr
Void f() -> any

Object Factory Structure

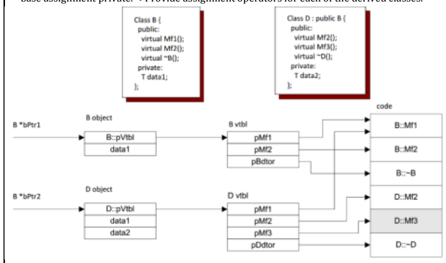
```
class AbstractProduct
{public:
 virtual ~AbstractProduct() {}
 virtual id ident()=0;
 virtual std::string OpA()=0;
 virtual size_t OpB()=0;};
struct Interface1{
 virtual ~Interface1() {}
 virtual std::string OpC() { return "";
}};
struct Interface2{
 virtual ~Interface2() {}
 virtual std::string OpD() { return "";
}};
class ConcreteProductA : public AbstractProduct, public Interface1{public:
 id ident(); std::string OpA();
 size_t OpB(){ return sizeof(*this);}
 std::string OpC();private: char buf[256];};
class ConcreteProductB : public AbstractProduct, public Interface2
{public:id ident();std::string OpA();
 size_t OpB() { return sizeof(*this);}
 std::string OpD();
private: char buf[1024];};
AbstractProduct*
Factory::CreateProduct(id productID)
{ switch(productID)
 { case 1 : return new ConcreteProductA;
 case 2 : return new ConcreteProductB;
  // can add more products here
 default : return 0;} }
Int main() {Factory f;
 AbstractProduct* pPr;
pPr = f.CreateProduct(1);
 if(pPr) {
  std::cout << "\n 1st pr" << pPr->OpA()
        << ". Its size is " << pPr->OpB() << " bytes.";
UseInterfaces(pPr);
 delete pPr; } } }
void UseInterfaces(AbstractProduct *pAP)
{
 Interface1* pI1 =dynamic_cast<Interface1*>(pAP);
 if(pI1) std::cout << "\n " << pI1->OpC();
 else std::cout << "\n can't use Interface1";Interface2* pI2 = dynamic_cast<Interface2*>(pAP);
 if(pI2)std::cout << "\n " << pI2->OpD();
 else std::cout << "\n can't use Interface2";}
```

If clients consist of a lot of code (millions of lines) that use these products (think of new parts of an evolving implementation) we don't want to rebuild each client every time a product implementation changes.
- Adding new products requires this Factory to be recompiled
Clients need only relink as no text changes in files they include
- New clients can use the new products - Modifying a Product implementation will only cause Clients to relink. Factory must recompile (not a problem since they are small). - If we build the products as Dynamic Link Libraries, the clients won't even have to relink.

WidgetFactory:

```
struct IWidget{
 virtual void doWork()=0; virtual ~IWidget() {}};
class Widget1 : public IWidget{
public: virtual void doWork();
 virtual ~Widget1();};
void Widget1::doWork(){
 std::cout << "\n Widget1 working";}
struct WidgetFactory
{ static IWidget* CreateWidget(int WidgetIndex);};
IWidget*
WidgetFactory::CreateWidget(int WidgetIndex){
 switch(WidgetIndex) {
  case 1: return new Widget1();
  case 2: return new Widget2();
  default:  return 0; }}
IWidget* pWidget = WidgetFactory::CreateWidget(1);
 if(pWidget) {
  pWidget->doWork(); delete pWidget; }
 else std::cout << "\n can't create widget1";
```

Interface/ Protocol Class :
Abstract class with no data members, no constructor and at least one pure virtual function.
A protocol class (C++ interface) provides a language for clients to use when interacting with any of its derived class instances.
Hookup:->A hook is a base class that supports modification of a library's behavior by Applications designed to use the library, without modifying any of the library's code.
->The Hook class provides a protocol for application designers to use and a set of virtual methods that are overridden to provide required application behavior.
->One very common usage provides a way for applications to respond to events that occur within the library.
Mixin:A mixin class provides a specialized set of behaviors intended to be used through multiple inheritance with other classes.
Mixins can be used to provide reference counting, locking, memory management, and other specialized behaviors as services to any class that needs them.
Koenig Lookup: If you supply a function argument of class type, then to find the function name the compiler is required to look not just in the local or surrounding scopes, but also in the namespace that contains the argument's type. For this reason, for the string class, packaged in namespace std, which has an operator<< we can say: std::cout << myString; instead of: std::operator<<(std::cout,myString);
Share Common Code: ->A base class may provide default implementations of some or all of the base class protocol. These, then, are shared by all derived classes that do not override the defaults. ->Disallow assignment of derived instances to base instances or derived instances of one type to an instance of another type through base pointers or references.
->You can make the base class abstract by including a pure virtual function and make the base assignment private. ->Provide assignment operators for each of the derived classes.



Building Software Components : Dynamic Linked Library:
->Components are packages that export only two things: 1) an interface, and 2) an object factory. Clients can create instances of all the objects the package needs to implement its services using the object factory. The factory returns a package interface pointer, bound to its internal implementation. The client uses the interface to access package services.
->The interface allows clients to bind to a service abstraction without binding to any implementation detail. The factory supports initialization of the service without binding its clients to that startup process. Hence, clients are completely isolated from the component implementation.
-> Using the component structure, e.g., exposing only interfaces and object factories, a component at any level can be modified and rebuilt without causing any design or compilation breakage. We simply need to relink the component into the system.
If we build each of the components as dynamic link libraries (DLLs) loaded by an executive, then we don't even need the link phase. We simply modify and rebuild the component and copy its new DLL into the directory where the executive looks for libraries to load.
That replaces the old component with the new one. Since libraries are loaded when the executive starts, the new part begins its service.
Singleton holds a static reference to the single instance, so any instance of the Singleton class, declared in any scope will provide access to the same shared instance.
Global access supports sharing across scopes, but the shared instance must be thread-safe if shared between two or more threads that run concurrently. The Singleton does not ensure that.
Project 1:

```
public:
using path = std::set < std::string > ;
using pathIterator = std::set<std::string>::iterator;
pathIterator pathReference;
using fmap = std::map < std::string, std::list<pathIterator> > ;
using iterator = fmap::iterator;
void DataStore::save(const std::string &absfilename) {
unsigned found = absfilename.find_last_of("/\\");
std::string fpath = absfilename.substr(0, found);
std::string fname = absfilename.substr(found + 1);
bool is_in = pathSet.find(fpath) != pathSet.end();
if (!is_in) pathSet.insert(fpath);
for (pathReference = pathSet.begin(); pathReference != pathSet.end(); ++pathReference)
{ //If the input file path is present in set
if (*pathReference == fpath){ // find if the filename is already present. If not, insert
bool is_in = filemap.find(fname) != filemap.end();
if (!is_in){ fileCounter++;std::list<pathIterator> pRef;
pRef.push_back(pathReference);filemap.insert(make_pair(fname, pRef));}
else{// Retrieve the Iterator list and add the path reference to the list and update map
std::list<pathIterator> pRefList = filemap.at(fname);
bool found = (std::find(pRefList.begin(), pRefList.end(), pathReference) != pRefList.end());
if (!found){fileCounter++;pRefList.push_back(pathReference);filemap[fname] = pRefList;}}}}}

void FileManager::processInput(){try {path_ = "."; if (argc_ >= 2) {std::string arg = argv_[1];
if (arg[0] != '/' && (arg.find("*") == std::string::npos)) {
path_ = argv_[1];}}for (int i = 1; i < argc_; ++i) {std::string arg = argv_[i];
//If options add to option vector
if (arg == "/s" || arg == "/d" || std::regex_match(arg, std::regex("(/f)(.*)"))){
if (std::regex_match(arg, std::regex("(/f)(.*)"))) {searchString = arg.erase(0, 2);
addOption("/f");}else {addOption(arg);}}// else If string has *, then add to pattern vector
else if ((arg.find("*") != std::string::npos)) {
addPattern(arg);}//else add everything except first argument to option pattern vector
else {if (i != 1) {addPattern(arg);}}}}

searchFiles(path_);
bool is_in = std::find(options_.begin(), options_.end(), "/s") != options_.end();
if (is_in){searchDirectories(path_);}

void FileManager::searchFiles(const std::string& path) {
bool res = FileSystem::Directory::setCurrentDirectory(path);
for (auto patt : patterns_) { std::vector<std::string> files;
if (res) {files = FileSystem::Directory::getFiles
(FileSystem::Directory::getCurrentDirectory(), patt);
for (auto f : files){ std::string p = FileSystem::Path::getFullFileSpec(f);
store_.save(p);}}}}

//----< search directories in the path >--------------------------------
void FileManager::searchDirectories(const std::string& path) {
std::string curDir = FileSystem::Directory::getCurrentDirectory();
std::vector<std::string> directories = FileSystem::
Directory::getDirectories(curDir); std::vector<std::string> newDir;
std::string test = FileSystem::Directory::getCurrentDirectory();
for (auto f : directories){ if (!((f == ".") || (f == ".."))){
dirCounter++; std::string res = test + "\\" + f;
FileSystem::Directory::setCurrentDirectory(res);
searchFiles(res);newDir = FileSystem::Directory::getDirectories();
if (newDir.size() >= 1) {searchDirectories(".");}}}}

void Catalogue::searchAndSaveFile(const std::string absolutePath, const std::string key) {
FileSystem::File in(absolutePath);
in.open(FileSystem::File::in); boolean found = false;
while (in.isGood()) {
std::string filetxt = in.readAll();
if (filetxt.find(key) != std::string::npos) {found = true;break;}}
if (found) {fileSet.insert(absolutePath);}}
```