CSCI 1061U
Programming Workshop 2

Structures and Classes
Learning Objectives

• Structures
  • Structure types
  • Structures as function arguments
  • Initializing structures

• Classes
  • Defining, member functions
  • Public and private members
  • Accessor and mutator functions
  • Structures vs. classes
Structures

- 2\textsuperscript{nd} aggregate data type: struct

- Recall: aggregate meaning "grouping"
  - Recall array: collection of values of same type
  - Structure: collection of values of different types

- Treated as a single item, like arrays

- Major difference: Must first "define" struct
  - Prior to declaring any variables
Structure Types

• Define struct globally (typically)

• No memory is allocated
  • Just a "placeholder" for what our struct will "look like"

• Definition:
  
  ```
  struct CDAccountV1  // Name of new struct "type"
  {
    double balance;  // member names
    double interestRate;
    int term;
  };
  ```
Declare Structure Variable

• With structure type defined, now declare variables of this new type:
  CDAccountV1 account;
  • Just like declaring simple types
  • Variable account now of type CDAccountV1
  • It contains "member values"
    • Each of the struct "parts"
Accessing Structure Members

• Dot Operator to access members
  • account.balance
  • account.interestRate
  • account.term

• Called "member variables"
  • The "parts" of the structure variable
  • Different structs can have same name member variables
    • No conflicts
Structure Example:

Display 6.1  A Structure Definition (1 of 3)

```cpp
1   //Program to demonstrate the CDAccountV1 structure type.
2   #include <iostream>
3   using namespace std;

4   //Structure for a bank certificate of deposit:
5   struct CDAccountV1
6       {
7       double balance;
8       double interestRate;
9       int term; //months until maturity
10      
11      void getData(CDAccountV1& theAccount);
12      //Postcondition: theAccount.balance, theAccount.interestRate, and
13      //theAccount.term have been given values that the user entered at the keyboard.
```

An improved version of this structure will be given later in this chapter.
Structure Example:

Display 6.1  A Structure Definition (2 of 3)

```c++
14  int main()
15  {
16      CDAccountV1 account;
17      getData(account);
18      double rateFraction, interest;
19      rateFraction = account.interestRate/100.0;
20      interest = account.balance*(rateFraction*(account.term/12.0));
21      account.balance = account.balance + interest;
22      cout.setf(ios::fixed);
23      cout.setf(ios::showpoint);
24      cout.precision(2);
25      cout << "When your CD matures in ">
26          << account.term << " months,
27          << "it will have a balance of $" 
28          << account.balance << endl;
29      return 0;
30  }
```

(continued)
Structure Example:
Display 6.1  A Structure Definition (3 of 3)

```cpp
//Uses iostream:
void getData(CDAccountV1& theAccount)
{
    cout << "Enter account balance: ";
    cin >> theAccount.balance;
    cout << "Enter account interest rate: ";
    cin >> theAccount.interestRate;
    cout << "Enter the number of months until maturity: ";
    cin >> theAccount.term;
}

SAMPLE DIALOGUE

Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity: 6
When your CD matures in 6 months, it will have a balance of $105.00
```
Structure Pitfall

• Semicolon after structure definition
  • ; MUST exist:
    struct WeatherData
    {
      double temperature;
      double windVelocity;
    }; ← REQUIRED semicolon!
  • Required since you "can" declare structure variables in this location
Structure Assignments

• Given structure named CropYield

• Declare two structure variables: CropYield apples, oranges;
  • Both are variables of "struct type CropYield"
  • Simple assignments are legal:
    apples = oranges;
    • Simply copies each member variable from apples into member variables from oranges
Structures as Function Arguments

• Passed like any simple data type
  • Pass-by-value
  • Pass-by-reference
  • Or combination

• Can also be returned by function
  • Return-type is structure type
  • Return statement in function definition sends structure variable back to caller
Initializing Structures

• Can initialize at declaration
  • Example:
    ```
    struct Date {
      int month;
      int day;
      int year;
    }
    Date dueDate = {12, 31, 2003};
    ```
  • Declaration provides initial data to all three member variables
Classes

• Similar to structures
  • Adds member FUNCTIONS
  • Not just member data

• Integral to object-oriented programming
  • Focus on objects
    • Object: Contains data and operations
    • In C++, variables of class type are objects
Class Definitions

• Defined similar to structures

• Example:

```cpp
class DayOfYear // name of new class type
{
   public:
      void output(); // member function!
      int month;    // member variable
      int day;      // member variable

};
```

• Notice only member function’s prototype

• Function’s implementation is elsewhere
Declaring Objects

• Declared same as all variables
  • Predefined types, structure types

• Example:
  \[ \text{DayOfYear today, birthday;} \]
  • Declares two objects of class type DayOfYear

• Objects include:
  • Data
    • Members month, day
  • Operations (member functions)
    • output()
Class Member Access

• Members accessed same as structures

• Example:
  
  ```
  today.month
  today.day
  ```

  • And to access member function:
    ```
    today.output(); ← Invokes member function
    ```
Class Member Functions

• Must define or "implement" class member functions

• Like other function definitions
  • Can be after main() definition
  • Must specify class:
    ```
    void DayOfYear::output()
    {...
    ```
    • :: is scope resolution operator
    • Instructs compiler "what class" member is from
    • Item before :: called type qualifier
Class Member Functions Definition

- Notice output() member function’s definition (in next example)
- Refers to member data of class
  - No qualifiers
- Function used for all objects of the class
  - Will refer to "that object’s" data when invoked
  - Example:
    today.output();
      - Displays "today" object’s data
Complete Class Example:

Display 6.3  Class With a Member Function (1 of 4)

```cpp
// Program to demonstrate a very simple example of a class.
// A better version of the class DayOfYear will be given in Display 6.4.
#include <iostream>
using namespace std;

class DayOfYear {
public:
    void output();
    int month;
    int day;
};

int main()
{
    DayOfYear today, birthday;
    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;
```

Member function declaration

Normally, member variables are private and not public, as in this example. This is discussed a bit later in this chapter.
## Complete Class Example:

### Display 6.3  Class With a Member Function (2 of 4)

```cpp
25     cout << "Today's date is ";
26     today.output();
27     cout << endl;
28     cout << "Your birthday is ";
29     birthday.output();
30     cout << endl;
31     if (today.month == birthday.month && today.day == birthday.day)
32         cout << "Happy Birthday!\n";
33     else
34         cout << "Happy Unbirthday!\n";
35     return 0;
36 }
37 //Uses iostream:
38 void DayOfYear::output( )
39 {
40     switch (month)
41     {
42         case 1:
43             cout << "January "; break;
44         case 2:
45             cout << "February "; break;
46         case 3:
47             cout << "March "; break;
48         case 4:
49             cout << "April "; break;
```

Calls to the member function `output`

Member function definition
Complete Class Example:

**Display 6.3** Class With a Member Function (3 of 4)

```cpp
    case 5:
      cout << "May "; break;
    case 6:
      cout << "June "; break;
    case 7:
      cout << "July "; break;
    case 8:
      cout << "August "; break;
    case 9:
      cout << "September "; break;
    case 10:
      cout << "October "; break;
    case 11:
      cout << "November "; break;
    case 12:
      cout << "December "; break;
    default:
      cout << "Error in DayOfYear::output. Contact software vendor.";
    }
  }
  cout << day;
  }
```
Complete Class Example:

Display 6.3 Class With a Member Function (4 of 4)

**Display 6.3 Class with a Member Function**

**SAMPLE DIALOGUE**

Enter today’s date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today’s date is October 15
Your birthday is February 21
Happy Unbirthday!
Dot and Scope Resolution Operator

• Used to specify "of what thing" they are members

• Dot operator:
  • Specifies member of particular object

• Scope resolution operator:
  • Specifies what class the function definition comes from
A Class’s Place

• Class is full-fledged type!
  • Just like data types int, double, etc.

• Can have variables of a class type
  • We simply call them "objects"

• Can have parameters of a class type
  • Pass-by-value
  • Pass-by-reference

• Can use class type like any other type!
Encapsulation

• Any data type includes
  • Data (range of data)
  • Operations (that can be performed on data)

• Example:
  int data type has:
  Data: -2147483648 to 2147483647 (for 32 bit int)
  Operations: +,-,*,/,%,,logical,etc.

• Same with classes
  • But WE specify data, and the operations to be allowed on our data!
Abstract Data Types

• "Abstract"
  • Programmers don’t know details

• Abbreviated "ADT"
  • Collection of data values together with set of basic operations defined for the values

• ADT’s often "language-independent"
  • We implement ADT’s in C++ with classes
    • C++ class "defines" the ADT
  • Other languages implement ADT’s as well
More Encapsulation

• Encapsulation
  • Means "bringing together as one"

• Declare a class → get an object

• Object is "encapsulation" of
  • Data values
  • Operations on the data (member functions)
Principles of OOP

• Information Hiding
  • Details of how operations work not known to "user" of class

• Data Abstraction
  • Details of how data is manipulated within ADT/class not known to user

• Encapsulation
  • Bring together data and operations, but keep "details" hidden
Public and Private Members

• Data in class almost always designated private in definition!
  • Upholds principles of OOP
  • Hide data from user
  • Allow manipulation only via operations
    • Which are member functions

• Public items (usually member functions) are "user-accessible"
Public and Private Example

• Modify previous example:
  ```cpp
class DayOfYear
{
public:
  void input();
  void output();
private:
  int month;
  int day;
};
• Data now private
• Objects have no direct access
Public and Private Example 2

• Given previous example

• Declare object:
  
  ```cpp
  DayOfYear today;
  ```

• Object `today` can ONLY access public members
  
  • `cin >> today.month;  // NOT ALLOWED!`
  
  • `cout << today.day;    // NOT ALLOWED!`
  
  • Must instead call public operations:
    
    • `today.input();`
    
    • `today.output();`
Public and Private Style

• Can mix & match public & private

• More typically place public first
  • Allows easy viewing of portions that can be
    USED by programmers using the class
  • Private data is "hidden", so irrelevant to users

• Outside of class definition, cannot change
  (or even access) private data
Accessor and Mutator Functions

• Object needs to "do something" with its data

• Call accessor member functions
  • Allow object to read data
  • Also called "get member functions"
  • Simple retrieval of member data

• Mutator member functions
  • Allow object to change data
  • Manipulated based on application
Separate Interface and Implementation

- User of class need not see details of how class is implemented
  - Principle of OOP \( \rightarrow \) encapsulation

- User only needs "rules"
  - Called "interface" for the class
    - In C++ \( \rightarrow \) public member functions and associated comments

- Implementation of class hidden
  - Member function definitions elsewhere
  - User need not see them
Structures versus Classes

• Structures
  • Typically all members public
  • No member functions

• Classes
  • Typically all data members private
  • Interface member functions public

• Technically, same
  • Perceptionally, very different mechanisms
Thinking Objects

• Focus for programming changes
  • Before → algorithms center stage
  • OOP → data is focus

• Algorithms still exist
  • They simply focus on their data
  • Are "made" to "fit" the data

• Designing software solution
  • Define variety of objects and how they interact
Summary 1

• Structure is collection of different types

• Class used to combine data and functions into single unit -> object

• Member variables and member functions
  • Can be public → accessed outside class
  • Can be private → accessed only in a member function’s definition

• Class and structure types can be formal parameters to functions
Summary 2

• C++ class definition
  • Should separate two key parts
    • Interface: what user needs
    • Implementation: details of how class works

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
CSCI 1061U
Programming Workshop 2

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

• Class definition with constructor:
  • `class DayOfYear
  {
    public:
      DayOfYear(int monthValue, int dayValue);
      //Constructor initializes month and day
      void input();
      void output();
  ...```
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:

```cpp
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"
#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 testData();
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        testData();
40    }
```

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

an explicit call to the constructor `DayOfYear::DayOfYear`

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
Class with Constructors Example:

Display 7.1  Class with Constructors (3 of 3)

```c++
41  DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
42  {  
43    testDate();
44  }
45  
46  DayOfYear::DayOfYear( ) : month(1), day(1)
47  { /*Body intentionally empty. */
48    //uses iostream and cstdlib:
49    void DayOfYear::testDate( )
50    {
51      if ((month < 1) || (month > 12))
52      { 
53        cout << "Illegal month value!\n";
54        exit(1);
55      }
56      if ((day < 1) || (day > 31))
57      {  
58        cout << "Illegal day value!\n";
59        exit(1);
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

• **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)

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

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    DayOfYear(int monthValue);
    DayOfYear();
    void input( );
    void output( );
    int getMonthNumber( );
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};
```

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
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
30 int main()
31 {
32     Holiday h(2, 14, true);
33     cout << "Testing the class Holiday.\n";
34     h.output();
35     return 0;
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/>}
```

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
Class Member Variable Example:

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

```cpp
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
52  DayOfYear::DayOfYear(int monthValue, int dayValue)
53  : month(monthValue), day(dayValue)
54  {
55      testDate( );
56  }
```
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          {cout << "Illegal month value!\n";
61            exit(1);}
62      if ((day < 1) || (day > 31))
63          {cout << "Illegal day value!\n";
64            exit(1);}
65  }
66  //Uses iostream:
67  void DayOfYear::output( )
68  {
69      switch (month)
70          {
71        case 1:
72          cout << "January "; break;
73        case 2:
74          cout << "February "; break;
75        case 3:
76          cout << "March "; break;
77        .
78        .
79        .
80      }
```

*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)

```
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
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 \(\rightarrow\) automatically inline

• Use for very short functions only

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

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
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;
};
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)

```cpp
#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)

```cpp
Server::Server(char letterName) : name(letterName)
{"Intentionally empty"}/

int Server::getTurn()
{
    turn++;  
    return turn;
}

bool Server::stillOpen()
{
    return nowOpen;
}

void Server::serveOne()
{
    if (nowOpen && lastServed < turn)
    {
        lastServed++;  
        cout << "Server " << name 
        << " now serving " << lastServed << endl;
    }

Since getTurn is static, only static members can be referenced in here.
```
Static Members Example:

Display 7.6 Static Members (4 of 4)

```java
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;
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)

```cpp
#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)

```cpp
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"
CSCI 1061U
Programming Workshop 2

Operator Overloading, Friends and References
Learning Objectives

• Basic Operator Overloading
  • Unary operators
  • As member functions

• Friends and Automatic Type Conversion
  • Friend functions, friend classes
  • Constructors for automatic type conversion

• References and More Overloading
  • << and >>
  • Operators: =, [], ++, --
Operator Overloading Introduction

- Operators +, -, %, ==, etc.
  - Really just functions!

- Simply "called" with different syntax:
  \( x + 7 \)
  - "+" is binary operator with \( x \) & 7 as operands
  - We "like" this notation as humans

- Think of it as:
  \( +(x, 7) \)
  - "+" is the function name
  - \( x, 7 \) are the arguments
  - Function "+" returns "sum" of its arguments
Operator Overloading Perspective

• Built-in operators
  • e.g., +, -, =, %, ==, /, *
  • Already work for C++ built-in types
  • In standard "binary" notation

• We can overload them!
  • To work with OUR types!
  • To add "Chair types", or "Money types"
    • As appropriate for our needs
    • In "notation" we’re comfortable with

• Always overload with similar "actions"!
Overloading Basics

• Overloading operators
  • VERY similar to overloading functions
  • Operator itself is "name" of function

• Example Declaration:
  
  ```cpp
  const Money operator +(  
      const Money& amount1,  
      const Money& amount2);  
  ```

  • Overloads + for operands of type Money
  • Uses constant reference parameters for efficiency
  • Returned value is type Money
    • Allows addition of "Money" objects
Overloaded "+"

• Given previous example:
  • Note: overloaded "+" NOT member function
  • Definition is "more involved" than simple "add"
    • Requires issues of money type addition
    • Must handle negative/positive values

• Operator overload definitions generally very simple
  • Just perform "addition" particular to "your" type
Money "+" Definition:

Display 8.1  Operator Overloading

• Definition of "+" operator for Money class:

```cpp
const Money operator +(const Money& amount1, const Money& amount2) {
    int allCents1 = amount1.getCents() + amount1.getDollars() * 100;
    int allCents2 = amount2.getCents() + amount2.getDollars() * 100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money can be negative.
    int finalDollars = absAllCents / 100;
    int finalCents = absAllCents % 100;

    if (sumAllCents < 0) {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }

    return Money(finalDollars, finalCents);
}
```
Overloaded "=="

• Equality operator, ==
  • Enables comparison of Money objects
  • Declaration:
    bool operator == (const Money& amount1,
                     const Money& amount2);
    • Returns bool type for true/false equality
  • Again, it’s a non-member function
    (like "+" overload)
Overloaded "==" for Money:

Display 8.1  Operator Overloading

• Definition of "==" operator for Money class:

```cpp
bool operator == (const Money& amount1, const Money& amount2) {
    return (amount1.getDollars() == amount2.getDollars())
    && (amount1.getCents() == amount2.getCents());
}
```
Constructors Returning Objects

- Constructor a "void" function?
  - We "think" that way, but no
  - A "special" function
    - With special properties
    - CAN return a value!

- Recall return statement in "+" overload for Money type:
  - return Money(finalDollars, finalCents);
    - Returns an "invocation" of Money class!
    - So constructor actually "returns" an object!
    - Called an "anonymous object"
Returning by const Value

• Consider "+" operator overload again:
  const Money operator + (const Money& amount1, const Money& amount2);
  
  • Returns a "constant object"?
  • Why?

• Consider impact of returning "non-const" object to see...→
Returning by non-const Value

- Consider "no const" in declaration:
  Money operator +(
    const Money& amount1,
    const Money& amount2);

- Consider expression that calls:
  m1 + m2
  - Where m1 & m2 are Money objects
  - Object returned is Money object
  - We can "do things" with objects!
    - Like call member functions...
What to do with Non-const Object

• Can call member functions:
  • We could invoke member functions on object returned by expression m1+m2:
    • (m1+m2).output(); //Legal, right?
      • Not a problem: doesn’t change anything
    • (m1+m2).input(); //Legal!
      • PROBLEM! //Legal, but MODIFIES!
  • Allows modification of "anonymous" object!
  • Can’t allow that here!

• So we define the return object as const
Overloading Unary Operators

• C++ has unary operators:
  • Defined as taking one operand
  • e.g., - (negation)
    • x = -y; // Sets x equal to negative of y
  • Other unary operators:
    • ++, --

• Unary operators can also be overloaded
Overload "-" for Money

- Overloaded "-" function declaration
  - Placed outside class definition:
    const Money operator - (const Money& amount);
  - Notice: only one argument
    - Since only 1 operand (unary)

- "-" operator is overloaded twice!
  - For two operands/arguments (binary)
  - For one operand/argument (unary)
  - Definitions must exist for both
Overloaded "-" Definition

• Overloaded "-" function definition:
  ```
  const Money operator -(const Money& amount)
  {
      return Money(-amount.getDollars(),
                    -amount.getCents());
  }
  ```

• Applies "-" unary operator to built-in type
  • Operation is "known" for built-in types

• Returns anonymous object again
Overloaded "-" Usage

• Consider:
  Money amount1(10),
  amount2(6),
  amount3;
  amount3 = amount1 – amount2;
  • Calls binary "-" overload

  amount3.output();      //Displays $4.00
  amount3 = -amount1;
  • Calls unary "-" overload

  amount3.output()       //Displays -$10.00
Overloading as Member Functions

• Previous examples: standalone functions
  • Defined outside a class

• Can overload as "member operator"
  • Considered "member function" like others

• When operator is member function:
  • Only ONE parameter, not two!
  • Calling object serves as 1st parameter
Member Operator in Action

- Money cost(1, 50), tax(0, 15), total;
  total = cost + tax;
  - If "+" overloaded as member operator:
    - Variable/object cost is calling object
    - Object tax is single argument
    - Think of as: total = cost.+(tax);

- Declaration of "+" in class definition:
  - const Money operator +(const Money& amount);
  - Notice only ONE argument
const Functions

• When to make function const?
  • Constant functions not allowed to alter class member data
  • Constant objects can ONLY call constant member functions

• Good style dictates:
  • Any member function that will NOT modify data should be made const

• Use keyword const after function declaration and heading
Overloading Operators: Which Method?

• Object-Oriented-Programming
  • Principles suggest member operators
  • Many agree, to maintain "spirit" of OOP

• Member operators more efficient
  • No need to call accessor & mutator functions

• At least one significant disadvantage
  • (Later in chapter...)
Overloading Function Application ()

• Function call operator, ( )
  • Must be overloaded as member function
  • Allows use of class object like a function
  • Can overload for all possible numbers of arguments

• Example:
  Aclass anObject;
anObject(42);
  • If ( ) overloaded \(\rightarrow\) calls overload
Other Overloads

• &&, ||, and comma operator
  • Predefined versions work for bool types
  • Recall: use "short-circuit evaluation"
  • When overloaded no longer uses short-circuit
    • Uses "complete evaluation" instead
    • Contrary to expectations

• Generally should not overload these operators
Friend Functions

• Nonmember functions
  • Recall: operator overloads as nonmembers
    • They access data through accessor and mutator functions
    • Very inefficient (overhead of calls)

• Friends can directly access private class data
  • No overhead, more efficient

• So: best to make nonmember operator overloads friends!
Friend Functions

• Friend function of a class
  • Not a member function
  • Has direct access to private members
    • Just as member functions do

• Use keyword *friend* in front of function declaration
  • Specified IN class definition
  • But they’re NOT member functions!
Friend Function Uses

• Operator Overloads
  • Most common use of friends
  • Improves efficiency
  • Avoids need to call accessor/mutator member functions
  • Operator must have access anyway
    • Might as well give full access as friend

• Friends can be any function
Friend Function Purity

• Friends not pure?
  • "Spirit" of OOP dictates all operators and functions be member functions
  • Many believe friends violate basic OOP principles

• Advantageous?
  • For operators: very!
  • Allows automatic type conversion
  • Still encapsulates: friend is in class definition
  • Improves efficiency
Friend Classes

• Entire classes can be friends
  • Similar to function being friend to class
  • Example:
    class F is friend of class C
    • All class F member functions are friends of C
    • NOT reciprocated
    • Friendship granted, not taken

• Syntax: friend class F
  • Goes inside class definition of "authorizing" class
References

• Reference defined:
  • Name of a storage location
  • Similar to "pointer"

• Example of stand alone reference:
  • int robert;
    int& bob = robert;
    • *bob* is reference to storage location for *robert*
    • Changes made to *bob* will affect *robert*

• Confusing?
References Usage

• Seemingly dangerous

• Useful in several cases:

• Call-by-reference
  • Often used to implement this mechanism

• Returning a reference
  • Allows operator overload implementations to be written more naturally
  • Think of as returning an "alias" to a variable
Returning Reference

• Syntax:
  double& sampleFunction(double& variable);
  • double& and double are different
  • Must match in function declaration and heading

• Returned item must "have" a reference
  • Like a variable of that type
  • Cannot be expression like "x+5"
    • Has no place in memory to "refer to"
Returning Reference in Definition

• Example function definition:
  double& sampleFunction(double& variable) {
    return variable;
  }

• Trivial, useless example

• Shows concept only

• Major use:
  • Certain overloaded operators
Overloading >> and <<

• Enables input and output of our objects
  • Similar to other operator overloads
  • New subtleties

• Improves readability
  • Like all operator overloads do
  • Enables:
    cout << myObject;
    cin >> myObject;
  • Instead of need for:
    myObject.output(); ...

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
Overloading $\gg$

- Insertion operator, $\ll$
  - Used with cout
  - A binary operator

- Example:
  cout $\ll$ "Hello";
  - Operator is $\ll$
  - 1\textsuperscript{st} operand is predefined object cout
    - From library iostream
  - 2\textsuperscript{nd} operand is literal string "Hello"
Overloading >>

• Operands of >>
  • Cout object, of class type ostream
  • Our class type

• Recall Money class
  • Used member function output()
  • Nicer if we can use >> operator:
    Money amount(100);
    cout << "I have " << amount << endl;
    instead of:
    cout << "I have ";
    amount.output()
Overloaded >> Return Value

• Money amount(100);
  cout << amount;
  • << should return some value
  • To allow cascades:
    cout << "I have " << amount;
    (cout << "I have ") << amount;
    • Two are equivalent

• What to return?
  • cout object!
    • Returns its first argument type, ostream
Overloaded >> Example:

Display 8.5 Overloading << and >> (1 of 5)

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

//Class for amounts of money in U.S. currency
class Money
{
public:
    Money( );
    Money(double amount);
    Money(int theDollars, int theCents);
    Money(int theDollars);
    double getAmount( ) const;
    int getDollars( ) const;
    int getCents( ) const;
    friend const Money operator +(const Money& amount1, const Money& amount2);
    friend const Money operator -(const Money& amount1, const Money& amount2);
    friend bool operator ==(const Money& amount1, const Money& amount2);
    friend const Money operator *(const Money& amount);
    friend ostream& operator <<(ostream& outputStream, const Money& amount);
    friend istream& operator >> (istream& inputStream, Money& amount);
private:
    int dollars; //A negative amount is represented as negative dollars and
    int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
```
Overloaded >> Example:  
Display 8.5 Overloading << and >> (2 of 5)

```cpp
25     int dollarsPart(double amount) const;
26     int centsPart(double amount) const;
27     int round(double number) const;
28  
29  int main()
30  {
31       Money yourAmount, myAmount(10, 9);
32       cout << "Enter an amount of money: ";
33       cin >> yourAmount;
34       cout << "Your amount is " << yourAmount << endl;
35       cout << "My amount is " << myAmount << endl;
36
37       if (yourAmount == myAmount)
38           cout << "We have the same amounts.\n";
39       else
40           cout << "One of us is richer.\n";
41       Money ourAmount = yourAmount + myAmount;
```
Overloaded >> Example:

**Display 8.5 Overloading << and >> (3 of 5)**

```cpp
Display 8.5 Overloading << and >>

```42``` cout << yourAmount << " + " << myAmount
" equals " << ourAmount << endl;
```

```cpp
44 Money diffAmount = yourAmount - myAmount;
45 cout << yourAmount << " - " << myAmount;
46 " equals " << diffAmount << endl;
```

```cpp
47 return 0;
48 }
```

*Definitions of other member functions are as in Display 8.1.
Definitions of other overloaded operators are as in Display 8.3.*

```cpp
49 ostream& operator <<(ostream& outputStream, const Money& amount)
50 {
51 int absDollars = abs(amount.dollars);
52 int absCents = abs(amount.cents);
53 if (amount.dollars < 0 || amount.cents < 0)
54 //accounts for dollars == 0 or cents == 0
55 outputStream << "$-";
56 else
57 outputStream << \\
58 outputStream << absDollars;
```

*In the main function, cout is plugged in for outputStream.*

*For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.*

`Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.`
Overloaded >> Example:

Display 8.5 Overloading << and >> (4 of 5)

```cpp
59    if (absCents >= 10)
60        outputStream << '.' << absCents;
61    else
62        outputStream << '.' << '0' << absCents;
63    return outputStream;
64
65    //Uses iostream and cstdlib:
66    istream& operator >> (istream& inputStream, Money& amount)
67    {
68        char dollarSign;
69        inputStream >> dollarSign; //hopefully
70        if (dollarSign != '$')
71            {
72                cout << "No dollar sign in Money input.\n";
73                exit(1);
74            }
75
76        double amountAsDouble;
77        inputStream >> amountAsDouble;
78        amount.dollars = amount.dollarsPart(amountAsDouble);
```

(continued)
Overloaded >> Example:

**Display 8.5** Overloading << and >> (5 of 5)

```java
79     amount.cents = amount.centsPart(amountAsDouble);
80     return inputStream;
81   }

**SAMPLE DIALOGUE**
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36
```

Returns a reference
Assignment Operator, =

• Must be overloaded as member operator

• Automatically overloaded
  • Default assignment operator:
    • Member-wise copy
    • Member variables from one object → corresponding member variables from other

• Default OK for simple classes
  • But with pointers → must write our own!
Increment and Decrement

• Each operator has two versions
  • Prefix notation: ++x;
  • Postfix notation: x++;

• Must distinguish in overload
  • Standard overload method → Prefix
  • Add 2d parameter of type int → Postfix
    • Just a marker for compiler!
    • Specifies postfix is allowed
Overload Array Operator, [ ]

• Can overload [ ] for your class
  • To be used with objects of your class
  • Operator must return a reference!
  • Operator [ ] must be a member function!
Summary 1

• C++ built-in operators can be overloaded
  • To work with objects of your class

• Operators are really just functions

• Friend functions have direct private member access

• Operators can be overloaded as member functions
  • 1\textsuperscript{st} operand is calling object
Summary 2

• Friend functions add efficiency only
  • Not required if sufficient accessors/mutators available

• Reference "names" a variable with an alias

• Can overload <<, >>
  • Return type is a reference to stream type