25.x — Chapter 25 summary and quiz

ALEX[1] FEBRUARY 19, 2025

And so our journey through C++'s inheritance and virtual functions comes to an end. Fret not, dear reader, for there are plenty of other areas of C++ to explore as we move forward.

Chapter summary

C++ allows you to set base class pointers and references to a derived object. This is useful when we want to write a function or array that can work with any type of object derived from a base class.

Without virtual functions, base class pointers and references to a derived class will only have access to base class member variables and versions of functions.

A virtual function is a special type of function that resolves to the most-derived version of the function (called an override) that exists between the base and derived class. To be considered an override, the derived class function must have the same signature and return type as the virtual base class function. The one exception is for covariant return types, which allow an override to return a pointer or reference to a derived class if the base class function returns a pointer or reference to the base class.

A function that is intended to be an override should use the override specifier to ensure that it is actually an override.

The final specifier can be used to prevent overrides of a function or inheritance from a class.

If you intend to use inheritance, you should make your destructor virtual, so the proper destructor is called if a pointer to the base class is deleted.

You can ignore virtual resolution by using the scope resolution operator to directly specify which class's version of the function you want: e.g. `base.Base::getName()`.

Early binding occurs when the compiler encounters a direct function call. The compiler or linker can resolve these function calls directly. Late binding occurs when a function pointer is called. In these cases, which function will be called can not be resolved until runtime. Virtual functions use late binding and a virtual table to determine which version of the function to call.

Using virtual functions has a cost: virtual functions take longer to call, and the necessity of the virtual table increases the size of every object containing a virtual function by one pointer.

A virtual function can be made pure virtual/abstract by adding "= 0" to the end of the virtual function prototype. A class containing a pure virtual function is called an abstract class, and can not be instantiated. A class that inherits pure virtual functions must concretely define them or it will also be considered abstract. Pure virtual functions can have a body, but they are still considered abstract.

An interface class is one with no member variables and all pure virtual functions. These are often named starting with a capital I.

A virtual base class is a base class that is only included once, no matter how many times it is inherited by an object.

When a derived class is assigned to a base class object, the base class only receives a copy of the base portion of the derived class. This is called object slicing.

Dynamic casting can be used to convert a pointer to a base class object into a pointer to a derived class object. This is called downcasting. A failed conversion will return a null pointer.

The easiest way to overload operator<< for inherited classes is to write an overloaded operator<< for the most-base class, and then call a virtual member function to do the printing.

Quiz time

1. Each of the following programs has some kind of defect. Inspect each program (visually, not by compiling) and determine what is wrong with the program. The output of each program is supposed to be “Derived”.

1a)

```
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    const char* getName() const { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() const { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base& b{ d };
    std::cout << b.getName() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[2]

1b)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    virtual const char* getName() const { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base& b{ d };
    std::cout << b.getName() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[2]

1c)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() override { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base b{ d };
    std::cout << b.getName() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[2]

1d)

```cpp
#include <iostream>

class Base final
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() override { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base& b{ d };
    std::cout << b.getName() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[2]

1e)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    virtual const char* getName() = 0;
};

const char* Derived::getName()
{
    return "Derived";
}

int main()
{
    Derived d{ 5 };
    Base& b{ d };
    std::cout << b.getName() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))[2]

1f)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    virtual const char* getName() { return "Derived"; }
};

int main()
{
    auto* d{ new Derived(5) };
    Base* b{ d };
    std::cout << b->getName() << '\n';
    delete b;

    return 0;
}
```

Show Solution (javascript:void(0))[2]

2a) Create an abstract class named Shape. This class should have three functions: a pure virtual print function that takes and returns a std::ostream&, an overloaded operator<< and an empty virtual destructor.

Show Solution (javascript:void(0))[2]

2b) Derive two classes from Shape: a Triangle, and a Circle. The Triangle should have 3 Points as members. The Circle should have one center Point, and an integer radius. Override the print() function so the following program runs:

```cpp
int main()
{
    Circle c{ Point{ 1, 2 }, 7 };
    std::cout << c << '\n';

    Triangle t{Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6 }};
    std::cout << t << '\n';

    return 0;
}
```

This should print:

```
Circle(Point(1, 2), radius 7)
Triangle(Point(1, 2), Point(3, 4), Point(5, 6))
```

Here's a Point class you can use:

```
class Point
{
private:
    int m_x{};
    int m_y{};

public:
    Point(int x, int y)
        : m_x{ x }, m_y{ y }
    {

    }

    friend std::ostream& operator<<(std::ostream& out, const Point& p)
    {
        return out << "Point(" << p.m_x << ", " << p.m_y << ')';
    }
};
```

Show Solution (javascript:void(0))[2]

2c) Given the above classes (Point, Shape, Circle, and Triangle), finish the following program:

```
#include <vector>
#include <iostream>

int main()
{
    std::vector<Shape*> v{
      new Circle{Point{ 1, 2 }, 7},
      new Triangle{Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6 }},
      new Circle{Point{ 7, 8 }, 3}
    };

    // print each shape in vector v on its own line here

    std::cout << "The largest radius is: " << getLargestRadius(v) << '\n'; // write
this function

    // delete each element in the vector here

    return 0;
}
```

The program should print the following:

```
Circle(Point(1, 2), radius 7)
Triangle(Point(1, 2), Point(3, 4), Point(5, 6))
Circle(Point(7, 8), radius 3)
The largest radius is: 7
```

Hint: You'll need to add a getRadius() function to Circle, and downcast a Shape* into a Circle* to access it.

Show Solution (javascript:void(0))[2]

2d) Extra credit: Update the prior solution to use a `std::vector<std::unique_ptr<Shape>>`. Remember that `std::unique_ptr` is not copyable.

h/t to reader surrealcereal for this idea.

Show Hint (javascript:void(0))[2]

Show Hint (javascript:void(0))[2]

Show Solution (javascript:void(0))[2]

Next lesson
26.1 Template classes

3

Back to table of contents

4

Previous lesson
25.11 Printing inherited classes using operator<<

5

6

AJAY.CHALLA

June 14, 2025 10:17 pm PDT

`QUIZ TIME!!`

1a)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    const char* getName(){ return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() const { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base& b{ d };
    std::cout << d.getName() << '\n';

return  0;
}
```

1b)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    virtual const char* getName() const { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base& b{ d };
    std::cout << d.getName() << '\n';

    return 0;
}
```

1c)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() override { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base b{ d };
    std::cout << d.getName() << '\n';

    return 0;
}
```

1d)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() override { return "Derived"; }
};

int main()
{
    Derived d{ 5 };
    Base& b{ d };
    std::cout << b.getName() << '\n';

    return 0;
}
```

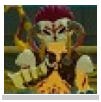
1e)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    const char* getName() override
    {
        return "Derived";
    }
};

int main(){
    Derived d{ 5 };
    Base& b{ d };
    std::cout << b.getName() << '\n';

    return 0;
}
```

1f)

```cpp
#include <iostream>

class Base
{
protected:
    int m_value;

public:
    Base(int value)
        : m_value{ value }
    {
    }

    virtual const char* getName() { return "Base"; }
};

class Derived : public Base
{
public:
    Derived(int value)
        : Base{ value }
    {
    }

    virtual const char* getName() { return "Derived"; }
};

int main()
{
    auto* d{ new Derived(5) };
    Base* b{ d };
    std::cout << b->getName() << '\n';
    delete b;

    return 0;
}
```

0 Reply



Sebby

June 11, 2025 8:08 am PDT

```cpp
#include <algorithm>
#include <iostream>
#include <memory>
#include <vector>

struct Point {
    int x;
    int y;
};
std::ostream& operator<<(std::ostream& out, const Point& point) {
    return out << "Point(" << point.x << ", " << point.y << ")";
}
class Shape {
  public:
    friend std::ostream& operator<<(std::ostream& out, const Shape& shape) {
        return shape.print(out);
    }

    virtual std::ostream& print(std::ostream& out) const = 0;
    virtual ~Shape() = default;
};
class Circle : public Shape {
    Point m_center{};
    int m_radius{};

  public:
    Circle(Point center, int radius) : m_center{center}, m_radius{radius} {
    }
    std::ostream& print(std::ostream& out) const override {
        return out << "Circle(" << m_center << ", radius " << m_radius << ')';
    }
    int radius() const {
        return m_radius;
    }
};
class Triangle : public Shape {
    Point m_vertex1{};
    Point m_vertex2{};
    Point m_vertex3{};

  public:
    Triangle(Point vertex1, Point vertex2, Point vertex3) : m_vertex1{vertex1},
m_vertex2{vertex2}, m_vertex3{vertex3} {
    }
    std::ostream& print(std::ostream& out) const override {
        return out << "Triangle(" << m_vertex1 << ", " << m_vertex2 << ", " <<
m_vertex3 << ')';
    }
};

int getLargestRadius(const std::vector<Shape*>& shapes) {
    std::vector<int> radii{};
    for (const auto& shape : shapes) {
        Circle* circle{dynamic_cast<Circle*>(shape)};
        if (!circle) {
            radii.push_back(0);
        } else {
            radii.push_back(circle->radius());
        }
    }
    std::sort(radii.begin(), radii.end(), [](int a, int b) { return (a > b); });
    if (radii.empty())
        return 0;
    return radii[0];
}
int getLargestRadius(const std::vector<std::unique_ptr<Shape>>& shapes) {
    std::vector<int> radii{};
    for (const auto& shape: shapes) {
        Circle* circle{dynamic_cast<Circle*>(shape.get())};
        if (!circle) {
```

```
            radii.push_back(0);
        } else {
            radii.push_back(circle->radius());
        }
    }
    std::sort(radii.begin(), radii.end(), [](int a, int b) { return (a > b); });
    if (radii.empty())
        return 0;
    return radii[0];
}

int main() {
    Circle c{Point{1, 2}, 7};
    std::cout << c << '\n';
    Triangle t{Point{1, 2}, Point{3, 4}, Point{5, 6}};
    std::cout << t << '\n';
    std::cout << "--------------------" << '\n';
    std::vector<Shape*> v{new Circle{Point{1, 2}, 7}, new Triangle{Point{1, 2},
Point{3, 4}, Point{5, 6}}, new Circle{Point{7, 8}, 3}};
    for (const auto* shapePtr : v) {
        if (shapePtr)
            std::cout << *shapePtr << '\n';
    }
    std::cout << "The largest radius is: " << getLargestRadius(v) << '\n';
    for (const auto* shapePtr : v) {
        delete shapePtr;
        shapePtr = nullptr;
    }
    std::cout << "----------------" << '\n';
    std::vector<std::unique_ptr<Shape>> v2{};
    v2.push_back(std::make_unique<Circle>(Circle{Point{1, 2}, 7}));
    v2.push_back(std::make_unique<Triangle>(Triangle{Point{1, 2}, Point{3, 4},
Point{5, 6}}));
    v2.push_back(std::make_unique<Circle>(Circle{Point{7, 8}, 3}));
    for (const auto& shapePtr : v2) {
        if (shapePtr.get())
            std::cout << *shapePtr << '\n';
    }
    std::cout << "The largest radius is: " << getLargestRadius(v2) << '\n';
    return 0;
}
```

Last edited 28 days ago by Sebby

0 Reply

Sanderson

April 21, 2025 10:39 am PDT

I had a question here: why can't we simply declare the vector using a initializer list and make the unique pointers there? There shouldn't be any copying operation I think?

```cpp
#include <algorithm>
#include <iostream>
#include <memory>
#include <vector>

class Shape
{
public:
    Shape() = default;
    virtual ~Shape() = default;
    virtual std::ostream& print(std::ostream& os) const = 0;
    friend std::ostream& operator<<(std::ostream& os, const Shape& s);
};

std::ostream& operator<<(std::ostream& os, const Shape& s)
{
    return s.print(os);
}

struct Point
{
    int x {};
    int y {};
};

std::ostream& operator<<(std::ostream& os, const Point& p)
{
    return os << "Point(" << p.x << ", " << p.y << ')';
}

class Triangle : public Shape
{
public:
    Triangle(const Point& p1, const Point& p2, const Point& p3)
        : m_p1 { p1 }, m_p2 { p2 }, m_p3 { p3 } {}
    std::ostream& print(std::ostream& os) const override
    {
        return os << "Triangle(" << m_p1 << ", " << m_p2 << ", " << m_p3 << ")";
    }
private:
    Point m_p1 {};
    Point m_p2 {};
    Point m_p3 {};
};

class Circle : public Shape
{
public:
    Circle(const Point& center, int radius)
        : m_center { center }, m_radius { radius } {}
    std::ostream& print(std::ostream& os) const override
    {
        return os << "Circle(" << m_center << ", radius " << m_radius << ")";
    }
    const int getRadius() const { return m_radius; }
private:
    Point m_center {};
    int m_radius {};
};

int getLargestRadius(const std::vector<Shape*>& shapes)
{
    if (shapes.empty()) return -1;
    auto largest {
        std::max_element(shapes.begin(), shapes.end(), [](const Shape* a, const Shape*
b) -> bool
        {
            auto circleA { dynamic_cast<const Circle*>(a) };
            auto circleB { dynamic_cast<const Circle*>(b) };
            if (circleA != nullptr && circleB != nullptr)
```

```
            {
                return circleA->getRadius() <= circleB->getRadius();
            }
            return false;
        })
    };
    if (largest != shapes.end())
        return dynamic_cast<Circle*>(*largest)->getRadius();
    return -1;
}

// int getLargestRadius2(const std::vector<Shape*>& shapes)
// {
//     if (shapes.empty()) return -1;
//     int largest {};
//     for (const auto* shape : shapes)
//     {
//         if (auto circle { dynamic_cast<const Circle*>(shape) })
//             largest = std::max(largest, circle->getRadius());
//     }
//     return largest;
// }

// Smart Pointer version
int getLargestRadius2(const std::vector<std::unique_ptr<Shape>>& shapes)
{
    if (shapes.empty()) return -1;
    int largest {};
    for (const auto& shape : shapes)
    {
        if (auto circle { dynamic_cast<const Circle*>(shape.get()) })
            largest = std::max(largest, circle->getRadius());
    }
    return largest;
}

int main()
{
    std::vector<Shape*> v{
        new Circle{Point{ 1, 2 }, 7},
        new Triangle{Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6 }},
        new Circle{Point{ 7, 8 }, 3}
    };

    std::vector<std::unique_ptr<Shape>> v2{};
    v2.push_back(std::make_unique<Circle>(Point{ 1, 2 }, 7));
    v2.push_back(std::make_unique<Triangle>(Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6
}));
    v2.push_back(std::make_unique<Circle>(Point{ 7, 8 }, 3));

    // Compilation error if declare as follows using a initializer list. Why?
   //  std::vector<std::unique_ptr<Shape>> v2{
   //     std::make_unique<Circle>(Point{ 1, 2 }, 7),
   //     std::make_unique<Triangle>(Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6 }),
   //     std::make_unique<Circle>(Point{ 7, 8 }, 3)
   // };
    // print each shape in vector v on its own line here
    for (const auto& p : v2)
    // {
    //     p->print(std::cout);
    //     std::cout << '\n';
    // }
        // std::cout << *p << '\n';
        std::cout << *(p.get()) << '\n';

    std::cout << "The largest radius is: " << getLargestRadius2(v2) << '\n'; // write
this function

    // delete each element in the vector here
    // for (auto p : v)
```

```
141 |       //      delete p;
        return 0;
    }
```

Nyac

March 14, 2025 12:25 am PDT

```cpp
/*About 2c another way , my add some new functions of find the max Radius of Circle*/
//class Point {/*No change*/};
//class Shape{/*No change*/};

class Circle:public Shape,public Point
{
    private:
    int radius{};
    public:
    int getRadius()
    {
        return radius;
    }

};
//This new add
inline auto getMoreRadius( Circle*a,Circle*b)
{
   auto Radiusa{a->getRadius()};
   auto Radiusb{b->getRadius()};
   auto LagrgeRadius{Radiusa>Radiusb?Radiusa:Radiusb};

   return LagrgeRadius;
}
//This function get the max radius of rhe Circle

//This function for cast the //Circle*
inline auto CasttoCircle(Shape*&shape)
{
   auto diver{dynamic_cast<Circle*>(shape)};
   return diver;
}
//new add;


//New add
inline auto findBiggestRadius(Shape*a,Shape*b)
{
    auto Smallradius{CasttoCircle(a)};
    auto Biggestradius{CasttoCircle(b)};
    auto isbiggerest{getMoreRadius(Smallradius,Biggestradius)} ;
    std::cout<<"the max radius of the circle is:"<<isbiggerest<<'\n';
    return isbiggerest;
}
//This function for print max //Radius of Circle;


template<class T>
auto getLargesRadius( std::vector<T>shapes)
{
      int a{};
     auto length{shapes.size()};
     auto intLength{static_cast<int>(length)};
     for (;a<intLength; a++)
     {
         std::cout<<*shapes[a];
     }
       std::cout<<'\n';

     return shapes;


}
//This function my way is not same to alex;
```

Last edited 3 months ago by Nyac

0 Reply

Nyac

Reply to Nyac [9] March 14, 2025 12:32 am PDT

```cpp
//This is completely code version;


#include <iostream>

#include <memory>
#include <ostream>

#include <string>
#include <vector>



class Shape
{

    public:
   const  auto getstr(std::string&str)const{return str;};
   auto length(int a)const{return a;};
    virtual std::ostream&print(std::ostream&out)const=0;
    friend std::ostream&operator<<(std::ostream&out,const Shape&shape)
    {
        return shape.print(out);
    }

     virtual ~Shape()=default;

    /*friend  bool operator<(const Shape&shape1,const Shape&shape2)
     {
         std::string a{"a"};
         std::string z{"z"};
         return shape1.getstr(a)<shape2.getstr(z);
     }*/

   /*friend  bool operator>(const Shape&shape1,const Shape&shape2)
     {
         std::string a{"a"};
         std::string z{"z"};
         return shape1.getstr(a)>shape2.getstr(z);
     }*/




};

/*auto getbigger( Shape*&shape, Shape*&shape1)
{
     int a{};
     int b{};
     return shape->length(a)>shape1->length(b);

}*/
 class Point
 {
     public:
    int m_x{};
    int m_y{};
    public:
    Point(int x=0,int y=0):m_x{x},m_y{y}{}
    friend std::ostream&operator<<(std::ostream&out,const Point&point)
    {
          out<<"Point("<<point.m_x<<','<<point.m_y<<')';
        return out;
    }
    auto setPoint( const Point&p)const
    {
       return p;

    }
```

```
};
class Circle:public Shape,public Point
{
   private:
   Point m_point{};
   int m_radius{};
   public:
   Circle(Point points=0,int radius=0):m_point{points},m_radius{radius}{}
  std::ostream&print(std::ostream&out) const override
   {

               out<<"Circle("<<m_point<<','<<"radiues "<<m_radius<<")\n";
               return out;
   }
   int getRadius()const
   {

      // auto LagrgeRadius{c.m_radius>c1.m_radius?c.m_radius:c1.m_radius};
     //  std::cout<<LagrgeRadius<<'\n';
        return m_radius;
   }




};
inline auto getMoreRadius( Circle*a,Circle*b)
{
   auto Radiusa{a->getRadius()};
   auto Radiusb{b->getRadius()};
   auto LagrgeRadius{Radiusa>Radiusb?Radiusa:Radiusb};

   return LagrgeRadius;



}
template<class U>
inline auto CasttoCircle(U*&shape)
{
   auto diver{dynamic_cast<Circle*>(shape)};
   return diver;
}
inline auto CasttoCircle(std::unique_ptr<Shape>&shape)
{
  auto diver{dynamic_cast<Circle*>(shape.get())};
  return diver;
}


class Triangle:public Shape,public Point
{
   private:

   Point m_pointa{};
   Point m_pointb{};
   Point m_pointc{};

   public:
   Triangle(Point pointa,Point pointb,Point
pointc):m_pointa(pointa),m_pointb(pointb),m_pointc(pointc){}
   std::ostream&print(std::ostream&out)const override
   {

       out<<"Triangle("<<m_pointa<<','<<m_pointb<<','<<m_pointc<<")\n";
       return out;
   }
   friend class Point;
```

```
 };


auto printShape(const Shape&shape)
{
    std::cout<<shape;


}
template<class T>
auto getLargesRadius(   const T&shapes)
{

     int result{};


        for (const auto*shape :shapes)
        {
            std::cout<<*shape;
            if(auto*ca{  dynamic_cast< const Circle*>(shape)})
             {
                 result=std::max(result,ca->getRadius());
            };


    }









        return result;
}

inline auto findBiggestRadius(Shape*a,Shape*b)
{
    auto Smallradius{CasttoCircle(a)};
    auto Biggestradius{CasttoCircle(b)};
    auto isbiggerest{getMoreRadius(Smallradius,Biggestradius)} ;
    std::cout<<"the radius of the circle is:"<<isbiggerest<<'\n';
    return isbiggerest;
}
auto findBiggestRadius(std::unique_ptr<Shape>a,std::unique_ptr<Shape>b)
{
    auto Smallradius{CasttoCircle(a)};
    auto Biggestradius{CasttoCircle(b)};
    auto isbiggerest{getMoreRadius(Smallradius,Biggestradius)} ;
    std::cout<<"the max  radius of the circle is:"<<isbiggerest<<'\n';
    return isbiggerest;

}


template<class T>
auto getLargesRadius( std::vector<T>shapes)
{
      int a{};
     auto length{shapes.size()};
     auto intLength{static_cast<int>(length)};
     for (;a<intLength; a++)
     {
         std::cout<<*shapes[a];
     }
      // std::cout<<'\n';
```

Hi Alex
think it
if you v
0

For 1f v
0

Took m
But at l
1

```cpp
#include <iostream>
#include <vector>
#include <memory>

class Point
{
private:
    int m_x{};
    int m_y{};

public:
    Point(int x, int y) : m_x{x}, m_y{y}
    {
    }

    friend std::ostream& operator<<(std::ostream& out, const Point& p)
    {
        return out << "Point(" << p.m_x << ", " << p.m_y << ')';
    }
};

class Shape
{
public:

    virtual std::ostream& printingSmthn(std::ostream& out) const = 0;

    friend std::ostream& operator<<(std::ostream& out, const Shape& shape)
    {
        return shape.printingSmthn(out);
    }

    virtual ~Shape() = default;

};

class Triangle: public Shape
{
private:
    Point m_point1;
    Point m_point2;
    Point m_point3;

public:
    Triangle(const Point& a, const Point& b, const Point& c)
     : m_point1{a}, m_point2{b}, m_point3{c}
    {
    }

    std::ostream& printingSmthn(std::ostream& out) const override
    {
        return out << "Triangle(" << m_point1 << ", " << m_point2 << ", " << m_point3
<< ')';
    }

};

class Circle: public Shape
{
private:
    Point m_centerPoint;
    int m_radius{};

public:
    Circle(Point center, int rad = 0)
     : m_centerPoint{center}, m_radius{rad}
     {
     }

     std::ostream& printingSmthn(std::ostream& out) const override
```

```
    {
        return out << "Circle(" << m_centerPoint << "Radius " << m_radius << ')';
    }

    int getRadius() const
    {
        return m_radius;
    }
};

int getLargestRadius(const std::vector<Shape*>& vec)
{
    int larger{-1};

    for (const auto& a: vec)
    {
        Circle* c{dynamic_cast<Circle*>(a)};

        if (c)
        {
            int num{c->getRadius()};

            if (num > larger)
                larger = num;
        }
    }

    return larger;
}

int main()
{
    std::vector<Shape*> v{
      new Circle{Point{ 1, 2 }, 7},
      new Triangle{Point{ 1, 2 }, Point{ 3, 4 }, Point{ 5, 6 }},
      new Circle{Point{ 7, 8 }, 3}
    };

    std::cout << *v[0] << '\n'
    << *v[1] << '\n'
    << *v[2] << '\n';

    std::cout << "The largest radius is: " << getLargestRadius(v) << '\n';

    for (const auto& a: v)
    {
        delete a;
    }

    return 0;
}
```

0 Reply

Asicx

October 5, 2024 2:33 pm PDT

Whoever came up with the `unique_pointer` idea is a sadist (just kidding:).
Any idea why `std::make_unique` doesn't work in this case, even if it is the recommended way to create unique_pointers ? The compiler log is hard to decipher.

0 Reply

Alex Author

Reply to Asicx [12] October 7, 2024 10:31 am PDT

Where is std::make_unique not working for you?

In quiz 2d, you can replace the 3 emplace_back calls with this if you prefer:

```
v.emplace_back(std::make_unique<Circle>(Point{1, 2}, 7));
v.emplace_back(std::make_unique<Triangle>(Point{1, 2}, Point{3, 4}, Point{5,
6}));
v.emplace_back(std::make_unique<Circle>(Point{7, 8}, 3));
```

1 Reply

Asicx

Reply to Alex [13] October 11, 2024 9:35 am PDT

Yeah, it does compile using your code. Weird, i was sure i tried all combinations but maybe i just did something like this: `std::make_unique<Shape>(...)`.

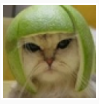
0 Reply

Masdas

October 3, 2024 2:21 pm PDT

In 2b) when you say "Overload the print() function", don't you mean override?

0 Reply



Alex Author

Reply to Masdas [14] October 3, 2024 7:05 pm PDT

Yep. Thanks for the correction!

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. javascript:void(0)
3. https://www.learncpp.com/cpp-tutorial/template-classes/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/
6. https://www.learncpp.com/chapter-25-summary-and-quiz/

7. https://www.learncpp.com/cpp-tutorial/function-template-specialization/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/chapter-25-summary-and-quiz/#comment-608526
10. https://www.learncpp.com/cpp-tutorial/chapter-25-summary-and-quiz/#comment-607855
11. https://www.learncpp.com/cpp-tutorial/chapter-25-summary-and-quiz/#comment-607830
12. https://www.learncpp.com/cpp-tutorial/chapter-25-summary-and-quiz/#comment-602732
13. https://www.learncpp.com/cpp-tutorial/chapter-25-summary-and-quiz/#comment-602809
14. https://www.learncpp.com/cpp-tutorial/chapter-25-summary-and-quiz/#comment-602669
15. https://g.ezoic.net/privacy/learncpp.com