Basic Classes

Rob Hackman and Xiao-bo Li

Winter 2024

University of Alberta

Table of Contents

What is a class

In object oriented programming one definition of the term `class` is "a type which aggregates data and behaviour over that data (functions) together."

In Python *every* data type was a class. In C no data types were classes.

In C++ we can make `structs` that are OOP classes[1].

[1] C++ has another keyword `class` that we'll talk about later.

What is an object

An *object* is an instantiation of a class — the class is the type and the object is a piece of data that is that type.

```
l = [1, 2, 3]
```

In the Python above `l` is an object of the type `List` which is a class.

What is a method

A *method* (or member function) is a function that belongs to a class.

```
l = [3, 1, 2]
l.sort()
```

In the Python above `l` is an object of the type `List` which is a class. The method `sort` is being called on the `List` object `l` which means the class `List` must provide that method.

What is a field

A *field* (or member variable) is a piece of data that belongs to a class.

```
struct Rect {
  int x, y, w, h;
};
```

In the C++ code above the class `Rect` has four fields named `x`, `y`, `w`, and `h`. Since `Rect` doesn't have any methods yet some may not consider it a class, but we'll add some.

Classes in C++

Unlike in C we can give our `struct` types methods!

```
struct Rect {
  int x, y, w, h;
  int area() {
    return w*h;
  }
};
```

Using methods

The code below shows how to use the `area` method provided by the `Rect` class.

```
Rect r{1,2,5,10};
Rect q{2,3,2,2};
cout << r.area() << endl;
cout << q.area() << endl;
```

Using methods

```
struct Rect {
  int x, y, w, h;
  int area() {
    return w*h;
  }
};
```

When using our `area` method we got the right value for both `q` and `r`.

But what do `w` and `h` mean in the code for the function `area`. *Whose* `w`? *Whose* `h`? How does the compiled code for `area` know when we want `q`'s and when we want `r`'s?

The implicit `this` parameter

```
struct Rect {
  int x, y, w, h;
  int area() {
    return this->w*this->h;
  }
};
```

Every (non-static) method has an implicit parameter named `this` which is a pointer to the object on which the method was called.

Accesses to members within methods implicitly assumes you are accessing the field of the object pointed at by the `this` pointer. As such the code here is functionally identical to the code on the previous slide.

Table of Contents

Aggregate Initialization

What was meant by the following lines?

```
Rect r{1,2,5,10};
Rect q{2,3,2,2};
```

This brace initialization syntax initializes the fields of our class in the order we've declared them in the class. So in this case we are providing the values for `x`, `y`, `w`, and `h` for each structure.

This is called *aggregate initialization* and is only allowed for certain types. You can use aggregate initialization for arrays as well as *simple* user created types that meet a set of requirements.

The problem with aggregate initialization

Aggregate initialization is fine for truly *aggregate types*, types that simply group together data with no specific logic.

Aggregate initialization allows the client programmer to create their data and set the fields to *any* values they want, even leaving values uninitialized if they want!

```
Rect r{1,2,-5,3};
Rect q{7,8};
Rect z;
```

Here `r` has a negative width, which is nonsensical! As well, for `q` the user has not specified what to initialize `w` and `h` to. Was `z` initialized at all? Does this make sense for a rectangle? Should we allow this?

Initializing an ADT

Most of the ADTs we create will have *invariants*. Except for simple data types we won't really want the client programmer to be able to initialize the fields of our data type with any values they want as they can with aggregate initialization.

In C we wrote a function that created returned a pointer to our ADT type on the heap, which we initialized, and hid the type from the client programmer so the could only create it through our initialization function — the client could then never place our ADT on their stack if they wanted.

In C++ we can enforce control over the initialization of our objects without restricting the client programmer's options if we don't want to.

Constructors

A constructor is a special method that is invoked automatically when an object is created.

To define a constructor define a method whose name is just the name of the class, include any parameters you want the client programmer to provide you with for initialization.

```
1  struct Rect {
2    int x, y, w, h;
3    Rect(int x, int y, int w, int h) {
4      this->x = x;
5      this->y = y;
6      this->w = w > 0 ? w : 1;
7      this->h = h > 0 ? h : 1;
8    }
9  };
```

Constructor comment

One small comment on the constructor function above. We had to specify `this->x = x` because the parameter had the same name as the field. If the parameter was named differently we would not require the specification `this->`.

This constructor now always initializes the width and height of any created `Rect` to be at least 1, if the client programmer provides a value that is less than 1 it is ignored and 1 is used instead.

As soon as we define a constructor for our class the client programmer *must* provide arguments for a defined constructor to initialize an object of our type. Since we've defined only this one constructor every `Rect` object must be initialized with all four integer values.

Initializing `Rect` objects

For any type that has at least one constructor when defining an object of that type the compiler will call one of the constructors, whichever one matches the arguments supplied.

If no constructor matches the arguments supplied it is a compilation error.

```
1  Rect a{1,2,3,4}; // OK
2  Rect b{1,2,3}; // Error
3  Rect c{5,6,7,8,9}; // Error
4  Rect d; // Error!
5  Rect e{1,2,-1, 0}; // OK
6  // Above becomes width 1 height 1
7  Rect *p = new Rect; // Error!
8  Rect *z = new Rect{1,2,3,4}; // OK
```

The Default Constructor

The zero-parameter constructor is called the *default constructor*. It is called whenever the client programmer initializes an object with 0 parameters.

Not every type needs to have a default constructor, if it doesn't make sense for your type (i.e. there is no meaningful "default" value of your type) don't include one!

For a default constructed `Rect` let's make it a 1x1 rectangle at (0,0).

The Default Constructor — Code

```
1  struct Rect {
2    ...
3    Rect() {
4      x = 0; y = 0;
5      w = 1; h = 1;
6    }
7  };
8
9  int main() {
10   Rect r; // Default ctor
11   Rect q{1,2,3,4}; // parameterized ctor
12   Rect z{}; // Default ctor
13 }
```

The Default Constructor — Code

```
1   struct Rect {
2     ...
3     Rect() {
4       x = 0; y = 0;
5       w = 1; h = 1;
6     }
7   };
8
9   int main() {
10    Rect r; // Default ctor
11    Rect q{1,2,3,4}; // parameterized ctor
12    Rect z{}; // Default ctor
13  }
```

Student Example

Let's create a quick class to represent a student, some fields we'll include

Student Example

Let's create a quick class to represent a student, some fields we'll include

- How many courses they're taking

Student Example

Let's create a quick class to represent a student, some fields we'll include

- How many courses they're taking
- Their GPA

Student Example

Let's create a quick class to represent a student, some fields we'll include

- How many courses they're taking
- Their GPA
- Their student ID

Student Example

Let's create a quick class to represent a student, some fields we'll include

- How many courses they're taking
- Their GPA
- Their student ID

The first two can change, the third cannot.

`const` fields

We can declare fields of our classes as `const`, that means once an object of that type has been initialized the field may not be changed. We'll do so with our student ID field.

```
1   struct Student {
2     int crsCount;
3     float gpa;
4     const int sID;
5     Student(int crsCount, float gpa, int sID) {
6       this->crsCount = crsCount;
7       this->gpa = gpa;
8       this->sID = sID;
9     }
10  };
```

`const` fields — problem!

When we try to compile our code including our `Student` class we get a compilation error, one that tells us we're leaving a `const` field uninitialized, and also trying to write to a `const field` in our constructor.

`const` fields — problem!

When we try to compile our code including our `Student` class we get a compilation error, one that tells us we're leaving a `const` field uninitialized, and also trying to write to a `const field` in our constructor.

Problem: by the time our constructor runs the fields have already been allocated and initialized — we're not declaring anything in our constructor just assigning the fields. The code of the constructor body is **too late** to initialize fields.

Steps of object creation

When an object is created there are three[1] steps that take place.

1. First, space for the object is allocated.
2. Second, fields are initialized.
3. Third, the constructor body runs.

By the time the constructor body runs it's too late to to initilize fields — it's already been done. We need some way to tell the compiler what values should be used in the second step.

[1] Four, really, but we won't discuss the last in this class.

The member initialization list (MIL)

C++ allows us to specify how fields should be initialized
in step 2 of object creation using the *member initialization list (MIL)*.

```
struct Student {
  int crsCount;
  float gpa;
  const int sID;
  Student(int crs, float gpa, int sID) : crsCount{crs},
      gpa{gpa}, sID{sID} {}
};
```

The member initialization list (MIL)

Since the MIL specifies how to initialize our fields, we should prefer to use it *always*, rather than just when we need to for fields that must be initialized like `const` fields.

Why initialize a field to one value, then overwrite it with a new value in the constructor body? Wasteful!

MIL initialization order — a note

Note: The order in which the initialization of fields specified in the MIL takes place is the *declaration order* of those fields in the class — not the MIL! Try the following code out and see the error!

```
struct Foo {
  int x, y;
  Foo(int yp) : y{yp}, x{y} {}
};

int main() {
  Foo f(10);
  cout << f.x << " " << f.y << endl;
}
```

Protection after construction

Our constructors allow us to enforce our invariants when the client programmer first creates an object of our type, but what about after that? Consider the following code:

```
int main() {
  Rect r{1,1,3,4};
  r.w = -10;
}
```

The client programmer has modified the `Rect` object so that it has a negative width violating our invariant that width and height must be at least 1!

The access problem

We have the same problem we had in C, that the client can access the internals of our ADT and ruin them.

In C we solved this problem by hiding the implementation of our ADT and only declaring it in the header file the client would use, but this limited the client to only heap allocating our ADT.

How can we solve this problem elegantly in C++?

Table of Contents

Access Specifiers

In C++ we can use *access specifiers* to specify which code can access the members of our classes.

- `public` members of a class can be accessed in any code, generally only the methods that make up our interface should be public.
- `private` members of a class can *only* be accessed by code that is part of the class (methods of that class). Generally our fields should be private, and some helper functions may be private too.
- `protected` members of a class can *only* be accessed by code that is part of the class or derived classes — will not be relevant to us in this course.

Using Access Specifiers

```
struct Rect {
 private:
  int x, y, w, h;
 public:
  Rect(int x, int y, int w, int h) :
    x{x}, y{y}, w{w > 0 : w ? 1},
    h{h > 0 : h ? 1} {}
  Rect() : x{0}, y{0}, w{1}, h{1} {}
};
```

Everything after `private:` is private, accessible only to methods of this class. Everything after `public:` is public, accessible to code anywhere.

Private fields — protecting invariants!

```
int main() {
  Rect r{1,1,3,4};
  r.w = -10;
}
```

Now that the fields of `Rect` are declared private this code won't compile! It's illegal for code that isn't within a `Rect` method to touch private data of a `Rect`.

`struct` default visibility

As we saw when we declared fields in our `struct` without specifying access then they were public. That is because the default access for a `struct` is `public`. So we also could have defined our `Rect` as such:

```
struct Rect {
  Rect(int x, int y, int w, int h) :
    x{x}, y{y}, w{w > 0 : w ? 1},
    h{h > 0 : h ? 1} {}
  Rect() : x{0}, y{0}, w{1}, h{1} {}
 private:
  int x, y, w, h;
};
```

Note we don't specify `public:` before the section containing our constructors, because a `struct` has default visibility public.

The `class` keyword

The keywords `struct` and `class` are identical *except* for their default access. A `class` defaults to `private` while a `struct` defaults to `public`. So, we could also write our `Rect` as follows:

```
class Rect {
  int x, y, w, h;
 public:
  Rect(int x, int y, int w, int h) :
    x{x}, y{y}, w{w > 0 : w ? 1},
    h{h > 0 : h ? 1} {}
  Rect() : x{0}, y{0}, w{1}, h{1} {}
};
```

Table of Contents

Classes in headers

Much like in C in C++ we also break our files up into header (interface) files and .cc (implementation) files.

In our header files we generally prefer not to implement our code, so that changes to that code don't require recompilation of files that include the header.

In C we saw how we could forward declare functions, and this holds true in C++. How do we forward declare and implement class methods however?

rect.h

```
class Rect {
  int x, y, w, h;
 public:
  Rect(int x, int y, int w, int h);
  Rect();
  int area();
};
```

Here is the header file for our `Rect` class. Note that in the method declarations, just as any function declaration, we don't need to include parameter names. However, in this case we have chosen to do so as it's useful information for the client programmer about which parameter represents which field.

rect.cc

```
#include "rect.h"
Rect::Rect(int x, int y, int w, int h) : x{x}, y{y},
  w{w > 0 ? w : 1}, h{h > 0 ? h : 1} {}

Rect::Rect() : x{0}, y{0}, w{1}, h{1} {}

int Rect::area() { return w*h; }
```

Here is the implementation file for our `Rect` class. Notice that each function name must be prefixed with `Rect::` (the class name and the scope resolution operator) to specify that the function we're writing is a method of the given class.

You must use the scope resolution operator to define methods you've declared inside of the class definition but define outside of it.

Table of Contents

Private fields and client access

The point of private fields is to limit client access to the internals of an ADT. However, this means any code outside of the class itself cannot access these fields. Sometimes this is a problem — consider

```
1  int main() {
2    Rect r{1,2,3,4};
3    cout << "(" << r.x << ", " << r.y << ")";
4    cout << r.w << "*" << r.h << endl;
5  }
```

Most of the time the client shouldn't be able to access the internals of the ADT — they shouldn't even know about them, but in some cases some access is warranted like printing out our `Rect`

What to do?

Accessor and Mutator Methods

It is commony in OOP to provide client code access to fields of an ADT through *accessor (aka getter)* and *mutator (aka setter)* methods.

A getter methods provide read access to a field, while a setter methods provide controlled write access to a field.

Sample getter/setter methods

```
1   class Rect {
2     int x, y, w, h;
3    public:
4     ...
5     // Do not return by reference!
6     int getWidth() { return w; }
7     void setWidth(int nw) {
8       if (nw < 1) return;
9       w = nw;
10    }
11  };
```

Note: the getter should *not* return a reference, otherwise the client may mutate your width freely with the returned reference.

When to provide getters/setters

When should you provide getters/setters?

- When the client programmer has reasonable need to read a fields value exactly provide a getter for that field.
- When the client programmer should be able to mutate a field directly (with protection of invariants) provide a setter.

If you are just providing methods so that the client can do a particular calculation or operation (like printing out your type) perhaps consider that it is better instead to just provide that operation, rather than providing general use getter/setters!

For example, we wanted to be able to print our `Rect` objects out — we don't necessarily need to be able to read the fields of these objects directly.

I/O operators and Access

Since we want to provide the client the ability to print out our `Rect` objects, better to provide an overloaded output operator than general getters. So we try the following

```
1   ostream &operator<<(ostream &out, const Rect &r) {
2     out << "(" << r.x << ", " << r.y << ")";
3     return out << r.w << "*" << r.h << endl;
4   }
```

But the code doesn't compile, because this function is not a part of the `Rect` class and as such cannot access private fields. So what do we do? We could try making it a `Rect` method.

Overloading operatings as methods

So we try the following code to overload the output operator as a method of class `Rect`

```
1   class Rect {
2     int x, y, w, h;
3    public:
4     ...
5     ostream &operator<<(ostream &out, const Rect &r) {
6       out << "(" << r.x << ", " << r.y << ")";
7       return out << r.w << "*" << r.h << endl;
8     }
9   };
```

But *this* doesn't compile telling us we've given `operator<<` too many operands for a valid overload — but we've only given it two?

The implicit `this` parameter and overload operator methods

When overloading an operator as a method the implicit object itself takes the place of one of the operands (that is the object pointed at by the implicit `this` parameter).

However, this object takes the place of the *first* operand. If we try to use this knowledge to implement our output operator however that causes a problem

```
1   class Rect {
2     int x, y, w, h;
3    public:
4     ...
5     ostream &operator<<(ostream &out) {
6       out << "(" << x << ", " << y << ")";
7       return out << w << "*" << h << endl;
8     }
9   };
```

Overloaded I/O operators as methods — no good

Since the above overload is implemented as a method the *first* operand is the implicit `Rect` object, which means to use this operator we have write code like the following

```
1   int main() {
2     Rect r{1,2,3,4};
3     r << cout;
4     (r << cout) << endl;
5   }
```

This code *works* but goes against the C++ paradigm for how the output operator should be used, it is *not* recommended to do this.

The problem

So, we have a problem

- If we implement the I/O operators as standalone functions they are unable to access the private data inside a `Rect` object, so they cannot work.
- If we implement the I/O operators as member functions then their operands are in the wrong order, which is undesirable.

So what do we do? Write getters/setters for every field just so the I/O operators can use them? But then *everyone* gets access to those getters/setters! Nope, the answer is...

friends

We can declare a function as a `friend` function, which means it is *not* a member of the class but we provide it unfettered access to our private data.

```
1  class Rect {
2    int x, y, w, h;
3   public:
4    ...
5    friend ostream &operator<<(ostream &, const Rect &);
6  };
```

Declaring a function inside a class as a `friend` does not make it a member function, it just informs the compiler that this function (when defined) will be allowed to access the private data of `Rect` objects.

`friend` method example

```
1   class Rect {
2     int x, y, w, h;
3    public:
4     ...
5     friend ostream &operator<<(ostream &, const Rect &);
6   };
7
8   ostream &operator<<(ostream &out, const Rect &r) {
9     out << "(" << r.x << ", " << r.y << ")";
10    return out << r.w << "*" << r.h << endl;
11  }
12
13  int main() {
14    Rect r{1,2,3,4};
15    cout << r << endl;
16  }
```

`friends` weaken encapsulation

Try to limit the number of `friends` you declare. Each `friend` is one more place where there could be code violating your invariants, and another place you have to search when tracking down bugs in your ADT. This is what it means when it is said that `friends` weaken encapsulation.