Unit 10

Object-Oriented Programming in C++

Static Members

CONTENTS

Objectives

At the end of this unit we will be able to:

- Create static data members
- Write static member functions

Static Data

A **static** data member is shared by all the objects of a class. Static data members are created before the start of `main`, and exist until the return from `main`.

A static data member may be accessed in two ways: a member of any object of the class (`any_object.member`), or as a part of the class itself (`class_name::member`). Static data members in a class's private section can only be accessed from the class's defining operations.

Static Data

A **static** data member is shared by all the objects of a class.

It can be accessed:

- with the "class-name::"

```
class_name::static_data
```

- as part of any object of the class

```
any_object.static_data
```

Declaring Static Data

Now that we have a simple class string working, we may want to try to change the implementation to make it more flexible, or more efficient, or both. The most efficient implementation may depend on which functions are called most often. For example, if `operator+` is called frequently, and the other functions only rarely, then we might choose to represent a string as a collection of small pieces of text (rather than one large block).

In our example, we will use a static data member to count the total number of calls to `operator+`. We will initialize this static member with the value 0, and each time the `operator+` function is called, it will increment `concat_calls`. We can then determine if a user's application is calling `operator+` frequently, and use that information to guide our choice of implementation of class `String`.

Declaring Static Data

```
class String {
public:
     String();
     String(const char *);
     String(const String &);
     ~String();

     const char *as_char_pointer() const;

     String &operator=(const String &);
     int  length() const;
     int  read();
     void print() const;

     const char &operator [](int) const;
     char & operator [](int);
     String substring(int start, int len) const;

     friend String operator+(const String &, const String &);
private:
     // a String is a sequence of up to
     // max_string_length non-null characters
     // followed by a null character

     char text[max_string_length+1];
   static int concat_calls;
};
```

Working with Static Data

Static members of a class must be defined in some source file[1]. At the point of definition, they may be initialized. Even though this initialization is not done inside one of the class's member functions, users can not change the initial value provided. If the users tried to provide a different initial value, the program could not be linked, because there would be two initial values for the same variable.

Member functions of the class can refer to the static member with just the member name (i.e., `concat_calls`). Friend functions must use the class name or an object to show which class's member they are referring to (i.e., `String::concat_calls` or `lhs.concat_calls`).

1. In C++ releases before 2.0, the declaration in the class served as the definition, of a static data member and initialization was impossible. Release 2.0 of C++ allows the omission of the static member definition, for compatibility with earlier releases.

Working with Static Data

```
#include "String.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int String::concat_calls = 0;

String operator+(const String &lhs, const String &rhs)
{
    String both;

    if (lhs.length() + rhs.length() > max_string_length) {
      fprintf(stderr, "RUN TIME ERROR: String too large\n");
      exit(1);
    }
  String::concat_calls++;

    strcpy(both.text, lhs.text);
    strcat(both.text, rhs.text);
    return both;
}
```

Static Member Functions

Static member functions are accessed with the same syntax as static data members. Since a static member function can be called without an invoking object, it can not use the keyword **this,** and it can not refer to non-static members of its "invoking object". It may, however, access non-static data members of objects of its class (local variables or arguments of the class type, for example), or static data members of its class.

Static Member Functions

Static member functions

- May be invoked
  - — with "class-name::"

```
class_name::static_function(args);
```

  - — with any object of the class

```
any_object.static_function(args);
```

- may not use the keyword **this**
- may not refer to non-static members of its "invoking object"

Static Member Functions (example)

The static member function `n_concatenations` will return the count of concatenations. We have chosen to write `n_concatenations` as a static member function because it does not need an invoking object. Why should we use the syntax `string_variable.n_concatenations()` to call this function, when it has no need for the variable `string_variable` ? We should be able to call it without using a `String` variable, in case we wish to discover the number of concatenations in a function that has no `String` variables.

Static Member Functions (example)

```
class String {
public:
     String();
     String(const char *);
     String(const String &);
     ~String();

     const char *as_char_pointer() const;

     String &operator=(const String &);
     int  length() const;
     int  read();
     void print() const;

     const char &operator [](int) const;
     char & operator [](int);
     String substring(int start, int len) const;

   static int n_concatenations();
     friend String operator+(const String &, const String &);
private:
     // a String is a sequence of up to
     // max_string_length non-null characters
     // followed by a null character

     char text[max_string_length+1];
     static int concat_calls;
};
```

Writing Static Member Functions

A static member function may refer to the static members of its class without specifying which class they belong to. It may also access the private members of any object of its class. For example, if `n_concatenations` declared a local `String` variable `s`, it would have access to `s.text`.

Writing Static Member Functions

```
#include "String.h"

int String::n_concatenations()
{
    return concat_calls;
}
```

Using Static Members

Using Static Members

```
#include "String.h"
#include <stdio.h>

main(int, char *[])
{
    String firstname = "Zaphod",
           lastname = "Beeblebrox",
           name;

    name = firstname + " " + lastname;
    ("Name is: " + name).print();

    printf("There were %d concatenations.\n",
        String::n_concatenations());

    return 0;
}
```

Summary

Summary

Static members

- are shared by all objects of the class
- may be either data or functions
- obey scope rules (public/private)

Lab Exercises

Lab Exercises

UNIT 10

Lab Exercises

1. Change to the *unit10/point* directory. Add private static data members to the class **Point** to count the number of calls to each constructor. Add a static member function that prints out the number of calls to each constructor, and the total number of calls to all constructors. Name this member function **print_stats**, and test it with the programs *test_stats.c* and *test_stats2.c*. These programs are in your *unit10/point* directory. The **print_stats** member function should be added to the *point.c* implementation file.

   You can compile and execute the test programs by entering '**make**' or you can compile and execute them directly using the commands:

```
$ CC -o test_stats test_stats.c point.c print.c
$ test_stats
$ CC -o test_stats2 test_stats2.c point.c print.c
$ test_stats2
```

SUMMARY	
DIRECTORY	unit10/point
DECLARATION	Point.h (modify), print.h
IMPLEMENTATION	point.c (modify), print.c
TEST PROGRAM	test_stats.c, test_stats2.c

FILE: **test_stats.c**

```
#include "Point.h"

main(int, char *[])
{
        Point p1(1, 1), p2(10, 10);
        Point p3;
        p3 = p1 + p2;

        Point::print_stats();

        return 0;
}
```

FILE: **test_stats2.c**

```
#include "Point.h"

Point global_var1(0, 0), global_var2(0, 1);

real_main(int, char *[])
```

```
{
        Point p1(1, 1), p2(10, 10);
        Point p3;
        p3 = p1 + p2;

        return 0;
}

main(int argc, char *argv[])
{
        real_main(argc, argv);

        Point::print_stats();

        return 0;
}
```

UNIT 10

Lab Exercises (Answers)

1. Static data have been added to count the number of calls, and a static function has been added to print out the data. Note that the function takes a default argument of type **FILE *** to allow the printing of statistics to a file.

FILE: **Point.h**

```
#include <stdio.h>

class Point {
public:
        Point(int x, int y);
        Point();
        ~Point();

        int x() const;
        int y() const;
        void set_to(int x, int y);

        Point operator+(const Point &) const;
        Point operator-(const Point &) const;

        friend Point operator*(int,  const Point &);
        friend Point operator*(const Point &, int);

        Point operator/(int) const;

        int operator==(const Point &) const;
        int operator!=(const Point &) const;

        Point &operator+=(const Point &);

        static void print_stats(FILE *fp = stdout);

private:
        int _x;
        int _y;

        static int default_ctor_count;
        static int other_ctor_count;
};

inline int Point::x() const
{
   return (_x);
}

inline int Point::y() const
{
   return(_y);
}
```

FILE: **point.c**

```
#include "Point.h"
#include <stdio.h>

int Point::default_ctor_count = 0;
int Point::other_ctor_count = 0;

void Point::set_to(int x, int y)
{
   _x = x;
   _y = y;
}

Point Point::operator+(const Point &p) const
{
   Point temp;
   temp._x = _x + p._x;
   temp._y = _y + p._y;
   return temp;
}

Point Point::operator-(const Point &p) const
{
   Point temp;
   temp._x = _x - p._x;
   temp._y = _y - p._y;
   return temp;
}

Point operator*(int i, const Point &p)
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
}

Point operator*(const Point &p, int i)
{
   Point temp;
   temp._x = i * p._x;
   temp._y = i * p._y;
   return temp;
}

Point Point::operator/(int i) const
{
   Point temp;
   temp._x = _x / i;
   temp._y = _y / i;
   return temp;
}

int Point::operator==(const Point &p) const
{
   return (_x == p._x && _y == p._y)?1:0;
}
```

```
int Point::operator!=(const Point &p) const
{
   return (_x != p._x || _y != p._y)?1:0;
}

Point &Point::operator+=(const Point &p)
{
   *this = *this + p;
   return *this;
}

Point::Point(int x, int y)
{
   _x = x;
   _y = y;
   other_ctor_count++;
}

Point::Point()
{
   default_ctor_count++;
}

Point::~Point()
{
}

void Point::print_stats(FILE *fp)
{
     fprintf(fp, "There were %4d calls to the default constructor,\n",
           default_ctor_count);
     fprintf(fp, "       and %4d calls to the other constructor,\n",
           other_ctor_count);
     fprintf(fp, " totalling %4d calls in all.\n",
           default_ctor_count + other_ctor_count);
}
```