Lecture #11 | Polymorphism: virtual functions

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Previously in Object-Oriented Programming

- OOP features
  - Abstraction
  - Inheritance
  - Polymorphism
  - Encapsulation
- Inheritance

Today's topic

- Miscellaneous C++ topics
  - Types of errors
  - (Implicit) type conversion in C++
  - -> operator
- A few comments on Object-Oriented Programming
- Virtual function
  - Pure virtual functions
  - Abstract class

Types of errors

- Syntax errors (or syntactic errors)
  - If a program does not follow the syntax of a programming language
  - Detected by compiler/interpreter before executing the program
  - Easiest to find and fix
- Runtime errors
  - Errors occur as the program executes
  - Errors are not detected until the flow of control reaches the problematic area of codes
  - Typically errors occurs when a certain condition holds (e.g., divide by zero)
- Semantic errors (or logical errors)
  - Usually program runs without any error, but it does not produce correct answers
  - Caused by a wrong design or implementation of the program
  - Most difficult to find and fix

Reference: http://www.edugrabs.com/types-of-errors-in-c-runtime-error/

Type conversion

- C++ implicitly converts type of operands
  - Promotion: a value in a smaller variable is assigned to a larger variable with no loss of data
  - Narrowing conversion (coercing): type conversions that lose information, e.g., `double` to `int`, `int` to `char`
    - Mainly due to C compatibility, and you need to pay attention to warnings!!!
    - Initialization list prevents implicit conversions

```
int i1 = 7.2;   // i1 becomes 7 (surprise?)
int i2 {7.2};   // error: floating-point to integer conversion
int i3 = {7.2}; // error: floating-point to integer conversion
(the = is redundant)
```

Reference: https://docs.microsoft.com/en-us/cpp/cpp/type-conversions-and-type-safety-modern-cpp

Type conversion (cont.)

- In many control flows where bool type is expected, contextual conversions are performed, e.g.,

  ```
  if (expression)
  ```

- *expression* is considered as `true` when it is evaluated as
  - `true`
  - a non-null pointer
  - any non-zero arithmetic value
  - a class type that defines an unambiguous conversion to an arithmetic, boolean or pointer type
- References
  - http://en.cppreference.com/w/cpp/language/implicit_conversion
  - https://docs.microsoft.com/en-us/cpp/cpp/if-else-statement-cpp

`->` operator and `this` pointer

- `-> operator`
  - `ptr->member` is an abbreviation of `(*ptr).member`
- `this` pointer
  - `this` points to the current object, where the pointer is used
  - When we mention member variables in member functions, we have used it implicitly

Once again… Why OOP?

- OOP is one of the mechanisms that provide SoC
  - In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern
- One of the mechanism to provide SoC is a separation of an interface and an implementation, roughly speaking
  - Interface (or design): class declaration (.h)
  - Implementation: class definition (.cpp)
- By separating an interface and an implementation, you can change (hopefully improve) your implementation without changing the interface
- Another important design principle of classes is keeping a class invariant (or a type invariant), i.e., an invariant used to constrain objects of a class
  - Will cover this later again when mentioning encapsulation

Reference: https://en.wikipedia.org/wiki/Separation_of_concerns

Pointers/references to the base class type of derived object

- A derived class is a base class, meaning it has all the member variable & functions
- Therefore, a derived object (i.e., an object of the derived class) can be referenced by the pointer to the base class

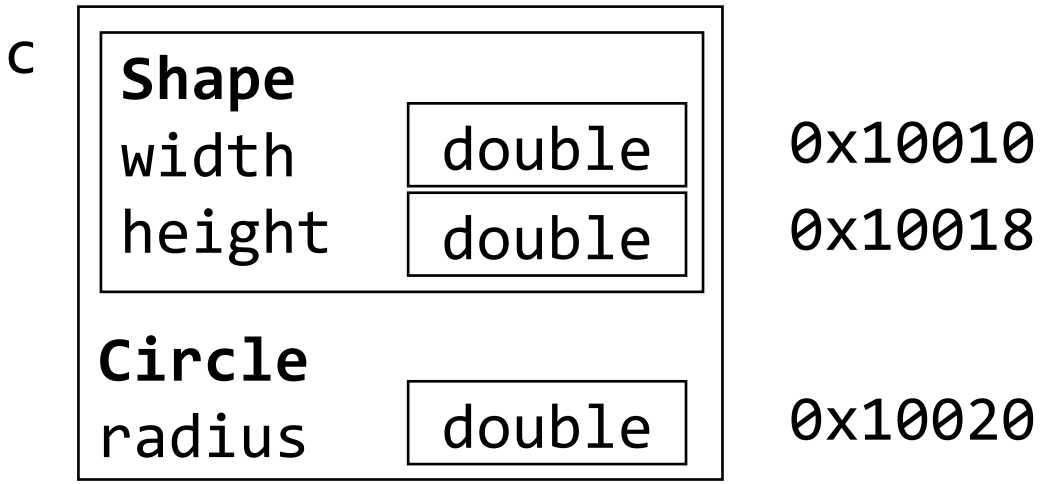
C++ object model

```
class Shape {
protected:
    double width;
    double height;
};

class Circle : public Shape {
private:
    double radius;
};

int main()
{
    Circle c;
}
```

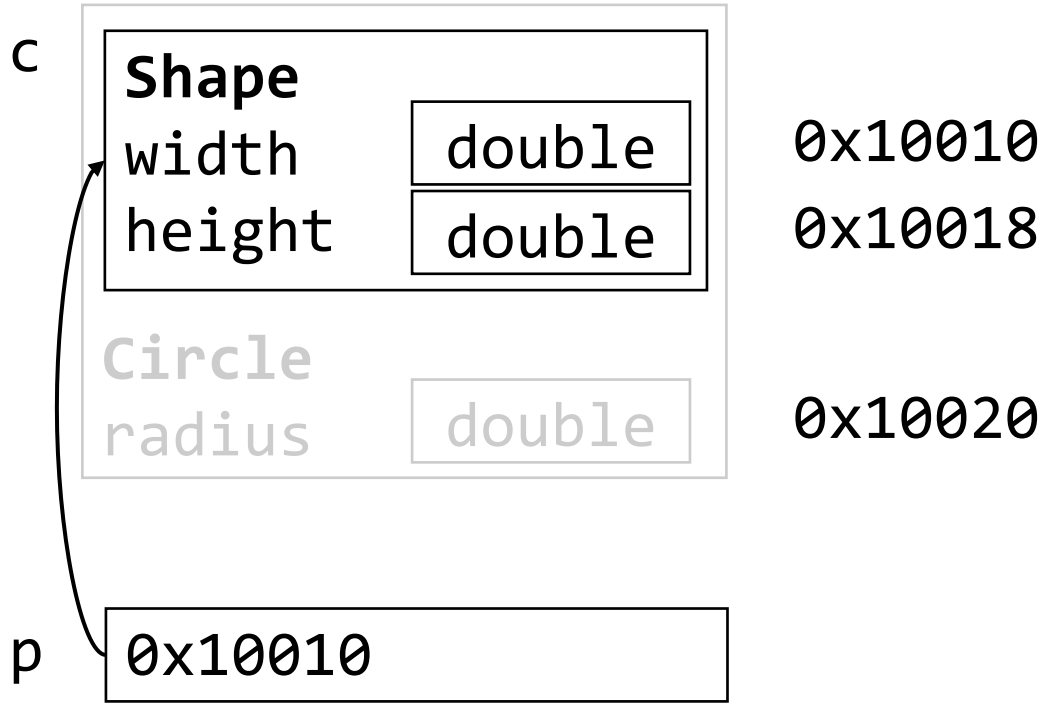
- What is stored in the memory
  - When e.g., `sizeof(double)==8`



C++ object model (cont.)

```
class Shape {
protected:
    double width;
    double height;
};
class Circle : public Shape {
private:
    double radius;
};
int main()
{
    Circle c;
    Shape* p = &c;
}
```

- What p and value pointed by p looks like to the compiler
  - Member variable/functions defined by Circle (e.g., radius) is NOT accessible with p



Example: shapes (from the previous lecture)

```
constexpr double pi = 3.1415926535;
class Shape {
protected:
    double width;
    double height;
public:
    Shape(double w, double h): width{w},
height{h} {}
    double getArea() { return 0.}
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h):
Shape{w, h} {}
    double getArea() { return width *
height; }
};
```

```
class Square : public Rectangle {
public:
    Square(double length) :
Rectangle{length, length} {}
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double radius) : Shape{radius
* 2., radius * 2.} { this->radius =
radius; }
    double getArea() { return width *
width * pi / 4.; }
    double getRadius() { return radius; }
};
```

Example: what would be the results?

```
#include "shapes.h"
int main()
{
    Shape* shapes[3];
    shapes[0] = new Rectangle(1, 2);
    shapes[1] = new Square(3);
    shapes[2] = new Circle(2);
    for (int i = 0; i < 3; i++)
        cout << "Area of shapes[" << i << "]="
             << shapes[i]->getArea() << endl;
}
```

What happens in the previous example

- Why aren't the proper `getArea()` functions invoked?
  - `shapes[i]` are pointers to the base class (`Shape`)
  - When function `getArea()` is invoked, `Shape::getArea()` is used for all the instances
- So what can be the remedy?
  - Use (lots of) if-else…?
  - Use virtual functions

Virtual functions and abstract classes

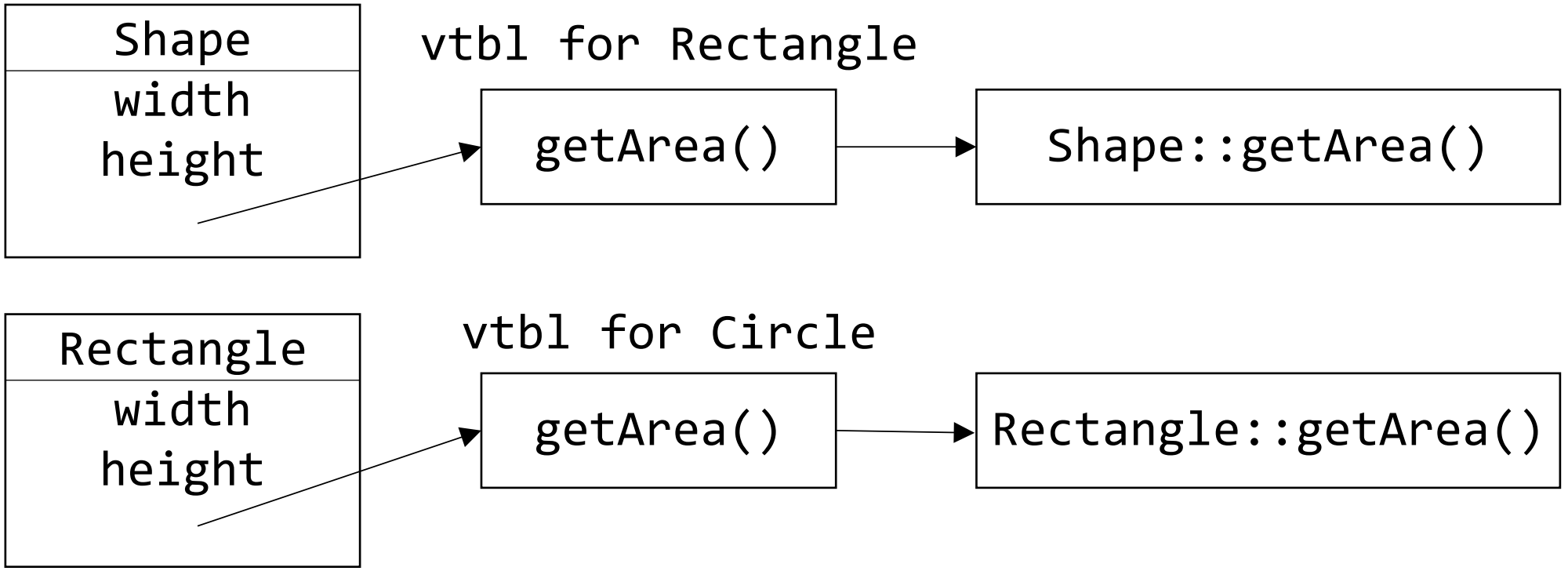
- ***Virtual functions*** allow the programmer to declare functions in a base class that can be redefined in each derived class
- A function from a derived class with the same name and the same set of argument types as a virtual function in a base is said to ***override*** the base class version of the virtual function
- ***A pure virtual function*** is a virtual function that is not defined in a base class
- A class with one or more pure virtual functions is ***an abstract class***, and no objects of that abstract class can be created

Example: virtual functions

```
class Shape {
    // ...
public:
    virtual double getArea() {}
    // virtual double getArea() = 0; // pure virtual function
};
int main() {
    Shape* arrayOfShapes[3];
    arrayOfShapes[0] = new Rectangle(1, 2);
    arrayOfShapes[1] = new Square(3);
    arrayOfShapes[2] = new Circle(2);
    for (int i = 0; i < 3; i++)
        cout << "Area of arrayOfShapes[" << i << "]="
             << arrayOfShapes[i]->getArea() << endl;
}
```

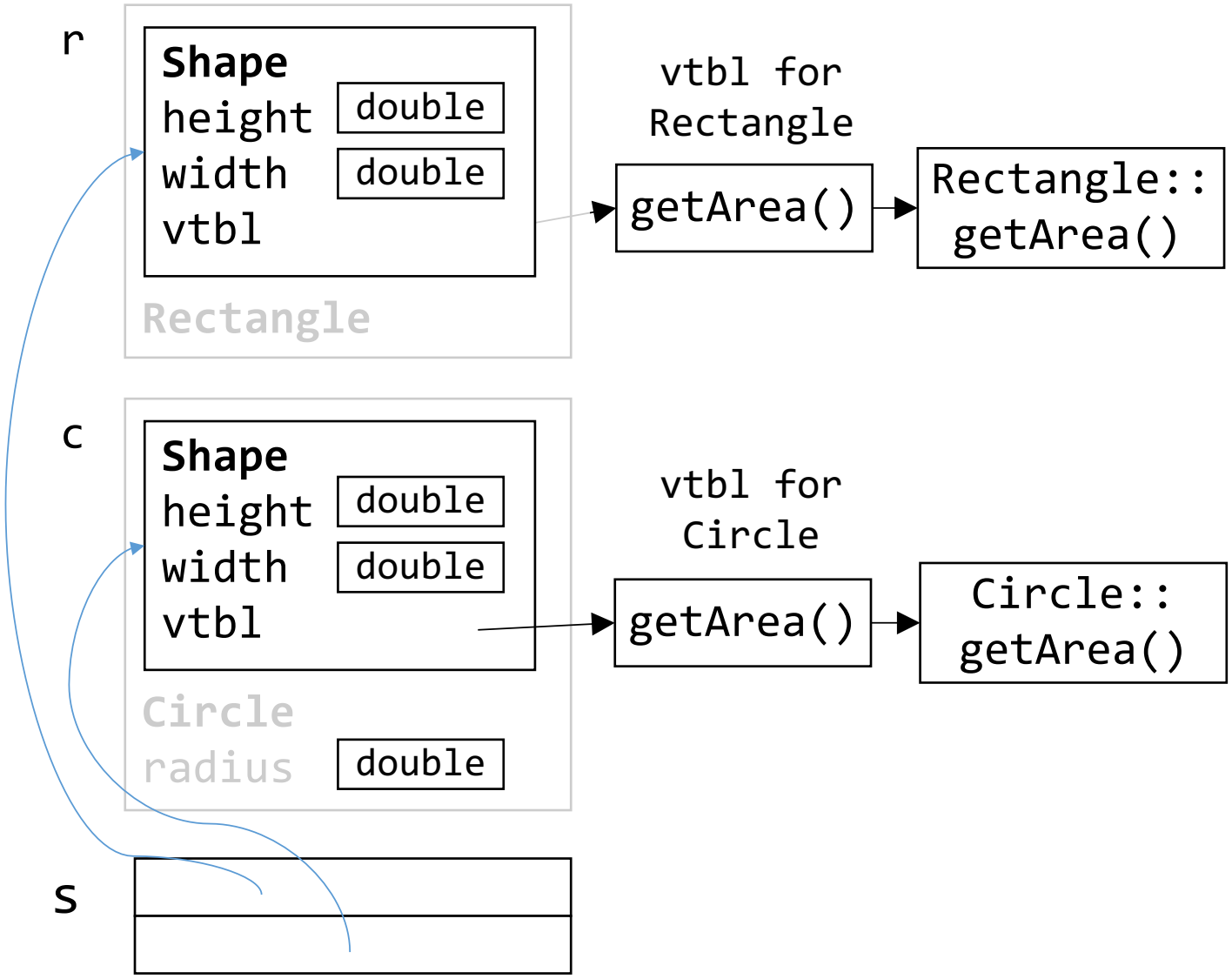
Reference: virtual function tables

- When an object is created, it obtains a pointer to a table with function pointers to all the virtual functions is also created, called ***the virtual function table*** (***vtbl***)
  - Implementation detail may vary by different types of compilers/systems



Example: virtual function tables

```
int main()
{
    Rectangle r {1, 2};
    Circle c {2};
    Shape* s[] {&r, &c};
    cout << s[0]->getArea()
         << endl;
    cout << s[1]->getArea()
         << endl;
}
```



Multiple inheritance

- A derived class may have more than one base class through multiple inheritance
  - c.f., many OOP languages prohibits multiple inheritance (e.g., Java), but uses interface
- Benefit of multiple inheritance
  - Shared interfaces
    - Often with abstract classes
    - Called *run-time polymorphism* or *interface inheritance*
  - Shared implementation
    - Called *implementation inheritance*
- But, use with caution; you may have lots of ambiguity

Example: multiple inheritance

```
class Base {
protected:
    int val;
};
class Derived : public Base {
protected:
    int val2;
};
class Derived2 : public Base {
protected:
    int val2;
};
```

```
class Diamond : public Derived,
public Derived2
{
    val2 = 42; // which one?
}
```

- Need to specify the name of the base class when two have variable with the same name

Reference: virtual function in Java

- Virtual functions
  - All methods are virtual functions by default
  - `final` keyword is used to indicate non-virtual functions
- Inheritance v.s. Interface
  - In Java, multiple inheritance is not allowed
  - Interfaces (similar to abstract class in C++) are used instead
  - A Java class can *inherit* only one base class, but it can *implement* multiple interfaces
- Note: the detailed implementation to support OO features may differ by OO languages
- Question: why do C++ and Java take the different approach?

Reading list

- Learn C++
  - Virtual functions: Ch. 12.1-3, 5-7
    - Recommended to read other sections
- (Optional, but recommended) C++ object model
  - https://github.com/CppCon/CppCon2015/tree/master/Presentations/Intro%20to%20C%2B%2B%20Object%20Model
- A class invariant: https://en.wikipedia.org/wiki/Class_invariant

ANY QUESTIONS?