

Inheritance - recap

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Pet{
public:
    virtual void eat() const { cout << "eating" << endl; }
protected:
    const string& getName() const { return name; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const { cout << "Cat is "; Pet::eat(); }
};

class Slug : public Pet {};
class Roach : public Pet {};

void funcByVal(Pet apet) { apet.eat(); }
void funcByRef(Pet& apet) { apet.eat(); } // polymorphic

int main() {
    Pet p;
    Cat c;
    Slug s;
    Roach r;

    cout << "Invoking by val: " << endl;
    funcByVal(p);
    funcByVal(c);

    cout << "Invoking by ref: " << endl;
    funcByRef(p);
    funcByRef(c);
}
```

Invoking by val:

eating

eating

Invoking by ref:

eating

Cat is eating

- The method "eat()" is defined in the base and also in the derived class:
 - Without "**virtual**", defining eat() in the derived class is a "**redefinition**".
 - With "**virtual**", defining eat() in the derived class is "**overriding**".
- When two methods have the same name:
 - Same number and type of parameters → overriding or redefinition
 - Different number or type of parameters → **hiding**
- A method in the derived class may invoke a method that's in the base class (overridable or not) by using the base class as the tag with the scope resolution operator
 - Without scoping, we get infinite recursion

Inheritance – recap cont.

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Pet{
public:
    virtual void eat() const { cout << "eating" << endl; }
protected:
    const string& getName() const { return name; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const override { cout << "Cat is "; Pet::eat(); }
};

class Slug : public Pet {};
class Roach : public Pet {};

void funcByVal(Pet apet) { apet.eat(); }
void funcByRef(Pet& apet) { apet.eat(); } // polymorphic

int main() {
    Pet p;
    Cat c;
    Slug s;
    Roach r;

    cout << "Invoking by val: " << endl;
    funcByVal(p);
    funcByVal(c);

    cout << "Invoking by ref: " << endl;
    funcByRef(p);
    funcByRef(c);
}
```

Invoking by val:

eating

eating

Invoking by ref:

eating

Cat is eating

- “**override**” tells the compiler to cause an error if the base class’ function is not overridable (i.e. not declared as virtual)
 - Useful when deriving from a class that’s in library or framework, that’s too large or difficult to track down.
 - When concerned about other developers (possibly in a different team) modifying the implementation of the base class.

Inheritance – recap cont. 1

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Pet{
public:
    virtual void eat() const { cout << "eating" << endl; }
protected:
    const string& getName() const { return name; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const override final { cout << "Cat is "; Pet::eat(); }
};

class Slug : public Pet {};
class Roach : public Pet {};

void funcByVal(Pet apet) { apet.eat(); }
void funcByRef(Pet& apet) { apet.eat(); } // polymorphic

int main() {
    Pet p;
    Cat c;
    Slug s;
    Roach r;

    cout << "Invoking by val: " << endl;
    funcByVal(p);
    funcByVal(c);

    cout << "Invoking by ref: " << endl;
    funcByRef(p);
    funcByRef(c);
}
```

Invoking by val:

eating

eating

Invoking by ref:

eating

Cat is eating

- “**final**” tells the compiler to cause an error if a class derived from Cat tries to override the method eat(), i.e. class Cat is the final class to override eat().

Inheritance – recap cont. 2

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Pet{
public:
    virtual void eat() const { cout << "eating" << endl; }
protected:
    const string& getName() const { return name; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const override final { cout << "Cat is "; Pet::eat(); }
};

class Slug : public Pet {};
class Roach : public Pet {};

void funcByVal(Pet apet) { apet.eat(); }
void funcByRef(Pet& apet) { apet.eat(); } // polymorphic

int main() {
    Pet p;
    Cat c;
    Slug s;
    Roach r;

    cout << "Invoking by val: " << endl;
    funcByVal(p);
    funcByVal(c);

    cout << "Invoking by ref: " << endl;
    funcByRef(p);
    funcByRef(c);
}
```

Invoking by val:

eating

eating

Invoking by ref:

eating

Cat is eating

Passing by value:

- The object is copied (through the stack) into the parameter whose type is Pet
- Since Cat had everything in Pet and some more, then copying will involve converting an object of type Cat into an object of type Pet
→ **Slicing**
- We loose all the attributes and methods by which Cat extended Pet.

Inheritance – recap cont. 3

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Pet{
public:
    virtual void eat() const { cout << "eating" << endl; }
protected:
    const string& getName() const { return name; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const override final { cout << "Cat is "; Pet::eat(); }
};

class Slug : public Pet {};
class Roach : public Pet {};

void funcByVal(Pet apet) { apet.eat(); }
void funcByRef(Pet& apet) { apet.eat(); } // polymorphic

int main() {
    Pet p;
    Cat c;
    Slug s;
    Roach r;

    cout << "Invoking by val: " << endl;
    funcByVal(p);
    funcByVal(c);

    cout << "Invoking by ref: " << endl;
    funcByRef(p);
    funcByRef(c);
}
```

Invoking by val:

eating

eating

Invoking by ref:

eating

Cat is eating

Passing by reference:

- No slicing
- Methods overridden in derived class remain intact.
- → supports polymorphism
- May be achieved by passing a reference or a pointer.

Inheritance – recap cont. 4

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Pet{
public:
    virtual void eat() const { cout << "eating" << endl; }
protected:
    const string& getName() const { return name; }
private:
    string name;
};

class Cat : public Pet {
public:
    void eat() const override final { cout << "Cat is "; Pet::eat(); }
};

class Slug : public Pet {};
class Roach : public Pet {};

void funcByVal(Pet apet) { apet.eat(); }
void funcByRef(Pet& apet) { apet.eat(); } // polymorphic

int main() {
    Pet p;
    Cat c;
    Slug s;
    Roach r;

    cout << "Invoking by val: " << endl;
    funcByVal(p);
    funcByVal(c);

    cout << "Invoking by ref: " << endl;
    funcByRef(p);
    funcByRef(c);
}
```

passed in a pet
Pet: cat is eating
passed in a pet
Pet is eating

Polymorphism:

- i.e. taking multiple forms
- The function “funcByRef” behaves differently based on the object reference passed to it.
- When we passed “c” → Pet: cat is eating
- When we passed “p”, → Pet is eating
- This is a polymorphic call.

Liskov's principle of substitutability

- Presented by Barbara Liskov in 1987
- An object of the base class may be replaced by an object of the derived class without breaking the program.
- e.g. If a function expects an object of type Pet, we should be able to pass it an object of the derived class (Cat) and not break it.
 - If the function misbehaves as a result → your class relationships is in violation of the Liskov substitutability principle (LSP).

Assigning objects

- Derived object assigned to base object:
 - Allowed
 - Results in slicing
- Base object assigned to derived object:
 - Not allowed

Assigning pointers

- Derived pointer assigned to base pointer:
 - Allowed
 - aka “Upcasting”
 - Safe
- Base pointer assigned to derived pointer:
 - Not allowed, unless we force it (use casting)
 - aka “Downcasting”
 - Unsafe and not recommended

Some terminology

- Static type = declared type
- Dynamic type = actual type

- Super class = parent class = base class
- Subclass = child class = derived class

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created, the base class's constructor is executed **first** (whether it's in init list or not) followed by the derived class's constructor
- When an object of a derived class is destroyed, its destructor is called first, then that of the base class.
 - i.e. in reverse order to constructor invocation

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading:

```
Cat::Cat(const string& name="unnamed") : Pet(name) { }
```

- Can also be done with inline constructors
- Must be done if base class has no default constructor

Inheritance – passing args to parent constructor

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Pet {
public:
    Pet(const string& aname = "unnamed") : name(aname) {}
    virtual void eat() const { cout << "eating" << endl; }
protected:
    const string& getName() const { return name; }
private:
    string name;
};

class Cat : public Pet {
public:
    Cat(const string& aname = "unnamed") : Pet(aname) {}
    void eat() const { cout << "Cat " << getName() << " is "; Pet::eat(); }
};

class Slug : public Pet {};
class Roach : public Pet {};

void funcByVal(Pet apet) { apet.eat(); }
void funcByRef(Pet& apet) { apet.eat(); } // polymorphic

int main() {
    Pet p("peeve");
    Cat c("clever");
    Slug s;
    Roach r;

    cout << "Printing the vector vp: " << endl;
    vector<Pet*> vp;
    vp.push_back(&p);
    vp.push_back(&c);
    vp.push_back(&s);
    vp.push_back(&r);
    for (size_t i = 0; i < vp.size(); i++) { vp[i]->eat(); }
}
```

Printing the vector vp:

eating
Cat clever is eating
eating
eating

- Invoking parent's constructor
- Child calling parent method

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class that has objects
- Abstract base class contains at least one pure virtual function:
`virtual void Y() = 0;`
- The `= 0` indicates a pure virtual function
- Must not be inlined in the base class

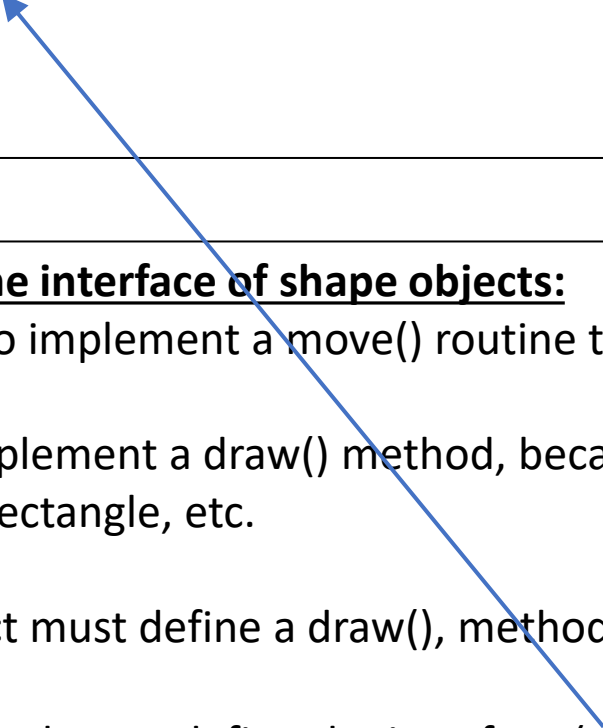
Abstract Base Classes and Pure Virtual Functions

- Abstract base class: class that can have no objects. Serves as a basis for derived classes that may/will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function

Interface example

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

class Shape { // Abstract class
public:
    Shape(int x, int y) : x(x), y(y) {}
    void move(int x, int y) { this->x = x; this->y = y; }
    virtual void draw() = 0; // abstract method = pure virtual method
private:
    int x, y;
};
```



We wish to define the interface of shape objects:

- We may be able to implement a move() routine that is valid for any shape object
- But we cannot implement a draw() method, because its implementation really depends on whether the shape is a circle, rectangle, etc.
- Every shape object must define a draw() method, which takes no parameter
- → Use an abstract class to define the interface (a class with at least one pure virtual function)

Interface example – cont.

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

class Shape { // Abstract class
public:
    Shape(int x, int y) : x(x), y(y) {}
    void move(int x, int y) { this->x = x; this->y = y; }
    virtual void draw() = 0; // abstract method = pure virtual method
private:
    int x, y;
};

class Triangle : public Shape {
public:
    Triangle(int x, int y) : Shape(x,y) {}
    void draw() {
        cout << "Will draw an object of type: ";
        /* stuff to print triangle */
        cout << "triangle" << endl;
    }
};

class Circle : public Shape {
public:
    Circle(int x, int y) : Shape(x,y) {}
    void draw() {
        cout << "Will draw an object of type: ";
        /* stuff to draw a Circle */
        cout << "circle" << endl;
    }
};

int main() {
    Triangle tri(3,4);
    tri.draw();
    Circle circ(10,10);
    circ.draw();
}
```

Will draw an object of type: triangle
Will draw an object of type: circle

- Classes “Triangle” and “Circle” must implement the draw() method, or else we get a compilation error
- → thus we enforced an interface by declaring the pure virtual methods of class “Shape”

Interface example – cont.

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

class Shape { // Abstract class
public:
    Shape(int x, int y) : x(x), y(y) {}
    void move(int x, int y) { this->x = x; this->y = y; }
    virtual void draw() = 0; // abstract method = pure virtual method
private:
    int x, y;
};

void Shape::draw() {
    // useful, common code
    cout << "Will draw an object of type: ";
}

class Triangle : public Shape {
public:
    Triangle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to print triangle */
        cout << "triangle" << endl;
    }
};

class Circle : public Shape {
public:
    Circle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to draw a Circle */
        cout << "circle" << endl;
    }
};

int main() {
    Triangle tri(3,4);
    tri.draw();
    Circle circ(10,10);
    circ.draw();
}
```

Will draw an object of type: triangle
Will draw an object of type: circle

- We can define (implement) a Shape::draw(), but the method is still a pure virtual method
- Useful for implementing common code that may be used by draw() methods of derived classes

Interface example – cont.

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

class Shape { // Abstract class
public:
    Shape(int x, int y) : x(x), y(y) {}
    void move(int x, int y) { this->x = x; this->y = y; }
    virtual void draw() = 0; // abstract method = pure virtual method
private:
    int x, y;
};

void Shape::draw() {
    // useful, common code
    cout << "Will draw an object of type: ";
}

class Triangle : public Shape {
public:
    Triangle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to print triangle */
        cout << "triangle" << endl;
    }
};

class Circle : public Shape {
public:
    Circle(int x, int y) : Shape(x,y) {}
    void draw() {
        Shape::draw();
        /* stuff to draw a Circle */
        cout << "circle" << endl;
    }
};

int main() {
    Shape aShape(3,4);
    Triangle tri(3,4);
    tri.draw();
    Circle circ(10,10);
    circ.draw();
}
```

```
Build started...
1>----- Build started: Project: Lect_01_B, Configuration: Release x64 -----
1>04.abstract_om.cpp
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(60,11): error C2259: 'Shape': cannot instantiate abstract class
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(23): message : see declaration of 'Shape'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(60,11): message : due to following members:
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(60,11): message : 'void Shape::draw(void)': is abstract
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(33): message : see declaration of 'Shape::draw'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(60,17): error C2259: 'Shape': cannot instantiate abstract class
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(23): message : see declaration of 'Shape'
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(60,17): message : due to following members:
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(60,17): message : 'void Shape::draw(void)': is abstract
1>C:\Dropbox\CS2124_OOP_2023_Spring\lect_code\07.Inheritance\04.abstra
ct_om.cpp(33): message : see declaration of 'Shape::draw'
1>Done building project "cs2124.vcxproj" -- FAILED.
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

- Compilation error when trying to instantiate
→ indeed it's still an abstract class despite
defining Shape::draw()