Polymorphism
Part 1
What is Polymorphism?

*Polymorphism* refers to a programming language’s ability to process objects differently depending on their *data type* or *class*.
Why polymorphism?

With polymorphism, one method can cause different actions to occur, depending on the type of the object on which the method is invoked.
Quotes from Deitel & Deitel’s

- The ability for objects of different classes related by inheritance to respond differently to the same message (i.e., member function call).

- The same message sent to many different types of objects takes on “many forms” - hence the term polymorphism.
How?

- Class Hierarchy: Inheritance
- Abstract Class
- Virtual Functions

Base-class pointer = derived class object (pointer)
How?

- Class Hierarchy: Inheritance
- Abstract Class
- Virtual Functions
Virtual Functions

- Virtual functions allow the programmer to declare functions in a base class that can be redefined in each derived class.

```cpp
class shape {
    ...
    public:
        virtual void draw();
    ...
}

class circle: public shape {
    ...
    public:
        virtual void draw();
    ...
}
```
Declare a virtual function

• Syntax 1:
  ```cpp
  virtual double earnings() = 0;
  ```

• Syntax 2:
  ```cpp
  virtual void print() const;
  ```

• A pure virtual function makes the corresponding class an abstract base class.
• No objects of an abstract base class can be instantiated.
• Classes from which objects can be instantiated are concrete classes.
More on Pure Virtual Function

- **No definition** given for pure virtual functions.

- If a class is derived from a class with a pure virtual function, and if no definition is supplied for that pure virtual function in the derived class, then the virtual function remains pure in the derived class. Consequently, the derived class is also an abstract class.
Although we cannot instantiate objects of abstract base classes, we can declare pointers and references to abstract base classes.

Such pointers and references can be used to enable polymorphic manipulations of derived-class objects.

Once a function is declared virtual, it remains virtual all the way down the inheritance hierarchy from that point, even if it is not declared virtual when a derived class overrides it.

It is suggested to explicitly declare inherited virtual functions virtual at every level of the hierarchy.
Virtual Functions

- A virtual function must be declared in a parent class

- **syntax**
  - `virtual function`
    - `virtual returnType functionName ( args1 ) {`
      - function body ;}
  - `pure virtual function`
    - `virtual returnType functionName ( args1 ) = 0;`
• **Virtual Functions**

**Declaration**

- A function name is preceded by the keyword `virtual`
  - Function name can only be used once in the parent class
  - Cannot overload virtual functions
  - Only class member functions can be declared virtual

- A function is virtual…..
  - If it is declared virtual
    - There is a base class function with the same signature
      - declared virtual

- Any or all class member functions (except constructors) can be declared virtual
EXAMPLE 5.1.5.

```cpp
virtual void f( ); //*** ERROR: not a method
int main ( ) {
    // ...
}
```

**Discussions:** contains an error because it declares f, a top-level function rather than a method, to be virtual.
Example

class C {
public:
virtual void m( ) ; // declaration--"virtual" occurs
   // . . .
};

void C::m( ) { // definition--"virtual" does not occur
   // . . .
}

Discussions: virtual method m is defined outside the class declaration. Therefore, the keyword virtual occurs only in its declaration.
Derived class object to base class object conversion

• A derived class object is an object of its base class
  
  Base class object pointer = derived-class object

• A derived class object contains more features than its base-class

• Note: the reverse is not true.

Derived-class object pointer ≠ base-class object (pointer)

Not enough information
Binding in C++

• Polymorphism and C++

  • Early
    • Binding occurs at compile time
      – Early binding polymorphism
        – Process of overloading members
  
  • Late
    • Binding occurs at runtime
      – Late binding polymorphism
        – The code to implement the method is chosen at runtime
          – Appropriate code chosen sending a message to
            the object .... Not to the pointer to the object
          – Implemented through virtual functions
Compile-Time Binding vs. Run-Time Binding

- A function’s name is associated with an entry point, the starting address of the code that implements the function.

- Compile-time binding lets the compiler determine what code is to be executed when a function name is invoked.

- Run-time binding is the binding of a function’s name to an entry point when the program is running.
# Compile-Time Binding

```cpp
#include <iostream>
using namespace std;
void sayHi();
int main() {
    sayHi();
    sayHi();
    return 0;
}
void sayHi() {
    cout <<"Hello, cruel world!"<< endl;
}
```
Requirements for C++ Polymorphism

• There must be an inheritance hierarchy

• The classes in the hierarchy must have a virtual method with the same signature

• There must be either a pointer or a reference to a base class. The pointer or reference is used to invoke a virtual method
// TradePerson.h
#ifndef TRADE_PERSON_H
#define TRADE_PERSON_H

class TradePerson {
public:
    virtual void sayHi();
};

class Tinker : public TradePerson {
public:
    virtual void sayHi();
};

class Tailor : public TradePerson {
public:
    virtual void sayHi();
};
#endif
// TradePerson.cpp
#include "TradePerson.h"
#include <iostream>
using namespace std;
void TradePerson::sayHi() {
    cout <<"Just hi"<< endl;
}
void Tinker::sayHi() {
    cout <<"Hi, I tinker"<< endl;
}
void Tailor::sayHi() {
    cout <<"Hi, I tailor"<< endl;
}
// Polymorphism Demo
#include "TradePerson.h"
#include <iostream>
using namespace std;
int main() {
    TradePerson* p;
    int which;
    do {
        cout <<"1 == TradePerson, 2 == Tinker, 3 == Tailor ";
        cin >> which;
    } while ( which < 1 || which > 3 );
switch ( which ) {
    case 1: p = new TradePerson; break;
    case 2: p = new Tinker; break;
    case 3: p = new Tailor; break;
}
p->sayHi();
delete p;
return 0;
// TestClient.cpp
#include "TradePerson.h"
#include <iostream>
#include <ctime>
using namespace std;

int main() {
    srand( time(0) );
    TradePerson* ptr[10];
    unsigned which, i;
    for( i=0; i<10; i++ ) {
        which = 1 + rand()%3;
        switch ( which ) {
        case 1: ptr[i] = new TradePerson; break;
        case 2: ptr[i] = new Tinker; break;
        case 3: ptr[i] = new Tailor; break;
        }
    }
}
Test Client 2 (2)

```cpp
for ( i=0; i<10; i++ ) {
    ptr[i]->sayHi();
    delete ptr[i];
}
return 0;
```
The main function contains a call to

\[ \text{srand}( \text{time}(0)) \];

in order to avoid the generation of the same random numbers each time it runs.

The keyword \textit{virtual} must be explicitly declared in the base class. Any derived class method with the \textit{same signature} will be automatically \textit{virtual}.
// TestClient.cpp

#include "TradePerson.h"
#include <iostream>
#include <ctime>
using namespace std;

void meet( TradePerson& person ) {
    person.sayHi();
}

int main() {
    srand( time(0) );
    unsigned which, i;
    TradePerson tradePerson;
    Tinker tinker;
    Tailor tailor;
    for( i=0; i<10; i++ ) {
        which = 1 + rand()%3;
}
switch ( which ) {
    case 1: meet( tradePerson ); break;
    case 2: meet( tinker ); break;
    case 3: meet( tailor ); break;
}

return 0;
Inheriting virtual Methods

```cpp
#include <iostream>
using namespace std;
class TradePerson {
public:
    virtual void sayHi() {
        cout << "Just hi" << endl;
    }
};
class Tinker : public TradePerson {
};
int main() {
    Tinker t;
    t.sayHi();
    return 0;
}
```
Exercise

• Write a class Shape where a virtual method draw is defined as printing out a suitable message like “Drawing Shape”. Then finish classes Rectangle and Circle derived from Shape. Define the method draw in these two classes as displaying suitable messages like “Drawing Rectangle” and “Drawing Circle.” Try to test the polymorphism of these three classes in different ways. Draw UML diagrams before coding.
• **Virtual Functions**

• When function in a class is declared virtual

  • Keyword *virtual* tells compiler
    • Don’t perform early binding
    • Install mechanisms to perform late binding

  • Compiler responds by creating
    • Table of function pointers
    • Installing a data member to the class to point to the table
• **Virtual Functions**

• The compiler created table is called the *vtable* (*vtbl*)
  - Contains pointers to all functions declared virtual within the class
  - and derived classes.

• Each class gets its own vtable

• A data member called the *vpointer* (*vPtr*)
  - Usually placed as the first element in object in memory.
  - Initialized to the starting address of the vtable.

• The function call through a base class pointer
  - Indexes into the vtable calls the function located at the address.
class A
{
    public:
        int i;
        virtual void f();
        virtual void g();
};

class A vtable

vtable[0]: &f()
vtable[1]: &g()
class A
{
    public:
        int i;
        virtual void f ( );
};

class B : public A
{
    public:
        virtual void f ( );    // override f ( )
        virtual void g ( );    // define g ( )
};
Virtual Pointers and Vtables

class B {
public:
    int data1;
    int data2;
    void m0( );
    virtual void m1( );
    virtual void m2( );
};

class D : public B {
public:
    int data3;
    virtual void m1( );
};
Virtual Pointers and Vtables

B object

<table>
<thead>
<tr>
<th>data1</th>
</tr>
</thead>
<tbody>
<tr>
<td>data2</td>
</tr>
</tbody>
</table>

D object

| data3       |

B::m0

vptr

vtable

0x7723

B::m1

B::m2

0x23b4

D::m1

0x99a7

0x23b4

Entry Points
Vtable

int main() {
    B b1;
    D d1;
    B* p;
    //... p is set to b1's or d1's address
    p->m1();
    //...
}

Run-Time Binding Using Vtable

• A program that uses run-time binding incurs a performance penalty in both space and time

• Pure object-oriented language incurs a relatively heavy penalty, because all functions are bound at run-time
Constructors Can Not Be virtual

class C {
public:
    virtual C(); //*** ERROR
    virtual C( int ); //***ERROR
    virtual ~C();
    virtual void m();
};
# Virtual Destructors (1)

// Virtual Destructor

#include <iostream>
using namespace std;
class A {
public:
    A() {
        cout << "A()" << endl;
        p = new char[5];
    }
    virtual ~A() {
        cout << "~A()" << endl;
        delete [] p;
    }
private:
    char* p;
};
virtual Destructors (2)

class Z : public A {
public:
    Z() {
        cout <<"Z()" << endl;
        q = new char[5000];
    }
    ~Z() {
        cout <<"~Z()" << endl;
        delete [] q;
    }
private:
    char* q;
};
virtual Destructors (3)

void f() {
    A* ptr;
    ptr = new Z();
    delete ptr;
}

int main() {
    for( unsigned i=0; i<3; i++ )
        f();
    return 0;
}
Exercise

- Write a class `Diagram` where a pointer `shapes` is used to maintain an array of `Shape` objects. In the constructor, `shapes` is newed as an array of two `Shape` objects, one is a `Rectangle` object, and the other a `Circle` object. Try to write the destructor to delete `shapes`. Also, define a method `draw` to call the method `draw` of every element in the array pointed by `shapes`. Test your programs. Remember to draw UML.
static Method Can Not Be virtual

class C {
public:
    static virtual void f(); //***ERROR
    static void g();
    virtual void h();
};
Name Overloading

Definition: Methods in the same class share the same name, but they have different formal parameter lists.

Name overloading always involves compile-time binding.
Name Overloading

class C {
public:
    C() { /* ... */ }
    C( int x ) { /* ... */ }
};
void f( double d ) { /* ... */ }
void f( char c ) { /* ... */ }
int main() {
    C c1;
    C c2( 26 );
    f( 3.14 );
    f( 'z' );
    // ...
    return 0;
}
Suppose that the base class B has a method m and its derived class D also has a method m with the same signature.

If the methods are **virtual**, run-time binding is at work in any invocation of m through pointers or references.

If the methods are **virtual**, the derived class method D :: m overrides the base class method B :: m.

If the methods are **not virtual**, compile time binding is at work in any invocation of m.
Name Overriding

#include <iostream>
using namespace std;

class B {
public:
    virtual void m() { cout <<"B::m"<< endl; }
};

class D : public B {
public:
    virtual void m() { cout <<"D::m"<< endl; }
};

int main() {
    B* p;
    p = new D;
    p->m();
    return 0;
}
**Name Hiding**

- Suppose that base class $B$ has a nonvirtual method $m$ and its derived class $D$ also has a method $m$.
- $D$’s local method $D::m$ is said to hide the inherited method $B::m$.
- Name hiding is particularly tricky if the derived class’s method has a *different signature* than the base class’s method of the same name.
Name Hiding (1)

```cpp
#include <iostream>
using namespace std;
class B {
public:
    void m( int x ) { cout << x << endl; }
};
class D : public B {
    void m() { cout << "Hi" << endl; }
};
int main() {
    D d1;
    d1.m();
    d1.m( 26 ); //***ERROR
    return 0;
}
```
#include <iostream>
using namespace std;
class B {
public:
    virtual void m( int x ) { cout << x << endl; }
};
class D : public B {
    virtual void m() { cout << "Hi" << endl;
}
int main() {
    D d1;
    d1.m();
    d1.m( 26 ); //***ERROR
    return 0;
}
Name Sharing

- Top-level functions with overloaded names
- Constructors with overloaded names
- Nonconstructor methods of the same class with the same name
- A derived class’s local virtual method overrides a virtual method inherited from the base class
- For overriding to occur, the methods must be virtual and have the same signature
Pure virtual Methods

class ABC {
public:
    virtual void open() = 0;
    int getCount() const { return n; }
private:
    int n;
};

class X : public ABC {
public:
    virtual void open() { /* ... */ }
};
class Y : public ABC {
};

ABC a1; //***ERROR
X x1;
Y y1; //***ERROR
Abstract Classes

class Shape {
protected:
public:
    virtual void draw( ) = 0;
    virtual ~Shape( ) { };
};
class Rectangle : public Shape {
public:
    void draw( ) { cout<< "Rectangle" <<endl; }
    ~Rectangle( ) {};
};
void display( Shape& s ) { s.draw(); } 
int main()
{
    Rectangle r;
    display( r );
    Shape* shape = new Rectangle;
    shape->draw();
    delete shape;
    return 0;
}
Constructors of Abstract Classes (1)

// Abstract Class Constructor
#include <iostream>
using namespace std;
class Shape {
public:
    Shape( int x0, int y0 ) {
        cout << "Shape constructor" << endl;
        xOrigin = x0;
        yOrigin = y0;
    }
    virtual void draw() = 0;
    virtual ~Shape() { cout << "Destructing Shape" << endl; }
private:
    int xOrigin;
    int yOrigin;
};
Constructors of Abstract Classes (2)

class Rectangle : public Shape {
public:
    Rectangle( int x0, int y0, int size1, int size2 ) :
        Shape( x0, y0 ) {
            cout << "Rectangle constructor" << endl;
            this->size1 = size1;
            this->size2 = size2;
        }
    void draw() { cout << "Drawing rectangle" << endl; }
    ~Rectangle() { cout << "Destructing Rectangle" << endl; }
private:
    int size1;
    int size2;
};
Constructors of Abstract Classes (3)

class Square : public Rectangle {
public:
    Square( int x0, int y0, int size ) :
        Rectangle( x0, y0, size, size ) {
            cout << "Square constructor" << endl;
        }
    virtual void draw() { cout << "Drawing Square" << endl; }
    virtual ~Square() { cout << "Destructing Square" << endl; }
private:
    int size;
};
Constructors of Abstract Classes (4)

```cpp
int main() {
    Shape* s;
    s = new Square( 0, 0, 5 );
    s->draw();
    delete s;
    return 0;
}
```
Abstract Base Classes

class BasicFile {
    public:
        virtual void open() = 0;
        virtual void close() = 0;
        virtual void flush() = 0;
};
class InFile : public BasicFile {
    public:
        virtual void open() { /*…*/ }
        virtual void close() { /* … */ }
        virtual void flush() { /*… */ }
};
• **Virtual Base Classes**

• Parent classes may have a common base class

```
Fruit
  
  Peach
  Plum

Fruit
  
  Peach
  Plum

Stem

Nectarine
```
• **Virtual Base Classes**

• **Problem:**

  • Fruit has a *stem* data member
  • Peach and plum each inherit a *stem* member from Fruit
  • Nectarine inherits a *stem* member from each
  • Could resolve using the scope operator
    - Plum::stem
    - Peach::stem
Virtual Base Classes

Solution:
- Declare Fruit as a *virtual base class*

Result:
- Only a single copy of the base class in the derivation hierarchy.
- Only a single copy of all inherited data members.
- Subsequent derivations point to shared members.
• **Virtual Base Classes - Specification**

• **Syntax**

```cpp
class DerivedClass : virtual accessSpec BaseClass

• DerivedClass    - The class being derived
• BaseClass       - The parent class
• Specification   - Specify base class member access
  • public
  • protected
  • private

• The keyword virtual identifies BaseClass as a virtual base class of DerivedClass.
```
Interfaces

- An abstract base class specifies a shared interface
- The interface is shared by all classes derived from the abstract base class
- The interface is used to specify design requirement
- The interface promotes code reuse
Exercise

• Rewrite class Shape as an Abstract Base Class. Test the polymorphism again with classes Shape, Rectangle, and Circle. Draw UML diagrams before coding.