2.4 Constructors & Destructors

Sue is working on her Date class but seems a bit confused. If the client does not call her `initialize()` function, the member variables in her class will remain uninitialized. What can she do to guarantee her `initialize()` function gets called? As she ponders this point, she stumbles upon constructors in the reading…

Objectives

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

- Describe situations when a constructor would be helpful in a class design
- Create a class with a default constructor, non-default constructor, and a copy constructor
- Create a destructor and describe when one might be used
- Be able to predict which constructors are called in a given situation

Prerequisites

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

- Create a class with member functions (Chapter 2.2)
- Use accessors and mutators in a class definition (Chapter 2.3)
- Explain the differences between pass-by-value, pass-by-pointer, and pass-by-reference (Procedural Programming in C++ Chapter 1.4, 3.3)

What are constructors and why you should care

A **constructor** is a method in a class that is guaranteed to be called when an object is instantiated. A **destructor** is a method in a class that is guaranteed to be called when an object is destroyed such as when it falls out of scope. These two methods are essential to the principle of encapsulation.

If a class designer is truly going to absolve the client from having to know anything about the implementation details of a class, then there must be some way to make such housecleaning chores as initializing variables invisible. Back to our `Date` class from Chapter 2.3, if the client fails to call `initialize()` then nothing will work!

There are several scenarios where constructors and destructors are very convenient. The need to initialize member variables necessitates constructors for almost every class. Classes allocating and freeing memory usually need both constructors and destructors to prevent memory leaks. Tools initiating complex operations such as setting up and tearing down graphics windows use constructors and destructors to ensure the client does not forget important procedures. There are so many uses for constructors and destructors that it is difficult to think of an application that would not benefit from them.

Constructor

A constructor is a special method in a class that is guaranteed to be called when an object is instantiated. It is unique because it has the following properties:

- **Name**: Constructors must have the same name as the class in which they are a member
- **Return value**: Constructors have no return type, not even `void`!

When an object is instantiated, the compiler selects the most appropriate constructor defined in the class to be called. The three types of constructors are the default constructor, non-default constructors, and the copy constructor.

Default constructor

The default constructor is the constructor lacking any parameters. Back to our `Card` class, the default constructor might be:

```
class Card
{
   public:
      Card()                        // default constructor
      {
         value = 0;
         assert(validate());        // make sure our Card is well formed
      }
... code removed for
};
```

Default Constructors are declared with no return type and no parameters.

Constructors can call functions or methods just like any other member function.

Observe how the method takes on the name of its class, that there is no return type, and this constructor takes no parameters.

Constructors have other syntactic nicety that the C++ language affords us: the **initialization section**. It provides a special way to initialize member variables outside the function body. The initialization section affords us this convenience; just list the member variables after the function name and specify the default value:

```
class Card
{
   public:
      Card() : value(0)             // this constructor is inline
      {
         assert(validate());
      }
... code removed for brevity ...
};
```

Variable to be initialized and the value to be assigned to it.

The variables in the initialization section get set immediately before the code in the body of the function is executed. Observe in the above example that `validate()` checks `value`. Clearly `value` must be set to zero first or our assert will fire.

Of course we do not need to make our constructors inline. The following will work as well:

```
Card :: Card() : value(0)           // non-inline constructors can also have
{                                   //    initialization sections.  The functionality
   assert(validate());              //    is exactly the same as the non-inline variant
}
```

Non-default constructors

Frequently we wish to instantiate an object and initialize it in one step. Consider, for example, the `ifstream` class. We can either create a `fin` object first and then attach it to a file…

```
{
   ifstream fin;                 // instantiate the fin object
   fin.open("data.txt");         // initialize it in a separate step
   …
}
```

… or we could instantiate `fin` and attach it to a file at the same time…

```
{
   ifstream fin("data.txt");     // instantiate and initialize all at once
   …
}
```

How is this done? The answer is that a special constructor was created for `ifstream` that takes a c-string as a parameter. This is a non-default constructor.

A non-default constructor is a constructor that takes a parameter. The parameter can be really anything, though it is usually a good idea to make it related to how the client would want to initialize an object. Back to our `Card` class, one default constructor might be to set the suit and the rank.

```
Card :: Card(int iSuit, int iRank)
{
   // make sure we are in the valid range
   if (iSuit >= (sizeof(SUITS) / sizeof(SUITS[0])) || iSuit < 0)
      iSuit = 0;
   if (iRank >= (sizeof(RANKS) / sizeof(RANKS[0])) || iRank < 0)
      iRank = 0;

   // assign
   value = iSuit * 13 + iRank;

   // paranoia
   assert(validate());
}
```

We can have as many non-default constructors as we choose. Another might take a string as a parameter.

```
Card :: Card(const string & s) : value(0)
{
   parse(s);
   assert(validate());
}
```

The client can create an object using a non-default constructor as you might expect:

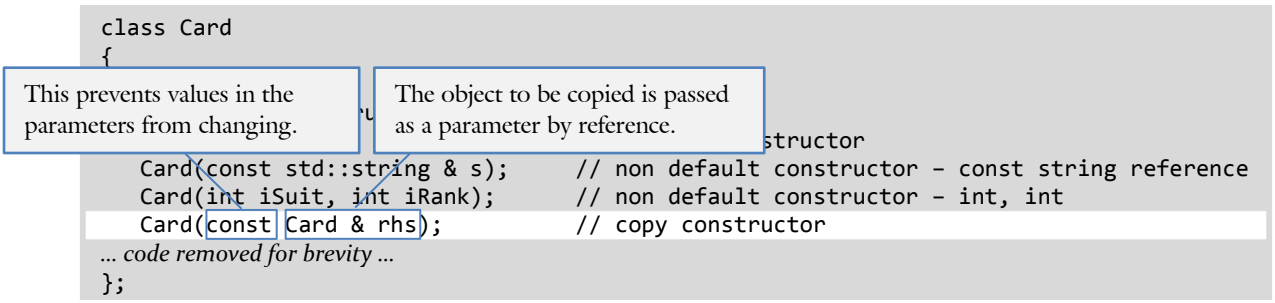
```
{
   // call the "int, int" constructor
   Card sixHearts(1, 4);           // iSuit == 1 which is a Heart
   …                               // iRank == 4 which is the six

   // call the "const string by-reference" constructor
   string s("8c");                // eight of clubs
   Card eightClubs(s);            // call with string("8c") as a parameter
   …
}
```

Copy constructor

A copy-constructor is a special constructor designed to make an exact copy of an object. Every copy-constructor must have the following properties:

- It must be a constructor.
- It takes exactly one parameter.
- The parameter must be the same data type as the class itself
- The parameter must be constant and by reference.

Thus there can be no more than one copy constructor for a given class. Back to our `Card` example, the declaration of the copy constructor would be:




```
class Card
{
                                                    structor
   Card(const std::string & s);     // non default constructor - const string reference
   Card(int iSuit, int iRank);      // non default constructor - int, int
   Card(const Card & rhs);          // copy constructor
... code removed for brevity ...
};
```

The complete solution for this example is available at 2-4-card.html:

```
/home/cs165/examples/2-4-card.cpp
```

Observe how it accepts as a parameter a constant `Card` by reference. The implementation would be:

```
Card :: Card(const Card & rhs)
{
   assert(rhs.validate());          // call the validate method of the "rhs" parameter
   value = rhs.value;               // do the actual work
   assert(validate());              // call the validate method of "this"
}
```

By convention, we call the parameter "`rhs`" which stands for "Right Hand Side." The reason for this will become apparent in Chapter 2.6 through 2.8. There are several scenarios when the copy constructor is called. The first is when the client wishes to make a copy of a given object:

```
{
   Card card1(string("qh"));        // create a queen of hearts with the non-default
                                    //     constructor: const string reference
   Card card2(card1);               // create a copy!
}
```

Another time the copy constructor is called is when an object is passed by-value. Recall that pass-by-value makes a copy of the parameter. How does the compiler know how to make a copy of an object? With the copy constructor of course!

```
void function(Card card)            // make a copy of Card.  Are you sure you want to
{                                   //     do that?  This should be const by-reference
   …
}
```

The final way the copy constructor is called is when an object is returned by-value:

```
Card makeCard(Card & card)          // it would be better to return a reference here:
{                                   //     Card & makeCard(Card & card)
   …
   return card;                     // return-by-value so we make copy is made here.
}                                   //     This should be return-by-reference instead.
```

Using constructors

As mentioned previously, constructors get called automatically when an object is instantiated. When no constructor is present, none gets called:

```
class Simple
{
   public:
      void display() { cout << value << endl; }
   private:
      int value;
};

int main()
{
   Simple s;               // no constructor is present so none gets called
   s.display();            // s.value is not initialized.
}
```

In other words, a default constructor is not created for classes lacking one and the object is instantiated without a constructor being called. However, if a contructor does exist for a class but the client does not specify one, an error will occur. Consider the case where a non-default constructor exists for our `Simple` class. Note that there is no default constructor:

```
class Simple
{
   public:
      Simple(int value) : value(value) {      }  // non-default constructor
   private:
      int value;
};
int main()
{
   Simple s;               // ERROR: no default constructor
}
```

In this example, we are trying to summon the default constructor but none exists, yielding a compile error:

```
2-4-errors.cpp: In function “int main()”:
2-4-errors.cpp:22: error: no matching function for call to “Simple::Simple()”
2-4-errors.cpp:12: note: candidates are: Simple::Simple(int)
2-4-errors.cpp:10: note:                 Simple::Simple(const Simple&)
```

Notice that, though only one constructor was provided (the non-default constructor taking an integer as a parameter), two are candidates. This is because, lacking a copy constructor, one is created that performs a simple copy of member variables. This may be what you want. However, if there is a dynamically allocated buffer in the class, a bug will likely result:

```
class String
{
   private:
      char * buffer;       // Warning: dynamically allocated array
... code removed for brevity ...
};
```

Instead of creating a new buffer, it will instead just copy the address. Thus both the new object and the old will refer to the same buffer. This means that changes made to one object will also be reflected in the other. You must always write a copy constructor for classes with dynamically allocated memory.

The complete solution is available at 2-2-errors.html or:

```
/home/cs165/examples/2-2-errors/
```

Destructor

While constructors are special methods that are called when an object is created, a destructor is a method that is called when an object is destroyed. Unlike with constructors, however, there can never be more than one destructor. Destructors have the following properties:

- The name is a combination of the class name with a tilde ~ on the front.
- There is no return type, not even `void`.
- There are no parameters.

Destructors are commonly used to free allocated memory, close files, terminate communications with external libraries such as graphic canvases, and a host of other things. None of these really apply to our `Card` class so the following destructor is a bit contrived:

```
class Card
{
  public:
   // various constructors
   Card();                          // default constructor
   Card(const std::string & s);     // non default constructor - const string reference
   Card(int iSuit, int iRank);      // non default constructor - int, int
   Card(const Card & rhs);          // copy constructor

   // destructor
   ~Card()                          // never more than one destructor
   {
      assert(validate());
   }
... code removed for brevity ...
};
```

Of course we could also define the destructor as a non-inline method:

```
Card :: ~Card()                     // notice the ~ before the name
{
   assert(validate());              // any code can go here, even a function call
}                                   // we can't return anything because no return type
```

Example 2.4 – Silly

Demo

This example will demonstrate how to predict the output of code involving creating objects from classes with constructors.

Problem

Consider the following class:

```
class Silly
{
   public:
      Silly()                       { cout << "Default constructor\n"; }
      Silly(const Silly & s)        { cout << "Copy constructor\n";    }
     ~Silly()                       { cout << "Destructor\n";         }
};
```

What is the output of the following code:

```
int main()
{
   Silly s1;                // line 1
   Silly s2(s1);            // line 2
   return 0;                // line 3
}
```

Solution

In Line 1, the default constructor is called because there are no parameters specified beside the `s1` variable. This means the following output will result:

```
Default constructor
```

In Line 2, we are creating another `Silly` object. This one will use the copy constructor because `s1` is passed to the constructor as a parameter. Therefore, the copy constructor will be called:

```
Copy Constructor
```

In Line 3, we leave the `main()` function and terminate the program. This means that the destructor for both `s1` and `s2` will be called:

```
Destructor
Destructor
```

See Also

The complete solution is available at 2-4-silly.html or:

```
/home/cs165/examples/2-4-silly.cpp
```

Example 2.4 – Silly with Functions

Demo

This example will demonstrate how to predict the output of code involving creating objects from classes with constructors when functions are called.

Problem

Consider the following class:

```
class Silly
{
   public:
      Silly()                        { cout << "Default constructor\n"; }
      Silly(const Silly & s)         { cout << "Copy constructor\n";    }
     ~Silly()                        { cout << "Destructor\n";         }
};
```

What is the output of the following code:

```
Silly function(Silly s)
{
   cout << "Function\n";           // function line 1
   return s;                       // function line 2
}

int main()
{
   Silly s;                        // main line 1
   function(s);                    // main line 2
   return 0;                       // main line 3
}
```

Solution

In "main line 1," the default constructor is called because there are no parameters specified:

```
Default constructor
```

In "main line 2," `function()` is called passing `s` as a by-value parameter. This calls the copy-constructor:

```
Copy Constructor
```

In "function line 1" we encounter the `cout` line in `function()`.

```
Function
```

In "function line 2," because `function()` is return-by-value, we need to create another copy of `s`. Additionally, the parameter that was pass-by-value needs to be destroyed.

```
Copy constructor
Destructor
```

We are back to "main line 2" where the return-by-value object needs to be destroyed.

```
Destructor
```

In "main line 3," the object `s` needs to be destroyed because we are exiting the function `main()`:

```
Destructor
```

See Also

The complete solution is available at 2-4-sillyWithFunctions.html or:

```
/home/cs165/examples/2-4-sillyWithFunctions.cpp
```

Example 2.4 – Time

Demo

This example will demonstrate how to use the default constructor, a non-default constructor, and the copy constructor in our `Time` class from previous chapters.

Problem

Modify the `Time` class from "Example 2.3 – Time" on page 139 to include a default constructor that sets the time to midnight, a non-default constructor taking hours and minutes as a parameter, and a copy constructor.

Solution

The class definition describing the three constructors:

```
class Time
{
   public:
     // Constructors
     Time() : minutes(0) { }                  // default constructor
     Time(int hours, int minutes = 0)         // non-default constructor
     {
        set(hours, minutes);
     }
     Time(const Time & rhs)                   // copy constructor
     {
        assert(rhs.validate());
        minutes = rhs.minutes;
        assert(validate());
     }
... code removed for brevity ...
};
```

The driver program is:

```
int main()
{
   // exercise the default constructor
   Time time1;
   cout << "Time1 is midnight - ";
   time1.display();
   cout << endl;

   // the non-default constructor
   Time time2(10 /*hours*/, 11 /*minutes*/);
   cout << "Time2 is in the morning - ";
   time2.display();
   cout << endl;

   // the copy constructor
   Time time3(time2);
   cout << "Time3 is the same as time2 - ";
   time3.display();
   cout << endl;

   return 0;
}
```

See Also

The complete solution is available at 2-4-time.html or:

```
b/home/cs165/examples/2-4-time.cpp
```

Example 2.4 – Write

Demo

This example will demonstrate how to use constructors and destructors for something other than initializing variables.

Problem

Create a class to write text to a file. The client of this class should not have to think about opening or closing a file; it should happen automatically.

Solution

The header file is the following:

```
#ifndef WRITE_H
#define WRITE_H

#include <fstream>        // necessary for the ofstream object
#include <string>         // necessary for the string parameters

class Write
{
   public:
      Write() : isOpen(false)      {       }
      Write(const std::string & fileName);
     ~Write();
      void writeToFile(const std::string & text);
   private:
      bool          isOpen;           // did we successfully open the file?
      std::ofstream fout;             // the file stream object
};
#endif // _WRITE_H_
```

Notice how the constructors and destructors take care of opening and closing the file. The implementation file is the following:

```
#include "write.h"
using namespace std;

Write::Write(const string & fileName) : isOpen(false)
{
   fout.open(fileName.c_str());
   isOpen = !(fout.fail());
}

void Write::writeToFile(const string & text)
{
   if (isOpen)
      fout << text << endl;
}

Write::~Write()
{
   if (isOpen)
      fout.close();
}
```

See Also

The complete solution is available at 2-4-write.html or:

```
/home/cs165/examples/2-4-write/
```

Problem 1

Given the following code:

```
class Silly
{
   public:
      Silly()                  { cout << "Default constructor\n"; }
      Silly(const Silly & s)   { cout << "Copy constructor\n";    }
     ~Silly()                  { cout << "Destructor\n";          }
      void method()            { cout << "Method\n";             }
};

int main()
{
   Silly s1;
   if (true)
   {
      Silly s2(s1);
      s2.method();
   }
   s1.method();
   return 0;
}
```

What is the output?

Please see page 149 *for a hint.*

Problem 2

Given the following code:

```
class Silly
{
   public:
      Silly()                  { cout << "Default constructor\n"; }
      Silly(const Silly & s)   { cout << "Copy constructor\n";    }
     ~Silly()                  { cout << "Destructor\n";          }
      void method()            { cout << "Method\n";             }
};

Silly function(Silly & s)
{
   s.method();
   return s;
}

int main()
{
   Silly s1;
   s1.method();
   function(s1);
   return 0;
}
```

What is the output?

Please see page 150 *for a hint.*

Problem 3 & 4

For the following class diagram:

3. Write the header file for the `Temp` class. Ensure the temperature is not below -273˚C

4. Write the source file for the `Temp` class that implements any of the methods not defined as inline in the class definition.

Please see page 151 *for a hint.*