In this Chapter:

2.5 Static

Sam is writing a program to keep track of the employees in a medium sized company. He decides to use a class to represent an employee. One property of this class will be the employee number. It doesn't matter what the number is for each employee, it is just important that each employee has a unique number. How can he do that? As he mulls over the problem for a while, he remembers reading something about the `static` modifier...

Objectives

By the end of this chapter, you will be able to:

- Explain what the `static` modifier does and when it can be useful
- Use `static` to solve programming problems in classes and in functions

Prerequisites

Before reading this chapter, please make sure you are able to:

- Describe the difference between a global, local, and member variable (Procedural Programming in C++ Chapter 1.2 and 1.4)
- Create a class definition to match a UML class diagram (Chapter 2.2)
- Articulate the difference between public and private member variables (Chapter 2.2)

What is `static` and why you should care

The `static` keyword is a modifier attached to a variable to indicate that only one copy of the variable will exist in a program. Perhaps this is best explained by example. Consider a function with a single integer local variable. The first time the function is called, the variable is uninitialized until a value is assigned. The second time the function is called, it is again uninitialized until a value is assigned. If, on the other hand, the `static` keyword is used, everything changes. The second time the function is called, the variable will remember the value from the first time it was called. This is because, no matter how many times the function is called, only one copy of the variable exists in memory. All instance of the function share the same variable. While `static` is a useful programming tool, it is infrequently used. The reason for this is twofold: first a programmer rarely encounters the situation when `static` would benefit him, and second there are other ways to accomplish the same thing. That being said, a member variable made `static` is a far more useful construct than a `static` local variable. The situation often arises when all the objects created from a given class need to share the same member variable. The `static` modifier makes this possible.

Static local variables

A local variable can be made `static` by the use of the `static` keyword:

```
void display()
{
    `static` int count = 0; // count is initialize to zero only
    // the first time display() is called
    cout << "This function has been called "
        << ++count << " times\n";
    // every time display() is called, we
} // will add one to count
```

So how does this work exactly? When the program begins execution, space is reserved for the `static` local variables. This space stays reserved until the program exits. Therefore, unlike a local variable that is created when the function is called and destroyed when the variable falls out of scope, `static` variables are "alive" the entire length of the program execution. However, unlike a global variable, they are only accessible from within the function in which they were defined.

There are a few instances when `static` might come in handy. One is for performance reasons. Since `static` variables only get initialized once, we can save the initialization cost by making it `static`.

```
bool playAgain()
{
    `static` string input; // because this is `static`, we will only
    cout << "Do you want to play again? "; // initialize this variable once.
    cin >> input;
    return (input == "yes");
}
```

Static can also be used to keep track of how the function was previously used.

```
int getScore()
{
    // fetch the current score
    int score;
    cout << "What is your score? ";
    cin >> score;
    // is this the highest score yet?
    `static` maxScore = 0; // only this function cares what the
    if (score > maxScore) // highest current score is. Why
    {
        cout << "Highest score yet!\n";
        maxScore = score;
    }
    return score;
}
```

Sue's Tips

One interesting thing about `static` local variables is that they automatically get initialized to zero even if the programmer neglects to explicitly do it. However, while you can depend on this initialization, it is unwise to do so. It is far better to be clear about your intentions and initialize it yourself.

Static member variables

Member variables can also be made `static`. If a class has a member variable that is made `static`, all objects made from that class will share that `static` member variable. The UML class diagram for a `static` is underline:

ClassName
+ publicStaticMemberVariable
+ publicMemberVariable
- privateStaticMemberVariable
- privateMemberVariable
+ publicMethod
- privateMethod

However, unlike their local variable cousins, declaration and initialization must occur separately.

Declaring `static` member variables

In the following example, we will create a class that has a member variable called temp and a `static` member variable called highest. Every object made from this class will have its own temp member variable, but they will all share the highest member variable.

```
class Temperature
{
    public:
        Temperature() : temp(0.0) { }
        void set(float temp)
        {
            if (temp > this->temp)
                highest = temp;
            this->temp = temp;
        }
        float get() const { return temp; }
        float getHighest() const { return highest; }
    private:
        float temp; // "temp" is not shared with
                                // other Temperature objects
        `static` float highest; // "highest" is `static` so it is
}; // shared with other objects
```

The size of an object is a computed by summing the size of all the member variables. At first glance, it may appear that sizeof(Temperature) $==$ 8 because each of the two floats takes four bytes of memory. However, since the highest member variable is shared, it is not part of Temperature's size. Thus sizeof(Temperature) == 4 .

Initializing `static` member variables

Static member variables are initialized outside the class definition. If, for example, the above Temperature class were to be used in a program, the highest member variable would have to be initialized:

```
#include "temperature.h"
float Temperature :: highest = 0.0; // initialization of `static` member variab
    int main()
{
    Temperature t;
    ... code removed for brevity ...
        return 0;
}
```

When to use `static`

The `static` keyword is a scope modifier. It serves to increase the scope from just one instance of a function or a class to all instances of the function or class. As we learned from CS 124, this can be a two-edged sword. The larger the scope, the harder it can be to find a bug with the variable. If, for example, a variable is global, it is difficult to tell what code is looking at the value or which code changes it. These questions are much easier to answer with local variables.

With `static`, there are now several levels of scope:

`global` variables	The largest level of scope. A global variable is accessible from anywhere in the program.
`static` member variable	All objects instantiated from a class share the same variable.
`public` member variable	All the methods in a class have access to the variable as well as functions having an object from that class.
`private` member variables	All the methods in a class have access to the variable.
`static` local variables	All instances of a single function share access to the variable.
By-reference parameter	Both the caller and the callee share the same variable.
Local variable	All the statements in a given function have access to the variable. Note that by-value parameters have essentially the same scope as local variables.
Blocks	It is possible to declare a variable that is only visible inside the body of an IF statement or in a FOR loop. These represent the smallest scope of any variable.

As a general rule, a programmer should use the smallest possible scope to solve a given programming problem. For example, never use a member variable when a parameter or a local variable will suffice. Try to declare counter variables inside the FOR loop rather than use a local variable for the same purpose.

One final note: `static` member variables have the largest scope of any variable with the exception of globals. Therefore, they should be used cautiously and when there is no better solution.

Example 2.5 - Point

This example will demonstrate how to use `static` to make all objects share the same limits. This is one of the most common uses of `static` in a class.

Consider a computer game where the size of the window is adjustable. Create a class called Point representing the position of an item in the game. The Point class should also know the bounds of the window (xMin through xMax and yMin through yMax) so it can determine if the item is off the screen.

First, the declarations of xMin and company in the Point class definition are:

```
class Point
{
... code removed for brevity...
private:
    float x; // horizontal position
    float y; // vertical position
    bool dead; // have we exceed our bounds?
    `static` float xMin; // minimum extent of the x position
    `static` float xMax; // maximum extent of the x position
    `static` float yMin; // minimum extent of the y position
    `static` float yMax; // maximum extent of the y position
};
```

To test this class, we need to instantiate a couple Point objects:

```
float Point::xMin = -10.0;
// initialize the `static` member
float Point::yMin = -10.0;
// variables. Though these
float Point::xMax = 10.0;
// look like global variables,
// they are not
int main()
{
    // create a legal point at zero, zero
    Point pt1; // bounds set to (-10, -10)
    cout << "Initial value: "; // to (10, 10)
    pt1.display();
    cout << endl;
    // move it to an illegal point
    pt1.setX(-20.0); // outside the xMin bounds
    cout << "After setting to (-20.0, 0.0), "
        << (pt1.isDead() ? "invalid" : "valid")
        << endl;
    // create another point that is also invalid
    Point pt2(0, 20.0); // also sharing the same bounds
    cout << "Second point at (0.0, 20.0) is " // as pt1, so this is
        << (pt2.isDead() ? "invalid" : "valid") // also invalid
        << endl;
    return 0;
}
```

The complete solution is available at 2-5-point.html or: /home/cs165/examples/2-5-point/

Example 2.5 - Card

Another common use of `static` is to configure a class for use in a single application. This is useful when all objects from the class share the same configuration setting.

Some card games consider the Ace to be higher than the King while others consider it to be lower than the Two. Create a card class that allows for either configuration.

There are two changes to card.h from "Example 2.4 - Card." The first is that we need two strings for the ranks.

```
const char RANKS_HIGH[] = "234567890jqka"; // aces high
const char RANKS_LOW[] = "a234567890jqk"; // aces low
```

The second is the addition of the `static` member variable called acesHigh:

```
class Card
{
... code removed for brevity ...
    private:
    // holds the value. Though there are 256 possible, only 52 are used
    unsigned char value; // internal representation
    `static` bool acesHigh; // is an Ace high, or low?
};
```

Next the getRank() method needs to be adjusted to point to the correct string:

```
char Card :: getRank() const
{
    // this is `static` because we should only need to initialize it once
    `static` const char * pRank = (acesHigh ? RANKS_HIGH : RANKS_LOW);
    return pRank[value % 13]; // point to the appropriate string
}
```

Finally we shall test the new class.

```
bool Card :: acesHigh = true; // initialized outside main()
int main()
{
    // where is the Ace of Diamonds?
    string sAceDiamonds("ad");
    Card cardAceDiamonds(sAceDiamonds);
    cout << "The Ace of Diamonds is at rank: "
            << cardAceDiamonds.iRank('a')
            << endl;
    return 0;
}
```

The complete solution is available at 2-5-card.html or: /home/cs165/examples/2-5-card/

Example 2.5 - Time

This example is much like “Example 2.5 - Card” in that the `static` member variable will be used to configure all the objects in the class.

The most common way to represent dinner time is "$6:00\text{pm}$." The military uses a 24 -hour clock and represents the same time as "$18:00$." Modify the Time class to either display military time or the traditional am/pm time.

The first change to the time class from “Example 2.4 - Time” is the `static` member variable isMilitary:

```
class Time
{
... code removed for brevity...
    `static` bool isMilitary;
};
```

Next we need to modify display() to call two variants displayMilitary() or displayCivilian():

```
void display() const
{
    // paranoia...
    assert(validate());
    if (isMilitary)
        displayMilitary();
    else
        displayCivilian();
}
```

Finally it is necessary to test our new Time class:

```
bool Time :: isMilitary = false; // Configure the time class
int main()
{
    Time time1; // start with midnight
    cout << "Time1 is midnight - ";
    time1.display();
    Time time2(9 /*hours*/, 11 /*minutes*/); // Next 9:11 am
    cout << "Time2 is in the morning - ";
    time2.display();
    Time time3(18 /*hours*/, 2 /*minutes*/); // Finally 6:02 pm
    cout << "Time3 is the afternoon - ";
    time3.display();
    return 0;
}
```

As a challenge, modify the driver program so it displays military time. This is accomplished by setting the isMilitary member variable. bool Time :: isMilitary = true;

The complete solution is available at 2-5-time.html or: /home/cs165/examples/2-5-time.cpp

Problem 1

Given the following code:

```
class Silly
{
    public:
        Silly()
        Silly(const Silly & s)
        ~Silly()
        void method()
};
Silly & function(Silly & s)
{
    cout << "Function\n";
    return s;
}
int main()
{
    Silly s1;
    s1.method();
    function(s1);
    s1.method();
    return 0;
}
```

What is the output?

Problem 2

What is the output of the following code:

```
int addto(int value)
{
    `static` int sum = 0;
    return sum += value;
}
int main()
{
    int array[] = { 3, 1, -5, 2 };
    for (int i = 0; i < 4; i++)
        addTo(array[i]);
    cout << addTo(0) << endl;
}
```

Problem 3

Given the following class:

```
class Bullet
{
    public:
        Bullet() { numBullet++; }
        ~Bullet() { --numBullet; }
        int getBullets() { return numBullet; }
    private:
        `static` int numBullet;
};
```

What is the output of the following code:

```
int Bullet :: numBullet = 0;
int main()
{
    Bullet b1;
    Bullet b2;
    {
        Bullet b3;
        Bullet b4;
        cout << b3.getBullets() << endl;
    }
    cout << b1.getBullets() << endl;
    return 0;
}
```

Challenge 4, 5, 6

You would like to build a class to handle group texting. While each object will have its own variables (ID, for example), they would share the same display text (an array of the last 20 messages).

5. Create a UML class diagram of the GroupText class.

6. From the above UML class diagram, define the class.

7. Implement any method which is not defined inline above.