Classes

Complex data types

Michael Burrell

April 23, 2024

Readings

Chapter 1 —1.5, 1.6

Chapter 2 —2.6

Chapter 7 —7.1, 7.2, 7.3, 7.4, 7.5, 7.6

Learning objectives

- Learn the syntax for defining new classes in C++
- Understand how to split a C++ file up into a .h and .cpp file
- See how objects can be useful to solve problems
- Use operator overloading for arithmetic-like objects
- Use `friend` to define how to print out objects

References

- Before we get into classes and objects, let's learn about references
- References aren't specific to objects, but they're used together often
- References are a convenient syntax when passing an argument by reference instead of by value

Example

With pointers

```
void foo(int *x) {
    *x *= 2;
}
// ...
int z = 3;
foo(&z);
```

With references

```
void foo(int &x) {
    x *= 2;
}
int z = 3;
foo(z);
```

References described

- A reference is a pointer, but the syntax used for it does not make it look like a pointer
- When you assign to a reference or use a reference in an expression, it is actually working with the value pointed to
- References cannot be “reseated”
  - It is not possible to change which value a reference is referring to
  - Effectively the pointer itself is `const`
- They are only used for function parameters and return types, and are only used for convenience

References example

Class exercise

Let's do an example with references. Let's see the Euclidian division example again.

Wrapping up references

- References are introduced with a & symbol in the type (instead of $*$)
- The syntax is more convenient than pointers sometimes
- We still need to use explicit pointers sometimes

What are classes?

- Classes take the idea of structs and put in more features
- Just like structs, they give the data representation for a complex data type
  - A data type which is composed of other data types
- Unlike structs, they go beyond data representation into code representation:
  - *methods* are functions which are defined within a class and operate on objects of that class
  - *constructors* define different ways of building an object
  - *destructors* can allow for automatic clean-up of an object

Why do we use classes?

- Classes allow for some practical benefits over structs
  - Constructors and destructors can automate some tasks
  - Inheritance (discussed later) offers a new type of polymorphism
- Classes also allow for some engineering/cultural benefits over structs
  - Grouping data representation and code together aids in encapsulation
  - With classes we will start to consider *visibility* of code

Classes and .h files

- It is possible to include an *entire* class in a .h file
    - Including all of the code for all of the methods
- Don't do this
- Although it will work, your code will be duplicated each time the header file is included
    - It will bloat your executable
    - You will get a deduction.

What goes in a .h file

In your .h file for a class, put:

- The class definition
- All member variables
- Prototypes for methods, constructors and destructors
- Member initializations for constructors
- Small “one liner” method bodies

Do *not* put:

- Larger method bodies

Method body goes in a .h file or not?

- By putting the body of a method in a .h file, you allow a compiler optimization called *inlining*
  - It avoids the overhead of a method call, which is useful in some performance-critical applications
- By putting the body of a method in the .cpp file instead, you avoid code bloat
- There's a tradeoff
- Choosing when to make this tradeoff comes with experience and profiling, which is beyond the scope of this course
  - In this course, we will allow (but not require) that method bodies be included in the .h file *only if* they are one simple statement

Example

student.h

```
class account {
    string id;
    int balance;
public:
    account(string id, int b=0) : id(id), balance(b) {}
    string get_id() const { return id; }
    void transfer_points_to(account, int);
};
```

Example

student.cpp

```
#include "student.h"

void account::transfer_points_to(account other, int amt) {
    this->balance -= amt;
    other.balance += amt;
}
```

Member initializations

```
account(string id, int b) : id(id), balance(b) {}
```

- In C++, constructors have a *member initialization* list before the curly braces in a constructor
- The syntax is:
  - First, the name of the *member variable* (e.g., `balance`)
  - Then, parentheses
  - Inside the parentheses, any *expression* given in terms of parameters or other member variables

Member initializations

Proper C++ code should:

- Initialize *all* member variables using member initialization syntax
  - If you don't, that member variable's default constructor will be implicitly invoked anyway
- Initialize them in the order they're declared (top down)
- Use the body of the constructor (between the curly braces) to do any further work *after* they've been given an initial initial value

Methods

- As mentioned previously, there are two places to define methods

  in the .h file — used only for short one-liner methods (in this course)

  in the .cpp file — the more usual place

- There are some syntactic considerations when defining methods in one place or the other

Inline definition

```
class complex {
    double r, i;
public:
    double magnitude() const { return sqrt(r*r + i*i); }
};
```

- When defining a function in the `.h` file, all class code is present
  - Don't worry too much about the `const` yet
  - I'll explain that in a minute
- Return type, method name, parameter list, then method body

In the .cpp file

complex.h

```
class complex {
    // ...
    double magnitude() const;
};
```

complex.cpp

```
double complex::magnitude() const {
    return sqrt(r*r + i*i);
}
```

It still must be *declared* with a prototype in the .h file.

In the .cpp file

complex.cpp

```
double complex::magnitude() const {
    return sqrt(r*r + i*i);
}
```

- Methods in a `.cpp` file are written at the global scope (outside of any `class` definition)
- They must include the name of the class, followed by a `::` double colon, followed by the name of the method
- Other than that, the syntax is the same

const

```
class complex {
    double r, i;
public:
    double magnitude() const; // accessor
    void rotate(double);      // mutator
}
```

- We have seen a method that has const after the parameter list
- This means that the object itself will not be changed by this method
  - It would be illegal to modify `r` or `i` inside a const method

const

```
class complex {
    double r, i;
public:
    double magnitude() const; // accessor
    void rotate(double);      // mutator
}
```

- Every method has a hidden parameter called `this`
- In `rotate`, `this` has a type `complex*`
- In `magnitude`, `this` has a type `complex const*`

```
class complex {
    double r, i;
public:
    double magnitude() const; // accessor
    void rotate(double);      // mutator
}
```

- By default, all members of a class are *private*
- The introduction of an access specifier (`public:`, in this case), will make all subsequent members have different visibility
- In this case, *r* and *i* are private, whereas *magnitude* and *rotate* are public

Access specifiers

private —the member is accessible only within the class. We will use this for most member variables, and only a few helper methods

protected —the member is accessible within this class and any child classes. We won't know what this means until we study inheritance

public —the member is accessible anywhere. We will use this for most methods, constructors, and the destructor

Operator overloading

- Similar to Python, C++ allows a feature called *operator overloading*
- We are allowed to redefine how operators (+, −, `!`, etc.) work for a particular data type
- It is used very frequently in C++
- We have actually already seen it with `cout` and `cin`
    - << and >> are overloaded operators
- We will now start defining our own operator overloadings

Example

```
complex &complex::operator+=(complex const &other) const {
    real += other.real;
    imaginary += other.imaginary;
}
```

- Operator overloading is done by naming a method that starts with operator
- After operator is the name of the operator to define for this method

Example

```
complex &complex::operator+=(complex const &other) const {
    real += other.real;
    imaginary += other.imaginary;
    return *this;
}
```

- We are constrained by the *number* of parameters
  - Binary operators (e.g., `+`) have one parameter
  - Unary operators (e.g., `!`) have no parameter
- But there is no restriction on the *types* of the parameters or return values

Example

```
complex &complex::operator+=(complex const &other) const {
    real += other.real;
    imaginary += other.imaginary;
    return *this;
}
```

- By convention, *updating operators* (+=, −=, ∗=, /=, %=, |=, etc.) return `*this` to be consistent with how these operators usually work

Conventions for arithmetic operators

Imagine your class is called X. By convention, the following are *usually* true:

Operator	**Return**	**Parameter**	`const?`
+=, −=, etc.	`X &`	`X const &`	
+, −, etc.	`X`	`X const &`	`const`
<, ==, etc.	`bool`	`X const &`	`const`
`-, !, ~`	`X`		`const`

Functions and methods

- In C++, there are multiple ways of overloading operators
- We've seen operator overloading with *methods*
    - The operator is overloaded *within* a class
    - The first operand of the operator is the implicit parameter (`this`) of the method
- We can do operator overloading with functions instead
    - The operator is overloaded *outside of* any class

Example

```
enum animal { rat, monkey };

ostream &operator<<(ostream &out, animal a) {
    return out << (a == rat ? "rat" : "monkey");
}

int main() {
    animal x = monkey;
    cout << x << endl;
    return 0;
}
```

Note that the functions have one explicit parameter, (instead of 1 implicit and 1 explicit).

cout

- Unlike Python, C++ does not have a `__str__` or `__repr__` method to convert directly to a string
- This is actually good from an efficiency standpoint
  - `cout` prints out objects without the memory overhead of constructing a string object first
- Instead, we overload how << works between an `ostream` object and our new data type

cout

```
ostream &operator<<(ostream &out, animal x) {
    return out << /* ... */;
}
```

By convention, any time we want to define how `cout` (and all instances of `ostream`) work with our custom object, we:

- Have a return type of `ostream &`
- Return a reference to the `out` object itself (or, equivalently, the result of using `out` with `<<`)
- The first parameter is of type `ostream &`
- The second parameter is either a value (if it's a very simple/small value) or a reference

cin

```
istream &operator>>(istream &in, animal &a) {
    string x;
    in >> x;
    if (x == "rat")          a = rat;
    else if (x == "monkey") a = monkey;
    else   in.setstate(std::ios_base::failbit);
    return in;
}
```

- Reading in is a little trickier than printing out
- Properly speaking, if you detect bad input, you should `putback` before setting `failbit`, but it is often tedious to do so

Friend functions

- Sometimes you want a function (external to your class) to be able to access your private variables
- Sometimes you want a different class to be able to access private variables
- In C++, this is done by using `friends`

Example friend function

fraction.h

```
class fraction {
    int num, denom;
public:
    /* ... */
    friend std::ostream &operator<<(std::ostream &,
        fraction const &);
};
```

fraction.cpp

```
ostream &operator<<(ostream &out, fraction const &f) {
    return out << f.num << "/" << f.denom;
}
```

Friend function notes

- The declaration of a `friend` *must* exist within the class definition
- If the body of the function is defined elsewhere, the keyword `friend` is no longer necessary

Friend classes

```
class binary_tree_node {
    int data;
    binary_tree_node *left, *right;
public:
    /* ... */
    friend class binary_tree;
};
```

Doing this would allow the binary_tree class full access to data, left, right in any of its code.

For the time being

- We won't use `friend classes` yet
- We will use `friend` functions occasionally to help overloading << for `cout`

Conclusion

- We can declare classes and construct objects
- We can overload operators
- We can print out objects by overloading <<