V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 21

—Slide 1—

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 2—

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 3—

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 4—

Deallocation

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

```
cell *temp = front;

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

—Slide 5—

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 6—

`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 7—

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 8—

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 9—

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{
     enter
     pop
     empty
  }
  ```

  ```
  stack{
     push
     pop
     empty
  }
  ```

—Slide 10—

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 11—

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;
                            }
```

—Last Slide—

Example

- `circlist` Revisited

```
class circlist{
public:          //visible outside
  boolean empty();

protected:       //visible to only derived classes
       circlist();
  void push(int);
  int  pop();
  void enter(int)

private:
  cell *rear;
};
```

[End of Lecture #21]