Classy

CS 211

Adam Sweeney

November 23, 2017

Wichita State University

Introduction

- As we saw with structures, relevant data can be gathered together, even if they are of different types
- But we ended up with some functions that now go where the `struct` goes, leaving us in the same predicament
- Turns out we can package those together as well

Agenda

- Classes
- Defining & implementing classes
- Using classes
- Collecting the basic principles of classes

Classes

Classes

- The biggest distinguishing feature is that a class allows us to collect data AND functions under a common name
- Collecting functions into a class along with data does change the way we approach writing our code a bit

A Basic Money Class

```
class Money {
public:
    Money();
    Money(int d, int c);
    Money(double money);
    void print();
    Money add(Money r);
private:
    int dollars;
    int cents;
};
```

- We can see some *struct*-ural similarties between a class and `struct`
- But there's a lot of new stuff going on here as well

A Basic Money Class

```cpp
class Money {
public:
    Money();
    Money(int d, int c);
    Money(double money);
    void print();
    Money add(Money r);
private:
    int dollars;
    int cents;
};
```

- We'll begin with the outermost part, the actual statement of a class definition
- It's almost identical to that of a `struct`, except that we type `class`
- We still have the { } which contain the data and functions, and we still need that terminating semi-colon

A Basic Money Class

```
class Money {
public:
     Money();
     Money(int d, int c);
     Money(double money);
     void print();
     Money add(Money r);
private:
     int dollars;
     int cents;
};
```

- Looking inside the `class`, we can see the two integers `dollars` and `cents`
- Except now, they appear under a line `private:`
- The indendation suggests that they belong to what looks like a private section
- All the functions are in a section called `public:` , and there are three extra functions with no return type

Public and Private?

- Our Money class is divided into two sections, public and private
- In a nutshell, the public section is accessible outside of the class, and the private section is not
- The idea here is that if data is worth holding in a class, it must also be worth keeping safe
- No matter where a C++ program is run, we expect a `double` to behave a certain way
    - Even if different systems implement doubles differently
- Class data should only be allowed to be manipulated in a controlled manner that we, as programmers, determine
- The public section is how others are able to use the class

A Bit of Vocabulary

- Recall procedural abstraction, the notion that functions should be independent black boxes
- For classes, the same idea is called *encapsulation*
  - A class holds all the data and functions it needs to be used properly
  - A class is self-contained (as much as possible, anyway)
  - A class can be dropped into another program and immediately used
- As we would expect, a class definition is called the *definition*
- The collected function bodies are called the *implementation*
- Any function or variable that belongs to a class is called a *member*
  - To more specific, we can have *member data* or *member functions*
- Classes are defined, and we call variables of a class type *objects*

Let's Look at a Class Function

```
void Money::print()
{
    if (dollars < 0 || cents < 0)
        cout << "(";

    cout << "$" << abs(dollars) << ".";

    if (abs(cents) < 10)
        cout << "0" << abs(cents);
    else
        cout << abs(cents);

    if (dollars < 0 || cents < 0)
        cout << ")";
}
```

- It's a pretty typical function that can output an amount of money to the screen
- The only new thing here is `Money::` appearing before the name of the function

Let's Look at a Class Function

```cpp
void Money::print()
{
    if (dollars < 0 || cents < 0)
        cout << "(";

    cout << "$" << abs(dollars) << ".";

    if (abs(cents) < 10)
        cout << "0" << abs(cents);
    else
        cout << abs(cents);

    if (dollars < 0 || cents < 0)
        cout << ")";
}
```

- Recall that the `print()` function is a member of the `Money` class
- The `::` operator is known as the *scope resolution operator*
- So, we are resolving the scope of the `print()` function so that it exists only within the `Money` class

Let's Look at a Class Function

```
void Money::print()
{
    if (dollars < 0 || cents < 0)
        cout << "(";

    cout << "$" << abs(dollars) << ".";

    if (abs(cents) < 10)
        cout << "0" << abs(cents);
    else
        cout << abs(cents);

    if (dollars < 0 || cents < 0)
        cout << ")";
}
```

- Recall that the private data members of the class are `dollars` & `cents`
- Note here that within a class function, I have direct access to those variables
- We only have this direct access from a class member function

Generic Look at Class Member Functions

```
ReturnedType ClassName::FunctionName(ParameterList)
{
    /*
     * Function body goes here
     * Function can use private data members directly
     */
}
```

- This is the general form of a class member function
- PITFALL: it is very easy to forget to scope the function to the class
- The compiler will give you an error stating that something is undefined if you forget

Defining & Implementing Classes

Defining & Implementing Classes

- We have had a look at a `Money` class
- We have discussed some of the new syntax
- Now we'll go a bit deeper

Building a Class

- Recall that a class is a collection of variables and functions
- When we declare an object, it would be nice if it was given to us in a valid, usable, and consistent state
- When objects are declared, we can think of them as needing to be built
- We can (and should) control that process to an extent

Constructors

```
class Money {
public:
    Money();
    Money(int d, int c);
    Money(double money);
    void print();
    Money add(Money r);
private:
    int dollars;
    int cents;
};
```

- The first three functions are a bit different
- There is no return type, and they are all named after the class
- These functions are known as *constructors*
- We can make sure that our objects are created consistently, and provide different ways to create objects

Constructors

```cpp
class Money {
public:
    Money();
    Money(int d, int c);
    Money(double money);
    void print();
    Money add(Money r);
private:
    int dollars;
    int cents;
};
```

- The first constructor `Money()`, is also known as the default constructor
- A default constructor never has parameters
- If we do not create **any** constructors, the compiler will provide a default constructor for us
- It is better to define it ourselves

Default Constructor

```
Money::Money()
{
    dollars = 0;
    cents = 0;
}
```

- The default constructor is used whenever a basic object is declared
- Like a string that is just declared, `string word;`
- Default values are up to us, but we want to choose values that make sense as defaults

More Constructors

```cpp
class Money {
public:
    Money();
    Money(int d, int c);
    Money(double money);
    void print();
    Money add(Money r);
private:
    int dollars;
    int cents;
};
```

- The next two constructors are parameterized constructors
- As the name implies, they have parameter lists that allow us to initialize our objects

Parameterized Constructors

```cpp
Money::Money(int d, int c)
{
    dollars = d;
    cents = c;
}

Money::Money(double money)
{
    int pennies = round(money * 100);
    dollars = pennies / 100;
    cents = pennies % 100;
}
```

- Parameterized constructors use their arguments to assign correct values to the appropriate data members

Constructors, Concluded

- It is generally the case that you will define more than one constructor for an object
- It is good practice to always provide the default constructor yourself
- The amount of constructors you write depends on how many ways to initialize your object make sense

A Function to Add Money

```
Money Money::add(Money r)
{
    int leftPennies = (dollars * 100) + cents;
    int rightPennies = (r.dollars * 100) + r.cents;

    int sum = leftPennies + rightPennies;
    Money temp(sum / 100, sum % 100);

    return temp;
}
```

- It is extremely similar to the addition function that was used for the Money `struct`
- A key difference is the fact that there is only a single parameter
- How can we add two Money objects if the function only has one?
    - The answer is that we do actually have two Money objects

Calling Objects, Redux

- Remember that class functions do **not** exist outside of the class
- The **only** way to call a class function is through an object
- The object that calls a class function is known as the *calling object*
- So, in the case of the `add()` function, we do actually have two Money objects to add together
- We have the calling object, and the Money object passed as an argument to the `add()` function

Looking at the Add Function Again

```
Money Money::add(Money r)
{
    int leftPennies = (dollars * 100) + cents;
    int rightPennies = (r.dollars * 100) + r.cents;

    int sum = leftPennies + rightPennies;
    Money temp(sum / 100, sum % 100);

    return temp;
}
```

- Note the difference in how `leftPennies` & `rightPennies` are assigned
- `leftPennies` is able to use the variables `dollars` & `cents` directly
- That is because these are variables that belong to the calling object

An Example

```
Money one(3, 50), two(2.24);
Money sum = one.add(two);
```

- Here, the Money object `one` is the calling object
- This means that the `add()` function is running from within `one`
- Because a class member function runs from within a calling object, it has direct access to the calling object's data members
- Looking back at the `add()` function, it should also be noted that it is able to directly access the private data members of the Money object `r`
- This is the property of class functions described earlier, where they have access to the private data of the class
- We can see that this is true, even for other objects *of the same type*, not just the calling object

A Couple Other Uses

- **If** you have the right constructor, you can do initialization like we would expect from a regular data type

```
int x = 5;
Money one = 5.67;
```

- This works **only** because we have a constructor that takes a double as a parameter
- This method of initialization or assignment will work any time there is a constructor that takes a single paramter
- This means that the constructor that takes two integers cannot be initialized or assigned in this way

I Did Say a Couple

- You can assign a Money object to another

```
Money one(3, 33);
Money two = one;
```

- The reasons for why this works is unimportant for an Intro course
- Unless you do want to know

- Refer to the `example.cpp` which is available on Blackboard if you want to play around with the Money class

Re-iterating the Principles

Re-iterating the Principles

- Beginning to learn about Object-Oriented Programming (OOP) brings with it a lot of new vocabulary and theoretical “baggage”
- We’ve discussed some best practices as we’ve gone through the course, but OOP takes it to another level
- So this section will review the principles that have been covered, and introduce another

Encapsulation

- Encapsulation is to OOP as Procedural Abstraction is to functions
- Our classes should be self-contained
- In fact, all the principles covered today fall under the umbrella of encapsulation

Data Members Should Be Private

- Encapsulation involves hiding as much information as possible from the user
- With functions, we used prototypes (forward declartions) to hide the function bodies
    - Users didn't need to know how they worked, just what the inputs and expected outputs were
- We can go further with classes by actually restricting access to parts of the class
- It is a best practice to make all data members of a class private

What if the User Needs to Change Private Data?

- As we've discussed, private data cannot be directly accessed outside of the class
- But sometimes we do want to allow the user to change that data
- The solution is public functions that can fetch or alter the private data for us

Accessors and Mutators

- Also known as getters and setters
- If we are going to allow users to know the values of private data, we write accessor functions
    - Also known as getters
    - They simply return the value of the data member
- If we are going to allow users to change the value of the private data, we write mutator functions
    - Also known as setters
    - These functions allow us to vet the change before applying it, or hide the real value from the user

Define the Default Constructor Yourself

- If no constructor is defined, then the compiler will provide a default one for you
- This default constructor can't read our minds, it doesn't know what good default values are
- If we define a single parameterized constructor and do not provide a default one ourselves, the compiler will not provide one for us
- This means that every object must be initialized, which may not be ideal behavior
- To avoid these possible issues, it is a best practice to always include a default constructor in your classes

The Concept of a Calling Object

- You won't get far in OOP if this evades you
- The only way to access a class function is through an object
- The object that is being used to call a class function is known as the calling object
- . . . Since it's the object calling the function