Lecture 4

Constructors and Other Tools
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

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

class Line {
public:
  void setLength(double len);
  double getLength();
private:
  double length;
};

Line::Line()
{
  cout << "Object is being created" << endl;
}

void Line::setLength(double len)
{
  length = len;
}

double Line::getLength()
{
  return length;
}

int main()
{
  Line line;
  line.setLength(6.0);
  cout << "Length of line: " << line.GetLength() << endl;
}
```
#include <iostream>
using namespace std;

class Line
{
public:
    void setLength(double len);
    double getLength();
    Line();
    Line(double len);
private:
    double length;
};

Line::Line()
{
    cout<<"Object is being created"<<endl;
}

Line::Line(double len)
{
    cout<<"Object is being created, length = " << len <<endl;
    length = len;
}

void Line::setLength(double len)
{
    length=len;
}

double Line::getLength()
{
    return length;
}

int main()
{
    Line line;
    Line line1(10.0);
    line.setLength(6.0);
    cout<<"Length of line: "<< line.getLength() << endl;
}
Constructor Equivalency

• Consider:
  - Line line1, line2
    line1.Line(10.0); // ILLEGAL!
    line2.Line(5.0); // ILLEGAL!

• Seemingly OK...
  - CANNOT call constructors like other member functions!
Constructor Code

• Constructor definition is like all other member functions:
  Line::Line(double len)
  {
    length=len;
  }

• Note same name around ::
  – Clearly identifies a constructor

• Note no return type
  – Just as in class definition
Explicit Constructor Calls

- Constructor can be called again after an object has been declared

```cpp
int main()
{
    Line line;
    Line line1(10.0);
    Line line2(5.0);
    line.setLength(6.0);
    cout<<"Length of line: "<< line.getLength() << endl;
    line1=Line(3.0); // explicit call to the constructor Line::Line
}
```
Alternative Definition

```cpp
Line::Line(double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

Line::Line(double len):length(len)
{
    cout << "Object is being created, length = " << len << endl;
}
```
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"
This definition of DayOfYear is an improved version of the class DayOfYear given in Display 6.4.

```cpp
#include <iostream>
#include <cstdlib>  // for exit
using namespace std;

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

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

    DayOfYear();  // default constructor
    // Initializes the date to January 1.

    void input();
    void output();
    int getMonthNumber();
    // Returns 1 for January, 2 for February, etc.
};
```
Class with Constructors Example:

Display 7.1 Class with Constructors (2 of 3)

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

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)**

```cpp
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 }

SAMPLE DIALOGUE

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

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

• Such a call returns "anonymous object"
  – Which can then be assigned
  – **In Action:**
    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)

```cpp
class Holiday
{
public:
    Holiday(); //Initializes to January 1 with no parking enforcement
    Holiday(int month, int day, bool theEnforcement);
    void output();
private:
    DayOfYear date;
    bool parkingEnforcement; //true if enforced
};

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

    return 0;
}

Holiday::Holiday( ) : date(1, 1), parkingEnforcement(false)
{ /*Intentionally empty*/

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

(continued)
Class Member Variable Example:

**Display 7.3  A Class Member Variable (3 of 5)**

```cpp
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  }
```
Class Member Variable Example:

Display 7.3  A Class Member Variable (4 of 5)

```cpp
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;
82          .
83          .
84          .
85          .
86          .
87          .
88          .
89          .
90          .
91      }
```

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

**Display 7.3** A Class Member Variable (5 of 5)

```cpp
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 $\rightarrow$ Overhead
  – Call-by-reference
    • Placeholder for actual argument
    • Most efficient method
    – Negligible difference for simple types
    – For class types $\rightarrow$ 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
Example

What are the differences and similarities between a call-by-value parameter and a constant call-by-reference parameter?

```c
void callByValue(int x);
void callByConstReference(const int& x);
```
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

```cpp
double Line::getLength() const
{
    return length;
}
bool isLarger(const Line& line1, const Line& line2)
{
    return (line1.getLength() < line2.getLength());
}
int main()
{
    Line line;
    Line line1(10.0);
    Line line2(5.0);
    line.setLength(6.0);
    cout << "Length of line: " << line.getLength() << endl;
    line1 = Line(3.0);
    if (isLarger(line1, line2))
        cout << "line1 is larger.\n";
    else
        cout << "line2 is larger.\n";
}
```
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!

```cpp
double Line::getLength() const
{
    return length;
}

class Line
{
    public:
        void setLength(double len);
        double getLength() const {
            return length;
        }
    private:
        double length;

    bool isLarger(const Line& line1, const Line& line2); Line(); 
    Line(double len);
};

inline void Line::setLength(double len)
{
    length = len;
}
```
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!
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 Data Member Example

```cpp
class Line
{
    public:
    static int objectCount;
    void setLength(double len);
    double getLength() const {
        return length;
    }
    bool isLarger(const Line& line1, const Line& line2);
    Line();
    Line(double len);

    private:
    double length;
};

// initialize static member of class Line
int Line::objectCount = 0;

int main()
{
    Line line;
    Line line1(10.0);
    Line line2(5.0);
    line1 = Line(3.0);
    cout << "Total objects:" << Line::objectCount << endl;
}
```
Static Function Member Example

class Line
{
   public:
   static int objectCount;
   static int getCount();
   void setLength(double len);
   double getLength() const {
      return length;
   }
   bool isLarger(const Line& line1, const Line& line2);
   Line();
   Line(double len);
   private:
      double length;
};

//initialize static member of class Line
int Line::objectCount = 0;

int Line::getCount()
{
   return objectCount;
}

int main()
{
   Line line;
   Line line1(10.0);
   Line line2(5.0);
   line1 = Line(3.0);
   cout << "Total objects: " << Line::getCount() << endl;
}
Static member

- can be initialize before an object is being created

```cpp
int main()
{
    cout<<"Before initialize objects:" <<Line::objectCount<<endl;
    cout<<"Before initialize objects:" <<Line::getCount()<<endl;
    Line line;
    Line line1(10.0);
    Line line2(5.0);
    line1=Line(3.0);
    cout<<"Total objects:" <<Line::getCount()<<endl;
}
```
#include <iostream>
using namespace std;

class Server {
public:
    Server(char letterName);
    static int getTurn();
    void serveOne();
    static bool stillOpen();
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server::turn = 0;
int Server::lastServed = 0;
bool Server::nowOpen = true;
Static Members Example:

Display 7.6  Static Members (2 of 4)

```cpp
int main( )
{
    Server s1('A'), s2('B');
    int number, count;
    do
    {
        cout << "How many in your group? ";
        cin >> number;
        cout << "Your turns are: ";
        for (count = 0; count < number; count++)
            cout << Server::getTurn( ) << ' ';
        cout << endl;
        s1.serveOne( );
        s2.serveOne( );
    } while (Server::stillOpen( ));

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

    return 0;
}
```
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.
Could the function defined as follows be added to the class Server in Display 7.6 as a static function? Explain your answer.

```cpp
void Server::showStatus()
{
    cout<<“Currently serving “<<turn<<endl;
    cout<<“server name”<<name<<endl;
}
```
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;
Vector Use

• \texttt{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 \texttt{push\_back}

• Member function \texttt{size()}
  – Returns current number of elements
Vector Example:

Display 7.7  Using a Vector (1 of 2)

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

int main() {
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
    << "Place a negative number at the end.\n";
    int next;
    cin >> next;
    while (next > 0) {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size() = " << v.size() << endl;
        cin >> next;
    }
```

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"
Quiz

Write a class named Ball. Ball should have two private member variables with default values: m_color ("Black") and m_radius (10.0). Ball should provide constructors to set only m_color, set only m_radius, and set both (define at outside of class). For this quiz question, do not use default parameters for your constructors. Also write a member function to print out the color and radius of the ball.
Main()

```c
int main()
{
    Ball def;
    def.print();

    Ball blue("blue");
    blue.print();

    Ball twenty(20.0);
    twenty.print();

    Ball blueTwenty("blue", 20.0);
    blueTwenty.print();

    return 0;
}
```