Classes

Lecture 17

Outline

1. Object Oriented Programming (OOP)
2. Making Your Own Classes
3. Member Access Level
4. Objects as Function Arguments
5. Calling Member Functions
6. Constructors

Review: File Streams

- Recall that **`ifstream`** and **`ofstream`** variables are used to represent files for input and output
- We use the **`open`**, **`fail`**, and **`close`** functions to work with file streams
- Functions are used on file stream variables differently, because they are actually C++ *classes*

File Stream Example

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream ofs;
    ofs.open( "tmp.txt" );
    if ( ofs.fail() )
    {
        cout << "File failed to open." << endl;
        return 1;
    }
    ofs << "Hello World" << endl;
    ofs.close();
    return 0;
}
```

ofs is a variable of type ofstream

ofs.open(string) is used to open a file

ofs.fail() is used to check if the file opened correctly

ofs.close() is used to close the file

File Stream Functions

- The **`open`**, **`fail`**, and **`close`** functions are part of the **`ofstream`** *class*

- When calling any of the class functions, you have to use a variable of type **`ofstream`** (termed an *instance* of the **`ofstream`** class; also called an *object*)
  - Generic form:
    `VARIABLE.FUNCTION( ARGUMENTS )`
  - Specific example: **`ofs.open( "tmp.txt" );`**

Object-Oriented Programming (OOP)

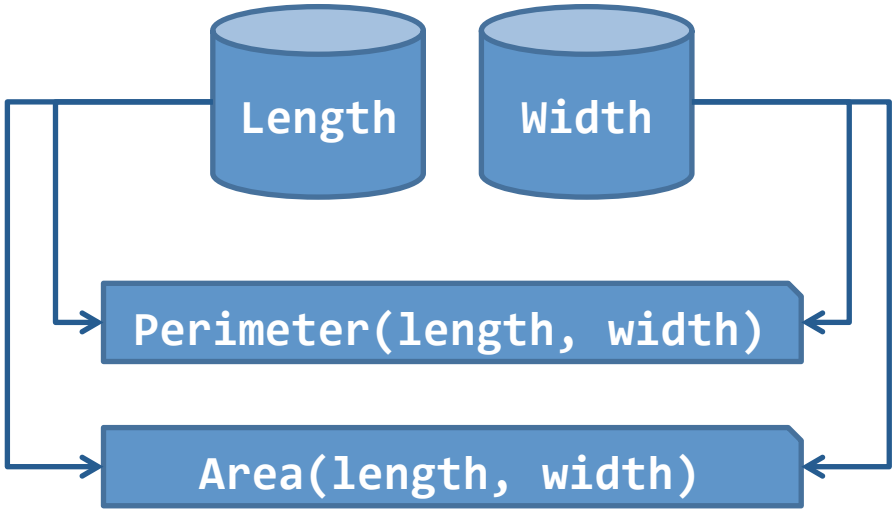
A programming paradigm based on the concept of "objects", which are data types that contain data, in the form of variables, called *member variables*; and code, in the form of functions, called *member functions*.

Examples of OO languages: C++, C#, Java, JavaScript, PHP, Python, Ruby, …

Rough Comparison with Procedural

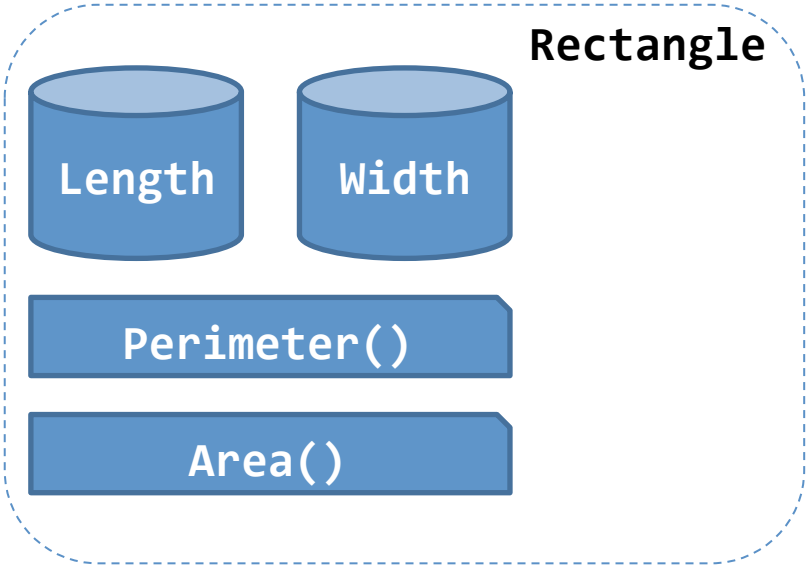
Procedural

- Variables and functions are independent (C, BASIC, PASCAL, …)



Object-Oriented

- When it makes sense, we group variables and functions together ("encapsulation")



OOP Terminology

Class. Like a "template," this identifies all the member variable(s) and function(s) that instances of this type have.

- *All rectangles have a length (double) and width (double), as well as functions to compute area and perimeter (each returning a double).*
- Many C++ libraries include class definitions (e.g. **`fstream`** includes **`ifstream`** and **`ofstream`**); you can also define your own.

Object. A specific instance of a class. Another way of saying this is a variable of type <insert class here>.

- *I have two variables of type Rectangle, one named rectA, the other rectB.*

Example Class

```
class rectangle
{
public:
    double length;
    double width;
};
```

- This defines a class named **`rectangle`**
- The **`public:`** line at the top of the class definition is important, and we will discuss it later
- The two **`double`** lines say that the rectangle class uses two **`double`** member variables named **`length`** and **`width`**

Member Variables

- Defining a member variable in a class is NOT a variable declaration
- You can not use those variables except in the context of a variable of the class type (an object)
- In other words, you can think of the member variables as sub-pieces of an object that you can only access as part of the object
- So, when you declare a variable of a class type, it automatically declares the member variables for that object for you
- Using the member variables is just like using any other variable of the same type

Using Member Variables

```
#include <iostream>
using namespace std;

class rectangle
{
public:
    double length;
    double width;
};

int main()
{
    rectangle r;
    r.length = 10.5;
    r.width = 5.25;
    cout << "r.length=" << r.length << endl;
    cout << "r.width=" << r.width << endl;
    return 0;
}
```

Definition of the `rectangle` class

Declaration of a variable named `r` of type `rectangle`

Give `r`'s `length` member variable a value of `10.5` and `r`'s `width` member variable a value of `5.25`

Multiple Class Objects

- You can declare more than one variable of a class type
- Each instance of a class variable has its own member variables that are completely separate from each other
- For example, declaring two **`rectangle`** objects actually declares four **`double`** variables (two **`length`** variables and two **`width`** variables)

Example

```
#include <iostream>
using namespace std;

class rectangle
{
public:
     double length;
     double width;
};

int main()
{
     rectangle r, s;
     r.length = 10.5;
     r.width = 5.25;
     s.length = 1.8;
     s.width = 0.3;
     cout << "r has length=" << r.length << " and width=" << r.width << endl;
     cout << "s has length=" << s.length << " and width=" << s.width << endl;
     return 0;
}
```

Declare two **rectangle** variables named **r** and **s**

Set the **length** and **width** member variables for **r**

Set the **length** and **width** member variables for **s**

Exercise

Define a class named **`triangle`**. It should have two member variables: **`base`** and **`height`**. Write a **`main`** function that declares a **`triangle`** variable and assigns values to the member variables.

Answer

```
#include <iostream>
using namespace std;

class triangle
{
public:
    double base;
    double height;
};

int main()
{
    triangle my_tri;
    my_tri.base = 1;
    my_tri.height = 2;
    return 0;
}
```

Member Functions

- In addition to member variables, classes can include member functions
- Member functions are like any other function, except that you have access to the member variables inside the function
- They are called using one object of the class, as we saw with **`ifstream`** and **`ofstream`** variables

Define Member Functions

```
class rectangle
{
public:
     void output()
     {
          cout << "length=" << length << " and width=" << width << endl;
     }

     double length;
     double width;
};
```

- We've added a member function, **`output`**, that returns nothing, takes no arguments, and prints out **`length`** and **`width`** values of the object
- Note that **`length`** and **`width`** are used directly because they are part of the same class as the function

Using Member Functions

```
#include <iostream>
using namespace std;

class rectangle
{
public:
     void output()
     {
          cout << "length=" << length << " and width=" << width << endl;
     }
     double length;
     double width;
};
int main()
{
     rectangle r, s;
     r.length = 10.5;
     r.width = 5.25;
     s.length = 1.8;
     s.width = 0.3;
     r.output();
     s.output();
     return 0;
}
```

Adding Another Member Function

```
class rectangle
{
public:
    void output()
    {
        cout << "length=" << length
             << " and width=" << width
             << endl;
    }

    double area()
    {
        return length * width;
    }

    double length;
    double width;
};
```

```
int main()
{
    rectangle r, s;
    r.length = 10.5;
    r.width = 5.25;
    s.length = 1.8;
    s.width = 0.3;
    r.output();
    s.output();
    cout << "r has area=”
         << r.area() << endl;
    cout << "s has area="
         << s.area() << endl;
    return 0;
}
```

Exercise

Add two member functions to your **`triangle`** class: **`output`** and **`area`**. **`output`** should print the values of the member variables. **`area`** should return the area of your triangle. Write a **`main`** function to test it.

Answer

```
#include <iostream>
using namespace std;

class triangle
{
public:
    void output()
    {
        cout << "base=" << base
             << " and height=”
             << height << endl;
    }

    double area()
    {
        return 0.5 * base * height;
    }

    double base;
    double height;
};
```

```
int main()
{
    triangle tri;
    tri.base = 1;
    tri.height = 2;
    tri.output();
    cout << "area="
         << tri.area() << endl;
    return 0;
}
```

Defining Member Functions

- The previous examples of member functions are defined directly in the class definition
- You can also break up the member function into a declaration and a separate definition like we typically do for regular functions
- The declaration goes in the class definition
- The function definition goes outside of the class definition, with the class name added to the function name
  - For example, the function name for the definition of the **`area`** function of **`rectangle`** would be **`rectangle::area`**

Examples (both produce identical results)

```
#include <iostream>
using namespace std;
class world
{
public:
    void hello();
};

int main()
{
    world w;
    w.hello();
    return 0;
}

void world::hello()
{
    cout << "Hello World" << endl;
}
```

```
#include <iostream>
using namespace std;

class world
{
public:
    void hello()
    {
        cout << "Hello World"
             << endl;
    }
};

int main()
{
    world w;
    w.hello();
    return 0;
}
```

Don't Forget the Class Name

- If you forget to add the class name to a member function definition that is outside of the class definition, C++ will not know that it is for that class
- The typical result is a linker error for unresolved symbols
- Note that you can have a member function that has the same name as non-member function

Example

```
#include <iostream>
using namespace std;
class world
{
public:
    void hello();
};

int main()
{
    world w;
    w.hello();
    return 0;
}

void hello()
{
    cout << "Hello World" << endl;
}
```

Missing the **`world::`** in front of the **`hello()`** name, so C++ thinks this is another function named **`hello`**

Another Example

```
#include <iostream>
using namespace std;

class world
{
public:
    void hello();
};

void hello();

int main()
{
    world w;
    w.hello();
    hello();
    return 0;
}
```

```
void world::hello()
{
    cout << "Hello World" << endl;
}

void hello()
{
    cout << "Hello" << endl;
}
```

Member Access Level

- Every member (variable or function) of a class has a fixed access level
- There are three access levels: `public`, `private`, and `protected`
- The access level determines where the variable or function can be used
- We will only be using `public` and `private` in this course, but you will learn about `protected` when you learn about inheritance

`public` Members

- So far, we've only been using `public` members
- These `public` members can be accessed directly from anywhere in your program
- You can use member variables just like any other variables
- You can use member functions just like any other functions

`private` Members

- Members that are `private` can only be accessed and used inside of member functions (`public` or `private`) of the same class
- You can use `private` member variables only in member functions of the class
- You can call `private` member functions only from other member functions of the class
- You can NOT use `private` members in non-member functions, such as `main`

Setting Member Access Levels

- All class members declared after an access level keyword (**`public`** or **`private`**) but before the next access level keyword have that access level
- If no access level is given, the members are **`private`**

```
class my_class
{
     int i; // private variable
public:
     void some_func(); // public function
     double x; // public variable
private:
     int fun(int c); // private function
     bool b; // private variable
public:
     double y; // public variable
};
```

Example

```
#include <iostream>
using namespace std;
class playing_card
{
public:
     void set(char s, char value);
     void print()
     {
          cout << rank << " of "
               << suit << endl;
     }
private:
     char suit;
     int rank;
};

int main()
{
     playing_card c;
     c.set( 'h', 'j' );
     c.print();
     c.suit = 'h'; // build error
     return 0;
}
```

```
void playing_card::set(char s, char value)
{
     suit = s;
     if ( value == 'j' )
     {
          rank = 11;
     }
     else if ( value == 'q' )
     {
          rank = 12;
     }
     else if ( value == 'k' )
     {
          rank = 13;
     }
     else if ( value == 'a' )
     {
          rank = 14;
     }
     else if ( value >= '2' && value <= '9' )
     {
          rank = value - '0';
     }
}
```

Why `private`?

- By making members **`private`**, you ensure that they are not used outside of the class member functions
  - This is typically done for all variables
- Functions are made **`private`** if they are only used internally in the class, and should not be called by a programmer that is utilizing the class
- In other words, **`private`** members are used to *hide the implementation details* of a class ("information hiding")

Objects as Function Arguments

- Class variables (objects) can be passed as function arguments, just like any other variable

- Inside a member function of the same class, you have be careful to use the correct variables (the ones for "this" object versus the ones for the function argument)

- You access members of the argument object like normal

Example

```
#include <iostream>
using namespace std;

class my_integer
{
public:
     void set_value(int new_v)
     {
          x = new_v;
     }
     void set_value(my_integer new_v)
     {
          x = new_v.x;
     }
     int get_value()
     {
          return x;
     }
private:
     int x;
};
```

```
int main()
{
     my_integer mine1, mine2;

     mine1.set_value( 15 );
     mine2.set_value( mine1 );

     cout << "mine1.x=”
          << mine1.get_value()
          << endl;
     cout << "mine2.x="
          << mine2.get_value()
          << endl;

     return 0;
}
```

Calling Member Functions

- You can call member functions of a class from other member functions of the same class
- When doing so, you use the function directly, just like when you use member variables
- That is, there is no variable name and a dot before the name of the function, because you are already in the context of the class

Example

```
#include <iostream>
using namespace std;

class temperature
{
public:
     void set(double temp)
     {
          temp_f = temp;
     }
     double get(char scale)
     {
          double temp;
          if ( scale == 'F’ )
          {
               temp = temp_f;
          }
          else
          {
               temp = to_celsius();
          }
          return temp;
     }
private:
     double to_celsius()
     {
          return ( 5.0 / 9 ) * ( temp_f - 32 );
     }
     double temp_f;
};

int main()
{
     temperature t;
     t.set( 50 );
     cout << t.get('F') << " degs F is ”
          << t.get('C') << " degs C" << endl;
     return 0;
}
```

Exercise

Modify the **`set`** function in the temperature class to have a second character argument that specifies the scale ('C' or 'F'). Add another private member function named **`to_fahrenheit`** that converts a temperature from celsius to fahrenheit to help you in the new **`set`** function.

Answer

```
#include <iostream>
using namespace std;
class temperature
{
public:
     void set(double temp, char scale)
     {
          if ( scale == 'F' )
               temp_f = temp;
          else
               temp_f = to_fahrenheit( temp );
     }
     double get(char scale)
     {
          double temp;
          if ( scale == 'F' )
               temp = temp_f;
          else
               temp = to_celsius();
          return temp;
     }
private:
     double to_celsius()
     {
          return ( 5.0 / 9 ) *
                 ( temp_f - 32 );
     }
     double to_fahrenheit(double c)
     {
          return ( 9.0 / 5 ) * c + 32;
     }
     double temp_f;
};

int main()
{
     temperature t;
     t.set( 50, 'C’ );
     cout << t.get('F')
          << " degs F is ”
          << t.get('C')
          << " degs C" << endl;
     return 0;
}
```

Exercise

Now, modify the temperature class so that the member variable stores the temperature in celsius rather than in fahrenheit

Answer

```
#include <iostream>
using namespace std;

class temperature
{
public:
     void set(double temp, char scale)
     {
          if ( scale == 'F' )
               temp_c = to_celsius( temp );
          else
               temp_c = temp;
     }
     double get(char scale)
     {
          double temp;
          if ( scale == 'F' )
               temp = to_fahrenheit();
          else
               temp = temp_c;
          return temp;
     }
private:
     double to_celsius(double f)
     {
          return ( 5.0 / 9 ) *
                 ( f - 32 );
     }
     double to_fahrenheit()
     {
          return ( 9.0 / 5 ) * temp_c + 32;
     }
     double temp_c;
};

int main()
{
     temperature t;
     t.set( 50, 'C’ );
     cout << t.get('F')
          << " degs F is ”
          << t.get('C')
          << " degs C" << endl;
     return 0;
}
```

Constructors

- Constructors are special member functions that are used for initialization, to *construct* an object
- A class can have multiple constructors that have different argument lists, but each object can only be initialized with one constructor
- The function name for a constructor is the same as the name of the class
- Constructors have no return values
- Except under very special circumstances, constructors should always be `public`

Example

```
class stock
{
public:
    stock(double init_value, int init_shares)
    {
        value = init_value;
        shares = init_shares;
    }
    void print()
    {
        cout << "You have "
             << shares << " shares worth a total of $"
             << value * shares << endl;
    }
private:
    double value;
    int shares;
};
```

Calling a Constructor

- Constructors are called automatically when you declare an object
- They can not be called after an object is declared (at least until you learn about dynamic allocation!)
- Only one constructor can be called per object, and one constructor is always called
- You specify the arguments in parentheses after the variable name
  - Example: `stock goog( 573.00, 5 );`

Example

```
#include <iostream>
using namespace std;

class stock
{
public:
     stock(double init_value, int init_shares)
     {
          value = init_value;
          shares = init_shares;
     }
     void print()
     {
          cout << "You have " << shares
               << " shares worth a total of $"
               << value * shares << endl;
     }
private:
     double value;
     int shares;
};
```

C++ automatically calls the constructor that takes two arguments (one **double** and one **int**)

```
int main()
{
     stock goog(573.00, 5);
     goog.print();
     return 0;
}
```

Exercise

Write a class named **`account`** which has a single member variable to store the balance of an account. Include a constructor that allows you to set the initial balance of the account. Write a **`main`** function to test it.

Answer

```
#include <iostream>
using namespace std;

class account
{
public:
     account(double initial_balance)
     {
          balance = initial_balance;
     }
     void print()
     {
          cout.precision( 2 );
          cout.setf( ios::fixed );
          cout << "$" << balance << endl;
     }
private:
     double balance;
};

int main()
{
     account checking( 15.50 );
     checking.print();
     return 0;
}
```

Multiple Constructors

- You can define as many constructors as you want for each class, so long as they conform to the normal function rules
- The argument lists have to be different, meaning different types or different numbers of arguments
  - Overloading!
- C++ automatically chooses the correct constructor based on the arguments provided

Example

```
#include <iostream>
using namespace std;
class stock
{
public:
     stock(double init_value)
     {
          value = init_value;
          shares = 0;
     }
     stock(double init_value, int init_shares)
     {
          value = init_value;
          shares = init_shares;
     }
     void print()
     {
          cout << "You have " << shares
               << " shares worth a total of $"
               << value * shares << endl;
     }
private:
     double value;
     int shares;
};
```

New constructor with a single **double** argument

Calling the new constructor

```
int main()
{
     stock goog( 573.00, 5 );
     stock msft( 27.02 );
     goog.print();
     msft.print();
     return 0;
}
```

Default Constructor

- One special constructor is the *default* constructor
- This is the constructor used when no arguments are provided at object declaration
  - Example: `stock csco;`
- If you define no constructors for a class, the compiler automatically adds a default constructor that does nothing
- If you define any constructors for a class (not necessarily a default constructor), the compiler does NOT add a blank default constructor for you

Example

```
#include <iostream>
using namespace std;

class stock
{
public:
     stock()
     {
          value = 0;
          shares = 0;
     }
     stock(double init_value)
     {
          value = init_value;
          shares = 0;
     }
     stock(double init_value,
           int init_shares)
     {
          value = init_value;
          shares = init_shares;
     }
     void print()
     {
          cout << "You have " << shares
               << " shares worth a total of $"
               << value * shares << endl;
     }
private:
     double value;
     int shares;
};

int main()
{
     stock goog( 573.00, 5 );
     stock msft( 27.02 );
     stock csco;
     goog.print();
     msft.print();
     csco.print();
     return 0;
}
```

Default constructor

Calling the default constructor

No Default Constructor

```
#include <iostream>
#include <string>
using namespace std;

class person
{
public:
     person(string fn, string ln)
     {
          first_name = fn;
          last_name = ln;
     }
     void name()
     {
          cout << "Name: " << first_name
               << " " << last_name << endl;
     }
private:
     string first_name;
     string last_name;
};
```

```
int main()
{
     person op( "Optimus", "Prime" );
     person um; // build error
     op.name();
     return 0;
}
```

Only constructor defined takes two arguments, so no default constructor

No default constructor means you cannot declare a person with no arguments

Automatic Default Constructor

```
#include <iostream>
#include <string>
using namespace std;

class user
{
public:
     void set_id(int new_id)
     {
          id = new_id;
     }
     void set_username(string un)
     {
          username = un;
     }
     void print()
     {
          cout << "username=" << username << endl;
          cout << "user id=" << id << endl;
     }
private:
     string username;
     int id;
};
```

No constructor defined, so a default constructor that does nothing is automatically added

Default constructor is called, which does nothing and initializes no member variables

```
int main()
{
     user u;
     u.print();
     u.set_id(2716057);
     u.set_username("bender");
     u.print();
     return 0;
}
```

Exercise

Modify your **`account`** class to include a default constructor that sets the balance to $0.00. Also add a function named **`adjust`** that allows you to adjust the balance by a positive or negative amount. Test the new functions in **`main`**.

Answer

```
#include <iostream>
using namespace std;

class account
{
public:
	account()
	{
		balance = 0;
	}
	account(double initial_balance)
	{
		balance = initial_balance;
	}
	void adjust(double adjustment)
	{
		balance = balance + adjustment;
	}
	void print()
	{
		cout.precision( 2 );
		cout.setf( ios::fixed );
		cout << "$" << balance << endl;
	}
private:
	double balance;
};
```

```
int main()
{
	account checking;
	checking.adjust( 1000.00 );
	checking.adjust( -125.00 );
	checking.print();
	return 0;
}
```

Wrap Up

- A class defines a complex variable type
  - Contains its own variables and functions that are only for use with objects of that class
- There are many predefined classes in C++ including string, **`ifstream`**, and **`ofstream`**
- You can also define your own classes
  - Often done to represent an entity in your program that requires more than one variable
- This is just the beginning of object oriented (OO) software development