G22.2210

Programming Languages: PL

B. Mishra
New York University.

Lecture # 10

—Slide 1—

Key Concepts of OOPS

- **Abstraction:** Interfaces abstract away from implementation details.
- **Encapsulation:** Ccontrolled access to the internal states. Consistent states and consistent state transition.
- **Modularity:** Each object has minimal dependence on other objects. (Inheritance, Friends, Public Access, Import/Export)
- **Reuse:** Code reuse through inheritance and polymorphism. Minimal Code modification.

—Slide 2—

Key Concepts of OOPS, Contd.

- **Invariants:** Universal properties that hold of an instance at all the time (except inside the class's method).
- **Mutability:** Allows one to design immutable objects (Java `final`, e.g., `Strings`) that cannot be changed once it is created.

—Slide 3—

Classes of objects in `C++`

- **Linked List Example**
- Dynamic Allocation and Deallocation (Involving Objects and Pointers)
- Features of this example:
  1. A **constructor** is called automatically when an object is created.
  2. Overloading of function names
  3. `new`: Creating objects
  4. `delete`: Deallocating objects
  5. **Friend**: A friendly class that is given the access to one's private members.

—Slide 4—

Pointers in C

- Pointer to `class cell`

```
cell *p;// p = pointer to an object of type cell

class cell{        +------+--+
   int info;       | info |--+-----> next
   cell *next      +------+--+
}
```

- **Note**

```
p -> info    === (*p).info
// The info field of the cell pointed to by p
cell c, d;      +----+--+      +----+--+
c.next = &d;    | c  | -+----->| d  |  |
                +----+--+      +----+--+
```

- **Further Note** `0` = Null pointer in `C++`.

  In the body of the object `this` (a special name) denotes a pointer to the object itself.

—Slide 5—

CONSTRUCTOR

```
class cell{
  cell(int i){info = i; next = this;}
  cell(int i, cell *n){info = i; next = n;}

 int info;
 cell *next;
}
```

- **Constructors can be overloaded:**

```
cell d(1, 0);                cell a(3);
```

- Allocation: **Operator `new` dynamically constructs an object.** "`new cell(1,0)`" creates an anonymous object of class `cell` and initializes by passing `(1,0)` to its constructor.

```
cell *front;
front = new cell(1,0);  front = new cell(2, front);
```

Creates a *singly-linked* list of length two pointed to by `front`.

—Slide 6—

Deallocation

- Operator `Delete` explicitly deallocates a previously allocated object.

```
cell *temp = front;

front = front -> next;
delete temp;
```


—Slide 7—

FRIENDS

- Recall: The members of a class are *private* unless they are explicitly declared to be public.
- However, a *friend declaration* within a class gives nonmember functions access to the private members of the class.
- **Example:**

```
class cell{
   friend class circlist;
   cell(int i){info = i; next = this;}
   ....
  int info;
  cell *next;
}
```

1. All the members of the class `cell` are private (by default).
2. They are accessible only to its *friend* `class circlist`.

—Slide 8—

circlist

- Built on top of `cell`, a friend.

```
class circlist{
   cell *rear;
public:
   circlist() {rear = new cell(0);}
   // Access the constructor
   // (private member) in class cell

   boolean empty(){
     return (boolean)(rear == rear -> next);}
   void push(int);
   int pop();
   void enter(int);
}
```

1. `push(int)` Adds a cell to the front of the list.
2. `enter(int)` Adds a cell to the rear of the list.
3. `pop()` Deletes a cell from the front of the list and returns its value.

—Slide 9—

Body of the class circlist

```
void circlist::push(int x){
   rear->next = new cell(x, rear->next);
}

void circlist::enter(int x){
   rear->info = x;
   rear = rear->next = new cell(0,rear->next);
}

int circlist::pop(){
   if(empty()) return 0;
   cell *front = rear->next;
   rear->next = front->next;
   int x = front->info;
   delete front;
   return x;
}
```


—Slide 10—

Nested Classes

Classes can be nested. However, because of `C++` scope rules it leads to *confusion*. **Poor style**.

- **Example**

```
char c;              //c in external scope
class X{
  char c;            //c in internal scope
  class Y{
    char d;
    void foo(char e){c = e;}
  };
  char baz(Y* q){return (q->d);}
  //Syntax error, d = private
}
```

- **Note:**
  Inner and outer classes have the same scope: class `X` and `Y` are at the same lexical level.

  ⇒ `c` in function `foo` refers to `c` in external scope.

  ⇒ `q->d` in function `baz` is attempting to access a private member of class `Y`.

—Slide 11—

Derived Classes

- **Inheritance Mechanism**
  Base Class, `B` ⇒ Derived Class, `D`

- `D` derives its variables and operations, by suitably modifying the properties of `B`. Declaration for `D` needs to mention only the changes that must be made to `B`.
- Example:
  Consider the base class `circlist` with members:

```
push
pop
enter
empty
```

One can easily obtain the derived classes `queue` and `stack` by suitably restricting the operations:

```
queue{                stack{
   enter                 push
   pop                   pop
   empty                 empty
}                     }
```

—Slide 12—

Access Control Mechanisms

- *Public*:
  Member is visible throughout its scope.
- *Private*:
  Member is visible to other members within its own class, only.
- *Protected*:
  Member is visible to other members within its own class and any class immediately derived from it.

—Slide 13—

Public & Private Bases Classes

- **Public Base Class** if its derived class maintains the visibility of all inherited members:

```
class <derived>: public <base>{
 <member-declarations> //visibility is kept
}
```

- **Private Base Class** if its derived class hides the visibility of all inherited members:

```
class <derived>: private <base>{
 <member-declarations> //visibility is lost
}
```

- **Note**

```
class b{                          class d: private b{
public:                           protected:
  int f;          ==>               int b::g;
  int g;                          public:
}                                   int b::f;
                                  }
```


—Slide 14—

Example

- `circlist` Revisited

```
class circlist{
public:
//visible outside

  boolean empty();

protected:
//visible to members of derived classes

       circlist();
  void push(int);
  int  pop();
  void enter(int)

private:
  cell *rear;
};
```


—Slide 15—

Derived Class `queue`

- `queue` example

```
class queue: private circlist{
public:
       queue(){}
  void enter(int x){circlist::enter(x);}
  int  exit(){return pop();}
       circlist::empty;
}
```

- **Note**: `enter` is overloaded. Full name has to be used.
- Following are **private** to queue: Inherited functions: `push`, `pop`, `enter`. Inherited variable `rear`. `rear` is available only to the inherited function.

—Slide 16—

Derived Class `stack`

- `stack` example

```
class stack: private circlist{
public:
       stack(){}
  void push(int x){circlist::push(x);}
  int  pop(){return circlist::pop();}
       circlist::empty;
}
```

- **Note**: `push` and `pop` are overloaded. Full names have to be used.
- Following are **private** to queue: Inherited functions: `push`, `pop`, `enter`. Inherited variable `rear`: Available only to the inherited functions.

—Slide 17—

Usage Example

```
main(){
   stack s;
   queue q;

   s.push(1);
   s.push(2);
   s.pop;

   q.enter(4);
   q.exit();
   q.enter(5);

        .
        .
        .
}
```

- **Note:** Members in the derived class cannot see the private members of its base class.

—Slide 18—

Virtual Functions

- Allows Object-Oriented Programming Style (OOPS) in `C++`
- **Basic idea**:

```
\\BASE CLASS          \\DERIVED CLASS
virtual fn()           ... fn()
  ...
... A(){              \\ inherits A, But A's
    fn;               \\ body uses the fn,
  }                   \\ defined here.
```

Suppose also that the virtual function `fn` is used in another member `F` of Base class.

Now a derived class that inherits `F`, gets an inherited instance of `F` that *normally* uses the same instance of `fn` (i.e., the one in the `BASE CLASS`) independent of whether `fn` is redefined in the Derived Class or not.

- But, in the case when `fn` is virtual, the rule is only to "use the virtual function body only as a default."

—Slide 19—

Example

- Example of a virtual Function

```
class Base{
public:
virtual char f(){return 'B';}
        char g(){return 'B';}
        void testF{cout << f() << "\n";}
        void testG{cout << g() << "\n";}
}

class Derive: public Base{
public:
  char f(){return 'D';}
  char g(){return 'D';}
}
```

—Slide 20—

Example (contd)

- Virtual Function

```
Base b; Derive d;

b.testF;          //=> B
b.testG;          //=> B
d.testF;          //=> D
d.testG;          //=> B
```

- **Remark on `d.testF`:**
  `testF` is inherited by `d`.

  When `testF` calls `f`—Since `f` is *virtual* in the `Base`, the body of `f` in `Derive` has to be used.

—Slide 21—

Usage: Virtual Functions

- `shape` ⇒ `circle` & `square`

```
class shape{
  point center; ...
public:
  void   move(point to){center = to; draw();}
  virtual void draw();
  virtual void rotate(); ...
}

class circle: public shape{
  int radius;
public:
  void draw();
  void rotate(){}; ...
}

class square: public shape{
  int side;
public:
  void draw(); ...
}
```

The `draw` used by different `shape`s (e.g., in `move`) is different.

—Slide 22—

Subtypes & Supertypes

- `S` = Subtype of `T` (`T` = Supertype of `S`), if any `S`-object (object of type `S`) is at the same time a `T`-object (object of type `T`).

  ⇒ Any operation that can be applied to a `T`-object can also be applied to an `S`-object.

```
Shapes
 => Polygons
   ==> Squares
 => Circles
```

- **Subtype Principle**: *An object of subtype can appear whenever an object of a supertype is expected.*

```
class S: public T{
   ...
}
```

- `S` can appear wherever public base class `T` is expected.

—Slide 23—

Parametric Polymorphism: `TEMPLATE`

- `Template` in `C++` allows the same code to be used with respect to *different types* where *the type is a parameter of the code body.*

```
template <class TYPE>
class stack{
public:
 stack():max_len(1000), top(EMPTY)
        {s = new TYPE[1000];}
 stack(int size):max_len(size), top(EMPTY)
        {s = new TYPE[size];}
 ~stack(){delete []s;}

 void push(TYPE c){s[++top] = c;}
 TYPE pop(){return (s[top--]);}
 TYPE top_of() const{return (s[top]);}
 boolean empty() const{return boolean(top==EMPTY);}
 boolean full() const{return boolean(top==max_len-1);}
private:
 enum {EMPTY = -1};
 TYPE* s;
 int max_len;
 int top;
}
```

—Slide 24—

Template Instantiation

- `reverse`

```
stack<char> stk_ch;
//1000 elements char stack

stack<char*> stk_str(200);
//200 element string stack

//Reversing a sequence of strings
void reverse(char * str[], int(n){
  stack<char*> stk(n);

  for(int i=0; i<n; i++)
    stk.push(str[i]);
  for(i=0; i<n; i++)
    str[i] = stk.pop();
}
```


—Slide 25—

Function Templates

- `copy`

```
template<class TYPE>
void copy(TYPE a[], TYPE b[], int n){
  for(int i = 0; i<n; i++)
    a[i] = b[i];
}

double f1[50], f2[50];
copy(f1, f2, 50);
```

- With two distinct class template arguments:

```
template <class T1, class T2>
boolean coerce(T1& x, T2& y){
 if(boolean b = (sizeof(x) >= sizeof(y)))
   x = (T1)y;
 return b;
}
```


—Last Slide—

Inheritance

- *Parameterized types can be reused through inheritance.*

```
class safe_char_stack: public stack<char>{
public:
  void push(char c){assert(!full());
                    stack<char>::push(c);}
  char pop(){assert(!empty());
             return(stack<char>::pop());}
};
```

- Other Template Arguments:
  Constant Expressions, Function Names, Strings,...

```
template<int n, class T>
class declare_array{
public: T a[n];
};

declare_array<50,int> x, y, z;
```

[End of Lecture #10]