Polymorphism

Introduction

> Any time we use a **base pointer** or a **base reference** to **access a derived object**, this is called polymorphism

Example

Once I define a function that accepts a (reference or pointer to a) `Person`, not only can I pass `Person` variables to that class, BUT I can also pass **any variable** that was derived from a `Person`

```
class Person {...};
class Student : public Person {...};

void SayHi(Person& p)
{
    cout << "Hello " << p.getName();
}

int main()
{
    Person p("Eric", 18);
    SayHi(p);

    float GPA = 1.6;
    Student s("David", 19, GPA);
    SayHi(s);
}
```

- `SayHi` treates variable`p` as if it referred to a `Person` variable

> Polymorphism ONLY WORKS when you use a **reference** or a **pointer** to pass an object!

Otherwise: Chopping

```
void SayHi(Person p) {...}

int main()
{
    Student s("Carey", 38, 3.9);
    SayHi(s);
}
```

- C++ chop off all the data/methods of the derived class, only sned the base (Person) parts of the variable s to the function

Functions Specific to Derived Classes are '*Invisible*'

Example

```
class Shape
{
    public:
        virtual double getArea()
        { return (0); } //Shape is a abstract base class...
}

class Circle : public Shape
{
    public:
        virtual double getArea()
        { return (3.14*m_rad*m_rad); }

        void setRadius(int newRad)
        { m_rad = newRad; }
    private:
        int m_rad;
}

class Square : public Shape
{
    public:
        virtual double getArea()
        {
            return (m_side*m_side);
        }
        void setSide(int side)
        { m_side = side; }
    private:
        int m_side;
}

void PrintPrice(Shape& x)
{
    cout << "Cost is: " $" << x.getArea()*3.25;
}

int main()
{
    Square s(5);
    PrintPrice(s);

    Circle c(10);
    PrintPrice(c);
}
```

- `PrintPrice` method THINKS that every variable passed in is JUST a `Shape`

    - CANNOT access functions specific to `Circle` or `Square`

      ```
      void PrintPrice(Shape& x)
      {
          x.setRadius(10); //ERROR
      }
      ```

- **If no `virtual` keyword**
    - C++ can't figure out which version to call -> just call the version defined in the base class

Inheritance versus Polymorphism

- Inheritance:
    - Publicly derive one or more classes from a common base class
    - All derived class, by def, inherit a *common set* of functions from base class
    - Each derived class may re-define any function
- Polymorphism
    - Use a base pointer/reference to access any variable that is of a type that is *derived from the base class*
    - The same function call automatcially causes different actions to occur -> depending on what type of variable is currently being referred/pointed to.

`Virtual` Keyword Cheat Sheet

When should you use the `virtual` keyword?

1. In `base` class any time you expect to redefine a function in a `derived` class
2. In `derived` classes any time you redefine a function (convention)
3. Destructor, ALWAYS in base class & derived class
4. No virtual constructors

Polymorphism and Pointers

> In general, you may point a base class pointer at a derived class variable

Example

```
class Person {...};

class Politician : public Person {...};
```

```
int main()
{
    Politician carey;
    Person *p;
    p = &carey; //LEGAL
    cout << p->getname(); //a Person method
}
```

> NEVER point a derived class pointer to a base class variable

```
int main()
{
    Politician *p;
    Person david;
    p = &david; //ILLEGAL!!!
}
```

Access base method that calls another overrode method with pointer

> C++ always calls the most-derived version of a function associated with a variable, as long as it's marked `virtual`

Example

```
class Geek
{
    public:
        void tickleme()
        {
            laugh();
        }
        virtual void laugh() { cout << "ha ha!"; }
};

class HighPitchGeek : public Geek
{
    public:
        virtual void laugh()
        { cout << "tee hee hee"; }
};
```

```
int main()
{
    Geek *ptr = new HighPitchGeek;

    ptr->tickleMe();
    delete ptr;
}
```

- A base pointer (Geek) is accessing a derived object
- Derived object `HighPitchGeek` did not define a `tickleMe()`
    - Calls base object function `tickleMe()`, which calls `laugh()`
- Which `laugh()`?
    - Call `HighPitchGeek`'s `laugh()`

Polymorphism and Virtual Destructors

Should always make sure that you use **virtual destructors** when you use inheritance/polymorphism

Example:

All professors think they're smart

All math profs keep a set of flashcards

```
class Prof
{
    public:
        Prof() { m_myIQ = 95; }
        virtual ~Prof()
        {
            cout << "I died smart: " << m_myIQ;
        }
    private:
        int m_myIQ;
};

class MathProf : public Prof
{
    public:
        MathProf()
        {
            m_pTable = new int[6];
            for (int i = 0; i < 6; i++) {
                m_pTable[i] = i*i;
            }
        }
        virtual ~MathProf()
        {
            delete [] m_pTable;
        }
    private:
        int *m_pTable;
};
```

```
int main()
{
    Prof *p;
    p = new MathProf;
    delete p;
}
```

- Call Prof's constructor first, then MathProf's constructor
    - Prof IQ data set -> MathProf array allocated, value set
- MathProf destructor first

What happens if destructors aren't virtual functions?

- Technically undefined behavior
- Typical behavior

    ```
    Prof *p;
    p = new MathProf;
    delete p;
    ```

    - Since not `virtual` -> call base class destructor
    - Calls `Prof` destructor
    - `~MathProf()` was NEVER CALLED
        - allocated array was never freed -> MEMORY LEAK!!!!

> Nonvirtual destructors are only problematic during polymorphism

```
class Person
{
    public:
        ~Person() { cout ...}
};

class Prof : public Person
{
    public:
        ~Prof() { cout ...}
};
```

```
int main()
{
    Prof carey;
}
```

- Both destructors will get called in this case because compiler KNOWS that this is a `Prof`

```
int main()
{
    Person *p = new Prof;
    delete p;
}
```

- Compiler DOES NOT KNOW `*p` points to a `Prof` because `p` is a `Person` poiner and `Person`'s destructor isn't virtual
    - Only `Person`'s destructor will be called

> Always use virtual destructors

> We have to declare and define (even an empty one) a virtual destructor for base classes always due to the steps of destruction

---

Pure Virtual Functions and Abstract Base Class

> A pure virtual function is one that has no actual code

We must define functions that are common to all derived classes in our base class in order to use polymorphism

- But these functions are never used, previously we defined a dummy version of these functions

Syntax

`=0;`

```
class Shape
{
    public:
        virtual float getArea() = 0;
        virtual float getCircum() = 0;
};
```

Define a **pure virtual function** -> base version of the function will never be called

```
//ALL OF THESE ARE ERROR
int main()
{
    Shape s;
    cout << s.getArea();
    cout << s.getCircum();
}
```

- No code -> cannot define anything, cannot call anything

> Dervied classes MUST re-define all pure virtual functions

> Make the base class function pure virtual if the base-class version of your function does not do anything useful

Abstract Base Class

If you define at least one pure virtual function in a base class, then the class is called an Abstract Base Class

Rules if you define an ABC:

1. Derived classes must either provide code for ALL pure virtual functions
2. Or derived class becomes an ABC itself

Example

```
class Robot //ABC
{
    public:
        virtual void talkToMe() = 0;
        virtual int getWeight() = 0;
};

class KillerRobot : public Robot //REGULAR CLASS
{
    public:
        virtual void talkToMe() { cout...}
        virtual int getWeight() { cout ...}
};

class FriendlyRobot : public Robot //ABC
{
    public:
        virtual void talkTome() { cout ... }
        //Effectively it has a pure virtual version of
        //getWeight() too
        virtual int getWeight() = 0;
};

class BigHappyRobot : public FriendlyRobot //REGULAR
{
    public:
        virtual int getWeight() { return 500; }
}; //inherits FriendlyRobot's talkToMe() and defines own other
```

- Pure virtual functions force the user to implement certain functions to prevent common mistakes

> ABCs can be used like regular base classes to implement polymorphism

```
void PrintPrice(Shape &x)...
```

- BUT cannot create a variable with ABC type