CSIS/COMP 1117B
Computer Programming

Inheritance

Object-Oriented Programming

- The main characteristics of object-oriented programming are:
  - encapsulation
    - a form of **information hiding** or **abstraction**
      - abstraction is supported by **class**
      - information hiding is supported using **private**
  - Inheritance
    - making program code **reusable**
  - polymorphism
    - a single name can have **multiple meanings** depending on ***context***
      - **template class** supports ***parametric polymorphism***
      - **overloading** supports a limited form of polymorphism

Inheritance Basics

- **Inheritance** is the process by which a new class – known as a **derived class** – is created from an existing class, called the **base class**.
- A derived class automatically has all the member variables and functions that the base class has and can have additional member functions and/or additional member variables.
- The base class is also known as the **parent class** or the **ancestor class** and the derived class is also known as the **child class**.

What is Inherited from the Base Class?

- All public and protected members of the base class are available to the derived class; except the followings:
  - constructors and destructor
    - the default constructor of the base is invoked *before* any constructor of the derived class when an object of the derived class is created
    - the destructor of the base is invoked at the *end* of executing the destructor of the derived class terminates
  - overloaded assignment defined on the base class
  - friends of the base class

Redefining Members of the Base Class

- Redefining a member variable of the base class in the derived class is allowed and the variable from the base class is available in the derived class as base::member
- When a member function of the base class is re-defined in the derived class:
  - if their **signatures** are the ***same***, similar to member variables
    - the signature of a function consists of the name of the function and the sequence of types in the parameter list, ignoring const and &
  - if their signatures are different, they are treated as overloaded functions in the derived class

A Simplified List Class

```
template <class any> class list {
private:
    struct node {
        any   *data;
        node *next;
    };
    node  *head, *tail;
protected:  // only accessible to derived classes
    void insert_at_head(any *t) { . . . }
    void insert_at_tail(any *t) { . . .}
    any *delete_at_head() { . . . }
public:
    bool empty_list() { return head == NULL; }
    list() { head = tail = NULL; }
    ~list() { . . . }
};
```

The Member Functions (1)

```
void insert_at_head(any *t) {
    node  *n = new node;
    n->data = t;
    n->next = head;
    if (head == NULL) tail = n;
    head = n;
}
void insert_at_tail(any *t) {
    node  *n = new node;
    n->data = t;
    n->next = NULL;
    if (head == NULL) head = n;
    else tail->next = n;
    tail = n;
}
```

The Member Functions (2)

```
any *delete_at_head() {
    if (head == NULL) return NULL;
    node *n = head;
    any   *t = head->data;
    head = head->next;
    if (head == NULL) tail = NULL;
    delete n;
    return t;
}
```

A String Stack

```
class str_stack : public list<string> {
public:
    void     push(string *s) { insert_at_head(s); }
    string*  pop() { return delete_at_head(); }
    bool     empty() { return empty_list(); }
    str_stack() {} // default constructor of base called
    ~str_stack() {}  // destructor of base called
};
```

A String Queue

```
class str_queue : public list<string> {
public:
    void  enqueue(string* s) { insert_at_tail(s); }
    string* dequeue() { return delete_at_head(); }
    bool  empty() { return empty_list(); }
    str_queue() { }
    ~str_queue() { }
};
```

What is the Type of a Derived Class?

- An object of a derived class type can be used anywhere that an object of any of its ancestor classes can be used.
  - for example, a str_stack object can be used anywhere a list<string> can be used
- If class Child is derived from class Ancestor and class Grandchild is derived from class Child, then an object of class Grandchild can be used anywhere an object of class Child can be used; and the object of class Grandchild can also be used anywhere that an object of class Ancestor can be used.

Polymorphism

- Polymorphism refers to the ability to associate ***multiple meanings*** to one function name by means of a special mechanism known as **late binding**.
- The technique of waiting until run-time to determine the implementation of a function is called late binding or **dynamic binding**.
- A **virtual *member* function** will be subjected to late binding; a member function is virtual if it is declared with the keyword virtual

Example

```
class      a {
private:
      char  x;
public:
      virtual void     f() { cout << "Virtual " << x << endl;}
      void  g() { cout << "Non virtual " << x << endl;}
      a() { x = 'a'; }
      ~a() { }
};
class      b :  public a {
private:
      char  y;
public:
      virtual void     f() { cout << "Virtual " << y << endl;}
      void  g() { cout << "Non virtual " << y << endl;}
      b() { y = 'b'; }
      ~b() { }
};
```

The Main Program

```
int   main() {
      a     *p = new a;
      b     *q = new b;

      cout << "First set of calls:\n";
      p->f();
      p->g();
      q->f();
      q->g();

      cout << "Second set of calls:\n";
      p = q;
      p->f();
      p->g();
      q->f();
      q->g();

}
```

The Output

```
First set of calls:
Virtual a
Non virtual a
Virtual b
Non virtual b
Second set of calls:
Virtual b
Non virtual a
Virtual b
Non virtual b
```

Pure Virtual Function

- A **pure** virtual function can be defined in an ancestor class as:

  virtual <type> <name> <parameter list> const ';'
- Any class derived from this (ancestor) class will have to supply the definitions of all the pure virtual functions defined in the ancestor class
  - a pure virtual function can ***only*** be called as a member of the derived class
- Different classes derived from the same ancestor class can have ***their own*** definitions of a pure virtual function.
- Typically used to specify a certain ***interface*** that all derived classes have to implement.

A Class for Geometric Objects

- The ancestor class defines common properties that are applicable to derived classes
  - geometric point as a pair of real numbers
    - methods involving points such as distance between two points
  - methods which are derived class specific, such a *show* for displaying an geometric object, will be defined as a virtual method
- Derived classes will be the various geometric objects: circle, triangle, rectangle, square, polygon, etc.