Programming in Modern C++

Module M16: static Members

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Weekly Recap

- Understood the core OOP features of C++: Class, Object, Attributes (Data members), Methods (Member functions), State of an Object, Encapsulation by Access Specifiers, get-set idioms for Information Hiding, various types of Constructors, Destructors, and copy mechanisms in terms of deep & shallow copy
- Understood the lifetime aspects of objects
- Understood how bit-wise `const`-ness can be modeled for objects and leveraged in the design with `const` objects, data members, and member functions, and how `mutable` can help model logical `const`-ness

Module Objectives

- Understand `static` data member and member function

Module Outline

static Data Member

static Data Member

- A `static` data member
  - is *associated with class* not with object
  - is *shared by all the objects* of a class
  - needs to be *defined outside the class scope* (in addition to the *declaration within the class scope*) to avoid linker error
  - *must be initialized* in a source file
  - is constructed before `main()` starts and destructed after `main()` ends
  - can be `private` / `public`
  - can be accessed
    - with the class-name followed by the scope resolution operator (`::`)
    - as a member of any object of the class
  - **virtually eliminates any need for global variables in OOPs environment**
- We illustrate first with a simple example and then with a Print Task where:
  - There is a printer which can be loaded with a paper from time to time
  - Several print jobs (each requiring a number of pages) may be fired on the printer

Program 16.01: static Data Member: Example

Non static Data Member

```
#include<iostream>
using namespace std;
class MyClass { int x; // Non-static
public:
    void get() { x = 15; }
    void print() { x = x + 10;
        cout << "x =" << x << endl ;
    }
};

int main() {
    MyClass obj1, obj2; // Have distinct x
    obj1.get(); obj2.get();
    obj1.print(); obj2.print();
}
---
x = 25 , x = 25
```

- x is a non-static data member
- x cannot be shared between obj1 & obj2
- Non-static data members do not need separate definitions - instantiated with the object
- Non-static data members are initialized during object construction

static Data Member

```
#include<iostream>
using namespace std;
class MyClass { static int x; // Declare static
public:
    void get() { x = 15; }
    void print() { x = x + 10;
        cout << "x =" << x << endl;
    }
};
int MyClass::x = 0; // Define static data member
int main() {
    MyClass obj1, obj2; // Have same x
    obj1.get(); obj2.get();
    obj1.print(); obj2.print();
}
---
x = 25 , x = 35
```

- x is static data member
- x is shared by all MyClass objects including obj1 & obj2
- static data members must be defined in the global scope

- static data members are initialized during program start-up

Program 16.02: static Data Member: Print Task (Unsafe)


```
#include <iostream>
using namespace std;
class PrintJobs { int nPages_;  /* # of pages in current job */ public:
    static int nTrayPages_; /* # of pages in the tray */ static int nJobs_; // # of print jobs executing
    PrintJobs(int nP): nPages_(nP) { ++nJobs_; cout << "Printing " << nP << " pages" << endl;
        nTrayPages_ = nTrayPages_ - nP;
    }                           // Job started
    ~PrintJobs() { --nJobs_; } // Job done
};
int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0;        // Definition and initialization -- no job to start with
int main() {
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
    PrintJobs job1(10);
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
    {
        PrintJobs job1(30), job2(20); // Different job1 in block scope
        cout << "Jobs = " << PrintJobs::nJobs_ << endl;
        cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
        PrintJobs::nTrayPages_ += 100; // Load 100 more pages
    }
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
}
```

Output:

```
Jobs = 0
Pages= 500
Printing 10 pages
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 490
Printing 30 pages
Printing 20 pages
Jobs = 3 // same nJobs_, nTrayPages_
Pages= 440
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 540
```

Program 16.03/04: Order of Initialization: Order of Definitions


```
#include <iostream>
#include <string>
using namespace std;
class Data { string id_; public:
    Data(const string& id) : id_(id)
    { cout << "Construct: " << id_ << endl; }
    ~Data()
    { cout << "Destruct: " << id_ << endl; }
};
class MyClass {
    static Data d1_; // Listed 1st
    static Data d2_; // Listed 2nd
};
Data MyClass::d1_("obj_1"); // Constructed 1st
Data MyClass::d2_("obj_2"); // Constructed 2nd

int main() { }
```

```
-----
Construct: obj_1
Construct: obj_2
Destruct: obj_2
Destruct: obj_1
```

```
#include <iostream>
#include <string>
using namespace std;
class Data { string id_; public:
    Data(const string& id) : id_(id)
    { cout << "Construct: " << id_ << endl; }
    ~Data()
    { cout << "Destruct: " << id_ << endl; }
};
class MyClass {
    static Data d2_; // Order of static members swapped
    static Data d1_;
};
Data MyClass::d1_("obj_1"); // Constructed 1st
Data MyClass::d2_("obj_2"); // Constructed 2nd

int main() { }
```

```
-----
Construct: obj_1
Construct: obj_2
Destruct: obj_2
Destruct: obj_1
```


- *Order of initialization of static data members does not depend on their order in the definition of the class. It depends on the order their definition and initialization in the source*

static Member function

static Member Function

- A `static` member function
  - does not have `this` pointer – not associated with any object
  - *cannot access* non-`static` data members
  - *cannot invoke* non-`static` member functions
  - can be accessed
    - with the class-name followed by the scope resolution operator (`::`)
    - as a member of any object of the class
  - is needed to read / write `static` data members
    - Again, for encapsulation `static` data members should be `private`
    - `get()-set()` idiom is built for access (`static` member functions in `public`)
  - *may initialize* `static` data members *even before any object creation*
  - *cannot co-exist with a non-*`static` *version* of the same function
  - *cannot be declared* as `const`
- We repeat the Print Task with better (safer) modeling and coding

Program 16.05: static Data & Member Function: Print Task (Safe)


```
// #include <iostream> using namespace std;
class PrintJobs { int nPages_; // # of pages in current job
    static int nTrayPages_; /* # of pages in the tray */ static int nJobs_; // # of print jobs executing
public: PrintJobs(int nP) : nPages_(nP) { ++nJobs_; cout << "Printing " << nP << " pages" << endl;
            nTrayPages_ = nTrayPages_ - nP; } // Job started
    ~PrintJobs() { --nJobs_; }                // Job done
    static int getJobs()          { return nJobs_; }  // get on nJobs_. Readonly. No set provided
    static int checkPages()       { return nTrayPages_; } // get on nTrayPages_
    static void loadPages(int nP) { nTrayPages_ += nP; } // set on nTrayPages_
};
int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0;        // Definition and initialization -- no job to start with
int main() { cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    PrintJobs job1(10);
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    {
        PrintJobs job1(30), job2(20); // Different job1 in block scope
        cout << "Jobs = " << PrintJobs::getJobs() << endl;
        cout << "Pages= " << PrintJobs::checkPages() << endl;
        PrintJobs::loadPages(100);    // Load 100 more pages
    }
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
}
```

Output:

```
Jobs = 0
Pages= 500
Printing 10 pages
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 490
Printing 30 pages
Printing 20 pages
Jobs = 3 // same nJobs_, nTrayPages_
Pages= 440
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 540
```

Counting Objects

- We illustrate another example and use for `static` data member and member function
  - Here we want to track the number of objects created and destroyed for a class at any point in the program
  - Naturally no object can keep this information. So we hold two `static` data members
    - `nObjCons_`: Number of objects created since beginning. It is *read-only and incremented in every constructor*
    - `nObjDes_`: Number of objects destroyed since beginning. It is *read-only and incremented in the destructor*
  - At any point (`nObjCons_ - nObjDes_`) gives the number of *Live* objects
  - In an alternate (less informative model) we may just maintain `static` data member `nLive_` which is *incremented in every constructor and decremented in the destructor*

Program 16.06: Count Objects


```
#include <iostream>
#include <string>
using namespace std;
class MyClass { string id_; // Object ID
    static int nObjCons_, nObjDes_; // Object history
public:
    MyClass(const string& id) : id_(id)
    { ++nObjCons_;
    cout << "ctor: " << id_ << " "; getObjLive(); }
    ~MyClass() { ++nObjDes_;
    cout << "dtor: " << id_ << " "; getObjLive(); }
    static int getObjConstructed()
    { return nObjCons_; }
    static int getObjDestructed()
    { return nObjDes_; }
    // Get number of live objects
    static int getObjLive() {
        int nLive = nObjCons_ - nObjDes_;
        cout << "Live Objects = " << nLive << endl;
        return nLive;
    }
};
int MyClass::nObjCons_ = 0;
int MyClass::nObjDes_ = 0;
```

```
int dummy1(MyClass::getObjLive()); // Before (main())
MyClass sObj("sObj");
int dummy2(MyClass::getObjLive()); // Before (main())
int main() { MyClass::getObjLive();
    MyClass aObj("aObj");
    MyClass *dObj = new MyClass("dObj");
    {
        MyClass bObj("bObj");
        delete dObj;
    }
    MyClass::getObjLive();
}
```

```
Live Objects = 0 // Before any object (dummy1)
ctor: sObj Live Objects = 1
Live Objects = 1 // Before main() (dummy2)
Live Objects = 1 // Enter main()
ctor: aObj Live Objects = 2
ctor: dObj Live Objects = 3
ctor: bObj Live Objects = 4
dtor: dObj Live Objects = 3
dtor: bObj Live Objects = 2
Live Objects = 2 // Exit main()
: aObj Live Objects = 1
dtor: sObj Live Objects = 0 // After all objecst
```

static vis-a-vis non-static

static **vis-a-vis** **non**-static

Comparison of static vis-a-vis non-static

static Data Members	Non-static Data Members
• Declared using keyword static	• Declared *without* using keyword static
• All objects of a class share the *same copy / instance*	• Each object of the class gets its *own copy / instance*
• *Accessed* using the *class name or object*	• *Accessed* only through an *object* of the class
• May be public or private	• May be public or private
• Belongs to the namespace of the class	• Belongs to the namespace of the class
• May be const	• May be const
• Are *constructed before* main() is invoked	• Are *constructed during* object construction
• Are *destructed after* (in reverse order) main() returns	• Are *destructed during* object destruction
• Are *constructed* in the order of definitions in source	• Are *constructed* in the order of listing in the class
• Has a *lifetime* encompassing main()	• Has a *lifetime* as of the lifetime of the object
• *Allocated* in *static* memory	• *Allocated* in *static*, *stack*, or *heap* memory as of the object

static Member Functions	Non-static Member Functions
• Declared using keyword static	• Declared *without* using keyword static
• *Has no* this pointer parameter	• *Has an implicit* this pointer parameter
• *Invoked* using the *class name or object*	• *Invoked* only through an *object* of the class
• May be public or private	• May be public or private
• Belongs to the namespace of the class	• Belongs to the namespace of the class
• *Can access* static data members and methods	• *Can access* static data members and methods
• *Cannot access* non-static data members or methods	• *Can* access non-static data members and methods
• Can be invoked anytime during program execution	• Can be invoked only during *lifetime* of the object
• Cannot be virtual or const	• May be virtual and / or const
• *Constructor* is static though not declared static	• *There cannot be* a non-static Constructor

Singleton Class

Singleton Class

- **Singleton** is a *creational design pattern*
  - ensures that *only one object* of its kind exists and
  - provides a *single point of access* to it for any other code
- A class is called a Singleton if it satisfies the above conditions
- Many classes are singleton:
  - President of India
  - Prime Minister of India
  - Director of IIT Kharagpur
  - CEO of a Company
  - ...
- How to implement a Singleton Class?
- How to restrict that user can created *only one* instance?

Program 16.07: static Data & Member Function
Singleton Printer


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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
private: bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; } // Private -- Printer cannot be constructed!
    static Printer *myPrinter_;               // Pointer to the Instance of the Singleton Printer
public: ~Printer() { cout << "Printer destructed" << endl; }
    static const Printer& printer(bool bw = false, bool bs = false) { // Access the Printer
        if (!myPrinter_) myPrinter_ = new Printer(bw, bs);            // Constructed for first call
        return *myPrinter_;                                           // Reused from next time
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};

Printer *Printer::myPrinter_ = 0;

int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);

    delete &Printer::printer();
}
```

Output:

```
Printer constructed
Printing 10 pages
Printing 20 pages
Printer destructed
```

Program 16.08: Using function-local static Data
Singleton Printer


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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
    bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; }
    ~Printer() { cout << "Printer destructed" << endl; }
public:
    static const Printer& printer(bool bw = false, bool bs = false) {
        static Printer myPrinter(bw, bs); // The Singleton -- constructed the first time

        return myPrinter;
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};
int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);
}
```

Output:

```
Printer constructed
Printing 10 pages
Printing 20 pages
Printer destructed
```

- Function local `static` object is used
- No memory management overhead – so destructor too get `private`
- This is called **Meyer's Singleton**

Module Summary

- Introduced `static` data member
- Introduced `static` member function
- Exposed to use of `static` members
- Singleton Class discussed