EE2-12 Software Engineering 2: Object-Oriented Programming

Week 6 - Polymorphism and Virtual Functions

Sahbi Ben Ismail

Imperial College London, Department of Electrical and Electronic Engineering

Imperial College London

Autumn Term 2018-19

Module Syllabus

Week 1	Classes and Objects I: Introduction
Week 2	Classes and Objects II: Constructors, and Operator Overloading
Week 3	More on Classes, Objects, and Operator Overloading
Week 4	Objects and Dynamic Memory
Week 5	Classes Relationships: Association, Aggregation/Composition and Generalisation (Inheritance)
Week 6	**Polymorphism and Virtual Functions**
Week 7	Generic Programming: Templates, and the Standard Template Library (STL)
Week 8	Exceptions Handling
Week 9	C++ to Java
Week 10	Revision

Week 6 Intended Learning Outcomes (ILOs)

By the end of this week you should be better able to:

Lecture

1. Design and write code with **polymorphic** behaviour using **inheritance** and **virtual** functions
2. Understand the difference between **redefinition**, **overloading** and **overriding** member functions
3. Understand the difference between **static (early) binding** and **dynamic (late) binding**

Lab

1. Understand the **constructor calling order** and the **destructor calling order**
2. Apply **polymorphism** to draw shapes (classes `shape`, `square` and `triangle`)
3. Apply **polymorphism** to play a mini-game (classes `Enemy`, `Ninja` and `Monster`)

Problematic: same function name, different behaviour I

```
1  #include <iostream>
2  using namespace std;
3
4  class Enemy {
5      public:
6          virtual void attack() { }
7  };
8
9  class Ninja : public Enemy {
10     public:
11         void attack() {
12             cout << "Ninja!"<<endl;
13     }
14 };
15
16 class Monster : public Enemy {
17     public:
18         void attack() {
19             cout << "Monster!"<<endl;
20         }
21 };
```

```
int main() {
    Ninja n;
    Monster m;
    Enemy* e1 = &n;
    Enemy* e2 = &m;

    e1->attack();
    e2->attack();
}

/* Output:
  $./prog
  Ninja!
  Monster!
*/
```

Problematic: same function name, different behaviour II

```
1  class shape {
2    public:
3      //virtual void draw() = 0; //what are the consequences?
4      virtual void draw();
5  };
6
7  class triangle : public shape {/*...*/};
8  class rectangle : public shape {/*...*/};
9  class circle : public shape {/*...*/};
10
11 int main {
12   shape* s1 = new triangle(/*...*/);
13   shape* s2 = new rectangle(/*...*/);
14   shape* s3 = new circle(/*...*/);
15
16   std::vector<shape*> v;
17   v.push_back(s1);  v.push_back(s2);  v.push_back(s3);
18
19   for (int i=0; i<v.size(); i++) {
20     v[i]->draw(); //what happens here?
21   }
22
23   for(int i=0; i <v.size(); i++){
24     delete v[i];
25   }
```

Outline

Outline

Bjarne Stroustrup - GoingNative 2013 - The Essence of C++

YouTube video https://www.youtube.com/watch?v=D5MEsboj9Fc

Week 4: Objects and Dynamic Memory (start at 12'00", until 15'20")

C++ in four slides:

- Map to hardware
- Classes
- Inheritance
- Parameterized

Week 6: Polymorphism and Virtual Functions (start at 44'41", until 47'44")

https://youtu.be/D5MEsboj9Fc?t=2680

- OOP
- Inheritance
- building deep hierarchies $\neq$ good programming

Outline

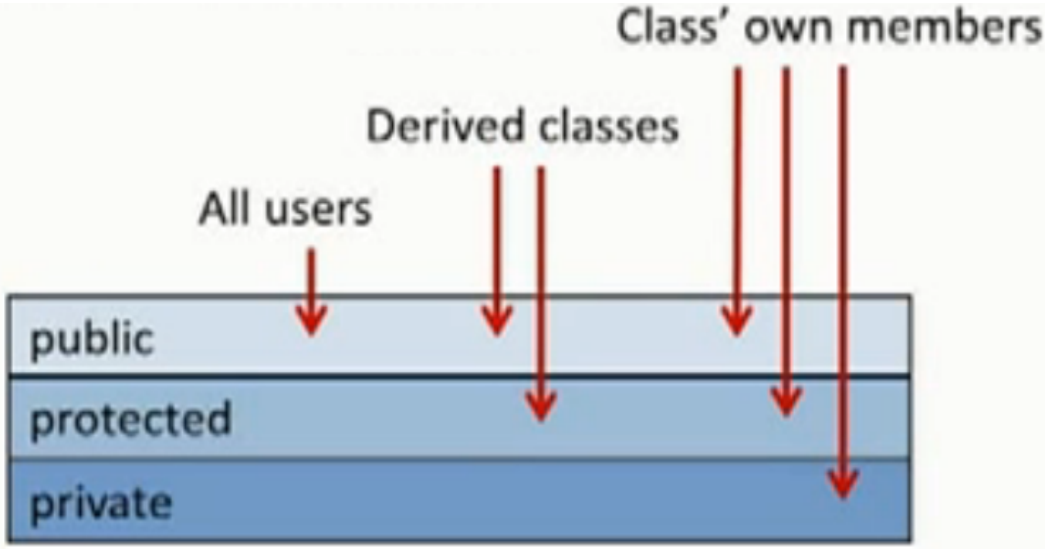
Inheritance: Basic mechanism

- a class B (**derived class**) can inherit from a class A (**base class**)

```
class A;

class B : public A {
  /*...*/
};
```

- some members (variables and functions) of the base class A may be used in the derived class B without re-writing any code [**re-usability**]
- as if the `public` and `protected` interfaces of the base class A are added to the interface of the derived class B

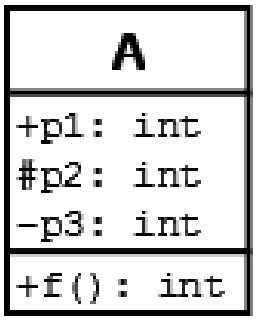
Access qualifiers: public protected private



Access from:

- Outside a class
- Inside a class
- Inside a derived class

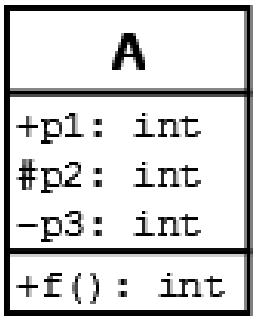
Access 1/3: from the outside of a class




```
class A {
  public:
    int f();
    int p1;

  protected:
    int p2;

  private:
    int p3;
};
```

```
int main() {
  A a;

  a.f(); // OK
  a.p1;  // OK
  a.p2 ; // ERROR
  a.p3 ; // ERROR
}
```

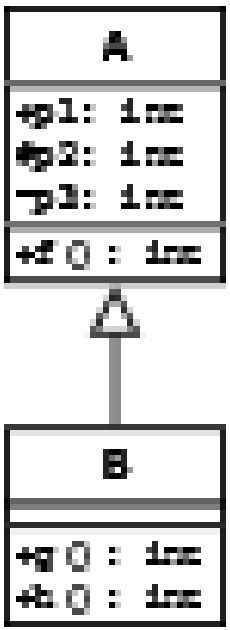
Access 2/3: from the inside of a class




```
class A {
  public:
    int f();
    int p1;

  protected:
    int p2;

  private:
    int p3;
};
```

```
int A::f() {
  return p1 + p2 +
         p3; // OK
}
```

Access 3/3: from the inside of a derived class




```
class A {
  public:
    int f();
    int p1;

  protected:
    int p2;

  private:
    int p3;
};
```

```
class B : public A {
  public:
    int g();
    int h();
};

int B::g() {
  return p1 + p2; // OK
}

int B::h() {
  return p3; // ERROR
}
```

Inheritance: Redefinition

Member functions may be redefined in the derived class.

```
class A {
  public:
    int f();

  protected:
    int n;
};

class B : public A {
  public:
    int f();
};
```

Inheritance: Enrichment (new features)

New members may be added to the derived class.

```
class A {
  public:
    int f();

  protected:
    int n;
};

class B : public A {
  public:
    int g();

  private:
    int m;
};
```

Order of calling constructors and destructors

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5    public:
6      A() { cout << "A()" << endl; }
7      ~A() { cout << "~A()" << endl; }
8  };
9
10 class B : public A {
11   public:
12     B() { cout << "B()" << endl; }
13     ~B() { cout << "~B()" << endl; }
14 };
15
16 int main () {
17   B b;
18 }
```

```
/* Output:
   $./prog
   A()
   B()
   ~B()
   ~A()
*/
```

Inheritance restrictions

a class B derived from a base class A does not inherit

- the constructor(s)
- the destructor
- the assignment operator

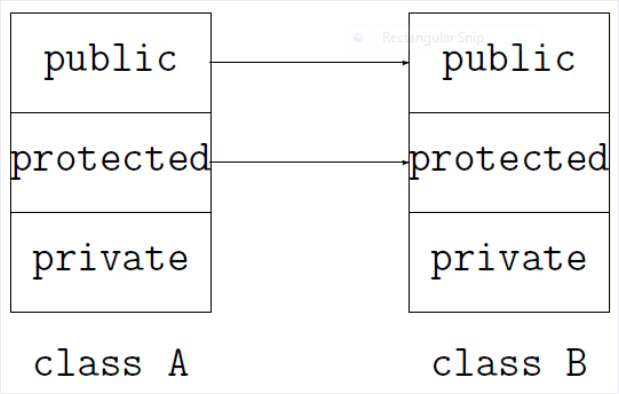
Inheritance and sub-typing

```
1  class A {/*...*/};
2
3  class B : public A {/*...*/};
4
5  class C : public A {/*...*/};
6
7
8  int main() {
9    A* aptr;
10   B* bptr;
11
12   aptr = new A();
13   aptr = new B();
14   aptr = new C();
15
16   //bptr = new A(); // ERROR (invalid conversion from 'A*' to 'B*'
17   bptr = new B();
18   //bptr = new C(); // ERROR (cannot convert 'C*' to 'B*')
19 }
```

public vs protected vs private inheritance I

public inheritance

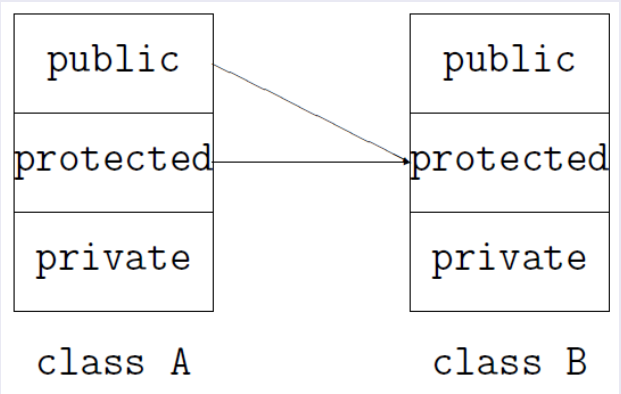
```
class A;

class B : public A {
  /*...*/
};
```



public vs protected vs private inheritance II

protected inheritance

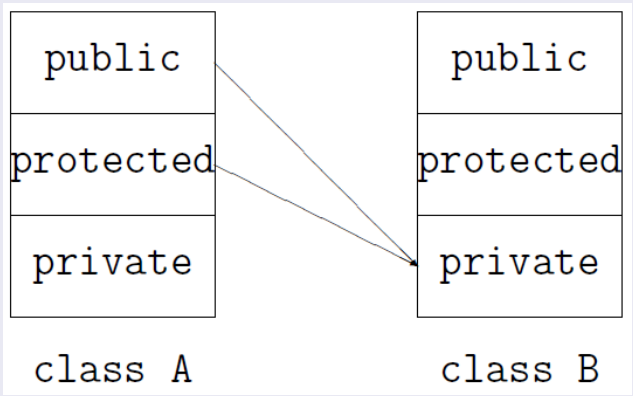
```
class A;

class B : protected A {
  /*...*/
};
```



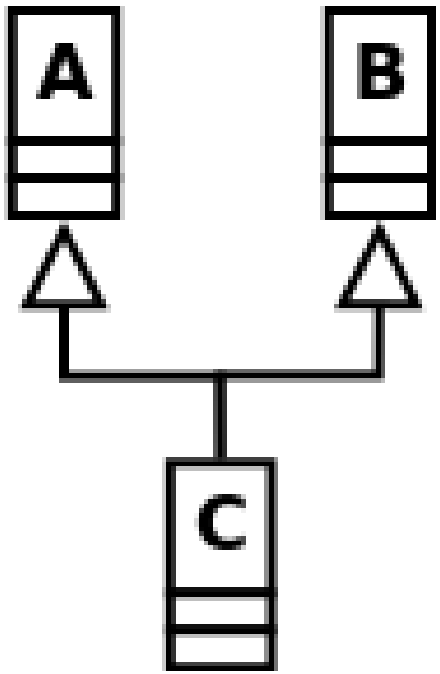
public vs protected vs private inheritance III

private inheritance

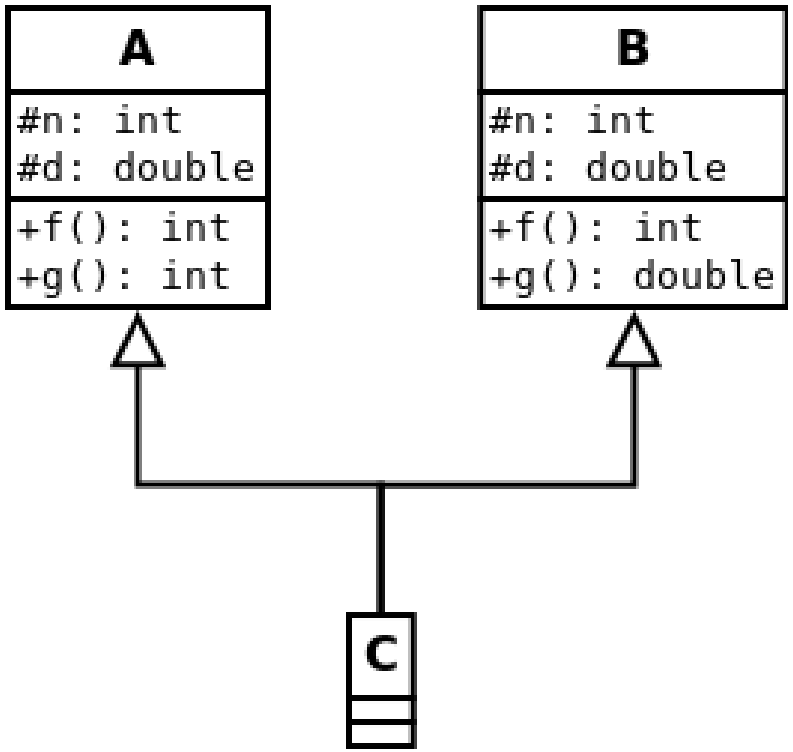
```
class A;

class B : private A {
  /*...*/
};
```



Multiple Inheritance I



Multiple Inheritance II



In-class activity: classes Ellipse and Circle

- Design two classes `Ellipse` and `Circle` using inheritance
- Discuss the issues of such an '*is-a*' relationship

In-class activity: conclusion

Other design choices

- `Circle` and `Ellipse` unrelated classes.
- `Circle` and `Ellipse` both inheriting from a third base class (e.g. `Shape`).

General guidelines

- Thinking of inheritance in terms of *is-a* relationship can be misleading.
- Sometimes a conceptual is-a hierarchy is not also suitable to be adopted in software design.
- Better to think of inheritance as *'can replace'* or *'extends'* (which is the keyword for inheritance in Java).

Outline

In-class introductory activity: saying hello :)

- Consider the classes hierarchy

`InternationalStudent` ▷ `Student` ▷ `Person`

`InternationalStaff` ▷ `Staff` ▷ `Person`

- EE2-12 lecture having a vector v of `pointers to Persons`

```
1  class Person {/*...*/}
2
3  class Student : public person {/*...*/}
4  class InternationalStudent : public Student {/*...*/}
5
6  class Staff : public person {/*...*/}
7  class InternationalStaff : public Staff {/*...*/}
8
9  int main {
10   std:vector<Person*> v; //in EE2-12 lecture
11   //...
12
13   for (int i=0; i<v.size(); i++) {
14     v[i]->say_hello(); //in mother tongue
15   }
16 }
```

Member functions calling

when to determine the member function to perform: static vs dynamic binding.

static binding

at compilation time

dynamic binding

at runtime

Static binding

- inheritance allows an implicit conversion of a derived class object to a base class object
- a **pointer to a base class object** can be redirected to point to any instance of a derived class
- but the type of this pointer was defined at **compilation time**
- the member functions to be performed at **runtime** are those defined at compilation time

Static binding - Example

```
1  class A {
2    public :
3      A() { cout << "A()" << endl; }
4      ~A() { cout << "~A()" << endl; }
5      int f() { return 'A'; }
6  };
7
8  class B : public A {
9    public :
10     B() { cout << "B()" << endl; }
11     ~B() { cout << "~B()" << endl; }
12     int f() { return 'B'; }
13 };
14
15 int main () {
16   A* pa = new A();
17   cout << pa->f() << endl;
18   delete pa;
19
20   cout << endl;
21
22   pa = new B();
23   cout << pa->f() << endl;
24   delete pa;
25 }
```

```
/* Output:
   $./prog
   A()
   65
   ~A()

   A()
   B()
   65
   ~A()
*/
```

Redefinition vs. overriding

- In both cases: function in base class and subclass, same name and signature, different behaviour.
- *Redefinition*: function is called on *object* declared and constructed as instance of subclass.
- *Overriding*: function is called *through pointer or reference* (to base class) on object constructed as instance of subclass.
- Overriding needs to be enabled (in C++ by default it's disabled): declaring member function `virtual` in base class.
- Once a member function is declared `virtual` it stays so all the way down in the inheritance hierarchy (still a good idea to mark it as such for readability).

Dynamic binding

- **virtual** functions allow to delay the choice of the most appropriate function call
- a **pointer to a base class object** can be redirected to point to any instance of a derived class
- the choice of the member function to be performed will be delayed to **runtime** if the function is declared **virtual** in the base class

Dynamic binding - Example

```
1  class A {
2    public :
3      A() { cout << "A()" << endl; }
4      ~A() { cout << "~A()" << endl; }
5      virtual int f() { return 'A'; }
6  };
7
8  class B : public A {
9    public :
10     B() { cout << "B()" << endl; }
11     ~B() { cout << "~B()" << endl; }
12     virtual int f() { return 'B'; }
13 };
14
15 int main () {
16   A* pa = new A();
17   cout << pa->f() << endl;
18   delete pa;
19
20   cout << endl;
21
22   pa = new B();
23   cout << pa->f() << endl;
24   delete pa;
25 }
```

```
/* Output:
   $./prog
   A()
   65
   ~A()

   A()
   B()
   66
   ~A()
*/
```

Dynamic binding and constructors/destructors

dynamic binding does not work for constructor(s)

It is too early!

A constructor cannot be declared `virtual`
(Error: `constructors cannot be declared 'virtual'`)

dynamic binding does not work for the destructor

It is too late!

- A destructor can be declared `virtual`

Dynamic binding with virtual destructor - Example

```
1  class A {
2    public :
3      A() { cout << "A()" << endl; }
4      virtual ~A() { cout << "~A()" << endl; }
5      virtual int f() { return 'A'; }
6  };
7
8  class B : public A {
9    public :
10     B() { cout << "B()" << endl; }
11     virtual ~B() { cout << "~B()" << endl; }
12     virtual int f() { return 'B'; }
13 };
14
15 int main () {
16   A* pa = new A();
17   cout << pa->f() << endl;
18   delete pa;
19
20   cout << endl;
21
22   pa = new B();
23   cout << pa->f() << endl;
24   delete pa;
25 }
```

```
/* Output:
   $./prog
   A()
   65
   ~A()

   A()
   B()
   66
   ~B()
   ~A()
*/
```

Problematic: same function name, different behaviour I

```
1  #include <iostream>
2  using namespace std;
3
4  class Enemy {
5      public:
6          virtual void attack() { }
7  };
8
9  class Ninja : public Enemy {
10     public:
11         void attack() {
12             cout << "Ninja!"<<endl;
13     }
14 };
15
16 class Monster : public Enemy {
17     public:
18         void attack() {
19             cout << "Monster!"<<endl;
20         }
21 };
```

```
int main() {
    Ninja n;
    Monster m;
    Enemy* e1 = &n;
    Enemy* e2 = &m;

    e1->attack();
    e2->attack();
}

/* Output:
  $./prog
  Ninja!
  Monster!
*/
```

Abstract Classes I

- if the declaration of a `virtual` member function ends with [= 0;], then any derived class has to define this function
  - such a function is called **pure virtual**

Definition

Abstract class: a class with at least one '**pure virtual**' member function.
(Note: the '=0' notation is just a keyword and does not imply that something is actually set to 0)

Consequence

An abstract class cannot be instantiated.
(what would happen when one of the pure virtual member functions is called, otherwise?)

Abstract Classes II

What for then?

- Abstract classes can still be used as a *type* for pointers and references (and sometimes it is exactly what is needed).
- Abstract classes are *'interfaces'* declaring what can be done on objects instances of classes implementing the abstract one

So: one (abstract) interface, many different (concrete) implementations

Abstract Classes III

Derived classes

- classes which inherit from an abstract class need to **override** the virtual member functions with an actual implementation in order to become 'concrete'.
- if a derived class does not override (all) the pure virtual function(s) of a base class, then it is also abstract

Abstract Classes - Example 1

```
1  class AbstractNotion {
2    public :
3      virtual int f(int n) = 0; //pure virtual
4  };
5
6  class Concrete : public AbstractNotion {
7    public :
8      virtual int f(int n) { return n * n; }
9  };
```

Abstract Classes - Example 2 - Shapes I

```
1  class shape {
2    public :
3      virtual double perimeter() = 0; //pure virtual
4      virtual double area() = 0; //pure virtual
5  };
```

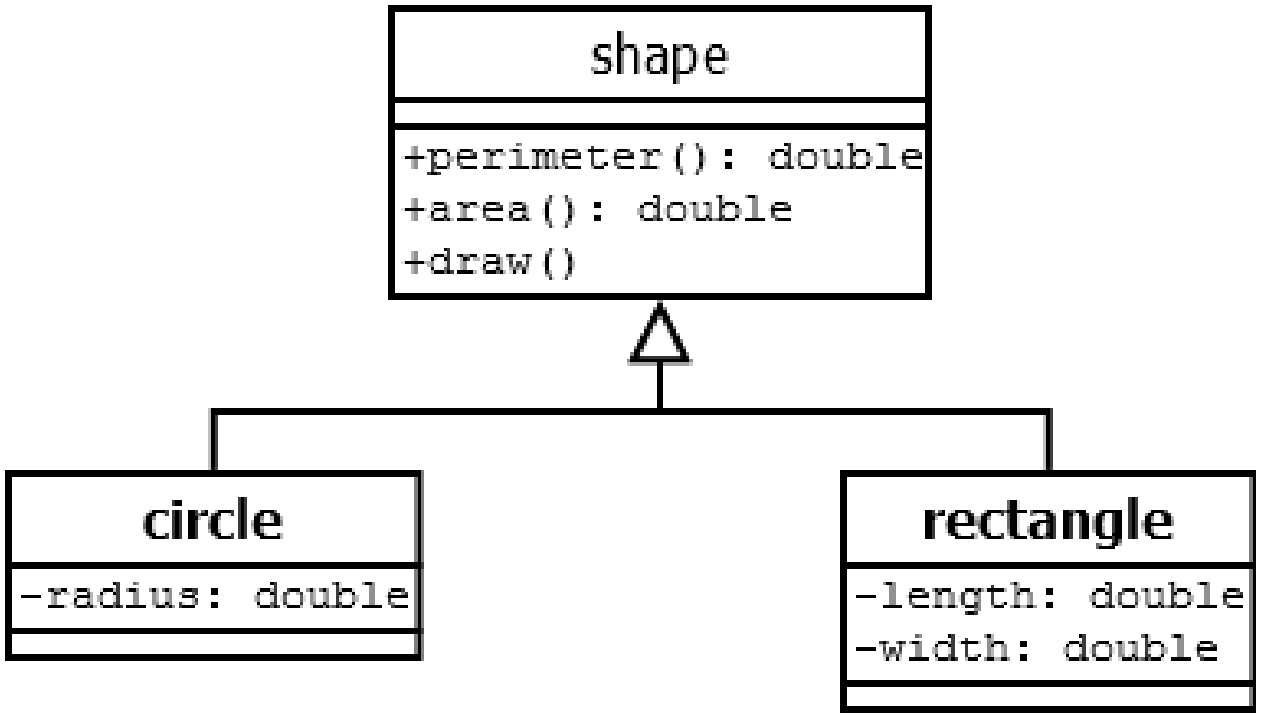
Abstract Classes - Example 2 - Shapes II

```
1  class circle : public shape {
2    public :
3      circle(double r=0) : radius(r) {}
4      virtual double perimeter();
5      virtual double area();
6
7    private:
8      double radius;
9  };
10
11 double circle::perimeter() {
12   return 2 * 3.14 * radius;
13 }
14
15 double circle::area() {
16   return 3.14 * radius * radius;
17 }
```

Abstract Classes - Example 2 - Shapes III

```
1  int main() {
2    //shape s; //ERROR
3          //error: cannot declare variable 's' to be of
               abstract type 'shape'
4
5    shape* s = new circle(1);
6    std::cout << "primeter: " << s->perimeter() << std::
         endl;
7    std::cout << "area: " << s->area() << std::endl;
8  }

   /* Output:
     $./prog
     primeter: 6.28
     area: 3.14
   */
```

Abstract Classes - Example 2 - Shapes



UML conventions:

- an abstract class name is highlighted in *italic shape*
- «*abstract*» stereotype

Problematic: same function name, different behaviour II

```
1  class shape {
2    public:
3      //virtual void draw() = 0; //what are the consequences?
4      virtual void draw();
5  };
6
7  class triangle : public shape {/*...*/};
8  class rectangle : public shape {/*...*/};
9  class circle : public shape {/*...*/};
10
11 int main {
12   shape* s1 = new triangle(/*...*/);
13   shape* s2 = new rectangle(/*...*/);
14   shape* s3 = new circle(/*...*/);
15
16   std::vector<shape*> v;
17   v.push_back(s1);  v.push_back(s2);  v.push_back(s3);
18
19   for (int i=0; i<v.size(); i++) {
20     v[i]->draw(); //what happens here?
21   }
22
23   for(int i=0; i <v.size(); i++){
24     delete v[i];
25   }
```

Outline

Summary I

OOP "Big Four"

- abstraction
- encapsulation
- **inheritance**
- **polymorphism**

Summary II

Polymorphic functions: Binding

- *Binding*: connecting a function call to a function body.
- *Early (static) binding*: performed (by compiler and linker) before the program is run.
- *Late (dynamic) binding*: (partially) occurring at runtime, based on the actual type of the object.
- Keyword `virtual` instructs the compiler to perform late binding on that function.

What to do next?

Next Lab

- More on inheritance (classes `Base`, `Derived` and `DerivedAgain`)
- Using polymorphism to draw shapes (classes `shape`, `square` and `triangle`)
- A mini-game: classes `Enemy`, `Ninja` and `Monster`
- `Makefile`: more variables, and project files reorganisation in different folders (`src/` `lib/` `inc/` `bin/`)

Next Lecture

Week 7 - Generic Programming: Templates, and the Standard Template Library (STL)