Lecture Ten

Virtual Functions

Ref: Herbert Schildt, Teach Yourself C++, Third Ed^{n} (Chapter 10)

© **Dr. M. Mahfuzul Islam**
Professor, Dept. of CSE, BUET

Pointers to Derived Classes

- A pointer declared as a **pointer to a base class** can also be **used** to point to any **class derived from that base**; however, the **reverse is not true**.
- A **type casting** can be used, but not recommended.

```
base *p;                    //base class pointer
base base_ob;               // object of base type
derived derived_ob;         // object of type derived
p = &base_ob;               // p points to base object
p = &derived_ob;            // p points to derived object
```

Introduction to Virtual Functions

- **A *virtual function* is a member function that is declared within a base class and redefined by a derived class. The keyword virtual is used in base class and the keyword is not needed in derived class.**
- **Virtual function implements one interface, multiple methods.**
- **A class that contains a virtual function is referred to as a *polymorphic class*.**
- **The determination of the type of object being pointed to by the pointer is made at run time.**

Introduction to Virtual Functions

```
class Father {
      char name[20];
public:
      Father(char *fname){ strcpy(name,fname);}
      void show(){
            cout << “Father: ” << name <<endl;
}
class Son: public Father {
        char name[20];
public:
        Son(char *sname, char *fname):
             Father(fname){ strcpy(name, sname);}
        void show(){
             cout << “Son: ” << name << endl;}
};
int main(){
       Father *fp, father(“Rashid”);
       Son son(“Robin”, “Rashid”);
       fp = father; fp->show();
       fp = son; fp->show();
}
```

OUTPUT:

```
Father: Rashid
Father: Rashed
```

```
class Father {
      char name[20];
public:
      Father(char *fname){ strcpy(name,fname);}
      virtual void show(){
            cout << “Father: ” << name <<endl;
}
class Son: public Father {
        char name[20];
public:
        Son(char *sname, char *fname):
             Father(fname){ strcpy(name, sname);}
        void show(){
             cout << “Son: ” << name << endl;}
};
int main(){
       Father *fp, father(“Rashid”);
       Son son(“Robin”, “Rashid”);
       fp = father; fp->show();
       fp = son; fp->show();
}
```

OUTPUT:

```
Father: Rashid
Son: Robin
```

Pure Virtual Functions

- A **pure *virtual function*** **has no definition relative to the base class. Only the function’s prototype is included. The general form is:**

  virtual type func-name (parameter-list) = 0;

```
class area {                    //Abstract class
      double dim1, dim2;
public:
      void setarea( double d1, double d2){
             dim1 = d1;    dim2 = d2;
      }
      void getdim( double &d1, double &d2){
             d1 = dim1;    d2 = dim2;
      }
      virtual double getarea() = 0;
}
```

```
class rectangle: public area {
public:
       double getarea(){
             double d1, d2;
             getdim(d1, d2);
             return d1 * d2;}
};
```

```
class triangle: public area {
public:
       double getarea(){
              double d1, d2;
              getdim(d1, d2); return 0.5*d1 * d2; }
};
```

```
int main(){
       area *p;       //area p; -> not permitted
       rectangle r;
       triangle t;

       r.setarea( 3.3, 4.5);
       t.setarea( 4.0, 5.0);

       p = &r; cout << p->getarea() << ‘\n’;
       p = &t; cout << p->getarea() << ‘\n’;
       return 0;
}
```

Virtual Destructors

```
class Father {
      char *name;
public:
      Father(char *fname){
           name = new char[ strlen(fname)+1 ];
           strcpy(name, fname);
      }
      virtual ~Father(){
           delete name;
           cout << “Father destroyed” << endl;
      }
      virtual void show(){
            cout << “Father: ” << name <<endl;
      }
}
```

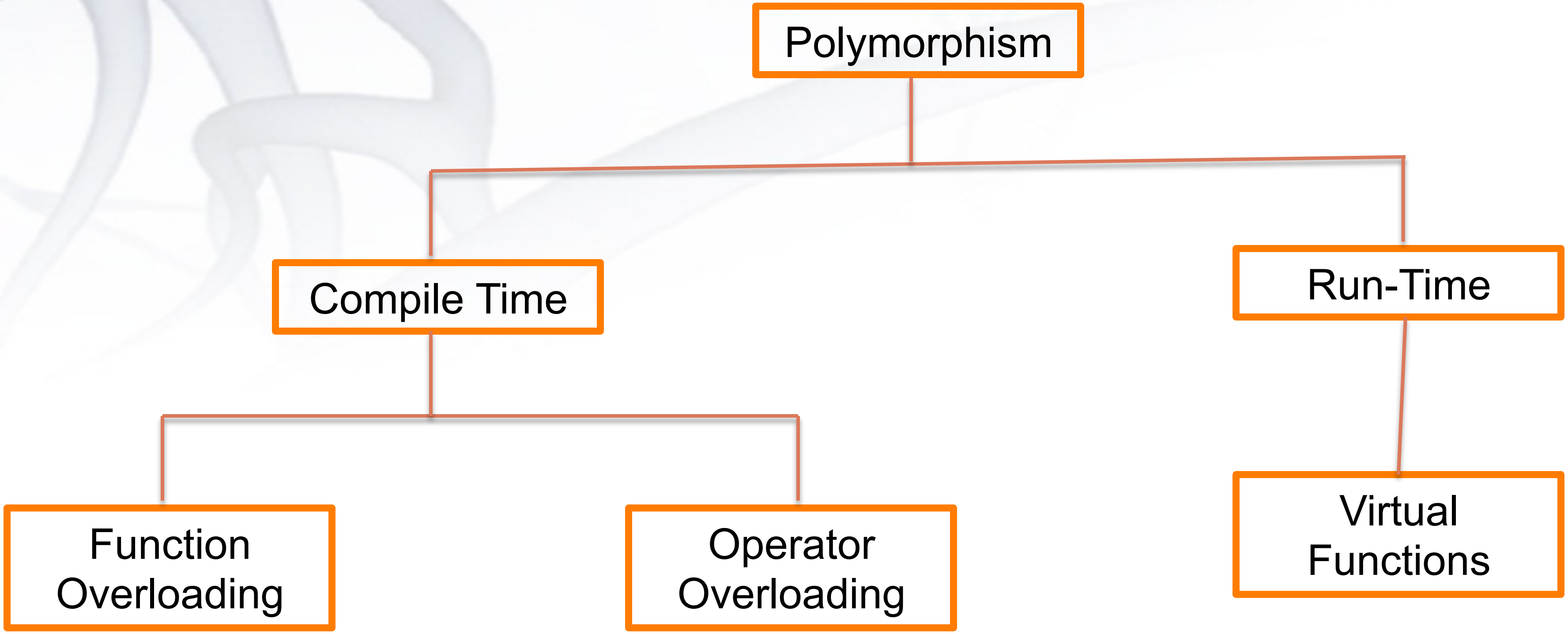
```
int main(){
       Father *fp, father(“Rashid”);
       Son son(“Robin”, “Rashid”);
       fp = father; fp->show();
       delete fp;
       fp = son; fp->show();
       delete fp;
}
```

```
class Son: public Father {
        char *name;
public:
        Son(char *sname, char *fname): Father(fname){
              name = new char[ strlen(sname)+1 ];
               strcpy(name, sname);
        }
        virtual ~Son(){
                delete name;
                cout << “Son destroyed” << endl;
        }
        virtual void show(){
             cout << “Son: ” << name << endl;
        }
};
```

OUTPUT:

```
Father: Rashid
Father destroyed
Son: Robin
Son destroyed
Father destroyed
```

Polymorphism Taxonomy



Polymorphism

Compile Time

Run-Time

Function Overloading

Operator Overloading

Virtual Functions

Applying Polymorphism

- **There are two terms linked with OOP: early binding and late binding.**
- **Early binding refers to those function calls that can be resolved during compilation. This method is faster but not flexible.**
- **Late binding refers to those function calls that can be resolved during run time. This method is slower but flexible.**

```
#include <iostream>
#include <cstdlib>
#include <ctype>
using namespace std;
```

```
class list {
public:
      list *head, *tail, *next;
      int num;

      list () { head = tail = next = NULL;}
      virtual void store(int i) = 0;
      virtual int retrieve() = 0;
};
```

Applying Polymorphism

```
class queue: public list {
public:
     void store(int i);
     int retrieve();
};
```

```
void queue::store(int i){
     list *item;

     item = new queue;
     if (!item){
           cout << “Allocation Error.\n”;
           exit(1);
    }
    item->num = i;
    if (tail) tail->next = item;
    tail = item;
    item->next = NULL:
    if (!head) head = tail;
}
```

```
int queue::retrieve(){
      int i;
      list *p;

      if (!head){
            cout << “List empty.\n”;
            return 0;
      }
      i = head->num;
      p = head;
      head = head->next;
      delete p;

      return i;
}
```

Applying Polymorphism

```
class stack: public list {
public:
	void store(int i);
	int retrieve();
};
```

```
void stack::store(int i){
	list *item;

	item = new stack;
	if (!item){
		cout << “Allocation Error.\n”;
		exit(1);
	}
	item->num = i;
	if (head) item->next = head;
	head = item;
	if (!tail) tail = head;
}
```

```
int stack::retrieve(){
	int i;
	list *p;

	if (!head){
		cout << “List empty.\n”;
		return 0;
	}
	p = head;
	while(p->next != tail) p = p->next;
	i = tail->num;
	tail = p;
	p = p->next;
	delete p;

	return i;
}
```

Applying Polymorphism

```
int main(){
    list *p;
    stack s_ob;
    queue q_ob;
    char ch;
    int i;

   for( i = 0; i < 10; i++){
        cout << “Stack or Queue (S/Q)?:”;
        cin >> ch;
        ch = tolower(ch);
        if (ch == ‘q’) p = &q_ob;
        else p = &s_ob;
        p->store(i);
   }
```

```
    cout << “Enter T to Terminate\n”;
    for(;;){
        cout << “Remove from stack or queue (S/Q):”;
        cin >> ch;
        ch = tolower(ch);
        if ( ch == ‘t’) break;
        if (ch == ‘q’) p = &q_ob;
        else p = &s_ob;
        cout << p->retrieve() << ‘\n’;
    }
    cout << ‘\n’;

    return 0;
}
```