ABSOLUTE C++

SIXTH EDITION

Walter Savitch

Chapter 7

Constructors and Other Tools

PEARSON

Learning Objectives

- Constructors
  - Definitions
  - Calling

- More Tools
  - const parameter modifier
  - Inline functions
  - Static member data

- Vectors
  - Introduction to vector class

Constructors

- Initialization of objects
  - Initialize some or all member variables
  - Other actions possible as well

- A special kind of member function
  - Automatically called when object declared

- Very useful tool
  - Key principle of OOP

Constructor Definitions

- Constructors defined like any member function

  - Except:

    1. Must have same name as class

    2. Cannot return a value; not even void!

Constructor Definition Example

- Class definition with constructor:
  - class DayOfYear
    {
    public:
        DayOfYear(int monthValue, int dayValue);
                //Constructor initializes month and day
        void input();
        void output();
        …
    private:
        int month;
        int day;
    }

Constructor Notes

- Notice name of constructor: DayOfYear
  - Same name as class itself!

- Constructor declaration has no return-type
  - Not even void!

- Constructor in public section
  - It's called when objects are declared
  - If private, could never declare objects!

Calling Constructors

- Declare objects:
  DayOfYear date1(7, 4),
           date2(5, 5);

- Objects are created here
  - Constructor is called
  - Values in parens passed as arguments to constructor
  - Member variables month, day initialized: date1.month → 7 date2.month → 5 date1.dat → 4 date2.day → 5

Constructor Equivalency

- Consider:
  - DayOfYear date1, date2
    date1.DayOfYear(7, 4);   // ILLEGAL!
    date2.DayOfYear(5, 5);   // ILLEGAL!

- Seemingly OK…
  - CANNOT call constructors like other member functions!

Constructor Code

- Constructor definition is like all other member functions:
  ```
  DayOfYear::DayOfYear(int monthValue, int dayValue)
  {
        month = monthValue;
        day = dayValue;
  }
  ```

- Note same name around ::
  - Clearly identifies a constructor

- Note no return type
  - Just as in class definition

Alternative Definition

- Previous definition equivalent to:

  ```
  DayOfYear::DayOfYear(                int monthValue,
                                       int dayValue)
          : month(monthValue), day(dayValue)  ←
  {...}
  ```

- Third line called "Initialization Section"

- Body left empty

- Preferable definition version

Constructor Additional Purpose

- Not just initialize data

- Body doesn't have to be empty
  - In initializer version

- Validate the data!
  - Ensure only appropriate data is assigned to class private member variables
  - Powerful OOP principle

Overloaded Constructors

- Can overload constructors just like other functions

- Recall: a signature consists of:
  - Name of function
  - Parameter list

- Provide constructors for all possible argument-lists
  - Particularly "how many"

Class with Constructors Example:
Display 7.1 Class with Constructors (1 of 3)

Display 7.1   **Class with Constructors**

```
1    #include <iostream>
2    #include <cstdlib> //for exit
3    using namespace std;

4    class DayOfYear
5    {
6    public:
7        DayOfYear(int monthValue, int dayValue);
8        //Initializes the month and day to arguments.

9        DayOfYear(int monthValue);
10       //Initializes the date to the first of the given month.

11       DayOfYear( );
12       //Initializes the date to January 1.

13       void input( );
14       void output( );
15       int getMonthNumber( );
16       //Returns 1 for January, 2 for February, etc.
```

This definition of **DayOfYear** *is an improved version of the class* **DayOfYear** *given in Display 6.4.*

default constructor

Class with Constructors Example:
Display 7.1 Class with Constructors (2 of 3)

```
17          int getDay( );
18      private:
19          int month;
20          int day;
21          void testDate( );
22      };
23
24      int main( )
25      {
26          DayOfYear date1(2, 21), date2(5), date3;
27          cout << "Initialized dates:\n";
28          date1.output( ); cout << endl;
29          date2.output( ); cout << endl;
30          date3.output( ); cout << endl;
31
32          date1 = DayOfYear(10, 31);
33          cout << "date1 reset to the following:\n";
34          date1.output( ); cout << endl;
35          return 0;
36      }
37
38      DayOfYear::DayOfYear(int monthValue, int dayValue)
39                          : month(monthValue), day(dayValue)
40      {
            testDate( );
        }
```

This causes a call to the default constructor. Notice that there are no parentheses.

an explicit call to the constructor DayOfYear::DayOfYear

Class with Constructors Example:
Display 7.1 Class with Constructors (3 of 3)

Display 7.1 Class with Constructors

```
41   DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
42   {
43       testDate( );
44   }

45   DayOfYear::DayOfYear( ) : month(1), day(1)
46   {/*Body intentionally empty.*/}

47   //uses iostream and cstdlib:
48   void DayOfYear::testDate( )
49   {
50       if ((month < 1) || (month > 12))
51       {
52           cout << "Illegal month value!\n";
53           exit(1);
54       }
55       if ((day < 1) || (day > 31))
56       {
57           cout << "Illegal day value!\n";
58           exit(1);
59       }
60   }
```

<Definitions of the other member functions are the same as in Display 6.4.>

SAMPLE DIALOGUE

Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31

Constructor with No Arguments

- Can be confusing

- Standard functions with no arguments:
  - Called with syntax: callMyFunction();
    - Including empty parentheses

- Object declarations with no "initializers":
  - DayOfYear date1;   // This way!
  - DayOfYear date();   // NO!
    - What is this really?
    - Compiler sees a function declaration/prototype!
    - Yes! Look closely!

Explicit Constructor Calls

- Can also call constructor AGAIN
  - After object declared
    - Recall: constructor was automatically called then
  - Can call via object's name; standard member function call

- Convenient method of setting member variables

- Method quite different from standard member function call

Explicit Constructor Call Example

- Such a call returns "anonymous object"

  - Which can then be assigned

  - **<u>In Action</u>**:
    DayOfYear holiday(7, 4);

    - Constructor called at object's declaration
    - Now to "re-initialize":
      holiday = DayOfYear(5, 5);
      - Explicit constructor call
      - Returns new "anonymous object"
      - Assigned back to current object

Default Constructor

- Defined as: constructor w/ no arguments

- One should always be defined

- Auto-Generated?
  - Yes & No
  - If no constructors AT ALL are defined → Yes
  - If any constructors are defined → No

- If no default constructor:
  - Cannot declare: MyClass myObject;
    - With no initializers

Class Type Member Variables

- Class member variables can be any type

    – Including objects of other classes!

    – Type of class relationship

        • Powerful OOP principle

- Need special notation for constructors

    – So they can call "back" to member
      object's constructor

Class Member Variable Example:
Display 7.3 A Class Member Variable (1 of 5)

Display 7.3   A Class Member Variable

```
1    #include <iostream>
2    #include<cstdlib>
3    using namespace std;

4    class DayOfYear
5    {
6    public:
7        DayOfYear(int monthValue, int dayValue);
8        DayOfYear(int monthValue);
9        DayOfYear( );
10       void input( );
11       void output( );
12       int getMonthNumber( );
13       int getDay( );
14   private:
15       int month;
16       int day;
17       void testDate( );
18   };
```

The class DayOfYear is the same as in Display 7.1, but we have repeated all the details you need for this discussion.

Class Member Variable Example:
Display 7.3 A Class Member Variable (2 of 5)

```
19   class Holiday
20   {
21   public:
22       Holiday( );//Initializes to January 1 with no parking enforcement
23       Holiday(int month, int day, bool theEnforcement);
24       void output( );
25   private:
26       DayOfYear date;
27       bool parkingEnforcement;//true if enforced
28   };

29   int main( )
30   {
31       Holiday h(2, 14, true);
32       cout << "Testing the class Holiday.\n";
33       h.output( );

34       return 0;
35   }
36
37   Holiday::Holiday( ) : date(1, 1), parkingEnforcement(false)
38   {/*Intentionally empty*/}

39   Holiday::Holiday(int month, int day, bool theEnforcement)
40                       : date(month, day), parkingEnforcement(theEnforcement)
41   {/*Intentionally empty*/}
```

member variable of a class type

*Invocations of constructors from the class **DayOfYear**.*

(continued)

Display 7.3   A Class Member Variable

```
42   void Holiday::output( )
43   {
44       date.output( );
45       cout << endl;
46       if (parkingEnforcement)
47           cout << "Parking laws will be enforced.\n";
48       else
49           cout << "Parking laws will not be enforced.\n";
50   }

51   DayOfYear::DayOfYear(int monthValue, int dayValue)
52                              : month(monthValue), day(dayValue)
53   {
54       testDate( );
55   }
```

```
56   //uses iostream and cstdlib:
57   void DayOfYear::testDate( )
58   {
59       if ((month < 1) || (month > 12))
60       {
61           cout << "Illegal month value!\n";
62           exit(1);
63       }
64       if ((day < 1) || (day > 31))
65       {
66           cout << "Illegal day value!\n";
67           exit(1);
68       }
69   }
70
71   //Uses iostream:
72   void DayOfYear::output( )
73   {
74       switch (month)
75       {
76           case 1:
77               cout << "January "; break;
78           case 2:
79               cout << "February "; break;
80           case 3:
81               cout << "March "; break;
                   .
                   .
                   .
```

The omitted lines are in Display 6.3, but they are obvious enough that you should not have to look there.

Display 7.3 **A Class Member Variable**

```
82            case 11:
83                cout << "November "; break;
84            case 12:
85                cout << "December "; break;
86            default:
87                cout << "Error in DayOfYear::output. Contact software vendor.";
88        }
89      cout << day;
90  }
```

SAMPLE DIALOGUE

Testing the class Holiday.
February 14
Parking laws will be enforced.

Parameter Passing Methods

- Efficiency of parameter passing
  - Call-by-value
    - Requires copy be made → Overhead
  - Call-by-reference
    - Placeholder for actual argument
    - Most efficient method
  - Negligible difference for simple types
  - For class types → clear advantage

- Call-by-reference desirable
  - Especially for "large" data, like class types

The const Parameter Modifier

- Large data types (typically classes)
  - Desirable to use pass-by-reference
  - Even if function will not make modifications

- Protect argument
  - Use constant parameter
    - Also called constant call-by-reference parameter
  - Place keyword *const* before type
  - Makes parameter "read-only"
  - Attempt to modify parameter results in compiler error

Use of const

- All-or-nothing

- If no need for function modifications
  - Protect parameter with const
  - Protect ALL such parameters

- This includes class member function parameters

Inline Functions

- For non-member functions:
  - Use keyword *inline* in function declaration and function heading

- For class member functions:
  - Place implementation (code) for function IN class definition → automatically inline

- Use for very short functions only

- Code actually inserted in place of call
  - Eliminates overhead
  - More efficient, but only when short!

Inline Member Functions

- Member function definitions
  - Typically defined separately, in different file
  - Can be defined IN class definition
    - Makes function "in-line"

- Again: use for very short functions only

- More efficient
  - If too long → actually less efficient!

Member Initializers

- C++11 supports a feature called member initialization
  - This feature allows you to set default values for member variables

```cpp
class Coordinate
{
  public:
        Coordinate();
  private:
        int x=1;
        int y=2;                    Member
};                                  Initializers
Coordinate::Coordinate()
{ }

Coordinate c1;        Initializes c1.x to 1 and c1.y to 2
```

Constructor Delegation

- C++11 allows one constructor to invoke another

```
Coordinate::Coordinate(int xval, int yval) : x(xval), y(yval)
{ }
Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

- The default constructor invokes the constructor to initialize x and y to 99,99

Static Members

- Static member variables
  - All objects of class "share" one copy
  - One object changes it → all see change

- Useful for "tracking"
  - How often a member function is called
  - How many objects exist at given time

- Place keyword *static* before type

Static Functions

- Member functions can be static
  - If no access to object data needed
  - And still "must" be member of the class
  - Make it a static function

- Can then be called outside class
  - From non-class objects:
    - E.g., Server::getTurn();
  - As well as via class objects
    - Standard method: myObject.getTurn();

- Can only use static data, functions!

Static Members Example:
Display 7.6 Static Members (1 of 4)

Display 7.6   **Static Members**

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

3    class Server
4    {
5    public:
6        Server(char letterName);
7        static int getTurn( );
8        void serveOne( );
9        static bool stillOpen( );
10   private:
11       static int turn;
12       static int lastServed;
13       static bool nowOpen;
14       char name;
15   };

16   int Server:: turn = 0;
17   int Server:: lastServed = 0;
18   bool Server::nowOpen = true;
```

Static Members Example:
Display 7.6 Static Members (2 of 4)

```cpp
19  int main( )
20  {
21      Server s1('A'), s2('B');
22      int number, count;
23      do
24      {
25          cout << "How many in your group? ";
26          cin >> number;
27          cout << "Your turns are: ";
28          for (count = 0; count < number; count++)
29              cout << Server::getTurn( ) << ' ';
30          cout << endl;
31          s1.serveOne( );
32          s2.serveOne( );
33      } while (Server::stillOpen( ));

34      cout << "Now closing service.\n";

35      return 0;
36  }
37
38
```

Static Members Example:
Display 7.6 Static Members (3 of 4)

Display 7.6   **Static Members**

```
39    Server::Server(char letterName) : name(letterName)
40    {/*Intentionally empty*/}

41    int Server::getTurn( )
42    {
43        turn++;
44        return turn;
45    }
46    bool Server::stillOpen( )
47    {
48        return nowOpen;
49    }

50    void Server::serveOne( )
51    {
52        if (nowOpen && lastServed < turn)
53        {
54            lastServed++;
55            cout << "Server " << name
56                << " now serving " << lastServed << endl;
57        }
```

*Since **getTurn** is static, only static members can be referenced in here.*

Static Members Example:
Display 7.6 Static Members (4 of 4)

```
58        if (lastServed >= turn) //Everyone served
59            nowOpen = false;
60    }
```

SAMPLE DIALOGUE

How many in your group? **3**
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? **2**
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? **0**
Your turns are:
Server A now serving 5
Now closing service.

Vectors

- **Vector Introduction**

  – Recall: arrays are fixed size

  – Vectors: "arrays that grow and shrink"

    • During program execution

  – Formed from Standard Template Library (STL)

    • Using template class

Vector Basics

- Similar to array:
  - Has base type
  - Stores collection of base type values

- Declared differently:
  - Syntax: vector<Base_Type>
    - Indicates template class
    - Any type can be "plugged in" to Base_Type
    - Produces "new" class for vectors with that type
  - Example declaration:
    vector<int> v;

7-40

Vector Use

- vector<int> v;
  - "v is vector of type int"
  - Calls class default constructor
    - Empty vector object created

- Indexed like arrays for access

- But to add elements:
  - Must call member function push_back

- Member function size()
  - Returns current number of elements

Vector Example:
Display 7.7 Using a Vector (1 of 2)

Display 7.7   **Using a Vector**

```cpp
1    #include <iostream>
2    #include <vector>
3    using namespace std;

4    int main( )
5    {
6        vector<int> v;
7        cout << "Enter a list of positive numbers.\n"
8             << "Place a negative number at the end.\n";

9        int next;
10       cin >> next;
11       while (next > 0)
12       {
13           v.push_back(next);
14           cout << next << " added. ";
15           cout << "v.size( ) = " << v.size( ) << endl;
16           cin >> next;
17       }
```

Vector Example:
Display 7.7 Using a Vector (2 of 2)

```
18          cout << "You entered:\n";
19          for (unsigned int i = 0; i < v.size( ); i++)
20              cout << v[i] << " ";
21          cout << endl;

22          return 0;
23      }
```

SAMPLE DIALOGUE

Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 −1
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8

Vector Efficiency

- Member function capacity()
  - Returns memory currently allocated
  - Not same as size()
  - Capacity typically > size
    - Automatically increased as needed

- If efficiency critical:
  - Can set behaviors manually
    - v.reserve(32); //sets capacity to 32
    - v.reserve(v.size()+10); //sets capacity to 10 more than size

Summary 1

- Constructors: automatic initialization of class data
  - Called when objects are declared
  - Constructor has same name as class

- Default constructor has no parameters
  - Should always be defined

- Class member variables
  - Can be objects of other classes
    - Require initialization-section

Summary 2

- Constant call-by-reference parameters
  - More efficient than call-by-value

- Can *inline* very short function definitions
  - Can improve efficiency

- Static member variables
  - Shared by all objects of a class

- Vector classes
  - Like: "arrays that grow and shrink"