Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 19/24

Session 19. More on Object-Oriented Programming: Polymorphic codes, Abstract classes and Class Hierarchies

- Access control: protected
- Protected members
- Polymorphic code
- Virtual functions
- Pure virtual functions and abstract classes
- Virtual destructors
- Class hierarchies
- Protected Inheritance
- Private inheritance
- Q&A

150 min (incl. Q & A)

Access control

Access control

- A member of a class can be *private*, *protected*, or *public*.

Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.

Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.

Access control

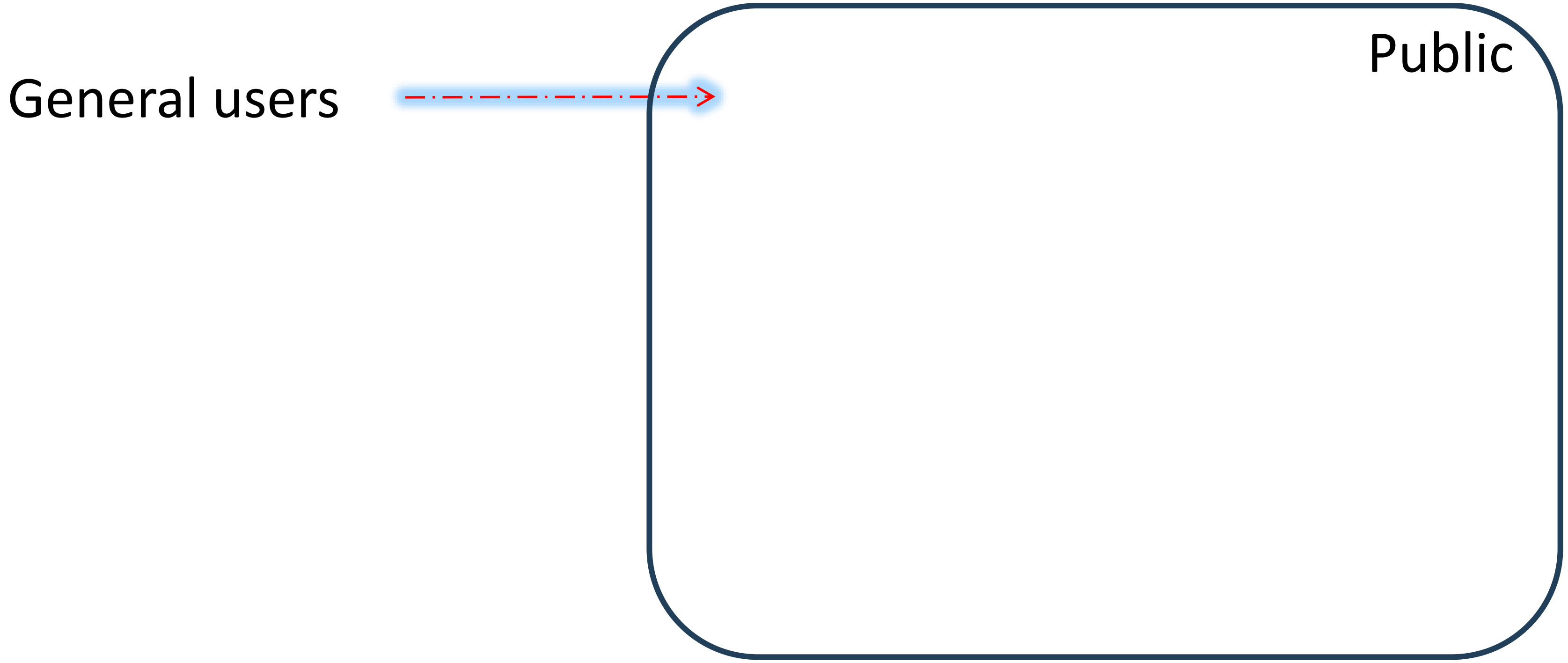
- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.
- If it is *public*, its name can be used anywhere without access restriction.

Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.
- If it is *public*, its name can be used anywhere without access restriction.

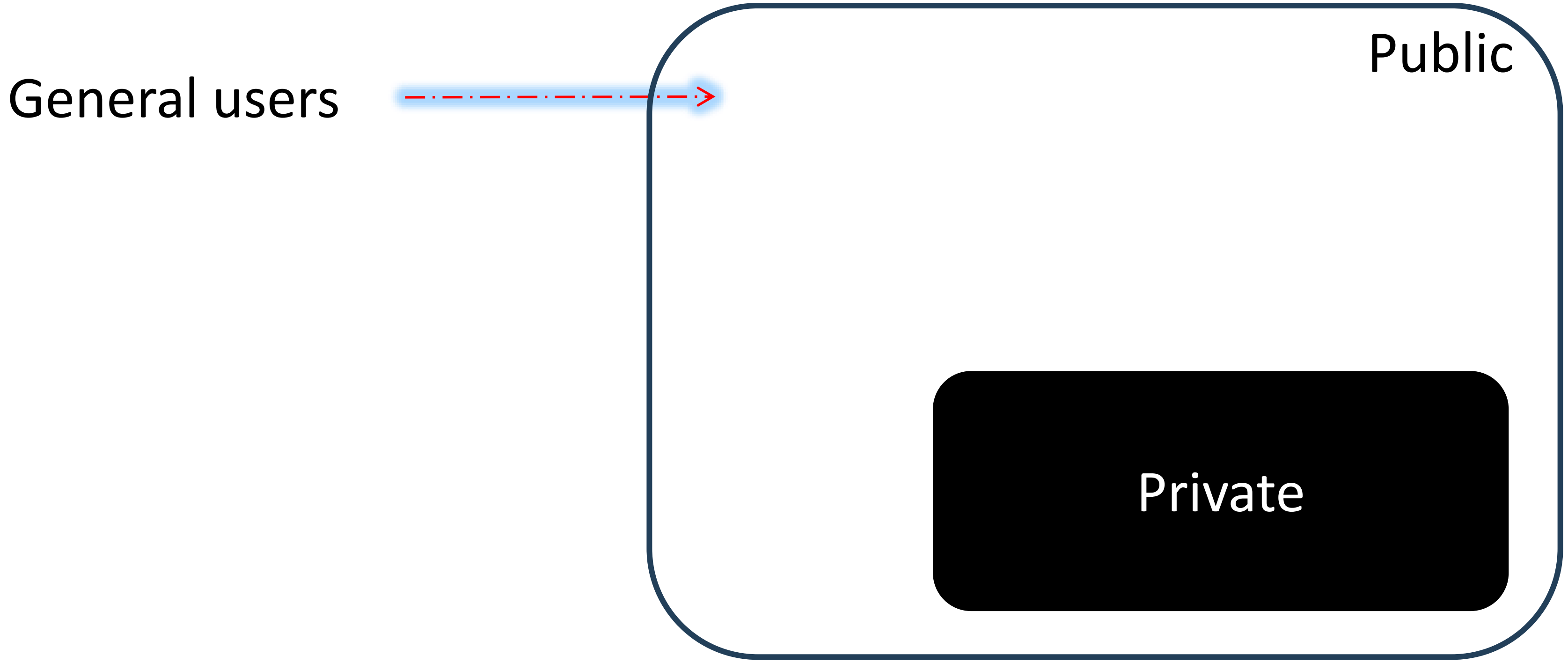
Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.
- If it is *public*, its name can be used anywhere without access restriction.



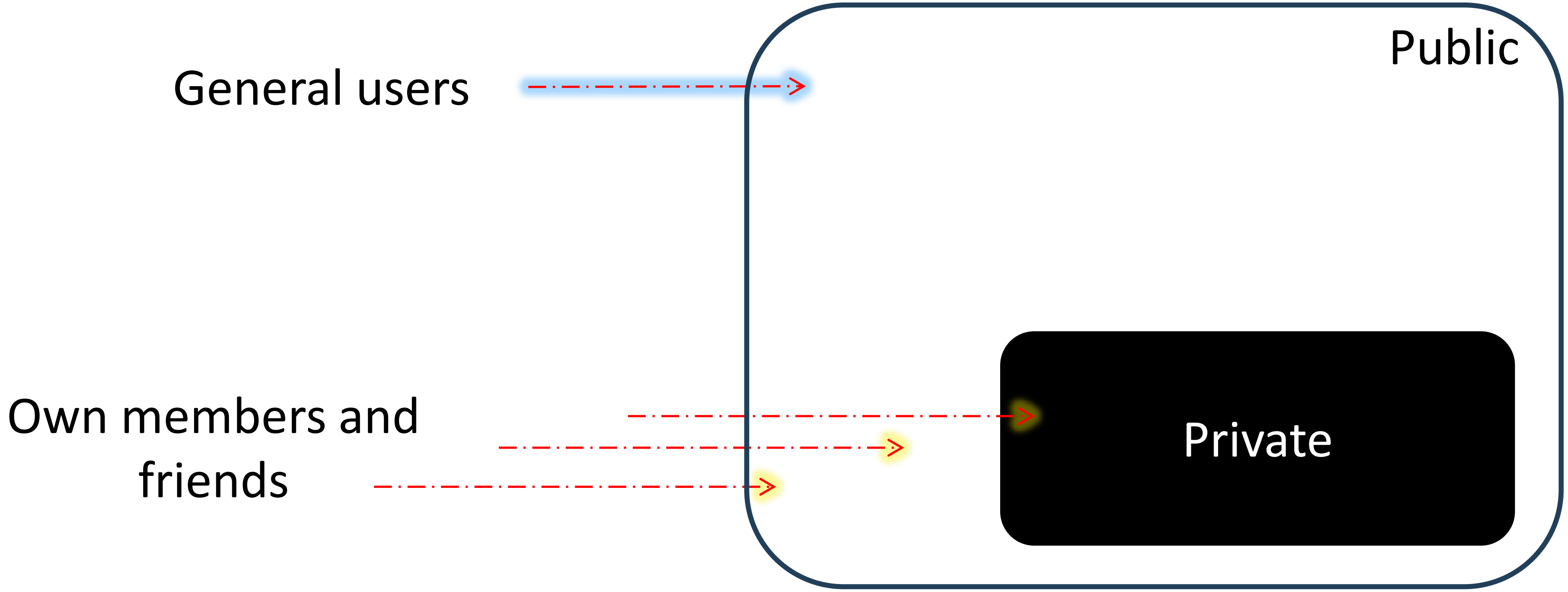
Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.
- If it is *public*, its name can be used anywhere without access restriction.



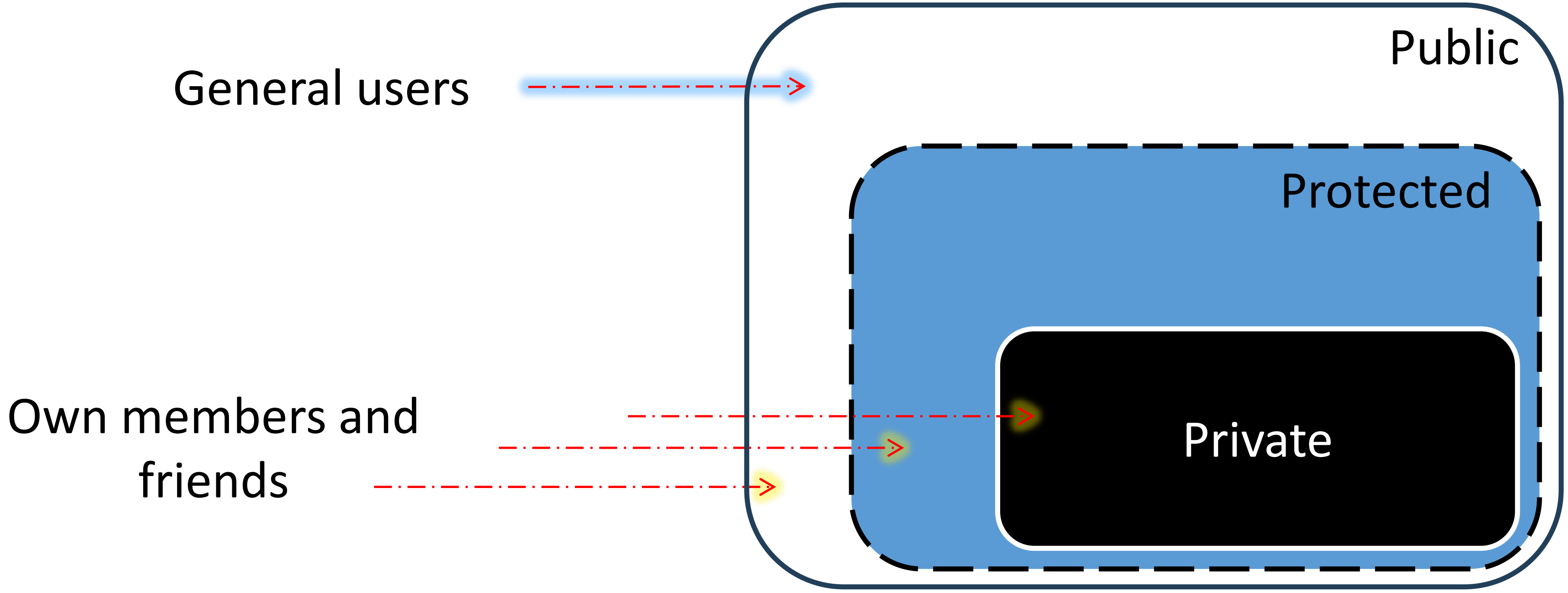
Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.
- If it is *public*, its name can be used anywhere without access restriction.



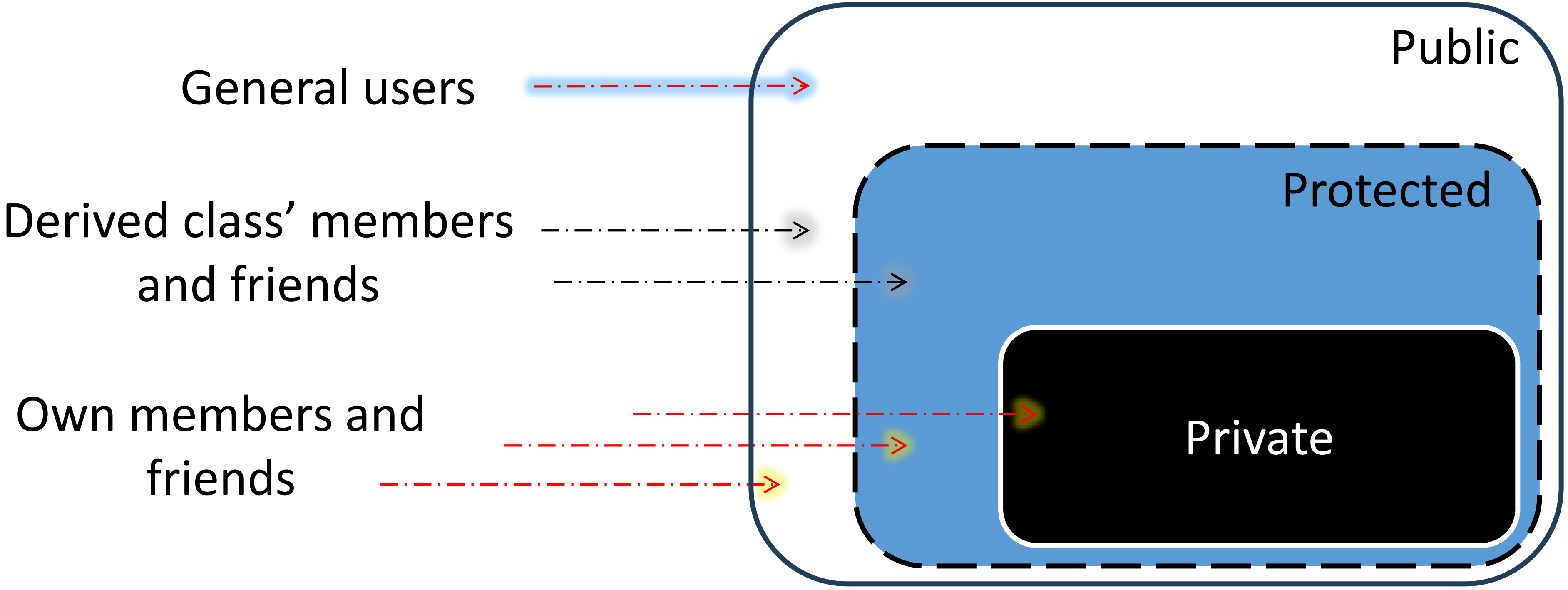
Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.
- If it is *public*, its name can be used anywhere without access restriction.



Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *protected*, its name can be used only by members and friends of the class in which it is declared and by members and friends of classes derived from this class.
- If it is *public*, its name can be used anywhere without access restriction.



Access control: protected

Access control: protected

- A protected member is like a public member to a member of a derived class, yet it is like a private member to class outside.

Access control: protected

- A protected member is like a public member to a member of a derived class, yet it is like a private member to class outside.
- Recap: Access to members vs. Member visibility

Access control: protected

- A protected member is like a public member to a member of a derived class, yet it is like a private member to class outside.
- Recap: Access to members vs. Member visibility
- Protected members access test:

Polymorphic code

Polymorphic code

- Important question: Given a pointer of type **Base***, to which derived type does the object pointed to really belong?

Polymorphic code

- Important question: Given a pointer of type **Base***, to which derived type does the object pointed to really belong?

```
Base* p = new Derived(); // polymorphic code
p->…
```

Polymorphic code

- Important question: Given a pointer of type **Base***, to which derived type does the object pointed to really belong?

- Example:

```
Base* p = new Derived(); // polymorphic code
p->…
```

1

```
#include <iostream>

class Base {
public:
  void f() { std::cout << "Base::f() called\n"; }
};

class Derived : public Base {
public:
  void f() { std::cout << "Derived::f() called\n"; }
};
```

Polymorphic code

- Important question: Given a pointer of type **Base***, to which derived type does the object pointed to really belong?

- Example:

```
Base* p = new Derived(); // polymorphic code
p->…
```

1

```
#include <iostream>

class Base {
public:
  void f() { std::cout << "Base::f() called\n"; }
};

class Derived : public Base {
public:
  void f() { std::cout << "Derived::f() called\n"; }
};
```

2

```
#include "Base_and_Derived.h"
int main()
{
  Base b;
  b.f();

  Derived d;
  d.f();

  Base* pb = new Derived();
  pb->f();
  delete pb;

  return 0;
}
```

Polymorphic code

- Important question: Given a pointer of type **Base***, to which derived type does the object pointed to really belong?

- Example:

```
Base* p = new Derived(); // polymorphic code
p->…
```

1

```
#include <iostream>

class Base {
public:
  void f() { std::cout << "Base::f() called\n"; }
};

class Derived : public Base {
public:
  void f() { std::cout << "Derived::f() called\n"; }
};
```

2

```
#include "Base_and_Derived.h"
int main()
{
  Base b;
  b.f();

  Derived d;
  d.f();

  Base* pb = new Derived();
  pb->f();
  delete pb;

  return 0;
}
```

```
Base::f() called
Derived::f() called
Base::f() called
```

Polymorphic code

- Important question: Given a pointer of type **Base***, to which derived type does the object pointed to really belong?

- Example:

```
Base* p = new Derived(); // polymorphic code
p->…
```

1

```
#include <iostream>

class Base {
public:
  void f() { std::cout << "Base::f() called\n"; }
};

class Derived : public Base {
public:
  void f() { std::cout << "Derived::f() called\n"; }
};
```

2

```
#include "Base_and_Derived.h"
int main()
{
  Base b;
  b.f();

  Derived d;
  d.f();

  Base* pb = new Derived();
  pb->f();
  delete pb;

  return 0;
}
```

```
Base::f() called
Derived::f() called
Base::f() called
```

- Calling base or derived member function test:

Inheritance Exercise

Prog.

آلفا
ALPHA

Type fields

Type fields

1

```
class Point {
  // as before
};
class Color { /* … */ };
// Shape concept: version 1
enum Kind { CIRCLE, TRIANGLE, RECTANGLE };
class Shape {
  Kind k; // type field
  Point Center;
  Color c;
public:
  Shape(Kind kk, Point Cen, Color col) :
      k(kk), Center(Cen), c(col) {}
  void draw();
  void rotate(int);
};
```

Type fields

1

```
class Point {
  // as before
};
class Color { /* … */ };
// Shape concept: version 1
enum Kind { CIRCLE, TRIANGLE, RECTANGLE };
class Shape {
  Kind k; // type field
  Point Center;
  Color c;
public:
  Shape(Kind kk, Point Cen, Color col) :
      k(kk), Center(Cen), c(col) {}
  void draw();
  void rotate(int);
};
```

2

```
void Shape::draw()
{
  switch (k) { // use type field
  case CIRCLE:
    // draw a circle
    break;
  case TRIANGLE:
    // draw a triangle
    break;
  case RECTANGLE:
    // draw a square
    break;
  }
}
```

Type fields

1

```
class Point {
   // as before
};
class Color { /* … */ };
// Shape concept: version 1
enum Kind { CIRCLE, TRIANGLE, RECTANGLE };
class Shape {
   Kind k; // type field
   Point Center;
   Color c;
public:
   Shape(Kind kk, Point Cen, Color col) :
         k(kk), Center(Cen), c(col) {}
   void draw();
   void rotate(int);
};
```

2

```
void Shape::draw()
{
   switch (k) { // use type field
   case CIRCLE:
      // draw a circle
      break;
   case TRIANGLE:
      // draw a triangle
      break;
   case RECTANGLE:
      // draw a square
      break;
   }
}
```

3

```
void Shape::rotate(int deg)
{
   switch (k) { // use type field
   case CIRCLE:
      // rotate a circle: just do nothing
      break;
   case TRIANGLE:
      // rotate a triangle
      break;
   /* … */
   }
}
```

Type fields cont.

4

```
// use Shape
void f()
{
     Color c;
     Point p;
     Shape s1(CIRCLE, p, c);
     s1.draw(); // Call draw: execute Circle case of switch statement
     Shape s2(RECTANGLE, p, c);
     s2.rotate(30); // rotate 30 degrees
}
```

ALPHA آلفا

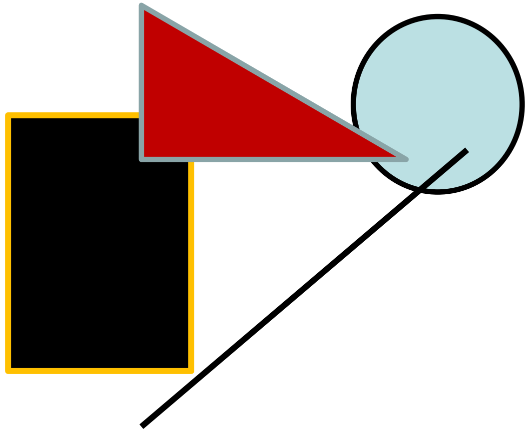
Commonality- another example

Commonality- another example

- Representing a circle and a triangle in a program without involving the notion of a shape would be to miss something essential.

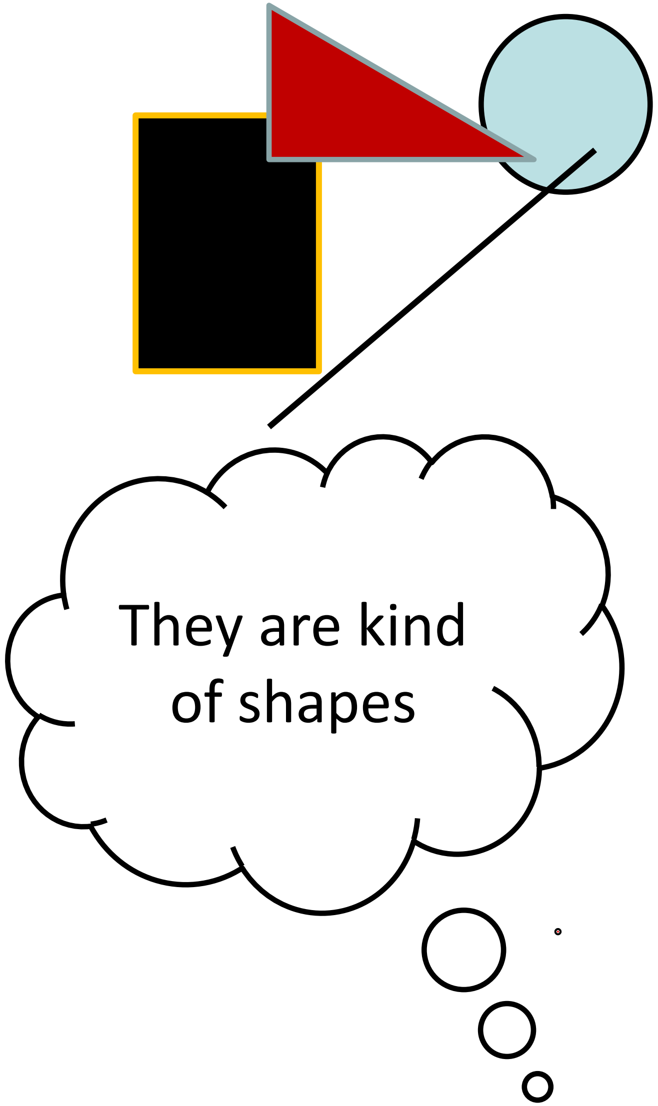
Commonality- another example

- Representing a circle and a triangle in a program without involving the notion of a shape would be to miss something essential.



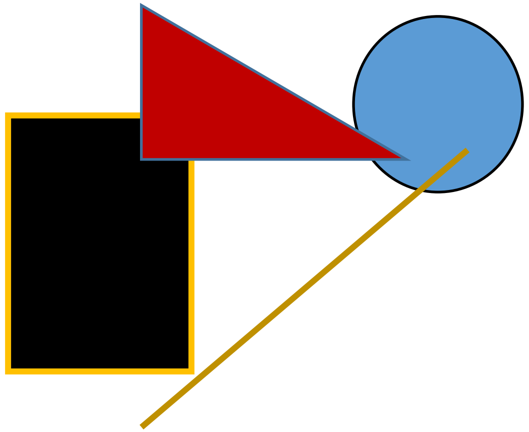
Commonality- another example

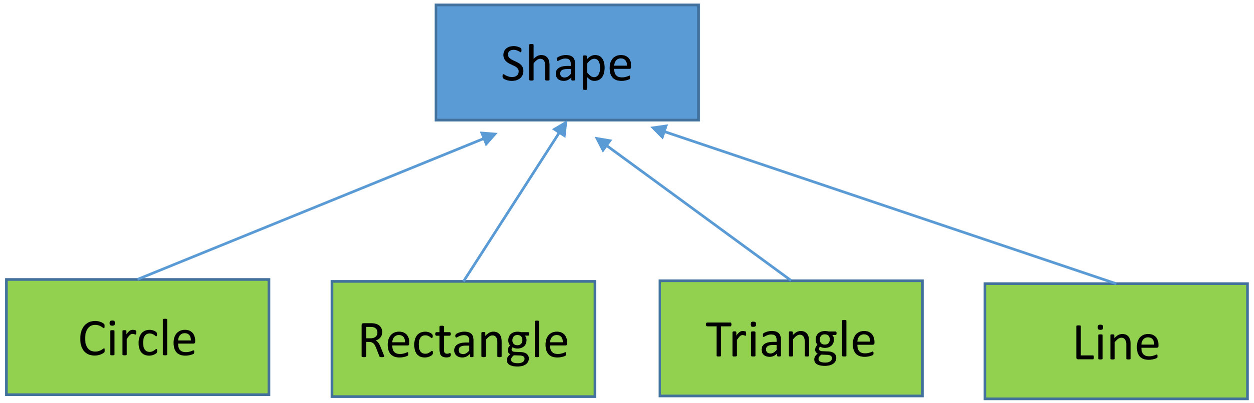
- Representing a circle and a triangle in a program without involving the notion of a shape would be to miss something essential.

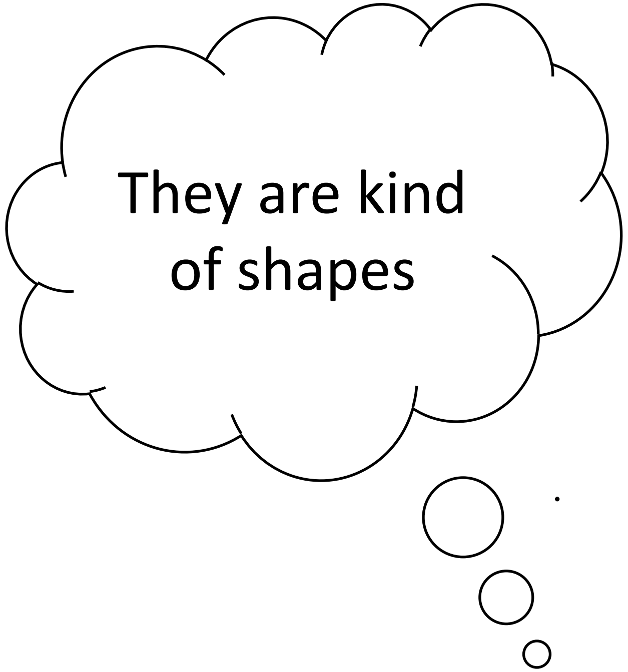


Commonality- another example

- Representing a circle and a triangle in a program without involving the notion of a shape would be to miss something essential.







Shape class hierarchy

- Shape
  - General properties: color, general operation
  - Specific properties: How to create circle, rectangle, …

1

```
// Shape concept: version 2
enum Kind { CIRCLE, TRIANGLE, RECTANGLE };
class Color { /* … */ };
class Shape {
protected:
  Kind k;
public:
  Shape(Kind kk) : k(kk) {}
  void draw();
  void rotate(int);
};
```

2

```
class Circle : public Shape {
  Point center;
  int radious;
  Color c;
public:
  Circle(const Point& cent = Point(0, 0), int rad = 1) : Shape(CIRCLE),
  Center(cent), radious(rad) { /* set color */ }
  void draw() { /* … */ }
  void rotate(int degree) { /* do nothing */ }
};
```

Shape class hierarchy cont.

3

```
class Triangle : public Shape {
   Point p1, p2, p3;
   Color c;
public:
   Triangle(Point p = Point(0, 0), Point q = Point(1, 0), Point r = Point(0, 1)) :
            Shape(TRIANGLE), p1(p), p2(q), p3(r) { /* set color */ }
   void draw() { /* … */ }
   void rotate(int degree) { /* … */ }
};
```

4

```
void Shape::draw()
{
     switch (k) {
     case CIRCLE:
     // draw circle
          (static_cast<Circle *>(this))->draw();
          break;
     case RECTANGLE:
     // draw rectangle
          (static_cast<Rectangle *>(this))->draw();
          break;
     case TRIANGLE:
     // draw triangle
          (static_cast<Triangle *>(this))->draw();
          break;
     }
}
```

5

```
// use shape
DrawAllShapes(vector<Shape *> v)
{
     for (i = 0; i < v.size(); i++) {
          v[i]->Draw();
     }
}
```

Type fields cont.

- Functions such as draw() must ''know about'' all the kinds of shapes there are.
- The code for functions such as draw() grows each time a new shape is added to the system.
- If we define a new shape, every operation on a shape must be examined and (possibly) modified.
- Adding a new shape involves ''touching'' the code of every important operation on shapes such as draw, rotate.
- The choice of representation of particular shapes can get severely cramped by the requirement that (at least some of) their representation must fit into the typically fixed-sized framework presented by the definition of the general type Shape.

Poor maintainability

☞ Avoid type fields.

Virtual member functions- An example

1

```
// Shape concept
class Shape {
public:
   // no constructor
   virtual void draw();
   virtual void rotate(int);
};
void Shape::draw() { }
void Shape::rotate(int i) { }
```

Overridden functions

2

```
class Circle : public Shape {
public:
   void draw() { /* … */ }
   void rotate(int d) { /* … */ }
};
```

3

```
class Rectangle : public Shape {
public:
   void draw() { /* … */ }
   void rotate(int d) { /* … */ }
};
```

4

```
class Triangle : public Shape {
public:
   void draw() { /* … */ }
   void rotate(int d) { /* … */ }
};
```

Overriding functions

5

```
DrawAllShapes(vector<Shape *> v)
{
   for (i = 0; i < v.size(); i++)
   {
      v[i]->Draw();
   }
}
```

- *virtual* means “may be redefined later in a class derived from this one”.
- *virtual* indicates that a member function can act as an interface to itself and the corresponding functions defined in classes derived from it.

Virtual functions- Introduction

Simula → C++

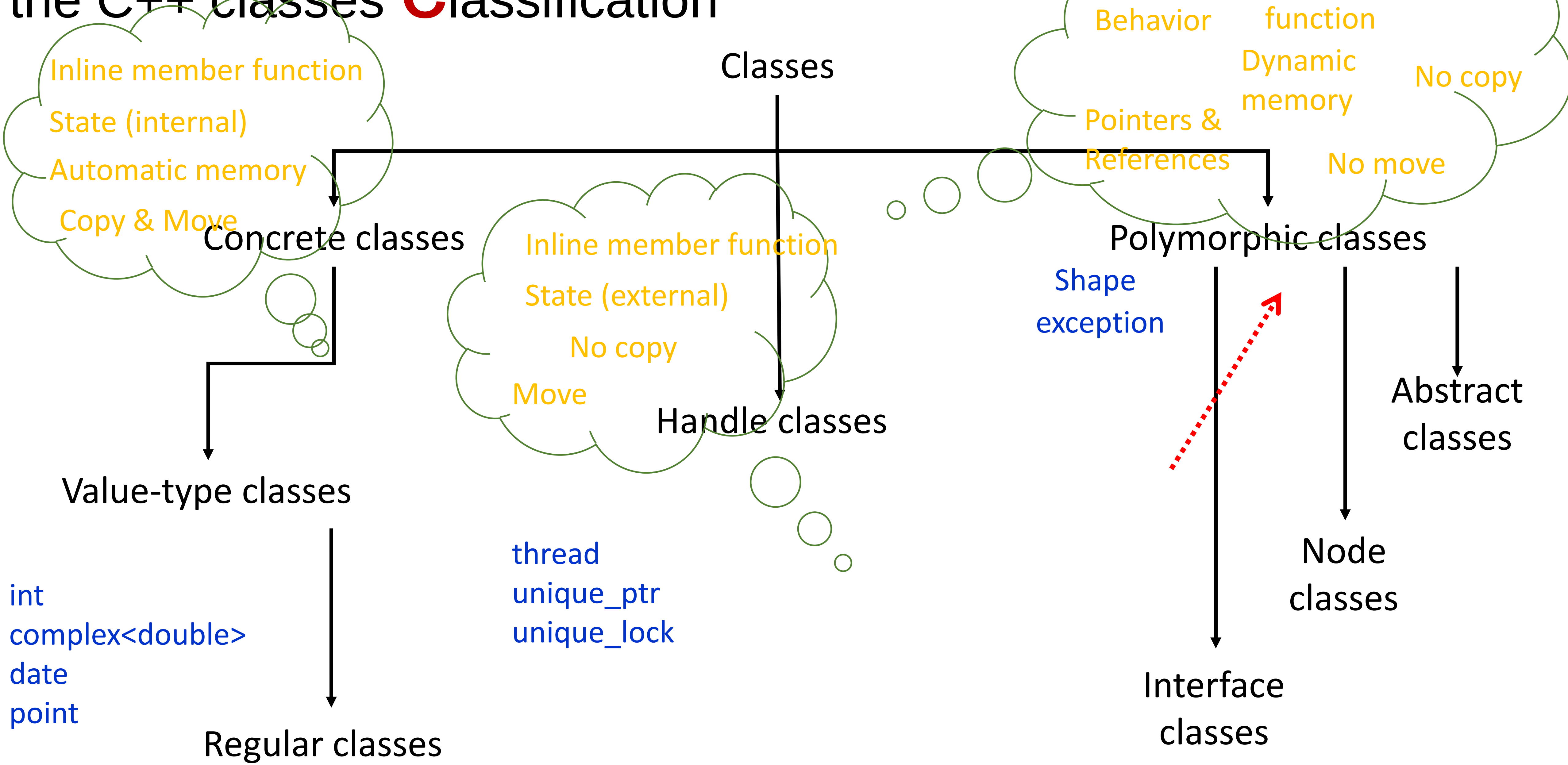
- Virtual (member) functions
- From an OO perspective, it is the single most important feature of C++.

Virtual functions

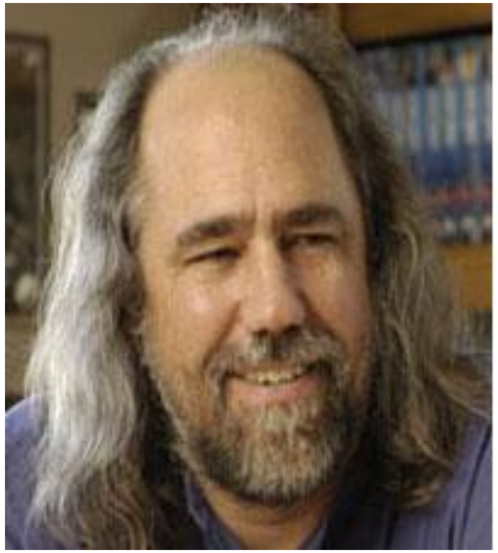
1. allow the programmer to declare functions in a base class that *can be redefined* in each derived class.
2. allow derived classes to replace the implementation provided by the base class.
3. the primary mechanism for run-time *polymorphism* in C++.

- The compiler and linker will guarantee the correct correspondence between objects and the function applied to them.
- A virtual member function is sometimes called a *method*.

the C++ classes Classification



Object-Oriented Programming



Object

Inheritance

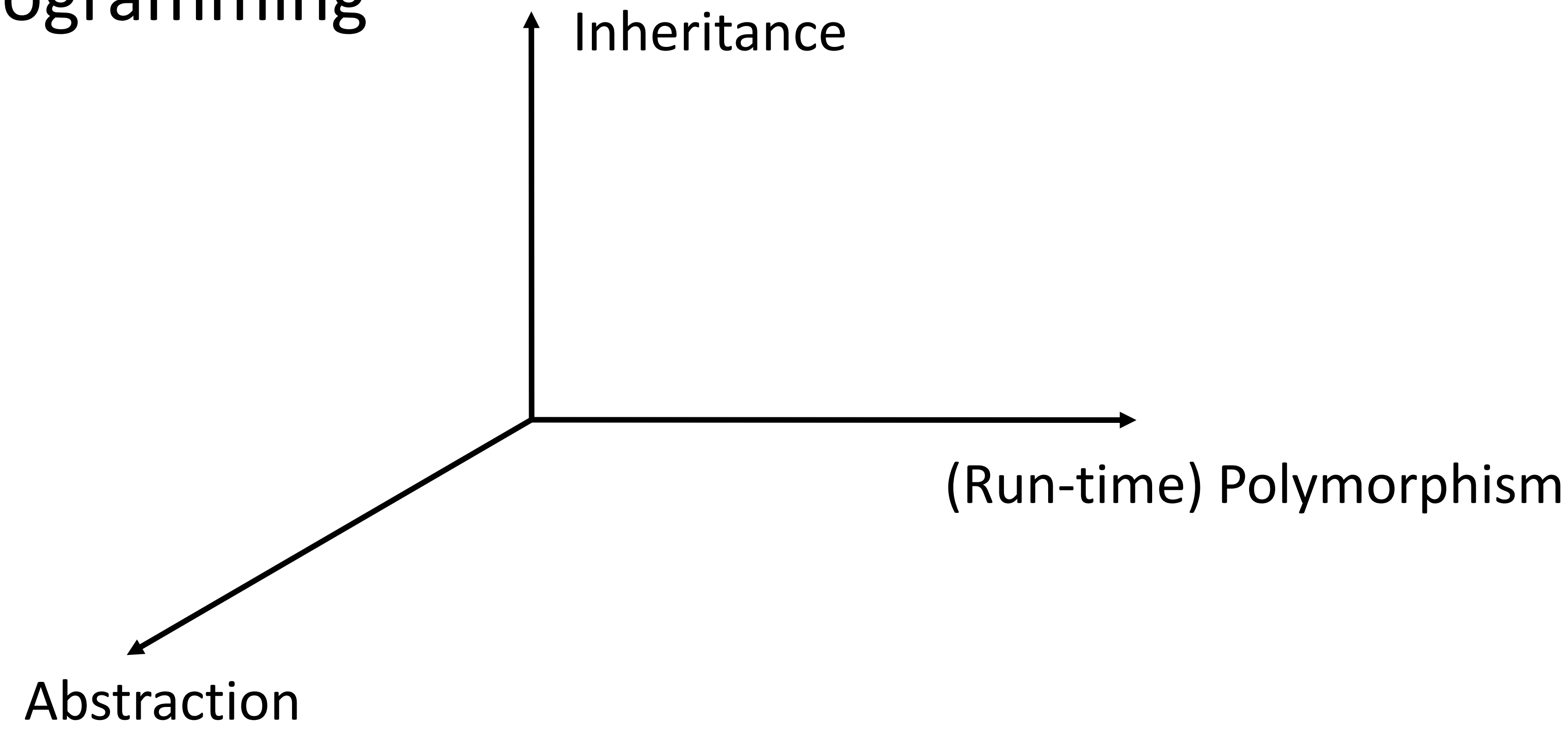
Class

Inheritance

(Run-time) Polymorphism

Abstraction

Object-Oriented Programming



- Abstraction: the ability to represent concepts directly in a program and hide incidental details behind well-defined interfaces.
- Inheritance: To build one class from another so that the new class can be used in place of the original.
- Polymorphism: the ability to provide the same interface to objects with differing implementations.

In OO context:

- Anytime you find yourself writing code of the form, “if the object is of type T1, then do something, but if it’s of type T2, then do something else, “ slap yourself. That isn’t The C++ Way. Yes, it’s reasonable strategy in C, in Pascal, even in Smalltalk, but not in C++, In C++, you use virtual functions.

  - Scott Meyers

Scott Meyers

Virtual member functions- Some rules

• the argument types specified for a virtual function in a derived class cannot differ from the argument types declared in the base.

```
class B {
public:
   virtual void f();
   virtual int g();
}
```

```
class D : public B {
public:
   void f();
   int g(double); // error
}
```

• A virtual function must be defined for the class in which it is first declared (unless it is declared to be a pure virtual function).

• A virtual function can be used even if no class is derived from its class, and a derived class that does not need its own version of a virtual function need not provide one.

Virtual vs. Non-virtual Member functions

- Non-virtual member functions are resolved:
  - statically (static binding)
  - at compile-time
  - based on type of pointer or reference to the object.

vs.

- Virtual member functions are resolved:
  - dynamically (dynamic binding)
  - at run-time
  - based on type of the object.

Pure virtual member functions

- All objects are represented by a class.
- All classes *do not necessarily* represent objects.

```
class Shape {
public:
  virtual void rotate(int);
  virtual void draw();
};
void Shape::draw() { error << "Shape::Draw\n"; }
void rotate(int i) {error << "Shape::Rotate\n"; }
```

Shape vs. Rectangle

Abstract concept — Concrete class/ object

```
Shape s; // silly: shapeless shape
s.draw(); // print error message
s.rotate(20); // print error message
```

- A better alternative is to declare the virtual functions of class Shape to be *pure virtual functions*.

```
// Shape concept
class Shape { // abstract class
public:
  virtual void rotate(int) =0; // pure virtual function
  virtual void draw() =0; // pure virtual function
  // ...
};
```

Abstract classes

- pure virtual function: virtual function that must be overridden in a derived class. Indicated by the curious *=0 syntax*.
- A class with one or more pure virtual functions is an abstract class.
- Decoupling Interface from Implementation
- An abstract class represents an interface. Direct support for abstract classes:

  - helps catch errors that arise from confusion of classes' role as interfaces and their role in representing objects;

  - supports a style of design based on separating the specification of interfaces and implementations.

- Examples:
  - Chess game: Piece
  - Portfolio Management: Asset
  - Operating system: File
  - Tehran Securities Exchange: Index
  - Nuclear Physics: Particle
  - Banking system: Account
  - Operating system: Character device
  - Hardware design: Logic gates

Interface inheritance vs. Implementation inheritance

- From the point of OO, C++ supports two kind of inheritance:
  - *Implementation inheritance*: to save implementation effort by sharing facilities provides by a base class
  - *Interface inheritance*: to allow different derived classes to be used interchangeably through the interface provided by a common base class
- The benefits of class hierarchy:

Implementation inheritance

A base class provides functions or data that simplifies the implementation of derived classes

vs.

An object of a derived class can be used wherever an object of a base class is required

Interface inheritance

Virtual destructor

```cpp
// Shape concept
class Shape {
public:
  // no constructor
  virtual void draw();
  virtual ~Shape() {}
};
void Shape::draw() { }
```

```cpp
class Circle : public Shape {
public:
  void draw() { /* … */ }
  ~Circle() { /* … */ } // overrides ~Shape()
};
```

```cpp
void user(Shape* p)
{
    p->draw();
    // …
    delete p; // invoke the appropriate dtor
}
```

override controls: Override

```
struct B {
    virtual void f();
    virtual void g() const;
    virtual void h(char);
    void k();    // not virtual
};

struct D : B {
    void f();    // overrides B::f()
    void g();    // doesn't override B::g() (wrong type)
    virtual void h(char); // overrides B::h()
    void k();    // doesn't override B::k() (B::k() is not virtual)
};
```

- To allow the programmer to be more explicit about overriding, we now have the "contextual keyword" override:

override controls: Override

```
struct B {
    virtual void f();
    virtual void g() const;
    virtual void h(char);
    void k();
};

struct D : B {
    void f() override;
    void g() override;
    virtual void h(char) override;
    void k() override;
};
```

- override is not an ordinary keyword but it is a *contextual* keyword, so you can still use it as an identifier:

```
int override = 7; // not recommended
```

override controls: final

- final specifier: Stopping inheritance
- Specifies that
  - a virtual function cannot be overridden in a derived class and
  - a class cannot be inherited from.

```
struct base {
    virtual void f();
    // …
};

struct derived: base {
    void f() final;
    // …
};
```

```
struct not_a_base final {
// …
};
```

- final is not an ordinary keyword but it is a *contextual* keyword, so you can still use it as an identifier:

```
int final = 7; // not recommended
```

final- more details

- If it is performance (in-lining) you want or you simply never want to override, it is typically better not to define a function to be virtual in the first place.

override Specifiers- virtual, pure virtual, final and override

- virtual: The function may be overridden.
- =0: The function must be virtual and must be overridden.
- override: The function is meant to override a virtual function in a base class.
- final: The function is not meant to be overridden.

- Language rule:

In the absence of any of these controls, a non-static member function is virtual if and only if it overrides a virtual function in a base class.

Covariant return types

- The return type of an overriding function shall be either identical to the return type of the overridden function or *covariant* with the classes of the functions. **from *Committee Draft***
- Return type relaxation rule:
- If the original return type was B*, then the return type of the overriding function may be D*, provided B is a public base of D. Similarly, a return type of B& may be relaxed to D&

```
class B {
public:
  virtual B* f();
  virtual B* g(int);
};

class D : public B {
public:
    D* f();
    D* g(int);
};
```

Covariant return type- an example

```
class Shape {
public:
    virtual ~Shape() { }
    virtual void draw() = 0;
    virtual void move() = 0;
    // ...
    virtual Shape* clone()  const = 0; // Uses the copy ctor
    virtual Shape* create() const = 0; // Uses the default ctor
};
```

```
class Circle : public Shape {
public:
    Circle* clone() const
    {
        return new Circle(*this); // make copy
    }
    Circle* create() const
    {
        return new Circle(); // make new object
    };
    // …
};
```

Virtual constructor

- There is no *direct support* for virtual constructor in C++.
- An idiom that allows you to do something that C++ doesn't directly support.
- Base facility to implement Abstract Factory pattern.

```
class Point {
public:
	virtual Point(); // error
};
```

1

```
class Shape {
public:
	virtual ~Shape() { }
	virtual void draw() = 0;
	virtual void move() = 0;
	// ...
	virtual Shape* clone()  const = 0; // Uses the copy ctor
	virtual Shape* create() const = 0; // Uses the default ctor
};
```

2

```
class Circle : public Shape {
public:
	Circle* clone() const
	{
		return new Circle(*this); // make copy
	}
	Circle* create() const
	{
		return new Circle(); // make new object
	};
	// …
};
```

3

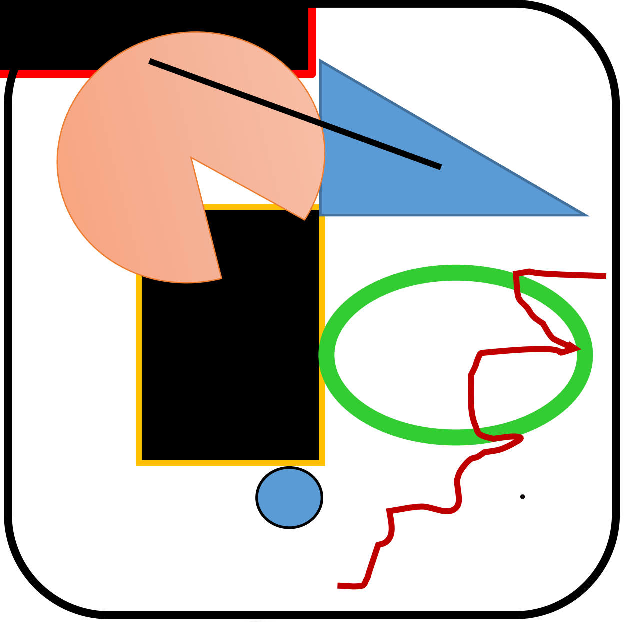
```
void user_code(Shape& s)
{
	Shape* s2 = s.clone();
	Shape* s3 = s.create();
	// ...
	delete s2;
	delete s3;
}
```

Virtual Constructor Test

ألفا ALPHA

Object-oriented and other programming styles together

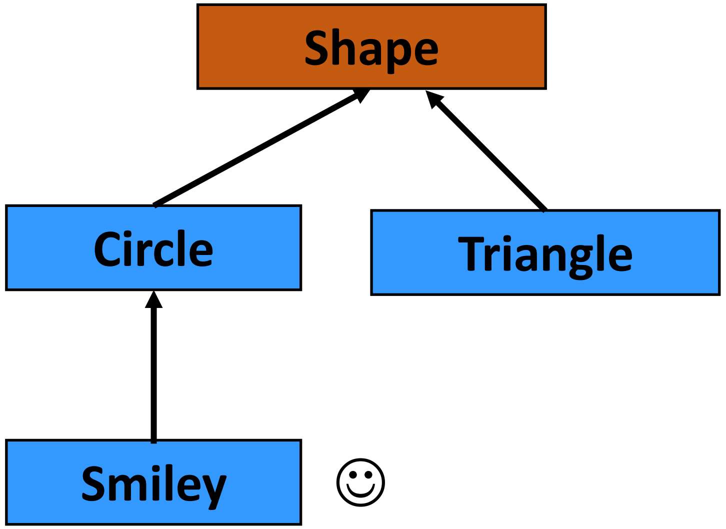
```
void rotate_and_draw(vector<Shape*>& vs, int r)
{
     // method #1
     for_each(vs.begin(),vs.end(), [](Shape* p) { p->rotate(r); }); // rotate all elements of vs
     // method #2
     for (Shape* p : vs) p->draw(); // draw all elements of vs
}
```



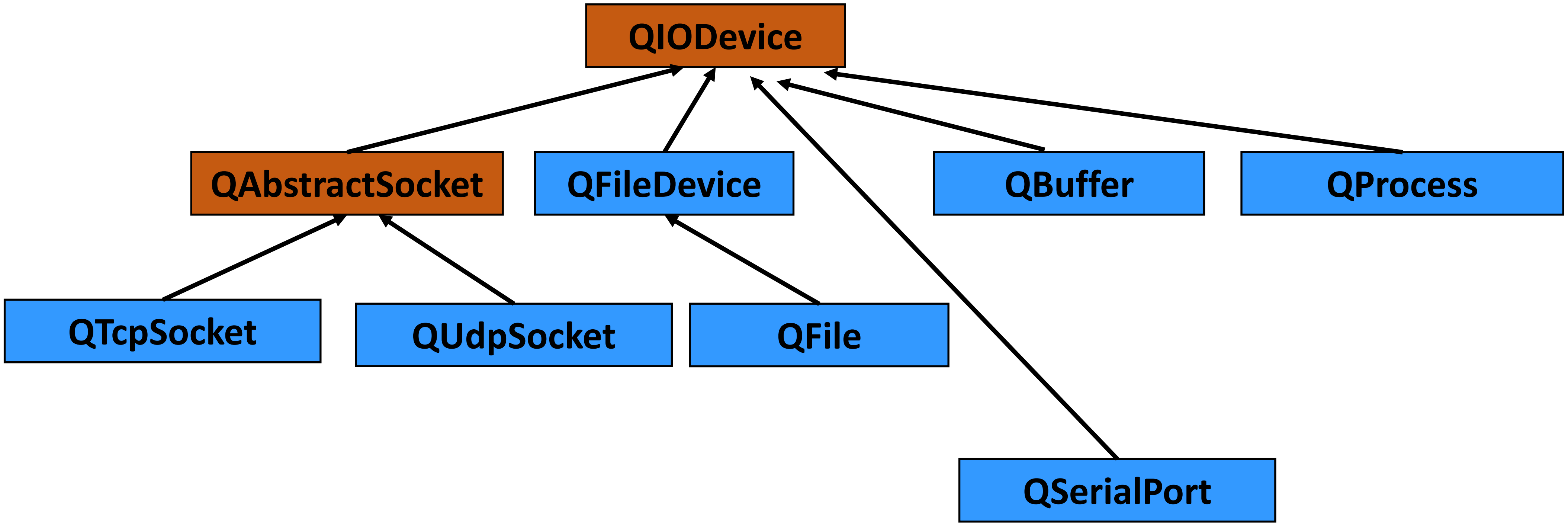
- Is this *object-oriented*? Of course it is; it relies critically on a class hierarchy with virtual functions.
- It is *generic*? Of course it is; it relies critically on a parameterized container (**vector**) and the generic function **for_each**.
- Is this *functional*? Sort of; it uses a lambda (the **[]** construct).
- Is this *Procedural*? Of course it is, the function rotate_and_draw.
- It is modern C++!

Class hierarchy

- A class hierarchy is a set of classes ordered in a lattice created by derivation. We use class hierarchy to represent concepts that have hierarchical relationship.
- A rose is a kind of flower and a flower is a kind of plant.
- A smiley face is a kind of a circle which is a kind of a shape.



Class hierarchy- An example from Qt



Accessibility of base classes and base class members

 # class member access specifier = 3
 # Base class member access specifier = 3
3 * 3 = 9 different states

```
class B {
private:
  int i;
protected:
  char c;
public:
  double d;
};
```

```
class D1 :public B {
};

class D2 :protected B {
};

class D3 :private B {
};
```

Derivation

Base class member Access Specifier

	public	protected	private
private	*private*	*private*	*private*
protected	*protected*	*protected*	*private*
public	*public*	*protected*	*private*

Inheritance　　Protected interface　　Composition

Private inheritance

```
class Foo : private Bar {
public:
    // …
};
```

- private inheritance is a syntactic variant of composition (AKA aggregation and/or has-a).

- Composition

```
class Engine {
public:
    Engine(int num_cylinders);
    void start();
};

class Car {
public:
    Car() : e_(8) { }
    void start() { e_.start(); }
private:
    Engine e_; // Car has-a Engine
};
```

- Private inheritance

```
class Car : private Engine { // Car has-a Engine
public:
    Car() : Engine(8) { }
    using Engine::start; // Start this Car by starting its Engine
};
```

Private inheritance vs. Simple composition

```
class Engine {
    // …
};

class Car {
    Engine e_; // Car has-a Engine
    // …
};
```

vs.

```
class Car : private Engine {
public:
    using Engine::start;
};
```

- Similarities

- In both cases there is exactly one Engine member object contained in every Car object
- In neither case can users (outsiders) convert a Car* to an Engine*

- Distinctions

- The simple-composition variant is needed if you want to contain several Engines per Car
- The private-inheritance variant can introduce unnecessary multiple inheritance
- The private-inheritance variant allows access to the protected members of the base class
- The private-inheritance variant allows Car to override Engine's virtual functions

ألفا ALPHA

Protected inheritance

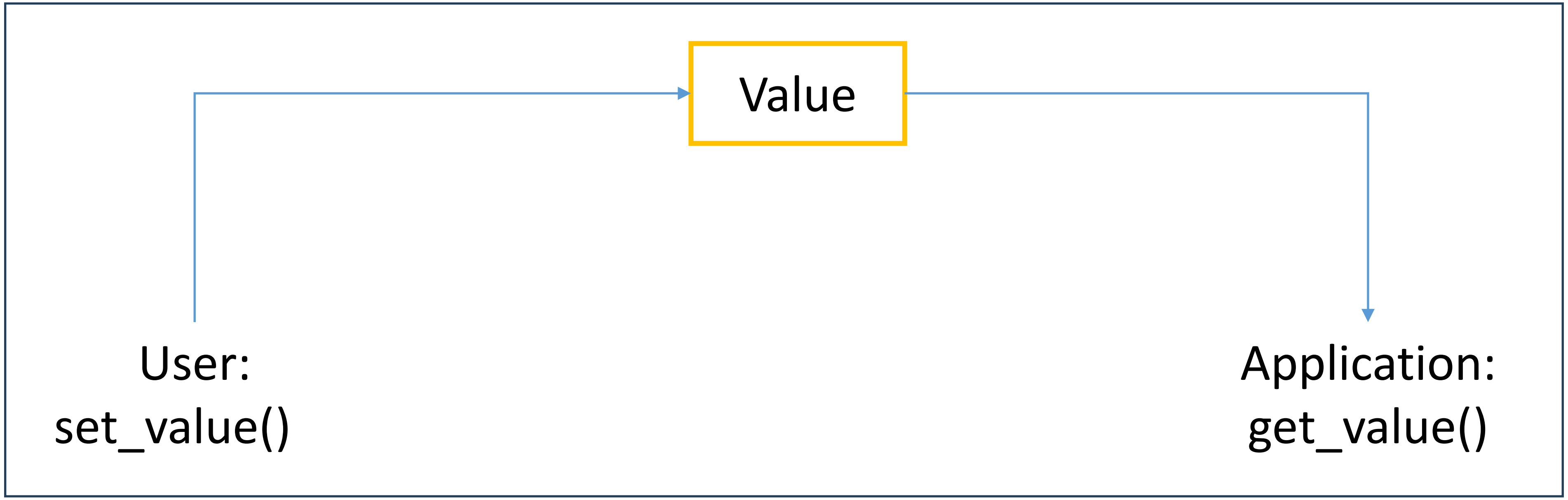
```
class Foo : protected Bar {
public:
    // …
};
```

Protected interface

- A class has two distinct interfaces for two distinct sets of clients:
  - It has a public: interface that serves unrelated classes
  - It has a protected: interface that serves derived classes

Design of Class Hierarchies: An example

- Problem: Provide a way for a program (“an application”) to get an integer value from a user.
- Invariant: Low <= Val < High
- Solution: Virtual user-interface system



- Graphical User Interface terminology: Controls or Widgets: Edit box, Text box, and …

IntValBox class

- Round 0

1

```
class IntValBox {
	int val;
	int low, high;
	bool changed{false}; // in-class member initialization
public:
	IntValBox(int ll, int hh) : val{ll}, low{ll}, high{hh} {}
	int get_value() { changed = false; return val; }
	void set_value(int i) { changed = true; val = i; }
	void reset_value(int i) { changed = false; val = i; }
	void prompt() { /* … */ }
	bool was_changed() const { return changed; }
	// other member functions
};
```

2

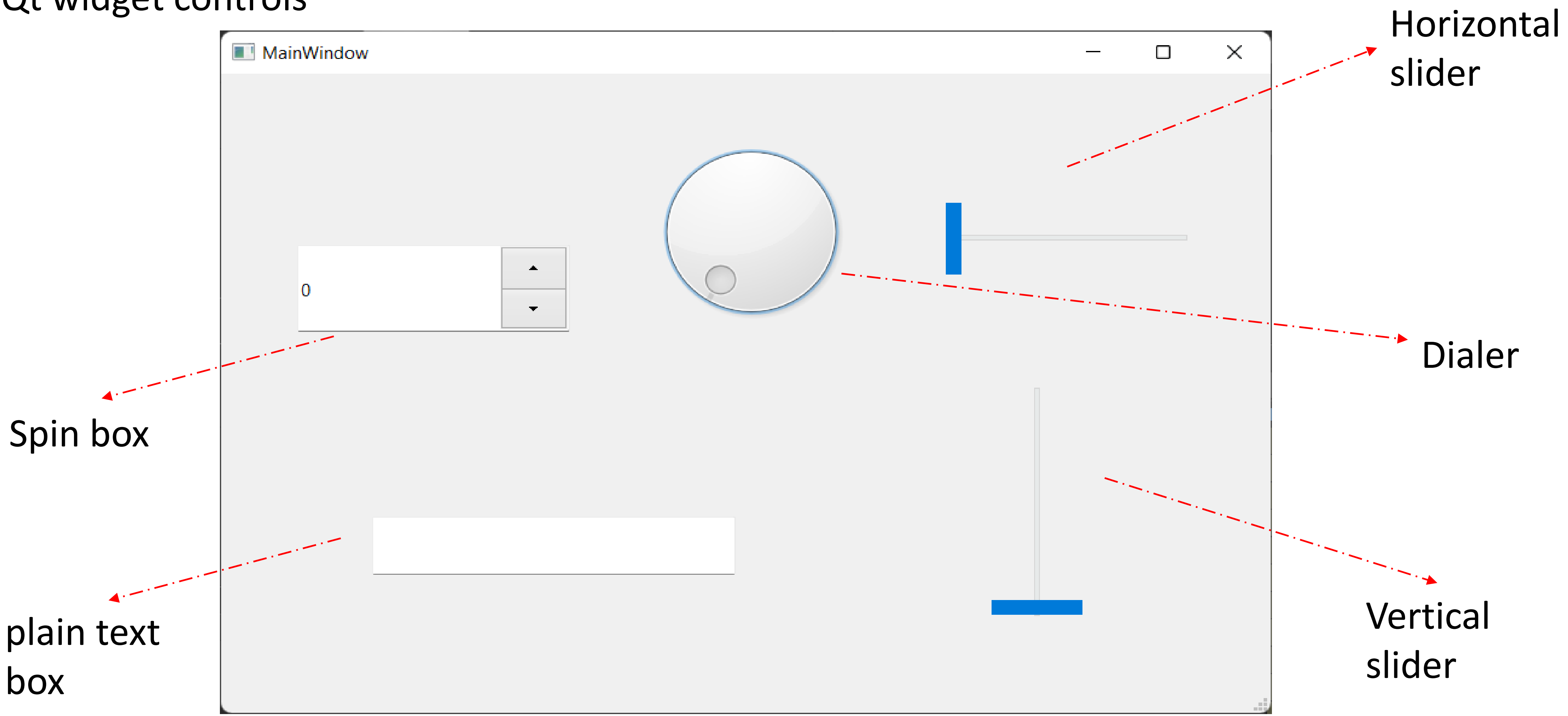
```
void interact(IntValBox b)
{
	b.prompt(); // alert user
	int i = b.get_value();
	if (b.was_changed()) {
		// new value: do something
	} else {
		// do something else
	}
}

void user()
{
	IntValBox b;
	interact(b);
}
```

- Problems: There are a lot of value box like widgets: Text box, Slider box, Dial box, Spin box

different kind of Input Values classes

- Qt widget controls



IntValBox classes

- Round 1

1

```
class IntValBox {
protected:
    int val;
    int low, high;
    bool changed{false}; // in-class member initialization
public:
    IntValBox(int ll, int hh) : val{ll}, low{ll}, high{hh} {}
    virtual int get_value() { changed = false; return val; }
    virtual void set_value(int i) { changed = true; val = i; }
    virtual void reset_value(int i) { changed = false; val = i; }
    virtual void prompt() { /* … */ }
    virtual bool was_changed() const { return changed; }
    virtual ~IntValBox() {}
    // other member functions
};
```

2

```
class IntValBoxSlider :public IntValBox {
    // private part
public:
    IntValBoxSlider(int ll, int hh);
    virtual int get_value() override;
    virtual void set_value(int) override;
    virtual void reset_value(int) override;
    virtual void prompt() override;
    virtual bool was_changed() const override;
    virtual ~IntValBox() {}
    // other member functions
};
```

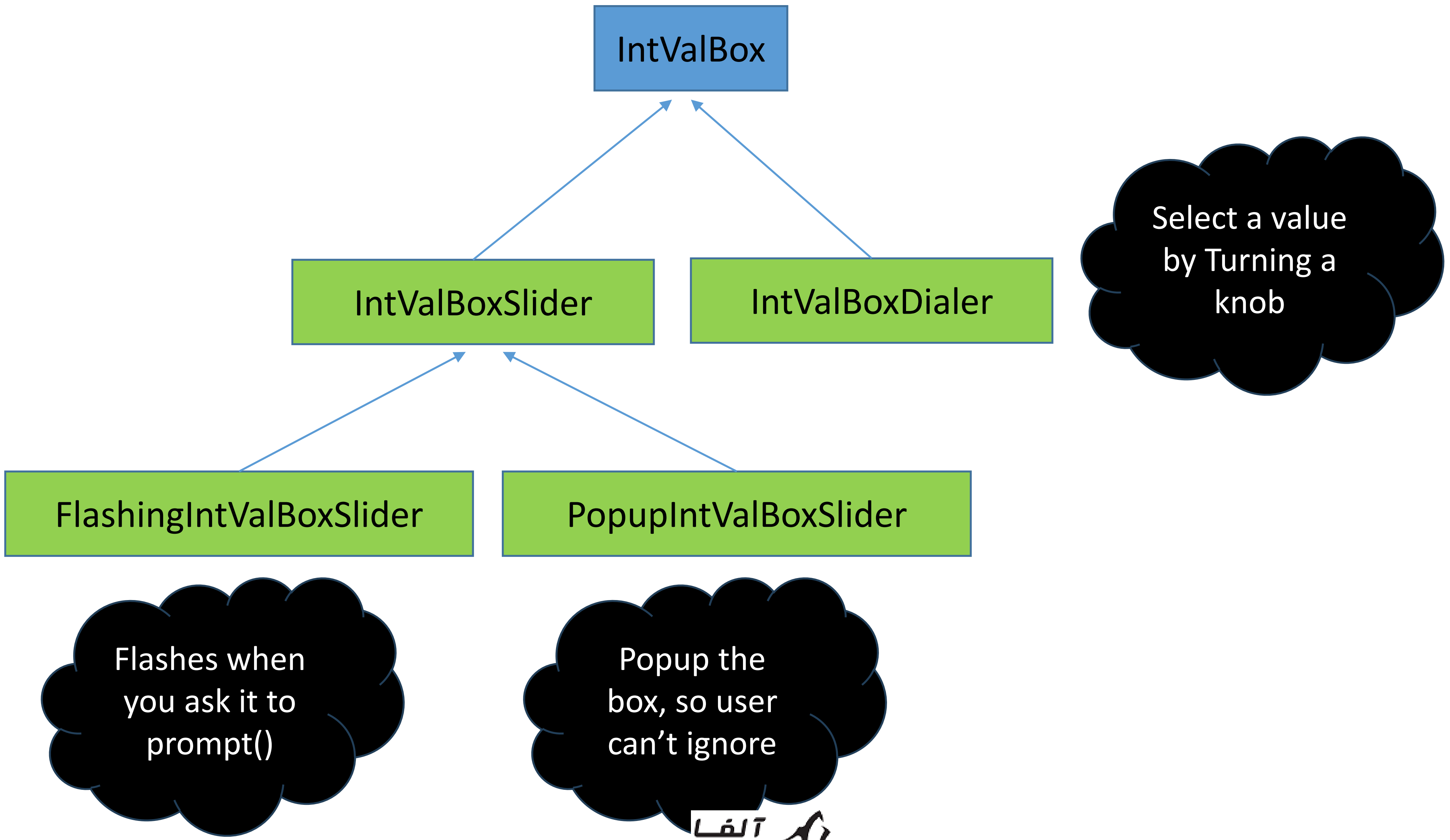
3

```
void interact(IntValBox* pb)
{
    pb->prompt(); // alert user
    int i = pb->get_value();
    if (pb->was_changed()) {
        // new value: do something
    } else {
        // do something else
    }
}

void user()
{
    unique_ptr<IntValBox> bp = new {IntValSliderBox{0, 100}};
    interact(bp);
}
```

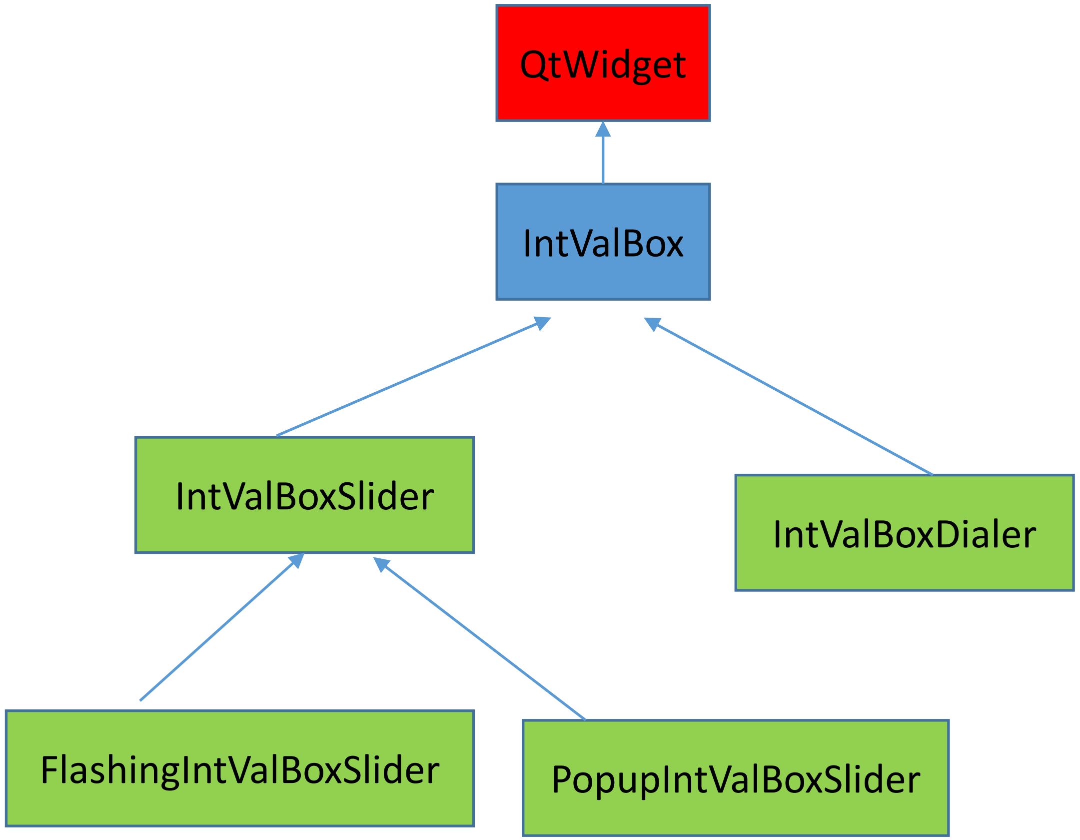
آلفا ALPHA

IntValBox class hierarchy

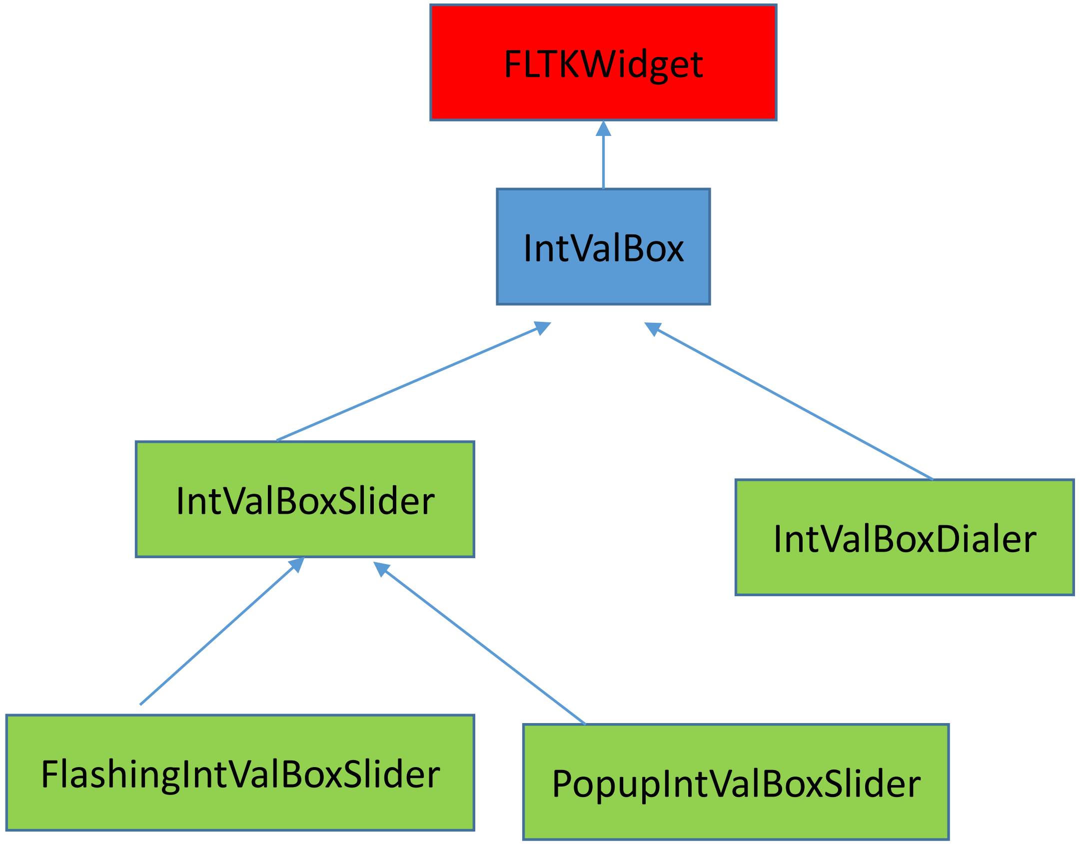


Graphics stuff

- The implementation of Widgets: OpenGL, .NET Framework, Qt, wxWidgets, FLTK, …
- Qt



- FLTK



Implementation

```
class IntValBox :public QtWidget { /* … */ };
class IntValBox :public FLTKWidget { /* … */ };
class IntValBox :public wxWidgetWidget { /* … */ }
// …
```

- The implementation detail leakage.
- Having many versions: maintenance nightmare.
- Every derived class shares the basic data declared in IntValBox.
- Changes to class QtWidget may force users to recompile or even rewrite their code.

Abstract IntValBox

```
class IntValBox {
public:
     virtual int get_value() =0;
     virtual void set_value(int i) =0;
     virtual void reset_value(int i) =0;
     virtual void prompt() =0;
     virtual bool was_changed() const =0;
     virtual ~IntValBox() {}
     // other member functions
};
```

```
class IntValBoxSlider :public IntValBox, :protected QtWidget {
public:
     IntValBoxSlider(int, int);
     virtual int get_value() =0;
     virtual void set_value(int i) =0;
     virtual void reset_value(int i) =0;
     virtual void prompt() =0;
     virtual bool was_changed() const =0;
     virtual ~IntValBox() {}
     // other member functions
};
```

...

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant

Q & A