OO Programming and C++

- Introduction to OO
- C++ from C
  - Reference types
  - Memory management
- Classes, friends
- Operator Overloading
- Inheritance
- Some OO design
- Virtual functions and polymorphism
- (Time permitting) Templates
- (Time permitting) Exception handling

Readings: Chapters 6 and 7 of Sethi

Brief History of the OO World

- the OO paradigm is really a natural outgrowth of the modularity/ADT movement in software engineering (1960's & 70's)
- some concepts originated outside of the systems world, esp. in AI and DB work
- the first "real" OO language was Simula-67 (simulation language)
- then came Smalltalk, the most famous of all OO languages
- also: C++, Objective-C, Object-Pascal, Eiffel, CLOS, Modula-3, OOTuring
- languages supporting modules and encapsulation, but lacking full classes and instantiation are called <u>object-based</u>
  e.g. Turing proper, Ada

Object-Oriented Software Construction

In a nutshell:

> View the system in terms of the basic <u>entities</u>, their associated operations, and the interrelationships between entities.

Key idea: <u>encapsulation</u> ($\approx$ modularity++)

- What are the fundamental data objects in the system?
- What do they look like?
- How do they interact?
- What routines do I need?
- How should I implement them?
  Start from scratch?
  Can I use something I already have?

Advantages of OOSD

- recognizes and supports the iterative and recursive nature of software development
- boundaries between development stages are blurred
- localization of code also eases maintenance
  - easier to debug, understand
  - small change $\Rightarrow$ localized consequences.
- <u>reuse</u> possible:
  - reuse libraries ("holy grail" of OO)
  - cannibalization of old code
  - bend/extend/generalize existing routines
  - also: design reuse, analysis reuse, testing reuse, etc.

What is an OO Language?

Answer: One that has objects.

An object is similar to a module:

- has a hidden internal state, and exported routines
- can only alter internal parts indirectly, via a call to an exported routine.

Major difference:

- a module defines only one "item".
  Can't create multiple copies.
  $\rightsquigarrow$ idea of a class

C++ - Overview

- Classes
  - data hiding
  - guaranteed initialization and finalization
  - dynamic typing
  - user controlled memory management
  - overloaded operators
- Other advantages
  - Better type checking
  - Overloaded function names
  - free storage management
  - default function arguments
  - references as well as pointers

Reference Types

type & *identifier* ;

Reference are declared by prefixing an identifier with the *address of* operator &

References *must* be initialized and are thereafter *immutable*

Reference types are like pointers, but designed to work better with Objects
  automatic dereferencing
  reference behaves like an *alias*

References used primarily for function arguments and return values

References for Parameters

void F(**int** & A, **char** & B) ;

A and B are declared as *reference* parameters.
Compiler automatically generates address of actual parameter.
(Eliminates problems with forgotten & at point of call

Use reference parameters when:
- function must modify actual parameter
  e.g. to return more than one value
- to avoid passing large object by value

References for return values

const T & G(...) ;

The function G returns a reference to an object of type T.

The **const** declaration prevents accidental modification of value returned.

Use functions returning references:
- to avoid returning large objects by value
- when an address (*lvalue*) is required

Beware of:
- returning reference to local object
- returning lvalues unless you mean it

Reference returning functions can be used where lvalues are required

C++ new and delete

new *objectSpec*
delete *object*
delete [] *arrayObject*

---

new allocates storage for the object specified by *objectSpec*.
objectSpec can refer to a builtin type like int, or to a class or class template.
Dynamic size information can be provided for arrays and classes with parameterized initializers.
Parameters must be provided for class templates.

new returns the null pointer if no storage is available.
Well written programs check every pointer returned by **new** .

delete releases storage that was allocated by **new** .
The two forms are necessary to distinguish between arrays and pointers.

C++ new and delete Examples

```
class X {
    X(int size = 100) ;
    ...
}

X *x1p, x2p ;
int * z1, *z2[] ;

x1p = new X
x2p = new X(200);
z1  = new int ;
z2  = new int[300];

delete x1p ;
delete [] z2 ;
```


Classes, Objects and Instantiation

A <u>class</u> definition is really a template:

- can <u>instantiate</u> individual elements or <u>objects</u> with same structure as needed.

The real power of classes comes with reuse:

- can define new classes from old ones with minimal effort
- can extend/generalize/reshape existing code without destroying it

Structures in C

Structures in C++ are similar to structures in C - declaration begins with `struct` and consists of a sequence of declarations enclosed within braces:

```
struct complexx {
   int real;
   int imag;
}
```

Can define variables of this struct:

```
complexx n1, n2;
```


Structures in C++

Members of struct can be functions.

Special function *constructor*, has the same name, used to initialize object automatically upon creation.

```
struct complexy {
   complexy() {real = 0; imag = 0;}
   void printNum();
   int real;
   int imag;
}
```

Note that constructor has no specified type!

Body of function

- Can be included within the structure (*in-line*, expanded at point of use)
- Can be specified outside of the class definition. Then need name-resolution operation `::` which connects member to its structure.

```
void complexy::printNum() {
   /* body of printNum */
}
```

- Can be *virtual* - normally provided by inheritors of class `complexy` (later).

So, member functions have access to data and functions in structure.

```
complexy num1;
num.printNum()
```


Classes and Structures

Structures and classes are closely related. The following two are equivalent:

```
struct x { <member-declarations> };
class x {
  public:
     < member-declarations >
};
```

By default, all members of struct are public, all members of class are private.

So, the following two are equivalent:

```
class x { <member-declarations> };
struct x {
  private:
     < member-declarations >
};
```

Visibility of member names

- **public** - visible to code outside class definition.
- **private** - not visible outside class definition.
- **protected** - behave the same way as **private**, except in "derived classes" (later).

The same keywords work for structs. The only difference is what is true on default.

class className {
  public :
  // visible to all clients
  protected:
  // visible to *inheritors* of this class
  private :
  // visible only to member functions
}

Visibility of member names (cont'd)

Example:

```
class complex {
   public:
     complex() {real = 0; imag = 0;}
     void printNum();
   protected:
     int realPart() {return real;}
     int imagPart() {return imag;}
   private:
     int real, imag;
     int incrementNum(){return real+1;}
}
```


C++ Class Storage

- A **class** declaration defines the local data for a class object and a set of functions that operate on this data.
- Each class *object* has its own copy of the local data for the class.
- For reasons of space efficiency there should only be *one* copy of the functions for a class
- Each call of a function in the class is associated with a specific class object.
  The functions need to be able to access the local data for the particular class object associated with the call.
- The *this* pointer is the C++ solution to the problem.

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.

Class local data can be declared **static** if one copy is shared by all class objects.

C++ this Pointer Example

```
class thisDemo {
   public
      void munch( ... ) ;
   private
      int data1, data2 ;
}

void thisDemo::munch( ... ) {
    ...
    this->data1 = this-> data2 ;
    ...
}

thisDemo A  ;        // an object
thisDemo B[10] ;    // array of objects

A.munch( ... ) ;
B[5].munch( ... ) ;
```


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.
- Style - put **friend** declarations at head of class declaration.
- Class must explicitly grant access to each friend.

C++ friend declaration

class Grantee {
   friend class *friendClass* ;
   friend *type fFunct*(. . .);
   friend *type fClass*::*fFunct*(. . .); }

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

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

Use friend access only when absolutely necessary.

C++ Class Integration

- Want classes to integrate well with the rest of the language.
  Class objects should have all the characteristics of builtin data types.
- Achieve this through:
  - Careful initialization of class data.
  - Overloading operators to provide compatible operations on class objects
  - Providing type conversions to/from existing data types and classes.
  - Providing iterators to systematically process data in the object.
  - Overloading input and output operations for convenient IO.
  - Using **private** to protect class data.
  - Careful finalization of class objects.

C++ Overloading

- Overloading allows multiple definitions to be given for a function or an operator. The C++ compiler automatically selects the appropriate definition for each particular use.

- Overloading is used to provide common function names (e.g Copy, Move, Add, Print) that apply to many different kinds of objects

- Overloading of operators (e.g. =, ==, + [ ], etc) allows familiar operators to be used on new types of data objects defined by classes.

C++ Function Overloading

void F(**int** A, **char** B) ;
void F(**char** * D , **char** E) ;

For function overloading just declare different instances of the function *with the same name*

Compiler selects instance of F depending on use in call

Examples of Function Overloading

```
complex::complex() {
   real = 0; imag = 0;
}
complex::complex(int i) {
   real = i; imag = 0;
}
complex::complex(int i, int j) {
   real = i; imag = j;
}
```

These don't have to be member functions.

```
void readAll (int & i) {
  cout << "Enter a number\n";
  cin >> i;
}
void readAll (char & c) {
  cout << "Enter a character\n";
  cin >> c;
}
```


Operator Overloading

type operator*Op* (...)

---

type is the type of value returned by the operator. May need to be a reference to some class

op is an operator being overloaded. Can overload the following operators:

```
+  -  *  /  %  ^  &  |  ~
!  =  <  >  += -+ *= /= %=
<= >= && !! ++ -- ->*, etc.
```

This function takes parameters just like any other. Can have multiple overloads for same operator.

If first parameter (left operand) is not a member of the class, the operator should not be declared as a member function.

Operator Overloading Examples

Two version of addition for complex numbers:

```
complex complex::operator+ (complex y) {
   complex c;
   c.real = real + y.real;
   c.imag = imag + y.imag;
   return (c);
}
complex operator+ (complex x, complex y) {
   complex c;
   c.real = x.real + y.real;
   c.imag = x.imag + y.imag;
   return (c);
}
```

Call: `A = X + Y;`

Assignment Example:

```
complex& complex::complex operator= (const complex & from) {
         real = from.real;
         imag = from.imag;
    }
```


C++ Class Object Initialization

- A class object is initialized by initializing its data members. If all data members are public then this could be done using a C style initialization list.
- C++ supports a mechanism for automatic initialization of class objects.
- A special class member function called a *constructor* is automatically called whenever a class object is defined or allocated using **new** .
- A constructor is declared as a member function *with the same name as the class that contains it*
- The constructor is responsible for correctly initializing the class object
- Construction functions can be overloaded to allow for different initialization situations.

C++ Class Object Destruction

- May need to clean up internal state when a class object is about to be destroyed through scope exit or explicit **delete** .
- A special class member function called a *destructor* can be defined to deal with this situation.
- A destructor is declared as a member function *with the same name as the class that contains it* but *the name is preceded by a ˜*
- The destructor is responsible for correctly deinitializing the object.
- There can only be one destructor per class.

Constructor/Destructor Example

```
class myString {

  public

    // two constructors
    myString(int size = 255 )
       { len = size ; s = new char[size+1]};
    myString( char * si  )
       { len = strlen(si) ;
         s = new char[len+1];
         strncpy(s,si,len);
       };
   ~myString() { if(s) delete s };

 private
    int  len ;      // length
    char * s ;    // string

}
```

C++ Assignment

- The normal interpretation of assignment (=) is bit copy.
  This is *almost certainly wrong* if object contains pointers.
  - Memory leaks from storage that is allocated by never freed.
  - Program crashes from multiple deallocations of the same memory.
- Example

```
{
    myString a(10) ;
    myString b(20) ;
    ...
    a = b ;
    ...
}
```

- a.s is deleted twice and b.s never

C++ Assignment (Cont'd)

- Try to redefine assignment for the class myString

```
void operator=(myString& a){
   if( this == &a )
       return ;      //   X = X ;
   delete s ;        //   Clean up storage
   s = new char[size = a.len + 1] ;
   strcpy(s, a.s);
};
```

- But this isn't a complete solution

```
{
   myString a(10) ;
   myString b = a ;
}
```

This = is *not* the assignment operator. Wouldn't work anyway because b.s has never been allocated.

C++ Assignment (cont'd)

- Also need a separate initializer.

```
void myString( myString & a)
  {
    len = a.len
    s = new char[ len + 1 ];
    strcpy( s, a.s );
  }
```

which takes care of initialization by copying.

- C++ compiler can't distinguish between pointer to a single object and pointer to the first element of a vector.
  Therefore, to support vectors of objects, a class X *must* define a constructor:

```
X()
```

with *no* arguments, not even defaults.

C++ Assignment - Summary

- If the destructor for a class X does anything non-trivial i.e. deallocate storage, then you will need to define *ALL* of:

```
X( ... )         // initializer
X()              // initializer for vectors
X( X & )         // initialize by assignment
operator=( X & ) // assignment
                 // AND all op= combinations
~X()             // destructor
```

- Function call and return will use the initialize by assignment function for parameters and return values.
- All of this is necessary to keep a classes local data from getting contaminated and to prevent memory leaks.

Type Conversions

- Want to be able to mix abstract data types defined by classes with existing types.

- Example, adding a new data type complex, want to mix complex arithmetic with existing float and double data

- Could overload all arithmetic operators with all permutations of complex vs. float/double operands. e.g
  operator +(double,complex)
  operator +(complex,double)
  . . .
  This gets tedious very quickly.
  Better way is to define type conversions.

Type Conversion

Class::**operator** *Type*() { . . . }

- *Class* is any class, *Type* is any type
- This declaration defines a conversion operation from objects of type *Class* to objects of type *Type*.
- Conversion functions are usually inline unless they're large.
- Conversions must be unique to be legal
- Neither argument types nor return type may be specified.
- Conversion functions can be called implicitly or explicitly (coercion).

Type Conversion Example

```
class complex {
   public:
     ...
     int operator int();  // error!
     operator int(int);   // error!
     operator int() { return real; }
};

void f(complex num) {
   int i = (int)a;
   i = (int a);
   i = a;
}
```


Input/Output Overloading

Overloading input and out operators is a bit tricky. All input is done by class istream, and all output - by class ostream. You need to include <stream.h> to use them.

Idea: make operators << and >> friends of your class and then redefine them separately.

<< and >> already know how to input/output simple objects.

I/O Overloading, Example

```
#include <stream.h>

class complex {
   public:
     ....
     friend ostream& operator<<(ostream&, const complex &);
     friend istream& operator>>(istream&...);
   private:
     int real, imag;
}

ostream& operator<< (ostream& o, const complex& num) {
   return o << num.real << "+" << num.imag << "i";
}

istream& operator>> (istream& i, complex& num) {
   char c;
   return i >> num.real >> c >> num.imag >> c;
}
```


Inheritance

So far, we've seen classes and objects.

One of the main applications of OO programming is reuse.

How do we design new classes from old with minimal effort?

Answer: Inheritance.

Single Inheritance

Inheritance mechanisms define objects as extensions of previously defined objects. The extension is said to inherit the variables and operations (members) of the previous object.

Two main issues:

- Information hiding
- Inheritance as subtyping

A class D can be defined as an extension to B by mentioning only the changes from B.

(B - base class, D - derived class.)

Syntax:

`class` <*derived*> : `public` <*base*>

`class` <*derived*> : `private` <*base*>

Information Hiding

Type of inheritance determines visibility of class members.

Public inheritance does not change visibility of class members and private makes them private:

Data Items	Type of Inheritance	
	Public	Private
public	public	private
private	–	–
protected	protected	private

"–" - does not inherit.

Inheritance as Subtyping

- each child class inherits all of the <u>attributes</u> (types, constants, variables, procs/funcs) of its parents
  - also holds recursively for descendants (inheritance is transitive)
- each descendant may add new attributes or redefine old ones
- some languages support <u>multiple inheritance</u>: a class can inherit (directly) from multiple parent classes
  Multiple inheritance is very tricky. Avoid it if possible.

Example of Inheritance

```
class base {
   public:
     int a, b;
}
class derived : public base {
   public:
     int b, c;
}
void f() {
   derived d;
   d.a = 1;
   d.base::b = 2;  // get redefined member
                   // of base class
   d.b = 3;
   d.c = 4;
   base* bp = &d;  // standard conversion:
                   // derived* to base*
}
```


Another Example

```
class A {public: void f(); };
class B : public A {};
class C : public B { public: void f(); };

void C::f() {
   f();      // Call C's f()
   A::f()    // Call A's f()
   B::f();   // still call A's f() since
             // that's the only f() in B
```


Yet another example

```
// File Inherit.h++

class Base_Class {
   public:
     int age;
     Base_Class(); // constructor
};

class Derived_Class : public Base_Class {
   public:
     int generation;
     Derived_Class();  // constructor
};
```

Output of Example

```
We are in main().

Creating a Base_Class object.
First, it has a random value for age -268436428.
Now, age is 35.

Meanwhile, back in main() ...
Creating a Base_Class object.
First, it has a random value for age 3.
Now, age is 35.
Creating a Derived_Class object.
Age is 35 whereas generation is random: -268436436
Now, age is 7.
... and generation is 1.
------------
mom.age is 35
child.age is 7
and child.generation is 1
```


Relationships between classes

Most important - *is a* and *has a*.

Is a is an inheritance. An object of a derived class type may also be treated as an object of the base-class type.

Has a is a composition. In this relationship, a class object has one or more objects of other classes as members.

Inherit.h++ (Cont'd)

```
Base_Class::Base_Class() {
  cout << "Creating a Base_Class object." << endl;
  cout << "First, it has a random value for age "
       << age << "." << endl;
  age = 35;
  cout << "Now, age is " << age << "." << endl;
}
Derived_Class::DerivedClass() {
  cout << "Creating a Derived_Class object.";
  cout <<  << endl << "Age is " << age << ;
  cout << " whereas generation is random: "
       << generation << endl;
  age = 7;
  generation = 1;
  cout << "Now, age is " << age << "." << endl;
  cout << "... and generation is " << generation
       << endl;
```


Main program for Example

```
#include <iostream.h>
#include "Inherit.h++"
main() {
  cout << "We are in main()." << endl << endl;
  Base_Class mom;
  cout << endl;
  cout << "Meanwhile, back in main() ..." << endl;
  Derived_Class child;
  cout << "---------------" << endl;
  cout << "mom.age is " << mom.age << endl;
  cout << "child.age is " << child.age << endl;
  cout << "and child.generation is "
       << child.generation << endl;
```

Examples of *Is A*

GraduateStudent UndergraduateStudent	*is a*	Student
Circle Triangle Rectangle	*is a*	Shape
FacultyMember StaffMember	*is a*	Employee

Examples of *Has A*

Has A - one class may have other classes as members, i.e. can create new classes as a composition of existing classes.

Example. Given classes `Employee`, `Birthdate`, `TelephoneNumber`.
`Employee` *is a* `Birthday`? NOT!
`Employee` *is a* `TelephoneNumber`? NOT!

But...
`Employee` *has a* `Birthday`
`Employee` *has a* `TelephoneNumber`

Use composition:

```
class Employee {
    Birthday birthday;
    TelephoneNumber number;
    String name;
};
```


Other Relationships - *Has Many*

"Many" - array or linked list. ALWAYS!
So, *has many* - composition with pointer to class.

Example:
`Card` *has many* `Notes`.
Create a class `Note`. Create a class `Notes` to store linked list of `Notes`. Include a pointer to `Notes` in definition of `Card`.

```
class Note {
    ....
};
class Notes {
   ...
   private:
      Note data;
      Notes* next;
};
class Card {
   ...
   private:
      Notes* notes;
};
```


Other Relationships, Cont'd

Want to reuse as much as possible.

Classes `Person` and `Car`:
`Person` *is a* `Car`? NOT! `Person` object contains a `Car` object? NOT! `Person` *uses a* `Car`.

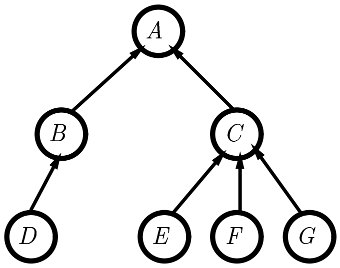
A function uses an object simply by issuing a function call to a member function of that object.

An object can also be *aware of* another object (call this *knows of*). Knowledge networks frequently have such relationships. One object can contain a pointer or a reference to another object to be aware of that object.

Often, an object A *knows of* object B indicates that A is `friend` of B.

Pointers and Inheritance

Inheritance defines a tree-like hierarchy on classes (actually a partial order)




```
A  *p ;
C  *q ;
```

Can instantiate `p` to point to an object of class `A` or any descendant of `A`.

Can instantiate `q` to point to an object of class `C` or any descendant of `C`, **but** `q` may not be instantiated to objects of classes `A`, `B` or `D`.

Static vs. Dynamic Types

```
A  *p ;
```

The declaration sets the static type of a pointer. The static type of a pointer remains unchanged throughout the life of the program. In this case, `p` has static type `pointer to A`.

```
p =   new D ;
```

The run-time bind sets the dynamic type of a pointer. The dynamic type is reset every time a `new` is done. An object/pointer may have several different dynamic types throughout its lifespan, although only one at a time. The dynamic type must be a descendant of the static type. In the above example, `p` is given the dynamic type `pointer to D`.

C++ Virtual Functions

- Allows to call a variety of functions using exactly the same interface.
- A mechanism for adding flexibility and generality to classes while providing abstraction and information hiding.
- Base (parent) class provides functions implementing its abstraction that are actually implemented by inheritors of the base class.
- Normal case is that inheritors *always* override the definition of virtual functions provided by the base class.

C++ Virtual Functions (cont'd)

- Base class can provide fall-back implementation of virtual function or explicitly declare it to be undefined. Undefined virtual functions are *Pure Virtual Functions*
- Classes containing pure virtual functions can never be used directly to create objects.

Virtual Function Example - easy

```
class base {
    public:
       virtual void doIt( int &, float )
          { cout << "base.doIt is useless" ; };
    protected:
       virtual int toIt( char * ) = 0 ;

    private:
       ...
};

class baseChild : public base {
    public:
       virtual void doIt( int &, float );
       int toIt( char * );
       virtual char * allDoneDoingIt() ;
    private:
       ...
};

class baseGrandChild : public baseChild {
    public:
        char * allDoneDoingIt();
    private:
        ...
};
```


Virtual Function Example - harder

```
struct base {
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
  void f();
}
class derived : public base {
   public:
     void vf1();
     void vf2(int); // hides base:vf2()
     void vf3();    // error: differs in return type only
     void f();
}
void g() {
   derived d;
   base* bp = &d;   // standard conversion:
                    // derived* to base*
   bp->vf1();       // calls derived:vf1
   bp->vf2();       // calls base:vf2
   bp->f();         // calls base::f
}
```


Lessons to Learn

- Interpretation of the call of a virtual function depends on the type of the object for which it is called

- Interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object.

This is the key feature of OO programming!

Now that we got your attention

- Base class defines an interface for which a variety of implementations are provided by derived classes

- A pointer to an object of a class can be passed into a context where the interface defined by one of its base classes is known but where the derived class is unknown.

- Virtual function mechanism ensures that the object is still manipulated by the functions defined for it (and not just by the functions defined for the base class).

- Only functions specified in the interface provided by the base class will be called and only if the actual arguments match the formal arguments required.

Polymorphism and Overloading

Polymorphism - the ability to call a variety of functions using exactly the same interface (types of parameters and return) - as it provided by virtual functions

Overloading - the ability to call a variety of functions with different interface. Can occur in non-OO languages since it does *not* depend on inheritance.

So, polymorphism is overloading++.

Case Study

Inheritance hierarchy: Points, Circles and Cylinders.

Circles inherit from points, and Cylinders inherit from Circles.

Goals:

- To study inheritance
- To study virtual functions
- To look at a "real" C++ application

Code is at

`http://www.cs.toronto.edu/~.../csc324/examples`

Organization of code

Files:

`shape.h`	-	definition of abstract base class Shape
`point.h`	-	definition of class Point
`point.C`	-	member function definitions for class Point
`circle1.h`	-	definition of class Circle
`circle1.C`	-	member function definitions for class Circle
`cylinder1.h`	-	definitions of class Cylinder
`cylinder1.C`	-	member function definitions for class Cylinder
`mainshapes.C`	-	driver for this case study

Base class Shape

```
// shape.h
// Definition of abstract base class Shape

class Shape {
public:
  virtual float area() const {return 0.0; }
  virtual float volume() const {return 0.0; }
  virtual void printShapeName() const = 0;
  // pure virtual
  virtual void print() const = 0;
  // pure virtual
};
```

Definition of class Point

```
// point1.H
#include <iostream.h>
#include "shape.h"

class Point : public Shape {
  friend ostream &operator<<(ostream &,
                             const Point &);
public:
// default constructor
  Point(float = 0, float = 0);
  void setPoint(float, float);
  float getX() const { return x; }
  float getY() const { return y; }
  virtual void printShapeName() const
{ cout << "Point: "; }
  virtual void print() const;
private:
  float x, y; // x and y coordinates of Point
};
```


Member functions for Point

```
// POINT1.C
#include <iostream.h>
#include "point1.h"

Point::Point(float a, float b)
{ setPoint(a, b); }

void Point:setPoint(float a, float b)
{ x = a; y = b;}

void Point::print() const
{ cout << '[' << x << ", " << y << ']';}

ostream &operator<<(ostream &output,
                    const Point &p)
{
  p.print();  // class print to output the object
  return output; // enables concatenated calls
}
```


Definition of class Circle

```
// CIRCLE.H
#include "point1.h"

class Circle : public Point {
  friend ostream &operator<<
          (ostream &, const Circle &);
public:
  // default constructor
  Circle(float r = 0.0, float x = 0.0,
         float y = 0.0);
  void setRedius(float);
  float getRadius() const;
  virtual float area() const;
  virtual void printShapeName() const
  { cout << "Circle: "; }
  virtual void print() const;
private:
  float radius;  // radius of Circle
};
```


Member functions for Circle

```
// Circle1.C
#include <iostream.h>
#include "circle1.h"

Circle::Circle(float r, float a, float b)
  : Point(a, b) // call base-class constructor
{ radius = r; }

void Circle::SetRadius(float r)
{ radius = r; }

float Circle::getRadius() const
{ return radius; }

float Circle::area() const
{ return 3.1415 * radius * radius; }
```

Member functions for Circle (cont'd)

```
void Circle::print() const
{
  cout << '[' << getX() << ", " << getY()
    << "]; Radius = " << radius;
}

ostream &operator<<
        (ostream &output, const Circle & c)
{
  c.print();  // call print to output the object
  return output; // enable concatenated calls
}
```


Definition of class Cylinder

```
// CYLINDR1.H
#include "circle1.h"

class Cylinder : public Circle {
  friend ostream &operator<<
            (ostream & const Cylinder &);
public:
  // default constructor
  Cylinder(float h = 0.0, float r = 0.0,
           float x = 0.0, float y = 0.0);
  void setHeight(float);
  virtual float area() const;
  virtual float volume() const;
  virtual void printShapeName() const
  { cout << "Cylinder: "; }
  virtual void print() const;
private:
  float height; // height of Cylinder
};
```


Member and Friend function definitions for Cylinder

```
// CYLINDR1.C
#include <iostream.h>
#include "cylindr1.h"

Cylinder::Cylinder(float h, float r,
          float x, float y) : Circle(r, x, y)
// call base-class constructor
{ height = h; }

void Cylinder:: setHeight(float h) {height = h;}

float Cylinder:: area() const
{
  // surface area of Cylinder
  return 2 * Circle::area() +
    2 * 3.1415 * Circle::getRadius() * height;
}
```


Member and Friend function definitions for Cylinder (cont'd)

```
float Cylinder::volume() const
{ return Circle::area() * height; }

void Cylinder::print() const
{
  Circle::print();
  cout << "; Height = " << height;
}

ostream & operator<<
    (ostream &output, const Cylinder& c)
{
  c.print(); // call print to output the object
  return output; // enables concatenated calls
}
```

Driver File

```
//MAINSHAPES.C
#include <ostream.h>
#include "shape.h"
#include "point1.h"
#include "circle1.h"
#include "cylindr1.h"

main()
{
  Point point(7,11);          // Create a Point
  Circle circle(3.5, 22, 8);  // Create a Circle
  // Create a Cylinder
  Cylinder cylinder(10, 3.3, 10, 10);
  point.printShapeName();  // static binding
  cout << point << endl;
  circle.printShapeName(); // static binding
  cout << circle << end;;
  cylinder.printShapeName(); // static binding
  cout << cylinder << "\n\n";
```


Driver File (cont'd)

```
  // array of base-class pointers
  Shape *shapeArray[3];

  // aim array members at derived-class objects
  shapeArray[0] = &point;
  shapeArray[1] = &circle;
  shapeArray[2] = &cylinder;

  // Loop through shapeArray and print the
  // shape name, area, and volume of each object
  // pointed by array, using dynamic binding.
  for (int i = 0; i < 3; i++) {
    shapeArray[i]->printShapeName();
    cout << endl;
    shapeArray[i]->print();
    cout << "\nArea = " << shapeArray[i]->area()
      << "\nVolume = " << shapeArray[i]->volume()
      << endl << endl;
  }
```


Output

```
Point: [7, 11]
Circle: [22, 8]; Radius = 3.5
Cylinder: [10, 10];
Radius = 33; Height = 10

Point:
[7, 11]
Area = 0.0
Volume = 0.0

Circle:
[22.0, 8.0]; Radius=3.5
Area = 38.48
Volume = 0.0

Cylinder:
[10.0, 10.0]; Radius=3.3; Height=10.0
Area = 275.77
Volume = 342.12
```


Templates

Reuse (again)!

E.g., can write a single *function template* for an array sort function and then have C++ automatically generate separate template functions that will sort an `int` array, sort a `float` array, sort an array of `char *` pointers, etc.

What can you do in C?

- *Function templates* - like macros. But has full C++ type checking.
- *Class templates*

Definition of Template Functions

`template`$<$*formal parameters*$>$
Each formal parameter proceeded with `class`.

Examples:

```
template<class T>...
template<class BorderType, class FillType>...

template<class T>
void printArray (T *array, const int count)
{
  for (int i = 0; i < count; i++)
    cout << array[i] << ' ';

  cout << endl;
}
```


Use of Template Functions

Idea - it has to be seamless.

```
main() {
  const int aCount = 7, bCount = 5, cCount = 6;
  int a[aCount] = {1, 2, 3, 4, 5, 6, 7};
  float b[bCount] = {1.1, 2.2, 3.3, 4.4, 5.5};
  char c[cCount] = "HELLO"; // 6th pos. for null
  printArray (a, aCount);   // integer template
  printArray (b, bCount);   // float template
  printArray (c, cCount);   // character template
}
```

Output:

```
1 2 3 4 5 6 7
1.1 2.2 3.3 4.4 5.5
H E L L O
```

The related functions generated from a function template all have the same name, so the computer uses overloading resolution to invoke the proper function.

Class templates

Allow software reusability by enabling type-specific versions of generic classes to be instantiated.

Example - stack. Can specify a template class and then instantiate for stack of integers, stack of characters, etc.

Another example - binary search tree. Can instantiate it for BST of integers, characters, floats, etc.

Case-Study - BSTs using Templates

Classes:

- `treenode` - contains data, left and right pointers

- `tree` - the tree itself.

Relationship between them?

"Has a" - tree has a pointer to tree node.

TreeNode

```
// TREENODE.H Definition of class TreeNode
#ifndef TREENODE_H
#define TREENODE_H

template<class NODETYPE>
class TreeNode {
   friend class Tree<NODETYPE>;
public:
   TreeNode(const NODETYPE &); // constructor
   NODETYPE getData() const;   // return data
private:
   TreeNode *leftPtr; // pointer to left subtree
   NODETYPE data;
   TreeNode *rightPtr; // pointer to right subtree
};
```


TreeNode - Cont'd

```
// Constructor
template<class NODETYPE>
TreeNode<NODETYPE>::TreeNode(const NODETYPE &d)
{
  data = d;
  leftPtr = rightPtr = 0;
}
// Return a copy of the data value
template<class NODETYPE>
NODETYPE TreeNode<NODETYPE>::getData()
const
{return data; }
#endif
```


Definition of template class Tree

```
#ifndef TREE_H
#define TREE_H
#include <iostream.h>
#include <assert.h>
#include "treenode.h"

template<class NODETYPE>
class Tree {
public:
  Tree();
  void insertNode(const NODETYPE &);
  void preOrderTraversal() const;
  void inOrderTraversal() const;
  void postOrderTraversal() const;
private:
  TreeNode<NODETYPE> *rootPtr;
```


Def. of Tree - Cont'd

```
  // utility functions
  void insertNodeHelp(TreeNode<NODETYPE> **,
                      const NODETYPE &);
  void preOrderHelp(TreeNode<NODETYPE> *) const;
  void inOrderHelp(TreeNode<NODETYPE> *) const;
  void postOrderHelp(TreeNode<NODETYPE> *) const;
};

template<class NODETYPE>
Tree<NODETYPE>::Tree() {rootPtr = 0;}

template<class NODETYPE>
void Tree<NODETYPE>::insertNode
(const NODETYPE &value)
{  insertNodeHelp(&rootPtr, value); }
```

```
template<class NODETYPE>
void Tree<NODETYPE>::
insertNodeHelp(TreeNode<NODETYPE> **ptt,
               const NODETYPE &value)
{
  if (*ptr == 0) {   // Tree is empty
    *ptr = new TreeNode<NODETYPE>(value);
    assert(*ptr != 0);
  }
  else  {            // Tree is not empty
    if (value < (*ptr)->data)
      insertNodeHelp (&((*ptr)->leftPtr), value);
    else {
      if (value > (*ptr)->data)
        insertNodeHelp(&((*ptr)->rightPtr), value)
      else
        cout << value << " dup" << endl;
    }
  }
}
```



```
template<class NODETYPE>
void Tree<NODETYPE>::preOrderTraversal() const
{
  preOrderHelp(rootPtr);
}

template<class NODETYPE>
void Tree<NODETYPE>::
preOrderHelp(TreeNode<NODETYPE> *ptr) const
{
  if (ptr != 0) {
    cout << ptr->data << ' ';
    preOrderHelp(ptr->leftPtr);
    preOrderHelp(ptr->rightPtr);
  }
}
```



```
template<class NODETYPE>
void Tree<NODETYPE>::inOrderTraversal() const
{
  inOrderHelp(rootPtr);
}

template<class NODETYPE>
void Tree<NODETYPE>::
inOrderHelp(TreeNode<NODETYPE> *ptr) const
{
  if (ptr != 0) {
    inOrderHelp(ptr->leftPtr);
    cout << ptr->data << ' ';
    inOrderHelp(ptr->rightPtr);
  }
}
```



```
template<class NODETYPE>
void Tree<NODETYPE>::postOrderTraversal() const
{
  postOrderHelp(rootPtr);
}

template<class NODETYPE>
void Tree<NODETYPE>::
postOrderHelp(TreeNode<NODETYPE> *ptr)
const
{
  if (ptr != 0) {
    postOrderHelp(ptr->leftPtr);
    postOrderHelp(ptr->rightPtr);
    cout << ptr->data << ' ';
  }
}
#endif
```

Driver to test class Tree

```
#include <iostream.h>
#include "tree.h"

main()
{
  Tree<int> intTree; int intVal;

  cout << "Enter 10 integer values:" << endl;
  for (int i = 0; i < 10; i++) {
    cin >> intVal;
    intTree.insertNode(intVal);
  }
  cout << endl << "Preorder traversal" << endl;
  intTree.preOrderTraversal();

  cout << endl << "Inorder traversal" << endl;
  intTree.inOrderTraversal();
```



```
  cout << endl << "Postorder traversal" << endl;
  intTree.postOrderTraversal();

  Tree<char> charTree; char charVal;

  cout << "Enter 10 character values: " << endl;
  for (int i = 0; i < 10; i++) {
    cin >> charVal;
    charTree.insertNode(charVal);
  }
  cout << endl << "Preorder traversal" << endl;
  charTree.preOrderTraversal();

  cout << endl << "Inorder traversal" << endl;
  charTree.inOrderTraversal();

  cout << endl << "Postorder traversal" << endl;
  charTree.postOrderTraversal();
  return 0;
}
```


Output

```
Enter 10 integer values:
3 6 2 7 10 0 4 5 15 2
2 dup
Preorder traversal
3 2 0 6 4 5 7 10 15
Inorder traversal
0 2 3 4 5 6 7 10 15
Postorder traversal
0 2 5 4 15 10 7 6 3

Enter 10 character values:
valuesofchar
Preorder traversal
v a l e c f h u s o
Inorder traversal
a c e f h l o s u v
Postorder traversal
c h f e o s u l a v
```


C++ Exception Handling

- 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 *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.

Example

```
//checking for a divide-by-zero error.
#include <iostream.h>

class DivideByZeroError {
public:
  DivideByZeroError() :
         message("Divide by zero") { }
  void printMessage() const { cout << message; }
private:
   const char *message;
};

float quotient(int num1, int num2);
{
  if (num2 == 0)
    throw DivideByZeroError();
  return (float) num1/num2;
}
```


Example - Driver Program

```
main() {
  cout << "Enter two integers: ";
  int number1, number2;
  cin >> number1 >> number2;
  try {
  // wraps the code that may throw an exception
     float result = quotient(number1, number2);
     cout << "Quotient is: " << result << endl;
  }
  catch (DivideByZeroError error) {
     // error handler
     cout << "ERROR: ";
     error.printMessage();
     cout << endl;
     return 1;  // terminate because of error
  }
  return 0;  // terminate normally
}
```

Output

```
Enter two integers: 7 3
Quotient is 2.333333

Enter two integers: 23 0
ERROR: Divide by zero
```


OO Computing in Review

OO Software Development:

- a more realistic model of software development
- more flexible, easier to deal with large, ongoing projects
- techniques can be applied at many levels
- reuse possible

OO Programming benefits include:

- modularity/encapsulation + instantiation
- inheritance and code reuse
- polymorphism

ood for slide eater

ood for slide eater