CSC 181F Lecture Notes

C++ Classes

- The *class* in C++ is essentially the module[a].
- The primary reason for classes is to provide *encapsulation* and *information hiding* so that large programs can be built from small separable pieces.
- In C++ classes are composed using a mechanism called *inheritance* which allows a hierarchy of classes that provide similar services.
- C++ distinguishes between the declaration of a class and an *instance* of the class which is often called a *class object*

[a] Recall Slides 278-289

C++ Classes

- A class specification has two parts:
  - A class declaration, which describes the data component, in terms of **data members** , and the public interface, in terms of **member functions**
  - The **class method** definitions, which describe how certain class member functions are implemented
- The principle is to separate the details of the implementation from the design of the interface.
- The implementation of the data representation or the member functions can be changed without changing the interface.

C++ Classes Declaration

```
class className {
    public :
    // visible to all clients
    private :
    // visible only to member functions
};
```

Public part is the interface the class provides to all clients.
Private part is visible only to functions declared in this class.

The declarations in a class can include data items, i.e. variables and constants, function declarations (prototypes) and complete function definitions[a].
A class containing only data items behaves like a struct or union.
Data members normally go into the private section.

Member functions can be defined in or outside of the class declaration that they belong to. Member functions defined inside class declaration will be **inline function** regardless of whether the keyword **inline** is used.

[a] Recall the distinction between function declaration and function definition

Class Data and Class Objects

- A **class declaration** creates a template for defining *class objects*. The declaration may include data and member functions that operate on the data.
  Think of a class declaration as an extended typedef
- The name of the class is used to declare **class objects** which are *variables* of the class type[a].
- A class containing only data declarations behaves like a **struct** or **union** .
  In C++, **struct** and **union** delcarations behave almost exactly like **class** declarations.
- Each class object gets its **own copy** of the data declared in the class.
  Think of a class object as a struct variable.
  Exception: there is only once instance of class data members declared with the **static** attribute. This instance is shared by all objects of the class.

[a] For space efficiency reasons there is usually only one copy of the member functions for the class.

C++ Classes Declaration Example

```
class Stack {
    public :
        void init( );       // Initialize the stack
        void push( int item );      // Push an item on the stack
        int pop( ) ;      // Pop an item from the stack
    private :
        int count;      // Number of items in the stack
        int data[STACK_SIZE] ;      // The items themselves
};
```

Note that a member declaration cannot contain an initializer.

```
        int count = 0 ;      // *** Error ***
```

Scope Resolution Operator

::X	External scope
CL::M	Class scope

- The first form is used to refer to a global variable X when this variable would ordinarily be inaccessible because of a name conflict with a variable having local or class scope.

```
    float X;      // global x
    void f( int N ) {
        float X ;      // local x
        X = 1.5 ;      // refers to local (f's) X
        ::X = 2.5 ;      // refers to global X
}
```

- The second form is used to refer to member M in class CL.

Implementing Class Member Functions

- When defining a member function outside the class declaration, use the scope operator (::) to identify the class to which the function belongs.
  We can use the same name for a member function for a different class.
- Class member functions can access the private components of the class
  A class method can use another class method in the same class without using the scope resolution operator.

Example:

```
void Stack::push( int item )
    {
        data[count] = item ;
        ++count ;
    }
```

```
void Pile::push( int item ) {
        pilePtr = new pileNode ;
        assert ( pilePtr ) ;
        pilePtr-> value = item ;
        pilePtr-> next = pileTop ;
        pileTop = pilePtr ;
    }
```

const Member Functions

- A programmer can declare a constant instance of a class. For example:
  const Stack conStack **;**
- Since the programmer manipulates data in the class by calling member functions, the compiler needs help to guarantee the **constantness** of the class is preserved.
- A member function can be declared to **not change** the internal data of a class by putting the keyword **const** *between* the functions argument list and the function body. It is an error to invoke a non-constant member function of a constant class.
- **Example:**

```
int Stack::size() const ;                       // Declaration
int Stack::size() const {    return count ;  }   // Definition
```

Using a Class

Can create a class object by declaring a class variable or by using **new** to allocate an object of a class type.

Example:

```
Stack s1 ;                  // define a stack object
Stack * sp ;                // pointer to a stack object
s1.init( ) ;                // invoke s1's init method
s1.push( 7 ) ;              // insert 7
s1.push( 12 ) ;             // insert 12
cout << s1.pop( ) ;         // remove and print 12
cout << s1.data[0] ;        // *** Error: data private ***
sp = new Stack[ 20 ] ;      // array of stacks
```

C++ friends

- Strict information hiding provided by **private** may be too strong in some cases.
- C++ provides a controlled way to grant access to private data to functions that are not member functions of the class.
- The mechanism is the **friend** declaration. A friend can be a specific function from another class or an entire class.
- **Good Style:** put the **friend** declarations at head of class declaration.
- A class must explicitly grant access to each friend.

C++ friend declarations

```
class Grantee {
    friend class friendClass ;
    friend type friendFunct( .. );
    friend type friendClass::friendFunct( .. );
}
```

The first form grants friend priviledges to the entire class *friendClass*

The second and third forms grant friend priviledges to the function *fFunct*

friend declarations *must* be used in some cases which will be described later.

Good Style: Use friend access **only when absolutely necessary**.

Friends Example

```
class F {
private :
    int adm ;
public :
    int f( void ) ;
}

class C {
private :
    int cdm ;
    friend class F ;    // class F a friend
public :
    int m( void ) ;    // class method
    friend int t( void ) ;    // friend
    friend int F::f( void ) ;    // method friend
};
```

Class Constructors

- Class constructors are special method functions for constructing new objects and assigning values to their data members.
- The compiler ensures that the constructor is invoked whenever an object is created. A constructor's name is the class's name.
  A constructor has no declared type. A class constructor may be overloaded to deal with different initialization conditions.
- The prototype for the constructor goes in the public section of the class declaration.
  If the constructor takes no arguments, it is called the **default constructor**.
- Once you define a constructor, a program must use it when creating an object.
  If you fail to define any constructors, the compiler provides a default constructor, one does nothing.

```
    Stack( ) { }
```

Class Constructors Example

```
class Stack {
    public :
        Stack( ) ;    // stack constructor
        void push( int item ) ;    // Push an item on the stack
        int pop( ) ;    // Pop an item from the stack
    private :
        int count ;    // Number of items in the stack
        int data[STACK_SIZE] ;    // The items themselves
} ;

// constructor definition
Stack::Stack()
{
    count = 0 ;
}
```

More Class Constructors Examples

```
class Color {
    private :
        float red ;
        float green ;
        float blue ;
    public :
        Color( ) { red = green = blue = 0.0 ; }    // inline
        Color( float r, float g, float b ) ;
        ...
} ;
Color::Color( float r, float g, float b )        // constructor with initial values
{
    red = r ; green = g ; blue = b ;
}
Color c1 ;                // Color::Color( ) constructor used
Color *c2 = new Color ;    // Color::Color( ) constructor used
Color c3( 1.0, 0.5, 0.0 ) ;    // Color::Color(float r, float g, float b ) constructor used
```

Class Destructor

- A destructor is automatically called whenever an object is destroyed, e.g., by going out scope or by using the **delete** operator.
- A destructor's job is to free any storage that a constructor dynamically allocates before the allocated storage becomes garbage.
- A destructor's name is the class name preceded by a tilde (~).
  A destructor can have no return value and have no declared type.
  A destructor can have no arguments.

Example:

```
    Stack::~Stack( void ) {
        if ( count != 0 )
            cerr << "Warning: Destroying a nonempty stack" << endl ;
    }
```

Constructor/Destructor Example

```
class myString {
    private :
        int len ;        // length
        char * s ;       // string
    public :
        // two constructors
        myString(int size = 255 )          // String S1 ;
            { len = size ; s = new char [size+1] } ;
        myString( char * si )          // String S2 = "initialValue" ;
            { assert( si ) ;
                len = strlen(si) ;
                s = new char [len+1] ;
                strncpy(s,si,len+1) ;
            } ;
        ~myString() { if (s) delete [ ] s } ;      // Destructor
} ;
```

HOW TO Construct and Destruct

- Use the constructor to initialize (if necessary) the internal data belonging to a class object.
- Overload the constructor to make sure you have dealt with all circumstances in which a class object gets created.
- Use the destructor to
  - Verify that the object has class object has been used properly. e.g. a Stack being deleted is empty.
  - Clean up (deallocate) any data that was dynamically allocated for the class object using **new** or malloc. *This is your tool for dealing with memory leaks.*
  - Put any global data structures used by the class object into a correct state. for example, close files.

Other C++ Operators

- Value construction operator: *type* (*expression*)
  an alternative to the cast operator.
- Pointers to class members must reference a specific class object. There are two special operators for doing this.
  - *classObject* **. *** *memberPtrVar*
    Dereferences memberPtrVar contained in classObject.
  - *classObjectPtr* **-> *** *memberPtrVar*
    Dereferences classObjectPtr to access some class object, then dereferences memberPtrVar in that class object.

C++ this pointers

className ***** this **;**

- For each **class** declaration, a pointer to the class named *this* is *automatically declared*.
- The *this* pointer behaves like a pointer to a **struct** where the struct contains the local data for the class object.
- The *this* pointer is automatically set every time a member function in the class is called.
- The member functions can use the *this* pointer to access the local data of the class object that caused their invocation.

C++ this Pointer Example

```
class Stock {
    private :
        double total_val ;
    ...
    public :
    ...
        const Stock & topval( const Stock & s ) const ;
}
const Stock & Stock::topval( const Stock & s) const
{
    if ( s.total_val > total_val )
        return s ;
    else
        return *this ;      // reference to self
}
Stock s1, s2 ;                  // two class objects
Stock top = s1.topval(s2) ;     // top = max( s1, s2 )
```

C++ Operator Overloading

type-name **operator** *opSymbol*(*parameters*)

- *type-name* is the type of value returned by the operator.
  (May need to be a reference in some cases.)
- *opSymbol* is the operator being overloaded. e.g. + **,** * **,** = etc.
- The function like parameters are the operands of the operator.
  Multiple overloads can be made for the same operator.
- For binary operators the C++ compiler uses the left operand of an operator to select among possibilities for an overloaded operator.

- There are two ways to define overloaded (binary) operators
- The overloaded operator is defined as the *member* function of some class.
  - The operator will be invoked when it appear with a *left operand* that is an object of the class.
  - The *this* pointer will refer to the left operand.
    A parameter of the right type will be required for the right operand.
- The overloaded operator is defined as a *nonmember* function
  - The standard rules for resolving references to overloaded functions will be used to determine when the operator function is invoked.
  - The function will require *two* parameters for the left and right operands.
  - You must use this form if the left operand cannot be an object of the class. e.g. overloading the << operator.

Examples:

```
int myString::operator < ( const myString & sRight ) const ;
int operator <= ( const mySTring & sLeft , const myString & sRight ) const ;
```

Operator Overloading Restrictions

- The overloaded operator must have at least one operand that is a user-defined type.
- You cannot use an operator in a manner that violates the syntax and semantic rules for the original operator.
  Cannot change the precedence of the operator.
  Cannot change the *arity* (unary or binary) of the operator .
  Cannot create new operators.
- You cannot overload the following operators:

sizeof	the sizeof operator	.	membership operator
.*	pointer-to-member operator	::	scope resolution operator
?:	conditional operator		

C++ Operator Overloading Example

```
class Pair {
private :
    double X , Y ;
    ...
public :
    Pair(void ) ;              // default constructor
    Pair(double H , double V) ;     // constructor
    Pair operator * (double N) const ;
    ...
}
Pair Pair::operator * ( double N ) const ;     // definition
{
    double MX , MY ;
    MX = N * X ;
    MY = N * Y ;
    Pair scaled = Pair( MX , MY ) ;
    return scaled ;
}
Pair V ;
...
Pair Q = V * 2.0 ;     // Q = V.operator * (2.0) ;
Q = 2.0 * V ;          // *** Error, not supported ***
```

Friends and Operator Overloading

- A *friend function* is a nonmember function that is allowed access to an object's private section.
  Good Style: declare the function in the class declaration using keyword **friend** . Don't use the keyword in the function definition.
- A friend function has to access an object explicitly by an argument.
  Use friend function for overloading an operator that takes two different types of operands or where the left operand is not of the class type.

```
friend Pair operator * ( double N , const Pair & A ) ;     // declaration
Pair operator * ( double N , const Pair & A ) {     // definition
    return A * N ;
}
Pair Q = 2.0 * V ;          // now supported
```

Friendly Operator Overloading Example

- To overload the << operator to display an object of class Class_Name, use a friend function with a definition:

```
ostream & operator<< ( ostream & os, const Class_Name & obj )
{
    os << ... ;
    return os ;
}
```

- cout is the ostream object and the prototype of << operator is defined as

```
ostream & operator<< ( typename ) ;
```

Example:

```
ostream & operator<< ( ostream & os, const Pair & V )
{
    os << "(x,y) = (", << V.X << " , " << V.Y << ")" ;
    return os ;
}
Pair V(4.0 , 5.0) ;
cout << V ;          // (x,y) = (4.0 , 5.0)
```

Type Conversions: Convert Constructors

- A one-parameter constructor is called a **convert constructor**.
- If the parameter is of type *T*, the convert constructor for class *C* converts type *T* to type *C*.
  That is, converting the argument to the internal representation in terms of the data members of the class *C*.
- In the following situations, convert constructors are used to convert *T* type to *C* type.
  - When a *C* object is initialized to a *T* type value.
  - When a type *T* value is assigned to a *C* object.
  - When a type *T* value is passed to a function expecting a *C* object argument.
  - When a function that's declared to return a *C* object tries to return a *T* value.

Convert Constructors Example

```
class Clock {
public :
    Clock() { hour = 12 ; min = 0 ; ampm = 0 ; }
    Clock( int ) ;
private :
    int hour, min, ampm ;      // ampm is 0 for AM and 1 for PM
} ;
Clock::Clock( int time )      // time is given as 24-hour time
{
    min = time % 100 ;
    hour = time / 100 ;
    if ( hour > 12 ) {
        hour -= 12 ;
        ampm = 1 ;
    } else
        ampm = 0 ;
}
// converting int to Clock using Clock::Clock( int )
Clock c = 1150 ;
...
c = 2330 ;
```

Type Conversions: Conversion Functions

operator *type-name* () **;**

- Conversion function are used to convert a class object to some other type, i.e *type-name* .
- Conversion function is a class member function, it has no declared return type and no arguments.
- The conversion function is automatically invoked when you assign a class object to a variable of that type or use the type cast operator to that type.

Conversion Function Example

```
Clock::operator int() const
{
    if (ampm == 1)
        return ( hour + 12 ) * 100 + min ;
    else
        return hour * 100 + min ;
}
Clock c = 2249 ;      // convert int to Clock using convert constructor
int time1 = c ;       // convert Clock to int using conversion function
int time2 = int ( c ) ;
```

Copy Constructors

Class_name(**const** *Class_name* & *)* **;**

- The copy constructor is invoked whenever a new object is created and initialized to an existing object of the same kind.
  The copy constructor is also used whenever a program generates copies of an object, *e.g.,* when function passes an object by value or when it returns an object.
- If you don't define a copy constructor, the compiler provides a default copy constructor which performs a member-by-member copy of the *nonstatic* members[a]
- **Good Technique:** If a class contains a static data member whose value changes when new objects are created, you should provide an explicit copy constructor.
- **Good Technique:** If a class contains members that are pointers initialized by **new** , then you should define a copy constructor that copies the pointed-to data instead of copying the pointers themselves.

[a] The *static* data members of a class are shared among all objects of the class.

Copy Constructors Example

```
class MyString {
public :
    MyString( const char *s ) ;
    MyString( ) ;

    ˜MyString( ) ;
    MyString( const MyString & st ) ;
private :
    char * str ;
    int len ;
    static int num_strings ;
}
MyString::MyString( const char *s )
{
    len = strlen( s ) ;
    str = new char [ len + 1 ] ;
    strcpy( str, s ) ;
    num_strings++ ;
}
```

```
MyString::MyString( const MyString & st )
{
    num_strings++ ;

    len = st.len ;
    str = new char [len+1] ;
    strcpy( str, st.str ) ;
}
MyString::˜MyString( )
{
    --num_strings ;
    if ( str )
        delete [] str ;
}
```

Overloading the Assignment Operator

- The assignment operator (=) is used when one object is assigned to another existing object.
- If you don't define an assignment operator, the compiler provides one which performs a member-by-member copy of the nonstatic members.
- **Good Technique:** You **should** define an overloaded assignment operator if the class data contains pointers or other data that requires special handling.
- If an overloaded assignment operator is defined,
  - It should check for self-assignment, i.e. X = X **;**
  - It should free memory formerly pointed to by the member pointers
  - It should copy the data, not just the address of the data
  - It should return a reference to the invoking object

Assignment Operator Example

```
MyString & MyString::operator =( const MyString & S )
{
    if ( this == & S )      // object assigned to itself
        return * this ;
    delete [ ] str ;      // free old string
    len = S.len ;      // copy string data
    str = new char [ len + 1 ] ;
    strcpy( str, S );
    return * this ;      // return reference to invoking object
}
```

More on Constructors

- Object initialization with **new**

  Class_name *ptr = **new** Class_name(val) **;**

  invokes the

  Class_name(Type);

  constructor, where Type is the type of val.

  Class_name *ptr = **new** Class_name **;**

  invokes the default constructor.

- Initializer lists
  - Class data members can be initialized with constructors.
  - Initialization takes place when the object is created.

    ```
    Class_name::Class_name( ... ) :member1( .. ), member2( .. ), ...
    {
        ...
    }
    ```
  - Must use this form to initialize a nonstatic **const** data member.
    Must use this form to initialize a reference data member.

Overloading Memory Management Operators

void * **operator new** (size_t size) **;**

void operator delete (**void *** objectPtr) **;**

- Both forms can be either a member function or a toplevel operator function.
- The overloaded **new** operators must return a **void** *.
  The first paramter in the overloaded **new** operator must be of type `size_t`.
  The value of this parameter is the size in bytes of the object created.
- The overloaded **delete** or **delete** [] operator must reuturn a **void** .
  The first parameter in the overloaded **delete** must be of type **void** *.
  The pointer objectPtr must point to the storage to be freed.

Looking Under the Hood

- To help you understand how C++ classes actually work, Slides 448 through 452 describe how the C++ compiler implements various C++ constructs in terms of C code.
- The left column shows the C++ program *as you would write it*.
- The right column shows the declarations and statements that the C++ compiler generates *internally* to implement the C++ constructs.
- Identifiers that start with # are internal names that are created by the compiler.
- This example doesn't necessarily correspond to any actual implementation of C++.

C++ Program	Implementation
class myStr {	**typedef** #myStrData ***** #myStrPtr **;**
public :	
friend ostream & ostream::**operator**	**friend** ostream & ostream::**operator**
<< (ostream & os **, const** myStr & S) **;**	<< (ostream & os **, const** myStr & S) **;**
myStr(**int** size = 255) **;**	myStr(#myStrPtr this **, int** size = 255) **;**
myStr(**char *** SI) **;**	myStr(#myStrPtr this **, char *** SI) **;**
int length( ) **;**	**int** length(#myStrPtr this) **;**
myStr **operator +** (**const** myStr & SR) **;**	myStr **operator +** (#myStrPtr this **,**
	const myStr & SR) **;**
private :	**struct** #myStrData
int len **;**	**int** len **;**
char * str **;**	**char *** str **;**
}	} **;**

C++ Program	Implementation
myStr S1 **;**	#myStrData S1 **;**
	myStr(& S1) **;**
myStr S2 = ”Hello World” **;**	#myStrData S2 **;**
	myStr(& S2 **,** ”Hello World) **;**
myStr S3(128) **;**	#myStrData S3 **;**
	myStr(& S3 **,** 128);
myStr S4[100] **;**	#myStrData S4[100]
	for (**int** #I = 0 **;** #I < 100 **;** #I++)
	myStr(& S4[#I]) **;**
myStr ***** S5 = **new** myStr	#myStrData ***** S5 **;**
	S5 = **new** #myStrData **;**
...	myStr(S5) **;**
delete S5 **;**	˜myStr(S5) **;**
	delete S5 **;**

C++ Program	Implementation
int myStr::length() {	**int** myStr::length(#myStrPtr this) {
return len **;**	**return** this-> len **;**
}	}
myStr(**char *** SI) {	myStr(#myStrPtr this **, char *** SI) {
assert (SI) **;**	**assert** (SI) **;**
len = strlen(SI) **;**	this-> len = strlen(SI) **;**
str = **new char** [len + 1] **;**	this-> str = **new char** [this-> len + 1] **;**
assert (str) **;**	**assert** (this-> str) **;**
strncpy(str **,** SI **,** len+1) **;**	strncpy(this-> str **,** SI **,** this-> len+1) **;**
}	}

C++ Program	Implementation
myStr myStr::**operator +** (**const** myStr & SR) {	myStr myStr::**operator +** (#myStrPtr this **,**
	const myStr & SR) {
assert (str & S.str) **;**	**assert** (this-> str & S.str) **;**
int outLeng = strlen(str) + strlen(SR.str) + 1 **;**	**int** outLeng =
	strlen(this-> str) + strlen(SR.str) + 1 **;**
myStr Sout(outLeng) **;**	#myStrData Sout **;**
	myStr(& Sout , outLeng) **;**
strcpy(Sout.str, str) **;**	strcpy(Sout.str, this-> str) **;**
strcat(Sout.str **,** SR.str) **;**	strcat(Sout.str **,** SR.str) **;**
return Sout **;**	**return** Sout **;**
}	}

C++ Program	Implementation
myStr S6 = ”Hello ” **;**	#myStrData S6 **;**
	myStr(& S6 **,** ”Hello ”) **;**
myStr S7 = ”World” **;**	#myStrData S7 **;**
	myStr(& S7 **,** ”World”) **;**
myStr S8 **;**	#myStrData S8 **;**
	myStr(& S8) **;**
S8 = S6 + S7 **;**	S8 = myStr::**+** (& S6 **,** & S7) **;**
int K = S8.length() **;**	**int** K = myStr::length(& S8) **;**

Templates in C++

- Templates are an effort-saving mechanism in C++
- Templates allow you to write an function once and apply it to many different types of data.
- Function Templates allow you to parameterize the definition of a function with one or more *type* parameters.
- Class Templates allow you to parameterize class (module) definitions with type and value parameters.
- Function templates make generating multiple function definitions simpler and more reliable.

Function Templates

template < *typeList* >

function definition

- **template** and the angle brackets are required.
  typeList is a comma-separated list of items of the form
  class *identifier*
- The *identifiers* may be used in the function definition any place that a *type-name* could be used.
  They *must* be used in the functionparameter list.
- When the function is called the compiler uses the *type* of the arguments to the call to instantiate an instance of the function with an appropriate body.
- The first use of each type argument in a call determines the type used in the call.

Function Template Example

```
template < class Type >
void swap( Type & A , Type & B ) {
          Type temp = A ;                    // Swap any A and B
          A = B ;
          B = temp ;
     }
...
int J , K ;
double X , Y ;
swap( J , K ) ;                              // swaps integers
swap( X , Y ) ;                              // swap doubles
```

Class Templates

```
template < templateParameters >
class {
     ..
}
```

- The class template construct defines a generic class with substitutable *type* and *constant expression* parameters
- The *templateParameters* is a list of items:
  class *typeName* identifier
  type-name identifier
- The **class** parameter defines a type parameter that can be substituted in the body of the class anywhere that a type is required.
- The second form of parameter defines an expression parameter that can be substituted in the class anywhere that an expression of type *type-name* is required.

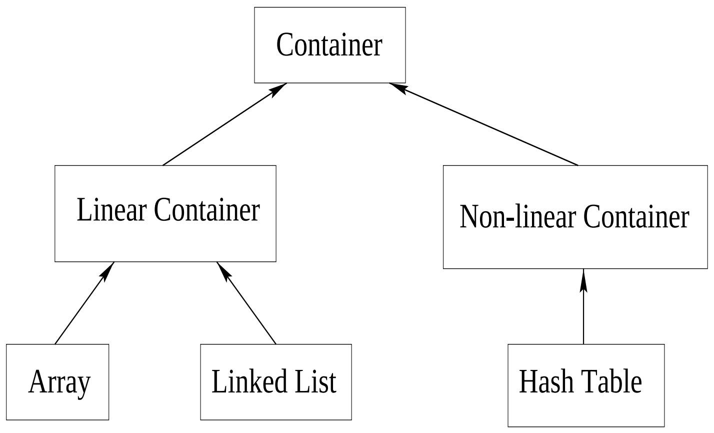
Class Template Example

```
template < class T , int size >
class Stack {
public :
    enum { FullStack = size, EmptyStack = -1 } ;
    Stack() ;
    void push( T newVal ) ;
    T pop( void ) ;
    int isEmpty( void ) ;
    int isFull( void ) ;
private :
    T items[ size ] ;
    int top ;
}
template < class T , int size >
Stack< T, size > ::Stack()
    {    top = EmptyStack ;    }
template < class T , int size >
void Stack< T, size > ::push( T newVal )
    {    items[ ++top ] = newVal ;    }
template < class T , int size >
T Stack< T, size > ::pop( void )
    {    return items[ top - - ] ;    }
Stack < int , 1000 > S1 ;    // Stack of 1000 integers
Stack < myString , 100 > S2 ;    // Stack of 100 myStrings
```

Inheritance

- Inheritance is the mechanism that is used to build a hierarchy of classes to perform useful work. Inheritance is different from *use* of a class to make objects.
- The class hierarchy is usually tree-like. At the root of the tree is the **base class** that defines the most general and least specific version of an interface.
- **Sub-classes** *inherit* an interface from a base-class. Inheritance means that the sub-class supports all of the data and function members provided by the base class.
- The sub-class may modify the inherited base-class interface in several ways
  - Add new member functions.
  - Add new data members.
  - Redefine member functions.
- Usually sub-classess (derived class) of the base class specialize the operations to implement a more specific instance of the base class

Class Hierarchy Example



Member Accessibility

- Each member of a class is either **private** , **public** , **protected** .
- **protected** member can be accessed only by methods within its class and within the derived class.
- All data members and methods of the base class, except for constructors, the destructor, and the overloaded assignment operator are included in the derived class.
- If a derived class adds a member with the same name as a member in the base class, the local member hides the inherited member.
- In general, inheritance can never *increase* the visibility of a member.

Inheritance

```
class B {       // base class
    ...
};
class D : access-specifier B {       // derived class
    ...
};
```

access-specifier is one of **public** , **protected** , or **private**

	public in B	**protected** in B	**private** in B
public	**public** in D	**protected** in D	**private** in D
protected	**protected** in D	**protected** in D	**private** in B
private	**private** in D	**private** in D	**private** in B

Inheritance Example

```
class B {
public :
    int x;
protected :
    int y;
private :
    int z;
};

class D : private B {
public :
    int z;       // hides B::z
    int w;
};
```

Constructor Under Inheritance

- Base class constructor handles for the “from the base class” part of the object and derived class constructor handles for the “added by the derived class” part of the object.
- Derived class constructor may invoke a base class constructor (if exists) explicitly.
- Let B be a base class and D be the derived class from B. When a D object is created one of the followings will occur.
  - If D has constructors but B has no constructors, then the appropriate D constructor will be used.
  - If D has no constructors but B has constructor, then B must have a default constructor, which will be used.
  - If D has constructors and B has a default constructor, then B’s default constructor will be used unless the D constructor explicitly invokes some other B constructor.
  - If D and B have constructors but B has no default constructor, then each D constructor has to invoke a B constructor explicitly.

Constructor Under Inheritance: Example

```
const int MaxLen = 100;
class B {       // base class
protected :
    char *name;
    int maxlen;
public :
    B() {       // B’s default constructor
        maxlen = MaxLen;
        name = new char[ maxlen ];
    }
};
class D : public B {       // derived class
public :
    // invoke B’s default constructor
    D( char *n ) : B() { strcpy( name, n ); }
};
D foo( ”foo” );
```

Polymorphism and Virtual Methods

- **Polymorphism** refers to the run-time binding of a pointer to a method.
  C++ supports polymorphism through **virtual** methods and pointers.
- A pointer to base class can point to a base class or to any derived class object without explicit casting.
  A pointer to a derived class object cannot point to a base class without explicit casting.
- For **virtual** methods with the same name, the system determines at run-time which of the methods to invoke.
  For non-virtual functions with the same name, the system determines at compiler-time which of the functions to invoke.
- **virtual** methods are declared with the keyword **virtual** .
  If a derived class redefines a **virtual** method, the redefined method must have exactly the same prototype as the base class method.

Virtual Methods Example

```
class B {
public :
    virtual void g();      // virtual method
    int h();
};
class D : public B {
    void g();      // virtual method
    int h();
};
main() {
    D d;
    B *ptr = &d;
    ptr->h();      // B::h invoked
    ptr->g();      // D::g invoked
}
```

Abstract Class and Pure Virtual Functions

- Abstract class is a base class which is required to have a derived class.
- Abstract class is not allowed to have objects that belong to it.
- Abstract class is specified by declaring a pure virtual function in the class's declaration.

```
class AC {      // abstract class
...
public :
    virtual void f( int ) = 0;      // pure virtual function
};
```

- The purpose of declaring a pure virtual function is to have derived classes inherit a function interface only.

Multiple Inheritance

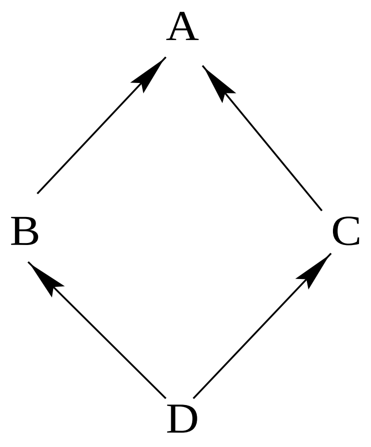
- In multiple inheritance, a derived class has multiple base classes.
- Derived class typically represents a combination of its base classes.

```
class iostream:   public istream, public ostream {
...
};
```

- Name conflicts are resolved using scope resolution operator.
- Derived class inherits multiple times from the same indirect base class. This problem can be avoided by using **virtual** base class.

Virtual Base Class Example

```
class A {    // base class
protected :
   int x;
   ...
};
class B: public virtual A {
   ...
};
class C: public virtual A {
   ...
};
class D: public B, public C {
   ...
};
```



Destructors Under Inheritance

- Derived class destructor is first exected and then the base class destructor is exected.
- Destructor must be **virtual** function whenver the following two conditions are met.
  - Program dynamically allocates a class object, e.g.,

    B *p = **new** D;
  - Constructor for the base and the derived class dynamically allocate separate storage.

Destructors Under Inheritance Example

```
class B {       // base class
public :
      B() { cout << "B's constructor\n"; }
      ~B() { cout << "B's destructor\n"; }
};
class D : public B {       // derived class
public :
      D() : B() { cout << "D's constructor\n"; }
      ~D() { cout << "D's destructor\n"; }
};
main() {
      D d;        // Printed output
                  // B's constructor
                  // D's constructor
                  // D's destructor
                  // B's destructor
}
```

C++ Exception Handling

- The exception handling mechanism in C++ provides a cleaner way to deal with error and exceptional conditions that arise during normal processing.
- *Use exception mechanism only for true exceptions not for general processing.*
- General mechanism:
  - User defined exception classes
  - **throw** statement to signal exception
  - **try** and **catch** to handle exceptions
- Exceptions can be defined as class objects.
- This is all relatively new to C++ and not widely implemented.

C++ throw Statement

throw *expression* ;

throw statement stops sequential execution and starts search for exception handler.

The *type* of the expression is used to determine the handler that is invoked.

The *value* of the expression is passed as a parameter to the handler that is invoked.

C++ try Statement

```
try {
    statements
}
catch ( .. ) { .. }
..
```

try statement associates a collection of catchers with a block of statements.

There may be multiple catchers, distinguished by their parameter lists.

Exception Handling Example

```
class MyString {
    char * S ;
public :
    enum { minSize = 1 , maxSize = 1000 } ;
    MyString() ;
    MyString( int ) ;
    ...
} ;
MyString::Mystring( int size )
{
    if ( size < minSize | | size > maxSize )
        throw ( size );
    s = new char [ size ];
    if ( s == NULL )
        throw ( "Out of Memory" );
}
void f( int N )
{
    try {
        MyString str( N );
    }
    catch ( char * errMsg ) {
        cerr << errMsg << endl;
        abort();
    }
    catch ( int k ) {
        cerr << "Out of range error: " << k << endl;
        f( string::maxSize );
    }
}
```

Namespaces

- Namespace is used to distinguish among identical global names, *e.g.,* two libraries may contain identical global names.
- To use namespace to resolve name conflicts, put global declarations in namespaces

```
namespace lib1 {
    void clr_screen();
    ...
}
namespace lib2 {
    void clr_screen();
    ...
}
```

- Namespace members can be referred to using scope resolution operator, `lib1::clr_screen()` or *using-declaration*:

```
using lib1::clr_screen;     // put lib1's clr_screen() into local namespace
using lib1;                 // make all the names in lib1 available
```