C++ Inheritance, Polymorphism

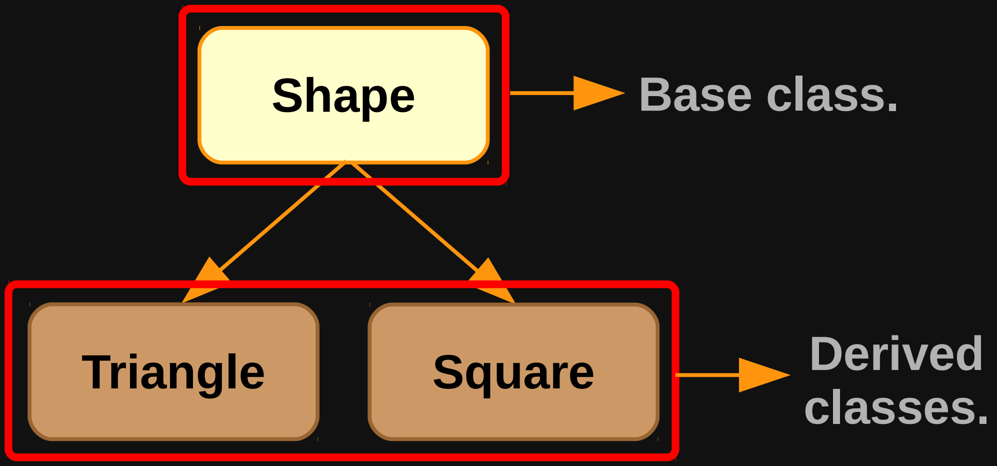
Lecture 8

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

Inheritance

- In C++, inheritance **implements** the idea of “**is a**” relationship.
- Examples



inheritance.cpp

```cpp
class Shape {
}

class Triangle : public Shape {
}

class Square : public Shape {
}
```

Inheritance

Derived classes inherit **function members** and **data members** from the **base class**.

color_
num_vertices_,
vertices_ are inherited from Shape

inheritance.cpp

```
...
class Shape {
public:
    Color color_;
    int num_vertices_ = 0;
    Vertex *vertices_ = nullptr;
}

class Triangle : public Shape {
}

class Square : public Shape {
}
...
```

```
struct Color {
    float color[4];
};
```

```
struct Vertex {
    float vertex[3];
};
```

inheritance.cpp

```
...
int main( void ) {
    Triangle t;
    t.color_[0] = ...;
    t.num_vertices_ = 3;
    t.vertices_ = new Vertex[3];

    Square s;
    s.color_[0] = ...;
    s.num_vertices_ = 4;
    s.vertices_ = new Vertex[4];

    return 0
}
```

What if we want to **protect `color_`** from **public (external)** access?

Inheritance

Accessor

Mutator

inheritance.cpp

```
...
class Shape {
public:
    Color getColor( void ) const {
        return color_;
    }

    void setColor( const Color &color ) {
        color_ = color;
    }

    int num_vertices_ = 0;
    Vertex *vertices_ = nullptr;
private:
    Color color_;
}
```

color_ can be accessed through publicly available member functions

color_ now is **private** and can not be accessed by derived classes

inheritance.cpp

```
...
int main( void ) {
    Triangle t;

    t.color_[0] = ...;

    t.setColor(...);
    Color ct = t.getColor();

    ...

    return 0
}
```

The actual values of **num_vertices_** and **vertices_** will depend on the **specialization** of the **Shape** class (**Triangle** or **Square**). **Is it possible to initialize and, simultaneously, keep them protected from external access**?

Inheritance

How to initialize inherited data members?

inheritance.cpp

```
...
class Shape {
public:
    Color getColor( void ) const {
        return color_;
    }

    void setColor( const Color &color ) {
        color_ = color;
    }

private:
    Color color_;
protected:
    int num_vertices_ = 0;
    Vertex *vertices_ = nullptr;
}
```

Protected members can be accessed from within derived classes, but are not visible externally.

inheritance.cpp

```
...
class Triangle : public Shape {
    Triangle ( void ) :
        num_vertices_( 3 )
    { ... }
}

class Square : public Shape {
    Square ( void ) :
        num_vertices_( 4 )
    { ... }
}
...
```

Inherited data members **cannot** be initialized in the initialization list of **derived classes**...

…because **inherited** data members are **initialized** by the **base class constructor**!

Let's talk a little bit about constructors....

Default Constructors

```
class Dummy1 {
}
```

Implicit default constructor → Inline public member function that will call the default constructors of the base class and non-static data members.

```
class Dummy2 {
public:
    Dummy2( void ) {}
}
```

Explicitly informed default constructor → Public member function that will call the default constructors of the base class and non-static data members.

```
class Dummy3 {
public:
    Dummy3( void ) {}

    Dummy3( int x ) { ... }
}
```

Explicitly informed default constructor

Custom constructor

```
class Dummy4 {
public:
    Dummy4( int x ) { ... }
}
```

Custom constructor → There is no default constructor.

Try declaring: `Dummy4 a{};`

Inheritance and Constructors

Constructors **are not** inherited.

ctorinheritance.cpp

```
class Dummy1 {
    Dummy1( void ) {}
}

class Dummy2 : public Dummy1 {
    Dummy2( void ) {
        Dummy1();
    }
}

int main( void ) {
    Dummy2 a{};
    return 0;
}
```

Try this code!

What happened here?

Actually, the call to `Dummy1()` **is not** interpreted as a call to a **member function** of the class `Dummy2`.
It is actually interpreted as the **creation** of a **temporary object** of the class `Dummy1`.

Inheritance and Constructors

Default base-class constructors are called **automatically** by the **derived-class** constructors.

ctorinheritance.cpp

```
class Dummy1 {
    Dummy1( void ) {
        std::clog << "Dummy1 default ctor..." << std::endl;
    }
}

class Dummy2 : public Dummy1 {
    Dummy2( void ) {
    }
}

...
```

Try this code!

Inheritance and Constructors

Base-class custom constructors can be called from within **initialization lists** of the **derived-class** constructors.

ctorinheritance.cpp

```
class Dummy1 {

    ...

    Dummy1( int x ) {
        std::clog << "Dummy1 custom ctor..." << std::endl;
    }
}

class Dummy2 : public Dummy1 {
    Dummy2( void ) :
        Dummy1{ 1 }
    {}
}

...
```

Try this code!

Back to Our Initial Problem...

... we want to initialize inherited data members!

inheritance.cpp (old)

```
...

class Shape {
public:

    ...

private:
    Color color_;

protected:
    int num_vertices_ = 0;
    Vertex *vertices_ = nullptr;
}
```

inheritance.cpp (new)

```
...

class Shape {
public:
    Shape( int num_vertices ) :
        num_vertices_{ num_vertices }
    {}

    ...

private:
    Color color_;

protected:
    int num_vertices_ = 0;
    Vertex *vertices_ = nullptr;
}
```

1º

We **create custom constructors** at the **base class**, that **initializes its data members**, and...

Back to Our Initial Problem...

inheritance.cpp

1º

```
...
class Shape {
public:
    Shape( int num_vertices ) :
        num_vertices_{ num_vertices }
    {}

    ...
}
```

inheritance.cpp

2º

```
...

class Triangle : public Shape {
public:
  Triangle( void ) :
      Shape{ 3 }
  {
    vertices_ = new Vertex[num_vertices_];
  };

  ~Triangle( void )
  {
    if ( vertices_ )
      delete [] vertices_;
  }

...
```

Consider **allocating** the dynamic arrays in the **base class**!

We must delete the dynamically allocated memory!

...make the **construtors** of the **derived classes** invoke the **base-class custom constructors** that **initialize** the **data members originally defined** at the **base class**.

Do **the same** for the **`Square`** class!

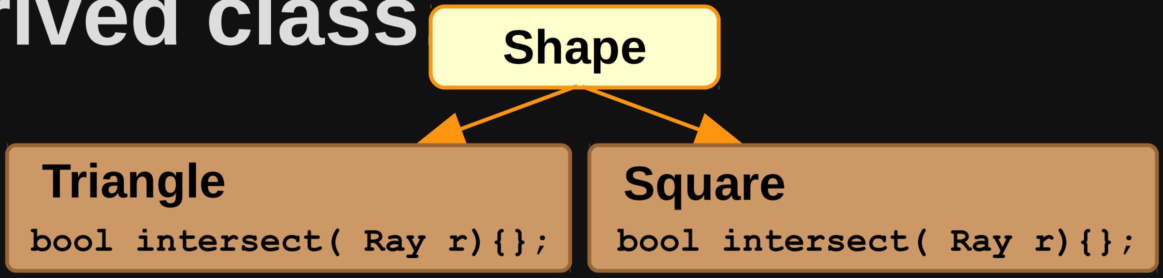
Computing Intersections

Suppose that, for **each shape** that the system is capable of representing, we wish to compute **its intersection point** when intersected with a **ray** (*i.e.* **line segment**).

IMPORTANT: Consider that the **procedure** used to estimate the **intersection point** between a **ray** and a **shape** is **distinct** for **each type of shape!**

1st Approach

Adding an **`intersect()`** member function to **each derived class**



intersect.cpp

```
...

class Triangle : public Shape {
public:

  ...

  bool intersect( const Ray& r ) {
    std::clog << "Intersect test triangle..." << std::endl;
    return true;
  }
  ...
}
```

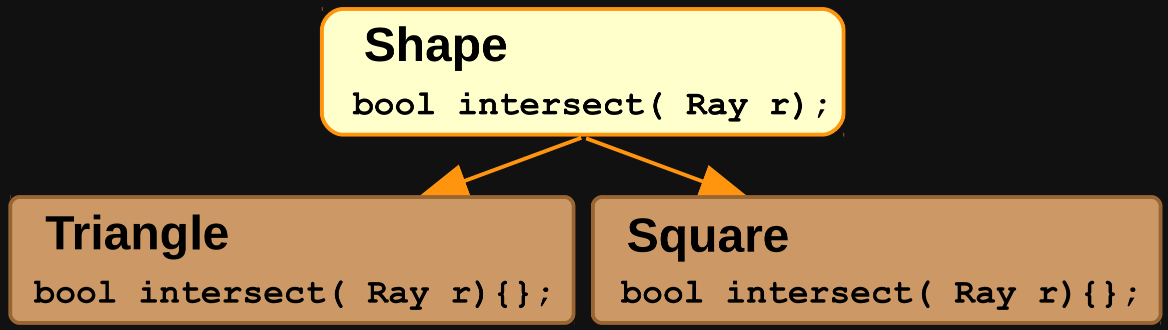
intersect.cpp

```
...

int main( void ) {

    Triangle t;
    t.intersect();

    Square s;
    s.intersect();

    Shape *shape = &t; // or &s
    shape->intersect();
}
```

`intersect()` can not be invoked from a **pointer** to the **base-class!**

2nd Approach

Adding an **`intersect()`** member function to the **base and derived classes**.



intersect.cpp

```
...

class Shape {
public:

  ...

  bool intersect( const Ray& r ) {
    std::clog << "Intersect test shape..." << std::endl;
    return true;
  }
  ...
}
```

intersect.cpp

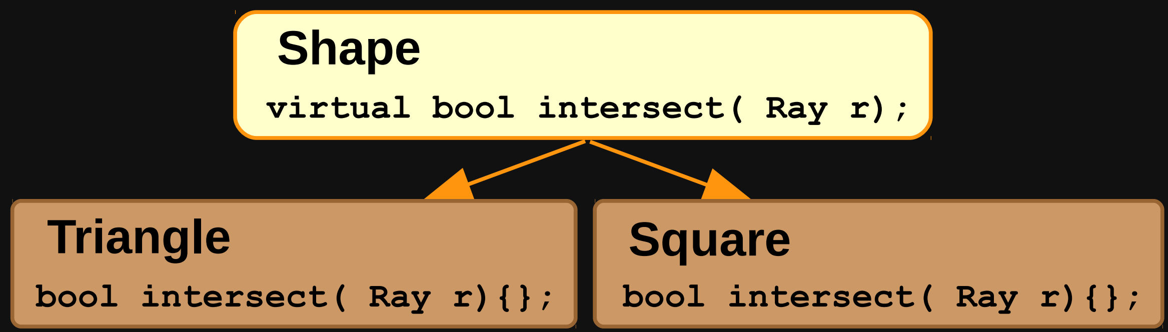
```
...

int main( void ) {

    Triangle t;
    t.intersect();

    Square s;
    s.intersect();

    Shape *shape = &t; // or &s
    shape->intersect();
}
```

The **`intersect()`** version that will be **invoked** is the one **defined** at the **base-class!**

Obviously, **it makes no sense!**

3rd Approach

Adding a **`virtual intersect()`** member function to the base class, and specific implementations at derived classes.



intersect.cpp

```
...

class Shape {
public:
  ...
  virtual bool intersect( const Ray& r ) {
    std::clog << "Intersect test shape..." << std::endl;
    return true;
  }
  ...
}
```

intersect.cpp

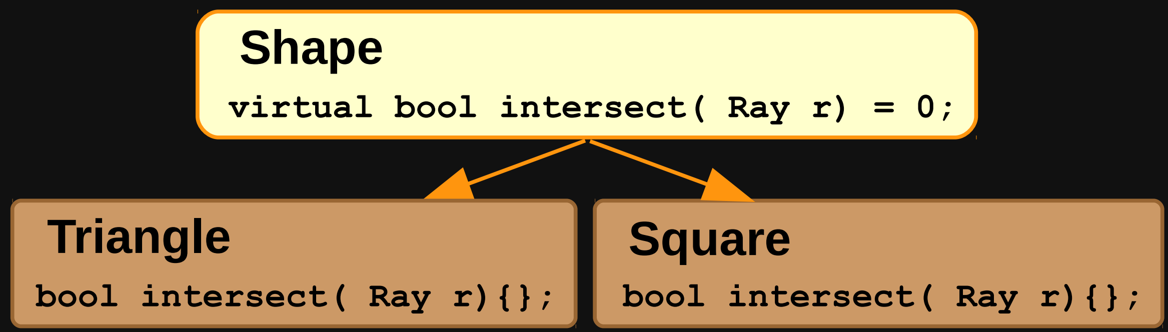
```
...

int main( void ) {

    Triangle t;
    t.intersect();

    Square s;
    s.intersect();

    Shape *shape = &t; // or &s
    shape->intersect();
}
```

The function will be **Invoked correctly** , according to the **object pointed at.**

We can **instantiate `Shape`**!

4th Approach

Adding a **pure `virtual intersect()`** member function to the base class (**abstract**), and specific implementations at derived classes.



intersect.cpp

```
...

class Shape {
public:

  ...

  virtual bool intersect( const Ray& r ) = 0;

  ...

}
```

The **invoked function** is **correct**, and we **can not instantiate `Shape`** (asbtract)!

intersect.cpp

```
...

int main( void ) {

    Triangle t;
    t.intersect();

    Square s;
    s.intersect();

    Shape *shape = &t; // or &s
    shape->intersect();
}
```

Polymorphism!

How Virtual Functions Work?

virtual.cpp

```
#include <iostream>

class Base {
public:
    virtual void f1( void ) {
        std::cout << "Base f() call...." << std::endl;
    }

     virtual void f2( void ) {};
};

class Derived : public Base {
public:
     void f1( void ) {
        std::cout << "Derived f() call...." << std::endl;
    }
};

...
```

virtual.cpp

```
...

int main( void ) {
    Base b;
    b.f1();

    Derived d;
    d.f1();

    Base *ptr;

    ptr = &b;
    ptr->f1();

    ptr = &d;
    ptr->f1();

    return 0;
}
```

What is the output of this code?

How does this work?

How Virtual Functions Work?

Base vtable

```
void f1( void );
void f2( void );
```

Derived vtable

```
void f1( void );
```

virtual.cpp

```
#include <iostream>

class Base {
public:
    *__vptr;
    virtual void f1( void ) {
        std::cout << "Base f1() call." << std::endl;
    }

    virtual void f2( void ) {};
};

class Derived : public Base {
public:
    *__vptr; (inherited)
    void f1( void ) {
        std::cout << "Derived f1() call." << std::endl;
    }
};

...
```

Pointer to the Base class vtable

Pointer to the Derived class vtable

How Virtual Functions Work?

virtual.cpp

```
...

int main( void ) {
    Base b;
    b.f1();

    Derived d;
    d.f1();

    Base *ptr;

    ptr = &b;
    ptr->f1();

    ptr = &d;
    ptr->f1();

    return 0;
}
```

Base vtable

```
void f1( void );
void f2( void );
```

Derived vtable

```
void f1( void );
```

virtual.cpp

```
#include <iostream>

class Base {
public:
  *__vptr;
  virtual void f1( void ) {
      std::cout << "Base f1() call." ...
  }

  virtual void f2( void ) {};
};

class Derived : public Base {
public:
  *__vptr; (inherited)
  void f1( void ) {
      std::cout << "Derived f1() call." ...
  }
};

...
```

What is {not} inherited?

- Derived classes inherit **function members** and **data members** from the “base” class, **except**:
  - Constructors.
  - Destructors.
  - Copy constructors.
  - Overloaded operators.
  - Overloaded function members.