Polymorphism and Virtual Functions

class Ball : public Sphere

Problems in Function Redefinition

```cpp
int main()
{
    Sphere mySphere();
    Ball myBall();
    Sphere *spherePtr;
    spherePtr = &mySphere;
    spherePtr->displayStatistics();
    ...
    spherePtr = &myBall;
    spherePtr->displayStatistics();
}
```
Virtual Functions

- Virtual functions are the answer
- Tells compiler:
  - “Wait until used in program”
  - “Then get implementation from object instance”
- Called late binding or dynamic binding
  - Virtual functions implement late binding
- Class that define virtual functions are extensible

Polymorphism

- Polymorphism refers to the ability to associate many meanings to one function by means of the late-binding mechanism. Thus, polymorphism, late binding, and virtual functions are really all the same topic.

Use Virtual Functions

```cpp
class Sphere {
public:
    virtual void displayStatistics(){
        cout << "It is Sphere\n";
    }
};
class Ball : public Sphere {
public:
    void displayStatistics(){
        cout << "It is Ball\n";
    }
};

text main()
{
    Ball myBall();
    Sphere *spherePtr = &myBall;
    spherePtr->displayStatistics();
}
```

OUTPUT:

```
It is Ball
```
Virtual: How?

- To write C++ programs:
  - Assume it happens by ‘magic’!
- But explanation involves late binding
  - Virtual functions implement late binding
  - Tells compiler to ‘wait’ until function is used in program
  - Decide which definition to use based on calling object
- Very important OOP principle

Overriding

- Virtual function definition changed in a derived class
  - We say it’s been ‘overridden’
- So:
  - Virtual functions changed: *overridden*
  - Non-virtual functions changed: *redefined*

Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we’ve seen
- One major disadvantage: overhead!
  - Uses more storage
  - Late binding is ‘on the fly’, so programs run slower
- So if virtual functions not needed, should not be used
getArea is not virtual; myBall.displayStatistics() calls Sphere::getArea

getArea is virtual; (a) mySphere.displayStatistics() calls Sphere::getArea;

(b) myBall.displayStatistics() calls Ball::getArea
Not All Definitions are Useful

- Base class might not have ‘meaningful’ definition for some of it’s members!

```cpp
class Employee {
    public:
        Employee(string tName, string tSsn);
        // other methods...
    void printCheck() const;
};
Employee::printCheck() {
    cout << "ERROR: Undifferentiated employee\n";
    exit(0);
}
```

Pure Virtual Functions

```cpp
class Employee {
    public:
        Employee();
        Employee(string tName, string tSsn);
        string getName() const;
        string getSsn() const;
        double getNetPay() const;
        void setName(string newName);
        void setSsn(string newSsn);
        void setNetPay(double newNetPay);
        virtual void printCheck() = 0;
    private:
        string name;
        string ssn;
        double netPay;
};
```

Abstract Base Classes

- Pure virtual functions require no definition
  - Forces all derived classes to define ‘their own’ version
- Class with one or more pure virtual functions is: abstract base class
  - Can only be used as base class
  - No objects can ever be created from it
    - Since it doesn’t have complete ‘definitions’ of all its members!
- If derived class fails to define all pure’s:
  - It’s an abstract base class too
Multiple Inheritance

class Base {
  public:
    virtual void print() = 0;
};

class DerivedOne : public Base {
  public:
    void print() {cout << "DerivedOne";}
};

class DerivedTwo : public Base {
  public:
    void print() {cout << "DerivedTwo";}
};

class Multiple : public DerivedOne, public DerivedTwo {
  public:
    void print() {DerivedTwo::print();}
};

Base

DerivedOne

DerivedTwo

Multiple

Multiple Inheritance

int main() {
    Multiple both; DerivedOne one; DerivedTwo two;
    Base *array[3];
    array[0] = &both;
    array[1] = &one;
    array[2] = &two;
    for (int i=0; i<3; i++)
        array[i]->print();
    return 0;
}

Error C2594: '=': ambiguous conversion from 'class Multiple *' to 'class Base *'
Virtual Base Class

```cpp
class Base {
    public:
        virtual void print() = 0;
};

class DerivedOne : virtual public Base {
    public:
        void print() {cout << "DerivedOne";}
};

class DerivedTwo : virtual public Base {
    public:
        void print() {cout << "DerivedTwo";}
};

class Multiple : public DerivedOne, public DerivedTwo {
    public:
        void print() {DerivedTwo::print();}
};

template

int main(){
    Multiple both; DerivedOne one; DerivedTwo two;
    Base *array[3];
    array[0] = &both;
    array[1] = &one;
    array[2] = &two;
    for (int i=0; i<3; i++)
        array[i]->print();
    return 0;
}
```
class Ball : public Sphere

int main()
{
    Ball myBall();
    Sphere *spherePtr;
    spherePtr = &myBall;
    spherePtr->getName();

    Ball *ballPtr;
    ballPtr = dynamic_cast <Ball *>(spherePtr);
    ballPtr->getName();

    // Error: Sphere has no function called getName()

    // "Downcasting" to a Ball pointer

    // Now it works
}