```cpp
#include <iostream>

class Rectangle {
public:
    Rectangle(int width, int height) : width_(width), height_(height) {}
    int width() { return width_; }
    int height() { return height_; }
    int area() { return width() * height(); }
private:
    int width_;
    int height_;
};

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

int main() {
    Square s(4);
    std::cout << s.area() << std::endl;
}
```

When will be printed when running this code? Please critizise the design.

§x.x, title

I get:

```
g++ -Wall scratch.cpp && ./a.out
16
```

When finding classes: the classification should be of aspects of the concepts that we model in our system, rather than aspects that may be valid in other areas. [p703]

Inheritance relationships like the one above is very common, but usually very wrong. First of all, public inheritance from a class without virtual functions is usually a sign of bad design. Worse is the assumption that a Square is a Rectangle in computer science. One way to demonstrate the flaw of this design is to introduce, set_height and set_width to Rectangle. Eg,

```
class Rectangle {
  // ...
  void set_width(int w) { width_ = w; }
  void set_height(int h) { height_ = h; }
  // ...
}
```

Then you can do things like:

```
Square s(4);
s.set_width(5);
```

and suddenly the invariant of Square is broken.

This demonstrates a violation of Liskov Substitution Principle.

```cpp
#include <iostream>

struct Foo {
    void f() const { std::cout << "Foo" << std::endl; }
};

struct Bar {
    void f() const { std::cout << "Bar" << std::endl; };
};

template <typename T> void f(const T & t) {
    t.f();
}

int main() {
    f(Foo());
    f(Bar());
}
```

what might happen if you try to compile, link and run this program?

§x.x, title

I get:

```
g++ -Wall scratch.cpp && ./a.out
Foo
Bar
```

Suppose you should try to do this without a template? Then you might want to consider creating a common class for Foo and Bar, say Object, eg:

```
#include <iostream>

struct Object { virtual void f() const = 0; };

struct Foo : Object { void f() const { std::cout << "Foo" << std::endl; } };
struct Bar : Object { void f() const { std::cout << "Bar" << std::endl; } };

void f(const Object & o) { o.f(); }

int main() {
    f(Foo());
    f(Bar());
}
```

```cpp
#include <iostream>

template<typename T> void p(T x) { std::cout << x; }

class Foo {
public:
    void do_it() { p(1); func(); }
private:
    virtual void func() { p(2); }
};

class Bar : public Foo {
public:
    void do_it() { p(3); func(); }
private:
    void func() { p(4); }
};

void run(Foo & f) {
    f.do_it();
}

int main() {
    Foo f;
    Bar b;
    run(f);
    run(b);
}
```

what might happen if you try to compile, link and run this program?

§x.x, title

I get:

```
g++ -Wall scratch.cpp && ./a.out
scratch.cpp:5: warning: ’class Foo’ has virtual functions but non-virtual destructor
scratch.cpp:12: warning: ’class Bar’ has virtual functions but non-virtual destructor
1214
```

If a virtual function is meant to be used only indirectly by a derived class, it can be left private. [p738]

How to get 1234? Eg, by adding virtual to line 7.

Would it make any difference if func() on line 16 was public?

```cpp
#include "Foo.hpp"
#include "Bar.hpp"

class Gaz {
    Foo * f;
    Bar & b;
public:
    // ...
};
```

Please critizise this code.

§x.x, title

To reduce the dependency from Gaz to Bar and Foo, you might consider, in this case, to replace the include statements with a forward declaration. [p760]

```cpp
#include <iostream>

// a silly implementation of a vector of integers
class IntVector {
    int v[1024];
public:
    int size() { return sizeof(v) / sizeof(v[0]); }
    int & operator[](int i) { return v[i]; }
};

// a silly attempt to make a safe version of IntVector
class SafeIntVector : public IntVector {
public:
    class OutOfRange {};
    int & operator[](int i) {
        if (i < 0 || i >= size())
            throw OutOfRange();
        return IntVector::operator[](i);
    }
};

int main() {
    SafeIntVector v;
    v[4] = 42;
    std::cout << v[4] << std::endl;
}
```

what might happen if you try to compile, link and run this program?

§x.x, title

I get:

```
g++ -Wall scratch.cpp && ./a.out
42
```

This is bad design, and is not the way to use subtyping. To wrap around an existing type you might consider composition and delegation, but you can also use private inheritance (also called implementation inheritance, since it does not change the type [p743]).

```cpp
#include <iostream>

class Action {
public:
    virtual void do_it() const = 0;
    virtual ~Action() { }
};

class SayGreeting : public Action {
    std::ostream & ostm_;
public:
    SayGreeting() : ostm_(std::cout) {}
    void do_it() const { ostm_ << "Hello" << std::endl; }
};

class SayGoodbye : public Action {
    std::ostream & ostm_;
public:
    SayGoodbye() : ostm_(std::cout) {}
    void do_it() const { ostm_ << "Good Bye!" << std::endl; }
};

void execute(const Action & action) {
    action.do_it();
}

int main() {
    execute(SayGreeting());
    execute(SayGoodbye());
}
```

what might happen if you try to compile, link and run this program?

§x.x, title

I get:

```
g++ -Wall scratch.cpp && ./a.out
Hello
Good Bye!
```

```
#include <iostream>

template<typename T> void p(T x) { std::cout << x; }

template <int min, int max> class IntRange {
    int value_;
public:
    class Error {};
    IntRange(int value) : value_(value) {
        if (value_ < min || value_ > max)
            throw Error();
    }
    IntRange operator=(int i) {
        return *this = IntRange(i);
    }
    operator int() {
        return value_;
    }
    // ...
};

int main() {
    IntRange<1,12> r(3);
    p(1);  r = 1;
    p(2);  r = 12;
    p(3);  r = 13;
    p(4);  r = 14;
}
```

what might happen if you try to compile, link and run this program?

§x.x, title

I get:

```
g++ -Wall scratch.cpp && ./a.out
terminate called after throwing an instance of 'IntRange<1, 12>::Error'
123
Compilation exited abnormally with code 134 at Tue Dec  4 01:23:41
```

Interface classes are often used to provide C++ interfaces to non-C++ code and to insulate application code from the details of libraries. [p781]

An interface class that controls access to another class or adjusts its interface is sometimes called a wrapper. [p781]