Delayed portfolio visibility

- I've (just) worked out why there is a delay
  - *The problem*: I push on Monday, but it doesn't show up for you till later
  - *The cause*: there was an intermediate step that only gets run occasionally. Only happens for Q3 portfolio…
    - I thought I had a great idea, 6 months ago
  - *The solution*: bypass the intermediate step
  - I can now force it to run: will happen after lecture
- To be fair the deadline needs to be moved back
  - They are quick, but still take time
  - Fri 14th 22:00 -> Mon 10th 22:00

Potential impact of strikes

- There *may* be strikes by academic staff in the union
  - Could be called off and nothing happens
- This *may* affect teaching after mid-term
  - You'll probably hear about it from college + department
- Academics have a three-way ethical/moral dilemma
  - ***Union***: my actions have quite large positive impact
  - ***Students***: moral responsibility to reduce negative impact
  - ***Personal***: not lose three weeks pay for nothing
- What follows is my *personal* position for this course
  - Not departmental/college policy
  - Not union policy or commitment
  - Says **nothing** about what other staff members do

    (We are not really supposed to say what we'll do)

Impact of strikes on **this** course

1. **No assessment will rely on anything that was not taught**
2. **There should be no impact on grades (up or down)**
   - *Either*: assessment has been changed to avoid un-taught material;
   - *Or*: teaching will be adapted to still deliver material
   - The extra work needed for this started before Christmas
     - You're already on an adapted version of the course
     - It's been/being re-written already to support this

- In the worst case we lose 6 out of 10 lectures
  - I will not deliver or re-schedule lectures if on strike
  - **If the strikes continue there are topics that will not be taught**
- *But*: no-one can stop you doing labs
  - I will not withhold lab materials if on strike
  - Anything assessed that is not in lectures will be in labs
  - You might find labs are a bit longer and include new content

Why use sub-types?

Why have Triangle and Square?

- Why not do everything in Polygon?
  - It can draw itself
  - It can translate itself
  - You can get the corners/vertices
  - Anything you can do to a Triangle you can do to a Polygon

- The more specialised sub-classes could offer:
  - Extra functionality not available in base class
  - More efficient storage/representation
  - More efficient implementation of behaviour

Special-casing functions

We can add extra functions on the base

```
class Shape
{
public:
    virtual ~Shape()
    {}

    virtual void draw(ostream &dst) const =0 ;

    virtual float area() const=0;
};
```

Special-casing functions

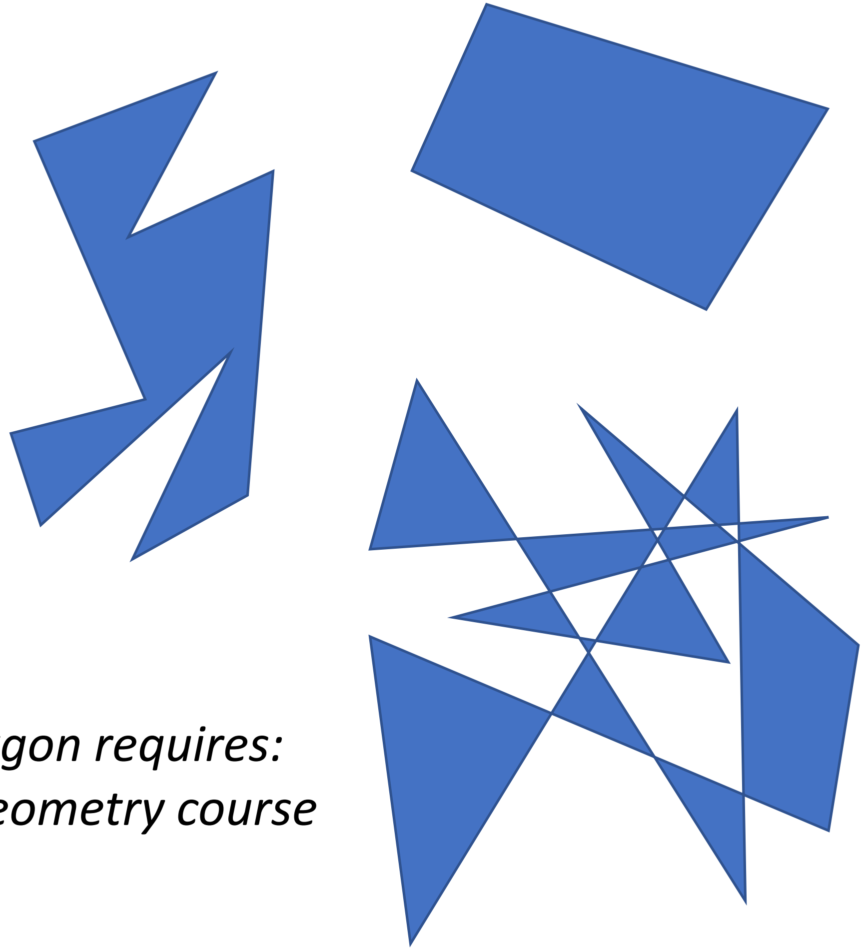
Then implement in derived classes

```
class Circle : public Shape
{
private:
    point m_centre;
    float m_radius;
public:
    float area() const
    {
        const float PI = 3.14159265358;
        return PI * m_radius * m_radius;
    }
};
```

Special-casing functions

General cases are often complex and slow

```
class Polygon : public Shape
{
private:
    vector<point> m_vertices;
public:
    float area() const
    {
        // TODO
    }
};
```



Calculating the area of an arbitrary polygon requires:
- An entire lecture of a computational geometry course
- Multiple auxiliary data-structures
- A few-hundred lines of code

Special-casing functions

Special cases can be fast and efficient

```
class Triangle : public Polygon
{
public:
    float area() const
    {
      assert(m_vertices.size()==3);

      point a=m_vertices[0], b=m_vertices[1], c=m_vertices[2];

      return (a.x*(b.y-c.y) + b.x*(c.y-a.y) + c.x*(a.y-b.y))/2;
    }
};
```

Inheritance
+ Constructors
+ Destructors

Object creation and destruction

- Each object instance has one concrete final type
  - But… it may have multiple base types

`Triangle` is a `Polygon` and a `Shape`

- Base types might still need constructors
  - Abstract base types might have data members
  - Concrete types can also be base types
- Base types might still need destructors
  - Might need to release resources from base class

Order of construction

- Object instances are constructed down the hierarchy
  - First construct the base class
  - Then construct the derived class
  - …
  - Finally: construct the concrete class

```
1. Shape::Shape()
2. Polygon::Polygon(const vector<point> &)
3. Triangle::Triangle(point p1, point p2, point p3)
```

```
int main()
{
    Triangle *tri=new Triangle({0,0},{0,1},{1,0});
    delete tri;
}
```

Order of destruction

- Object instances are constructed down the hierarchy
  - First call the concrete (most-derived) types destructor
  - Then call destructor of next base class
  - …
  - Then call final destructor of final base class

```
1. Triangle::~Triangle()
2. Polygon::~Polygon()
3. Shape::~Shape()
```

```
int main()
{
    Triangle *tri=new Triangle({0,0},{0,1},{1,0});
    delete tri;
}
```

Default constructors + destructors

- The compiler makes sure each step is followed
  - If you don't specify a constructor, the default is used
  - If you don't define a constructor, an implicit one is built
  - If you don't define a destructor, an implicit one is built

- The implicit (built-in) default constructor will:
  1. Call the default constructor of the base class
  2. Call the default constructor of member variables
- The implicit (built-in) destructor will:
  1. Call the destructor of member variables
  2. Call the destructor of the base class

Calling a specific base constructor

- Sometimes you need a non-default base constructor
  - You cannot call the base constructor like a function
  - It has to happen ***before*** the derived constructor runs
- You can call it using the base class name
  - Parameters can be used to supply constructor arguments

```
class Derived
    : public Base
{
    Derived()
        : Base()
    {}
};
```

```
class Derived
    : public Base
{
    Derived()
        : Base(4.5, “Hello”)
    {}
};
```

Constructing member variables

- Sometimes you want to construct member variables
- You can use the same syntax with member name

```
class MyClass
{
private:
    string m_string;
public:
    MyClass()
        : m_string("Hello")
    {}
};
```

```
class MyClass
{
private:
    string m_string;
    vector<int> m_vector;
public:
    MyClass()
        : m_string("Hello")
        , m_vector({1,2,3,4})
    {}
};
```

Destructors and Constructors

- Objects always have well defined lifetimes
  - Constructors are called in order
  - Destructors are called in reverse order
- You can control access to constructors
  - Protected constructors only accessible to derived classes
  - Stop certain objects being directly constructed
- You can manage inherited and aggregated classes
  - Explicitly call constructors on base classes
  - Explicitly construct member variables

The goal is always to maintain a valid state

Potential problems with inheritance

Some common problems

- Object slicing when copying/assigning
- “Hiding” rather than overriding
- Calling virtual methods in constructor
- Confusing static and dynamic method selection

Object slicing

- Objects are often copied or assigned
  - Passing parameters by value
  - Assigning local variables
  - Returning results by value
  - When allocating new variables

```
D f(B b)
{
    C c;
    c=b;
    return c;
}

int main()
{
    A a(4.5);
    E *pe=new E( f(a) );
}
```

Object slicing

- Objects are often copied or assigned
  - Passing parameters by value
  - Assigning local variables
  - Returning results by value
  - When allocating new variables

```
D f(B b)
{
    C c;
    c=b;
    return c;
}

int main()
{
    A a(4.5);
    E *pe=new E( f(a) );
}
```

Constructing type B from type A

Object slicing

- Objects are often copied or assigned
  - Passing parameters by value
  - Assigning local variables
  - Returning results by value
  - When allocating new variables

```
D f(B b)
{
    C c;
    c=b;
    return c;
}

int main()
{
    A a(4.5);
    E *pe=new E( f(a) );
}
```

Assigning type C from type B

Object slicing

- Objects are often copied or assigned
  - Passing parameters by value
  - Assigning local variables
  - Returning results by value
  - When allocating new variables

```
D f(B b)
{
    C c;
    c=b;
    return c;
}

int main()
{
    A a(4.5);
    E *pe=new E( f(a) );
}
```

Constructing type D from type C

Object slicing

- Objects are often copied or assigned
  - Passing parameters by value
  - Assigning local variables
  - Returning results by value
  - When allocating new variables

```
D f(B b)
{
    C c;
    c=b;
    return c;
}

int main()
{
    A a(4.5);
    E *pe=new E( f(a) );
}
```

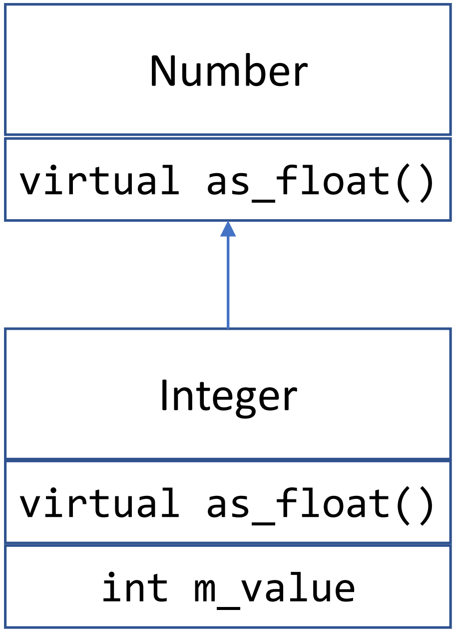
Constructing type E from type D

Object slicing

- Objects are often copied or assigned
  - Passing parameters by value
  - Assigning local variables
  - Returning results by value
  - When allocating new variables
- Inheritance makes this more complicated
  - We have static types and dynamic types
  - Construction uses the static type
  - Assignment usually uses the static type
    - Virtual assignment operators are legal but uncommon
- Object slicing: constructing Base from Derived
  - You may accidentally "slice" off parts of Derived class

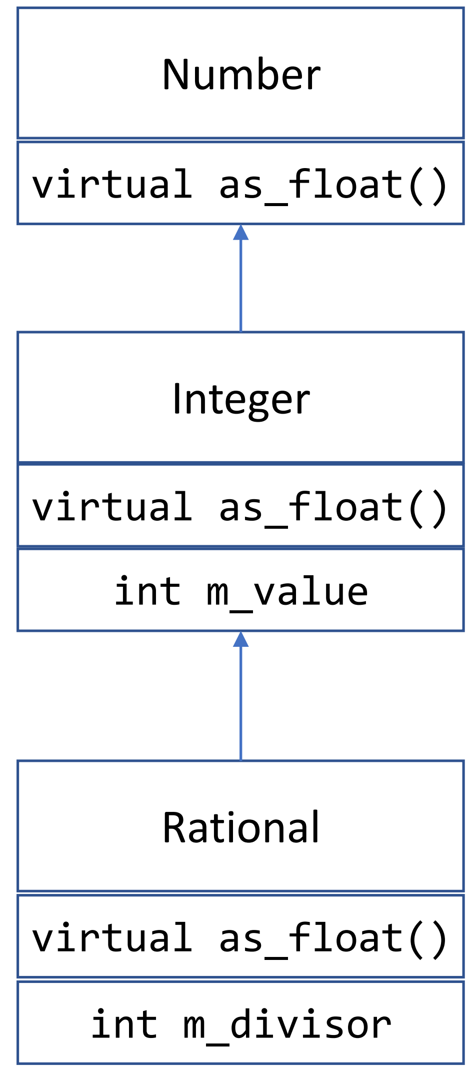
Object slicing : an example

```
class Number
{
public:
    virtual ~Number()
    {}

    virtual float as_float() const=0;
};

class Integer
    : public Number
{
protected:
    int m_value;
public:
    float as_float() const
    { return m_value; }
};
```



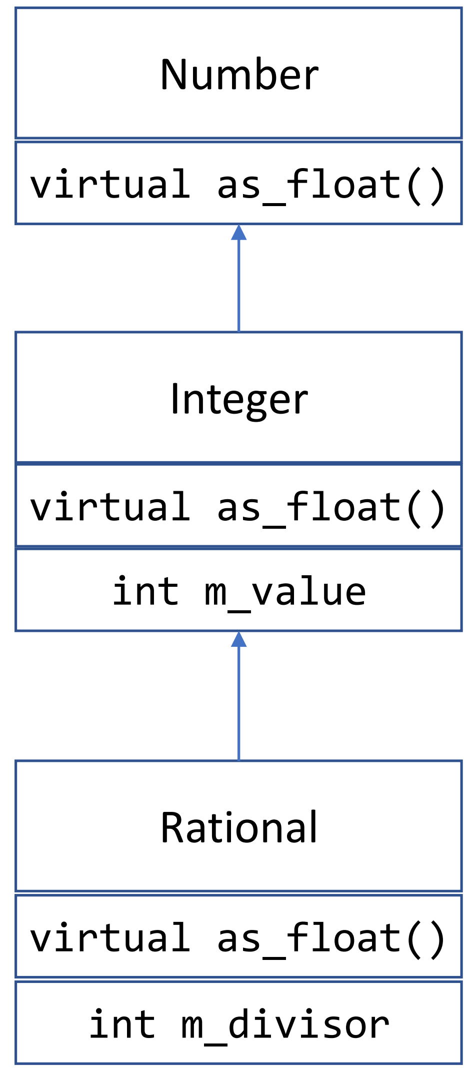
Object slicing : an example

```
class Integer
    : public Number
{
protected:
    int m_value;
public:
    float as_float() const
    { return m_value; }
};

class Rational
    : public Integer
{
protected:
    int m_divisor;
public:
    float as_float() const
    { return m_value / (float)m_divisor; }
};
```



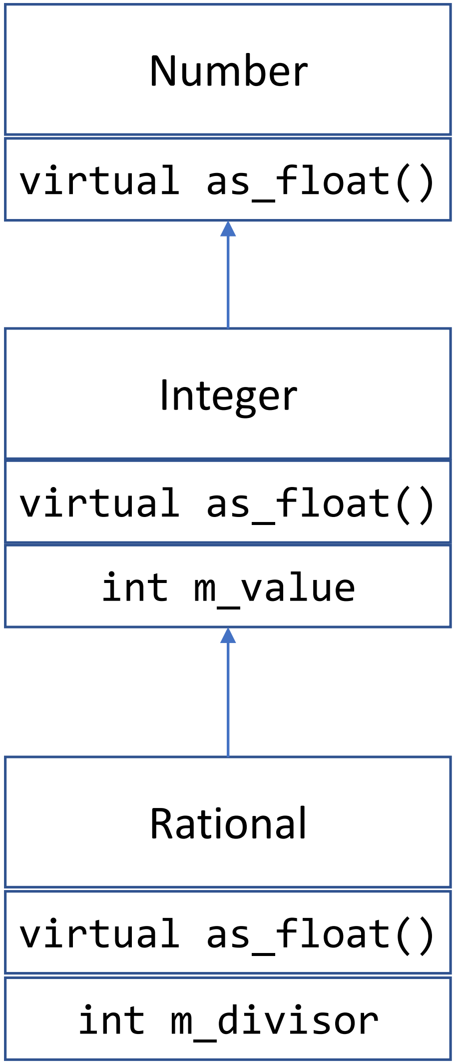
Object slicing : an example

```
ostream &operator(ostream &dst, Integer x)
{
    dst << x.as_float();
}

int main()
{
    Rational r{ 32 , 31}; // ~ 1.03226
    cout << r << endl;    // prints 31
}
```



Object slicing : an example

```
ostream &operator(ostream &dst, Integer &x)
{
    dst << x.as_float();
}

int main()
{
    Rational r{ 32 , 31}; // ~ 1.03226
    cout << r << endl;    // prints 1.03226
}
```



Avoiding object-slicing

1. Think carefully about inheritance
    - If you can slice off data, are the relationships correct?
    - Are the Rationals a sub-set of the Integers?
    - Might make sense to invert the relationship
        - Integers are derived from Rationals;
        - Plus: integers always have a denominator of 1
2. Control access to constructors and assignment
    - Do you really want the implicit copy constructor?
    - Should constructors be non-public
    - Should the base-class be abstract?

Hiding rather than overriding

You might think you are overriding a method,
but names are not enough: the compiler is very picky

```
class Integer
{
protected:
    int m_value;
public:
    float as_float() const
    { return m_value; }
};

class Rational
    : public Integer
{
protected:
    int m_divisor;
public:
    float as_float()
    { return m_value / (float)m_divisor; }
};
```

Hiding rather than overriding

You might think you are overriding a method,
but ***virtual*** is not enough: the compiler is very picky

```
class Integer
{
protected:
    int m_value;
public:
    virtual float as_float() const
    { return m_value; }
};

class Rational
    : public Integer
{
protected:
    int m_divisor;
public:
    float as_float()
    { return m_value / (float)m_divisor; }
};
```

Hiding rather than overriding

Everything must match to successfully override

```
class Integer
{
protected:
    int m_value;
public:
    virtual float as_float() const
    { return m_value; }
};

class Rational
    : public Integer
{
protected:
    int m_divisor;
public:
    float as_float() const
    { return m_value / (float)m_divisor; }
};
```

Hiding rather than overriding

- To successfully override you need an exact match
  1. Method is marked as virtual in the base class
     - If it's not virtual, then the derived class will *hide* the method
  2. Method in derived class matches virtual method
     - Name; Parameter types; Return type; Const or non-const
     - If declarations don't match, then you are hiding via an overload
- Pure virtual methods can help here
  - The compiler complains if you didn't successfully override it
- The `override` modifier lets you be explicit to compiler
  - A new feature of C++11 designed to help with accidental
  - "*If this method doesn't override something, then tell me*"

Hiding rather than overriding

Everything must match to successfully override

```
class Integer
{
protected:
    int m_value;
public:
    virtual float as_float() const
    { return m_value; }
};

class Rational
    : public Integer
{
protected:
    int m_divisor;
public:
    float as_float() const override
    { return m_value / (float)m_divisor; }
};
```

*This method is **intended** to override a method*

Hiding rather than overriding

Everything must match to successfully override

```
class Integer
{
protected:
    int m_value;
public:
    virtual float as_float()
    { return m_value; }
};

class Rational
    : public Integer
{
protected:
    int m_divisor;
public:
    float as_float() const override
    { return m_value / (float)m_divisor; }
};
```

<source>:15:33: error: non-virtual member function marked 'override' hides virtual member function

```
    float as_float() const override
^
```

<source>:6:24: note: hidden overloaded virtual function 'Integer::as_float' declared here: different qualifiers (unqualified vs 'const')

```
    virtual float as_float()
```

Virtual methods in constructor

- Objects change type during construction
  - We call the constructors for bases before derived
  - If we want to create a `Triangle` we have three constructors:
    1. `Shape::Shape()`
    2. `Polygon::Polygon()`
    3. `Triangle::Triangle(p1,p2,p3)`
  - Each constructor refines/extends from base to derived instance
- What happens if we call `` `draw()` `` in `Shape::Shape()`?
  - The instance is not *yet* a triangle
  - The point members are not *yet* initialised
- A general principle: “No virtual methods during constructor”
  - C++ does say what will happen, but it can easily confuse
- The same applies for destructors : no virtual calls

Confusing static and dynamic types

- For any instance you have two types:
  - ***Static type***: the type used to access instance in code
  - ***Dynamic type***: the true type of the instance itself

```
int main()
{
    int x;       // (x)      : Static=int,     Dynamic=int
    int *y=&x;   // (*x)     : Static=int,     Dynamic=int

    Base b1;     // (b1)     : Static=Base,    Dynamic=Base
    Base &b2=b1;// (base)   : Static=Base,    Dynamic=Base

    Derived d1; // (d1)     : Static=Derived, Dynamic=Derived
    Base &d2=d1;// (d2)     : Static=Base,    Dynamic=Derived
}
```

Confusing static and dynamic types

- Types control how methods are called
  - *Non-virtual* : depends on static type
    - Fixed at compile-time based on pointer or reference type
  - *Virtual* : depends on dynamic type
    - Selected at run-time based on actual instance's type
- General tips:
  - If you want virtual dispatch, pass references or pointers
  - Try to keep base classes abstract
    - Avoids you accidentally instantiating them
  - Avoid mixing overloading and virtual methods
    - *Overloading*: happens at compile-time
    - *Virtual methods*: happens at run-time
    - *Suggestion*: avoid multiple overloads of virtual methods
  - Logging: print things to cerr

Basic logging

```
class Integer
{
protected:
    int m_value;
public:
    float as_float() const
    {
       cerr << "Integer::as_float()" << endl;
       return m_value;
    }
};
```

Print logging information to `cerr`*,* ***not*** *to* `cout`*.*

cout = Genuine program output

cerr = Information about program execution

```
class Rational
    : public Integer
{
protected:
    int m_divisor;
public:
    float as_float() const override
    {
        cerr << "Rational::as_float()" << endl;
        return m_value / m_divisor;
    }
};
```

Objects : summary

Why use objects?

- Objects are there to provide interfaces
  - You cannot understand an entire code-base
  - You might be able to understand object interactions
- Bugs are usually down to corrupted state
  - Calling a function with an invalid parameter
  - Modifying a structure so that it has an invalid state
- *Encapsulation* provides abstracted states via functions
  - Users cannot access member data directly
  - Methods move objects between valid states
  - Users are not able to change or corrupt state

Objects and Polymorphism

“Polymorphism” : *using a single interface (API) to access multiple types or implementations*

We have seen three types of polymorphism:

- **Overloading** : “ad-hoc” polymorphism
- **Templates**: parametric polymorphism
- **Inheritance** : sub-type polymorphism

Ad-hoc polymorphism: *overloads*

- Define different functionality for a fixed set of types
  - We have to manually provide a definition for each type
  - The types must be known at compile-time
  - The function will be selected at compile-time

An example: overloading << for ostream

- Every class can exploit the interface
- You have to explicitly provide a new overload

Parametric polymorphism: *templates*

- Define functionality based on an unknown type T
  - Only one definition of function or class is needed
    - We don’t know what the type T is when writing code
  - The types must be known at compile-time
  - The compiler will specialise code at compile-time

- Many examples in the STL:
  - Container classes : `vector<T>, list<T>`
  - Algorithms and functions: `min<T>, sort<T>`

Sub-type polymorphism: *inheritance*

- Extend and refine functionality of a base type
  - The base class defines a general class of behaviour
  - Any number of classes can derived from the base
  - The choice of implementation is made at run-time

- We've seen examples of this:
  - *Shapes*: dynamic selection of draw() or area()
  - *Rover*: dynamic selection between svg and action output

How to choose?

- No-one can tell you when to use each approach
  - Inheritance is not always the answer
  - Objects are not always the answer
- Good interfaces comes from experience
  - Trying to design interfaces (and realising why they are bad)
  - Reading documentation for existing interfaces
  - Extending existing interfaces and libraries
- APIs are harder and more valuable than code
  - Anyone can implement a function
  - Designing a system is much harder

Where we are now

- You know everything needed to “do” OOP
  - Using objects
  - Designing objects
  - There are a few minor details left uncovered
    - Exceptions, Namespaces, Member types
    - We’ll encounter them organically
- Our main goal now: create programs that solve problems
  - Implement and analyse algorithms
  - Design interfaces and data structures
  - Collaborate with other people