64. Static Members

Objectives

- Understand static local variables
- Understand static members
- Write static members
- Access static members
- Write static methods
- Call static methods
- Understand when to use static members and methods

Static variables

First let me talk about static variables in general (which has nothing to do with classes.)

Static variables are declared before the program executes, and they stay in memory until the program halts. Functions are static, because the memory allocated for them is declared before the start of the program, and is only reclaimed once the program ends. Global variables are also static for the same reason.

Try the following program:

```
#include <iostream>

void f()
{
    static int x = 0;
    int y = 0;
    x++; y++;
    std::cout << x << ' ' << y << std::endl;
}

int main()
{
    f(); f(); f(); f();
    return 0;
}
```

What's happening??? Well ...

```
void f()
{
    static int x = 0; // this is executed ONCE when
                      // the program starts up
                      // and x STAYS AROUND even when
                      // you return from f(). x goes
                      // away and only when the
                      // program ends. So x is NOT
                      // automatic.
    ...
}
```

Exercise. Write a function `mybankaccount()` such that the first time you call `mybackaccount()`, it prints 0. If you call it the second time with 10, the function prints 10. If you call it the third time with 20, it prints 30. If you call it the fourth time with -5, it prints 25. In other words, it records the cumulative sum of the arguments sent into the function. Note that the first function call does not pass in a value while the remaining ones do. That tells you that `mybankaccount()` has a parameter with a default value. For instance you can use -9999 as the default value to mean “reset the cumulative sum to 0”.

Static members variables (in a class)

Besides static variables in a code block, classes can have static members as well. Let me give the concepts first.

(The code in this section is not runnable. You'll have to wait for the next section before I complete the code.)

Think of **static member variables** as **variables belonging to a class** and not to each object. There is only one copy for each static member. All objects of a class **share** the same static member. If `y` is a static member of a class and `obj1` and `obj2` are objects of that class, then `obj1.y` is the **same** as `obj2.y`.

To make a member static put the keyword `static` in front of the declaration of the member in the class declaration. Static members can be public or private.

For the next example, I'm going to make all members public just to simplify the discussion. Don't run the code yet – I'm going through the concept of static members and the code won't be runnable until the next section when I show you how to initialize static members.

First let me show you how to create static member variables. Here's class C:

```
// C.h
#ifndef C_H
#define C_H

class C
{
public:
    int x;            // regular class member variable
    static int y; // static class member variable
    static int z; // static class member variable
};

#endif
```

Look at the "`static`" words in the code. This class has two static members: `y` and `z`.

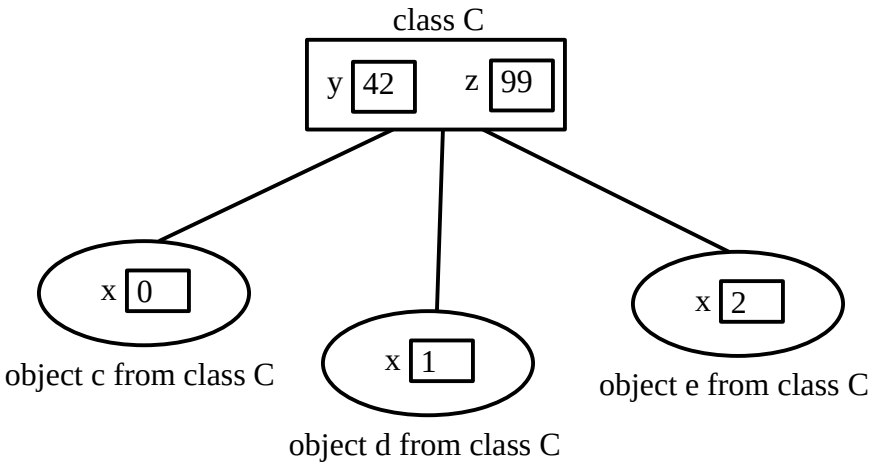
Now if I do this in `main()` (remember: don't run the code yet – just focus on the idea):

```
#include <iostream>
#include "C.h"

int main()
{
```

```
    C c, d, e; // 3 objects from class C
    ...
```

then the objects currently in memory and the static members of class `C` look like this (conceptually):



Note the following:

- Each of the 3 objects have their own member variable x. So … `c.x` is different from `d.x` which is different from `e.x`. When c changes its x to `1000`, `d.x` is still `1` and `e.x` is still `2`.
- All 3 objects **share** `y` and `z` (the **static** member variables). All 3 objects do have access to `y` and `z`. So when `c` changes `y` to `43` and `d` print `y`, `d` will be printing `43`.

Get it? It's really not that complex: regular member variables are not shared among objects while static member variables are shared across all objects.

Now although the objects `c`, `d`, `e` have access to the static members `y`, `z`, you should view the static member variables as variables belonging to the class `C`. So … although you can write `c.y` in your code (member `y` is public – look at definition of class `C`):

```
#include <iostream>
#include "C.h"

int main()
{
    C c, d, e; // 3 objects from class C
    std::cout << c.y << '\n';
    ...
```

it's more common to write `C::y`.

```
#include <iostream>
#include "C.h"

int main()
{
    C c, d, e; // 3 objects from class C
```

```
    std::cout << C::y << '\n'; // better than c.y
    ...
```

If y is a private member:

```
...
class C
{
public:
    int x;        // regular class member variable
    static int z; // static class member variable
private:
    static int y; // static class member variable
};
...
```

then this won't work:

```
#include <iostream>
#include "C.h"

int main()
{
    C c, d, e; // 3 objects from class C
    std::cout << C::y << '\n'; // BAD! BAD! BAD!
    ...
```

Now let me show you how to initialize the value of static member variables so that you can actually run the above code.

Initialization of static member variables

Here's our class C again:

```
// C.h
...
class C
{
public:
    int x;
    static int y;
    static int z;
};
...
```

The member variables y and z are static and have to be initialized even before any object is created.

Recall an earlier example of static variable in functions:

```
void f()
{
    static int x = 0; // this is executed ONCE when
                      // the program starts up
                      // and x STAYS AROUND even when
                      // you return from f(). x goes
                      // away and only when the
                      // program ends. So x is NOT
                      // automatic.
    ...
}
```

Static variables have to be initialized when you run your program. In the case of classes, the static members are initialized even before you create any object of that class. So the initialization of `y` and `z` are not done through objects.

The initialization is done through the class. There are two ways of doing it.

```
// C.cpp
#include "C.h"

int C::y(42);   // First method of initialization
int C::z = 99;  // Second method of initialization
```

Don't forget the `C::` in front of the `y`!!!

Note that the initialization code must be done in the implementation file, i.e., the cpp file, and not in the header file. (There's an exception – see later section on constant static member variables.)

WARNING: Do **not** include keyword static in cpp file.

```
// C.cpp
...

static int C::y(42); // WRONG!!!!!!
```

```
...
```

Static should only appear during declaration in the header file.

With the above you can now run this:

```
// C.h
#ifndef C_H
#define C_H

class C
{
public:
    int x;
    static int y;
    static int z;
};

#endif
```

```
// C.cpp
#include "C.h"

int C::y(42);
int C::z = 99;
```

```
#include <iostream>
#include "C.h"

int main()
{
    C c, d, e;
    std::cout << C::y << '\n';
    std::cout << C::z << '\n';

    // Change C::y through c ...
    c.y = 24;
    std::cout << C::y << '\n'; // and C::y is changed

    // Change C::y ...
    C::y = 42;
    std::cout << d.y << '\n'; // and d.y is changed

    // And in fact they are all the same ...
    std::cout << &C::y << ' '
              << &c.y << ' '
              << &d.y << '\n'

    return 0;
}
```

Static methods (i.e. static member functions)

Just like a regular member variable can be made static (so that it belongs to the class), a method can also made static.

To make a method static put the keyword `static` in front of the prototype of the method in the class declaration.

A static method or static member function is like a method that belongs to the class and not to individual objects. In general, this is how you decide if a method should be static or not …

If a method of a class `C` does not work with any regular member variable of `C`, then the method should be made static.

Exercise. Which method should be static?

```
class C
{
public:
    C(char name[]) { strcpy(name_, name); }
    void print0() const
    {
        std::cout << "hello 0\n";
    }
    void print1() const
    {
        std::cout << "hello " << name_ << std::endl;
    }
private:
    char name_[100];
};

int main()
{
    C c("john");
    c.print0();
    c.print1();
    return 0;
}
```

Here's our class `C` again:

```
// C.h
#ifndef C_H
#define C_H

class C
{
public:
    int x;
```

```
    static int y;
    static int z;
};

#endif
```

```
// C.cpp
#include "C.h"

int C::y(42);
int C::z = 99;
```

Note that in general, you want to hide all member variables, static or not. So I really should do this:

```
// C.h
#ifndef C_H
#define C_H

class C
{
public:
private:
    int x;
    static int y;
    static int z;
};

#endif
```

```
// C.cpp
#include "C.h"

int C::y(42);
int C::z = 99;
```

What if I might want to print the value of `C::y`? The following does not work any more since `C::y` is now private:

```
#include <iostream>
#include "C.h"

int main()
{
    std::cout << C::y << '\n'; // YIKES!!!

    return 0;
}
```

What should we do?

No problem... you just need a public method to return the value of `C::y`:

```
// C.h
...

class C
{
public:
    int get_y() const;
private:
    ...
};

...
```

```
// C.cpp
#include "C.h"

...

int C::get_y() const
{
    return y;
}
```

```
#include <iostream>
#include "C.h"

int main()
{
    //std::cout << C::y << '\n';

    C c;
    std::cout << c.get_y() << '\n';

    return 0;
}
```

Note that in this case, we're accessing `C::y` using `c`, which is a `C` object. It does work, but note the `C::get_y()` method works only with a static member variable. So it's better to make `C::get_y()` a static method like this:

```
// C.h
...

class C
{
public:
    static int get_y(); // remove "const"
private:
    ...
};

...
```

```
// C.cpp
#include "C.h"

...

int C::get_y() // remove “const”
{
    return y;
}
```

```
#include <iostream>
#include "C.h"

int main()
{
    std::cout << C::get_y() << '\n';

    return 0;
}
```

Easy right?

WARNING: Do **not** include the `static` keyword in the definition of the method in the cpp file. (But if the definition of the static method is in the class within the header file, i.e. it is inlined, then you have to include the keyword `static`.)

IMPORTANT WARNING ... Because static methods belong to the class rather than the object and only work with static member variables, **the body of the method must not refer to the `this` pointer.** There is no `this` pointer in the body of a static method. Also, you cannot do this:

```
// C.h
...

class C
{
public:
    static int get_y() const; // Huh?!?
private:
    ...
};

...
```

Because “const” refers to the fact the object making this method call cannot be changed by this method. But since the method should be called by the class, there's no object involved (even if you use a `C` object to invoke `get_y()`).

Although you can also call a static method through an object:

```
#include <iostream>
#include "C.h"

int main()
{
    std::cout << C::get_y() << '\n'; // BETTER

    C c;
    std::cout << c.get_y() << '\n';  // WORKS ... BUT

    return 0;
}
```

Exercise. Add a public static method C::set_y() to set the value of C::y in the above class C. Test your code.

Exercise. What's wrong here?

```
class C
{
public:
    C(int a) : x_(a) {}
    static void set() { x_ = a; }
    static C & clear()
    {
        x_ = 0;
        return (*this);
    }
private:
    static int x_;
};
```

Application: Keeping a count of objects which are alive

Suppose you want to maintain a count of the number of objects which are “alive”. This is what I mean:

```
int main()
{
    Car a;      // at this point, there's 1 Car obj
    Car b;      // at this point, there are 2 Car objs

    int x = 42;
    if (x < 42)
    {
        ...
    }
    else
    {
        Car a;  // at this point, there are 3 Car objs
        Car z;  // at this point, there are 4 Car objs
    }
                // at this point, there are 2 Car objs
    Car c;      // at this point, there are 3 Car objs
    Car * p = new Car;
                // at this point, there are 4 Car objs
    Car d = c;  // at this point, there are 5 Car objs
    delete p;   // at this point, there are 4 Car objs

    return 0;   // at this point, there are 0 Car objs
}
```

All you need to do is to declare a static member in Car to keep track of objects which are live:

```
class Car
{
public:

    static int numObjects;
    ...
private:
    ...
};

int C::numObjects(0);
```

Here's what you need to do:

- You want to increment each time an object is created, therefore `numObjects` should increment in each constructor, including the copy constructor.

- You also want to decrement `numObjects` each time an object is destroyed so you want to decrement `numObjects` in the destructor.

Exercise. Modify the default destructor to decrement `numObjects`. Also, modify the copy constructor so that you increment `numObjects`. Complete the following:

```
// Int.h
#ifndef INT_H
#define INT_H

class Int
{
public:
    Int(int a=0)
        : x(a)
    {
        numObjects++;
    }
private:
    static int numObjects;
    ...
};
#endif
```

```
#include "Int.h"

int Int::numObjects(0);
```

```
#include <iostream>
#include "Int.h"

int main()
{
    Int a;
    std::cout << Int::get_numObjects() << '\n'; // 1
    Int b;
    std::cout << Int::get_numObjects() << '\n'; // 2
    Int c = a;
    std::cout << Int::get_numObjects() << '\n'; // 3

    if (1)
    {
        Int d = a;
        std::cout << Int::get_numObjects()
                  << '\n'; // 4
    }
    std::cout << Int::get_numObjects() << '\n'; // 3

    return 0;
```

```
}
```

Application: Controlling default constructor using static object member

Now let me give you a slightly more complicated example where the static member is an object.

If you look at our Date class again, you recall that we use 1970/1/1 as a default date in the default constructor:

```
// Date.h
#ifndef DATE_H
#define DATE_H
class Date
{
public:
    Date(int=1970, int=1, int=1);
    ...
private:
    int yyyy_, mm_, dd_;
};
#endif
```

What if depending on the scenario of usage of the Date class, I sometimes want to change the behavior of the default constructor during runtime? In other words, occasionally, I might want to change the default constructor to initialize the Date object something other than 1970/1/1. In other words I want this to happen in my main():

```
#include "Date.h"

int main()
{
    Date date1; // date1 models 1970/1/1

    // do something here so that after this point
    // default Date constructor works differently

    Date date2; // date2 models 2000/12/25

    return 0;
}
```

How would you do that??? For sure, your program (while it's running) cannot possible change the default values in the constructor!!!

Well … here's I would do it …

Instead of **hardcoding** 1970/1/1 as default values in the constructor:

```
// Date.h
#ifndef DATE_H
#define DATE_H
class Date
{
public:
    Date(int=1970, int=1, int=1);
    ...
private:
    int yyyy_, mm_, dd_;
};
#endif
```

I store 1970/1/1 as three static integers in the Date class:

```
// Date.h

#ifndef DATE_H
#define DATE_H
class Date
{
public:
    Date(int=1970, int=1, int=1);
    ...
private:
    ...
    static int default_yyyy_;
    static int default_mm_;
    static int default_dd_;
};
#endif
```

And initialize these static members in the cpp file:

```
// Date.cpp
...

Date default_yyyy_(1970);
Date default_mm_(1);
Date default_dd_(1);

...
```

These three default integer static members can be used by the default constructor. For instance I can set the default values of the parameters in the constructor to 0 to mean “use the static default yyyy, mm, dd”:

```
// Date.h

#ifndef DATE_H
#define DATE_H
class Date
{
public:
    Date(int=0, int=0, int=0);
    ...
private:
    ...
    static int default_yyyy_;
    static int default_mm_;
    static int default_dd_;
};
#endif
```

and in the `Date` implementation file I do this:

```
//Date.cpp
#include "Date.h"

Date::Date(int yyyy, int mm, int dd)
{
    yyyy_ = (yyyy == 0 ? default_yyyy_ : yyyy);
    mm_ = (mm == 0 ? default_mm_ : mm);
    dd_ = (dd == 0 ? default_dd_ : dd);
}

...
```

Furthermore I can write static methods to modify these static values.

Whoa ... wait …

The three static integers in fact form a date … why not just bunch them up together into a default static `Date` object like this:

```
// Date.h

#ifndef DATE_H
#define DATE_H
class Date
{
public:
    Date(int=0, int=0, int=0);
    ...
private:
    ...
    static Date default_date_;
};
#endif
```

and initialize `default_date_` in the cpp files like this:

```
// Date.cpp
...

Date default_date_(1970, 1, 1);

...
```

The constructor can now use this static default date like this:

```
//Date.cpp
#include "Date.h"

Date::Date(int yyyy, int mm, int dd)
{
    yyyy_ = (yyyy == 0 ? default_date.yyyy_ : yyyy);
    mm_ = (mm == 0 ? default_date.mm_ : mm);
    dd_ = (dd == 0 ? default_date.dd_ : dd);
}

...
```

And we can change the default date like this:

```
// Date.h

#ifndef DATE_H
#define DATE_H
class Date
{
public:
    Date(int=0, int=0, int=0);
    ...
private:
    ...
    static void set_default_date(
```

```
                                int, int, int);
};
#endif
```

```
//Date.cpp
#include "Date.h"

...
void Date::set_default_date(int a, int b, int c)
{
    default_date_ = Date(a, b, c); // changing default
                                   // date
}
...
```

Exercise. Modify the above `Date` class so that the constructor uses the `default_date_` during the constructor and it uses an initializer list.

Exercise. Modify `Date` class by adding an integer `num_dates_` static member that keeps a count of the number of `Date` objects which are still in scope. In other words the value of `num_dates_` will tell us how many `Date` objects are still in memory.

Exercise. In the `IntArray` class, keep track of the number of bytes used by all objects of this class.

Exercise. In the `IntDynArray` class, keep track of the number of bytes used by all objects of this class.

Exercise. In the `IntDynArray` class, keep track of the maximum of number of bytes used by all objects in the class.

Exercise. In your `Date` class, keep a static `Date` object `earliest_date` so that the earliest `Date` object instantiated will have the same values as `earliest_date`. If the constructor is called with values which model a `Date` before `earliest_date`, then the value of `earliest_date` is used instead.

Exercise. You have a class with 200 methods that you are using for a computer game. The class is working well but you want to improve on the performance to get a smoother animation. The game is very complex and it's practically impossible to optimize every single one of the 200 methods. But you still have 1 month before the game is to be shipped and you think you can optimize one function. You want to focus on the most frequently

used method. You don't want to read the code and manually count the number of times a method is called. How would you figure out which method is used most?

Application: Special global objects

In class `vec2d`, suppose you **frequently** use a `vec2d` object with x value of 0 and y value of 0. Because of its values, let's call this concept (or object) the origin.

You can create `origin` as a public static `vec2d` object in `vec2d` class. This allows you do this in `main()`:

```
#include <iostream>
#include "vec2d.h"

int main()
{
    std::cout << vec2d::origin << '\n';
    return 0;
}
```

which has the same effect as

```
#include <iostream>
#include "vec2d.h"

int main()
{
    std::cout << vec2d(0, 0) << '\n';
    return 0;
}
```

except that the first method is faster since the second method has to create `vec2d(0,0)` during runtime and if you need `vec2d(0,0)` again and again and again and again it might be slower (especially if the class in question has a lot more member variables and consumes a lot more memory.)

Exercise. Create a public static `vec2d` object member in your `vec2d` class so that you can run this:

```
#include <iostream>
#include "vec2d.h"

int main()
{
    std::cout << vec2d::origin << '\n';
    return 0;
}
```

Next, note that the member variables of `vec2d::origin` do not change their values. Modify `vec2d::origin` so that it is a **constant** public static member object of `vec2d`.

Application: Debug printing flag for a class

Sometimes when you first write a class, you might need to print low level information on the objects created from that class. You can switch between usual printing and low level debug printing for this class by keeping a boolean flag in the class.

Here's out IntDynArr class for handling dynamic array of integer values:

```
// IntDynArr.h

#ifndef INTDYNARRAY_H
#define INTDYNARRAY_H

class IntDynArr
{
public:
    ...
    void print() const;
private:
    int * x_;
    int size_;
    int capacity_;
};

#endif
```

```
// IntDynArr.cpp

#include <iostream>

void IntDynArray::print() const
{
    std::cout << '[';
    for (int i = 0; i < size_ - 1; ++i)
    {
        std::cout << x_[i] << ", "
    }
    if (size_ - 1 >= 0)
    {
        std::cout << x_[size_ - 1];
    }

    std::cout << ']';
}
```

If `arr` is an object of the above class and the array `arr.x_` points to has values 2, 4, 6 (the size is 3), then `arr.print()` will print

```
[2, 4, 6]
```

While you're writing (and debugging!) the class, you might want to print `arr.size_` and `arr.capacity_` You might want to do this:

```
// IntDynArr.h
```

```
#ifndef INTDYNARRAY_H
#define INTDYNARRAY_H

class IntDynArr
{
public:
    ...
    void print() const;
    static void set_debug(bool);
private:
    static bool debug;
    int * x_;
    int size_;
    int capacity_;
};

#endif
```

```
// IntDynArr.cpp

#include <iostream>

bool IntDynArry::debug(false);

void IntDynArray::print() const
{
    std::cout << '[';
    for (int i = 0; i < size_ - 1; ++i)
    {
        std::cout << x_[i] << ", ";
    }
    if (size_ - 1 >= 0)
    {
        std::cout << x_[size_ - 1];
    }

    std::cout << ']';

    if (debug)
    {
       std::cout << size_ << ' '
                 << capacity_;
    }
}
```

Or you can even print all the value `arr.x_[0]`, ..., `arr.x_[arr.capacity_]` when `IntDynArry::debug` is `true`.

Your `main()` can then turn on/off debug printing:

```
...
int main()
{
    IntDynArray::set_debug(true); // turn on debug
                                  // printing

    IntDynArr arr;
    arr.print();

    IntDynArr::set_debug(false); // turn off debug
                                 // printing
    arr.print();
    return 0;
}
```

Initialization of static constant integer members (DIY)

You can initialize static, **constant** integer members (int, char, etc) in class declaration with constant expression. In other words instead of this:

```
// C.h
class C
{
public:
    static const int c;
};
```

```
// C.cpp
const int C::c(1);
```

You can do this:

```
// C.h
class C
{
public:
    static const int c = 1;
};
```

This “inline initialization” is valid only for integer constants (which includes character constants):

```
class C
{
public:
    static const int c1 = 1;       // OK
    static const int c2 = f(5.3); // ERROR
    static int c3 = 33;            // ERROR
    const int c4 = 4444;           // ERROR
    static const double c5 = 5.5; // ERROR
};
```

(For `c4`, a later C++ compiler which is C++11 compliant might allow you to initialize `c4` inside the class definition.)

Here's a valid use of static constant integer member:

```
class GameEngine
{
public:
    static const int version = 3;
...
};
```

You can also use `enum` to create static constant integer members:

```
class C
{
public:
```

```
    enum {c1=1, c2=22, c3=333, c4=44, c5=5};
};

int main()
{
    std::cout << C::c1 << std::endl;

    return 0;
}
```

Exercise. Experiment with this class:

```
class Employee {
public:
    enum Type {CEO, MGR, FULL_TIME, PART_TIME};
    Employee(char[], Type);
private:
    char name_[1024];
    Type type_;
};

int main()
{
    std::cout << Employee::CEO << std::endl;
}
```

Summary

Static members are members associated to the class rather than to objects.

All the objects share the static members. Static methods do not have `this` pointer. To make member or method static, include the keyword `static` before the declaration of member or method in the class declaration.

In the definition of static members and methods in the implementation file do not include `static` keyword.

If a static member `x` is public in `class C` , you can access it by `C::x` or `obj.x` if `obj` is a `C` object.

If a static method `m()` is public in a `class C`, you call it by `C::m()` or `obj.m()` where `obj` is a `C` object.

Some uses of static members include

- Object counter
- Default objects of a class to control default constructor
- Special global objects within a class