V22.0490.001
Special Topics: Programming Languages

B. Mishra
New York University.

Lecture # 22

—Slide 1—

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 2—

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 3—

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 4—

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 5—

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 6—

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 7—

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 8—

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 9—

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 10—

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`).

  $\Rightarrow$ 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 11—

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 12—

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 13—

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 #22]