

Introduction to Classes

Unit 4

Chapter 13

CS 2308
Fall 2016

Jill Seaman

1

13.2 The Class

- A class in C++ is similar to a structure.
 - It allows you to define a new (composite) data type.
- A class contains the following:
 - variables AND
 - **functions** (these manipulate the variables)
- These are called members
- A class declaration defines the member variables and the prototypes of the member functions.

2

Example class declaration

```
// models a 12 hour clock
class Time      //new data type
{
private:
    int hour;
    int minute;
    void addHour();

public:
    void setHour(int);
    void setMinute(int);
    int getHour() const;
    int getMinute() const;

    string display() const;
    void addMinute();
};
```

3

Access specifiers

- Used to control access to members of the class
 - public members can be accessed by functions inside AND outside of the class
 - private members can be called or accessed only from functions inside the class (the class's member functions) Private is the default setting for class members.
- Member variables are declared private, to hide their definitions from outside the class.
- Certain functions are declared public to provide controlled access to the hidden/private data.
 - these public functions form the interface to the class

Using const with member functions

- `const` appearing after the parentheses in a member function declaration specifies that the function will **not** change any data inside the object.

```
int getHour() const;
int getMinute() const;
string display() const;
```

- These member functions won't change hour or minute.
- Other functions may or may not change them.
- using `const` is optional.

5

Accessors and mutators

- Accessor functions
 - return a value from the object (without changing it)
 - can be defined using `const`.
 - a “getter” is a special accessor function that returns the value of **one** member variable
- Mutator functions
 - Change the value(s) of member variable(s).
 - a “setter” is a special mutator function that changes (sets) the value of **one** member variable.

6

Defining member functions

- Member function definitions usually occur after of the class definition.
- The name of each function is preceded by the class name and scope resolution operator (`::`)

```
void Time::setHour(int hr) {
    hour = hr;
}
```

hour appears to be undefined,
but it is a member variable of the Time class

7

Defining Member Functions

```
void Time::setHour(int hr) {
    hour = hr;          // hour is a member var
}
void Time::setMinute(int min) {
    minute = min;      // minute is a member var
}
int Time::getHour() const {
    return hour;
}
int Time::getMinute() const {
    return minute;
}

void Time::addHour() { // a private member func
    if (hour == 12)
        hour = 1;
    else
        hour++;
}
```

8

Defining Member Functions

```
void Time::addMinute() {
    if (minute == 59) {
        minute = 0;
        addHour(); // call to private member func
    } else
        minute++;
}

string Time::display() const {
    // returns time in a string formatted to hh:mm
    ostringstream sout; //include <sstream>
    sout.fill('0'); //padding char for setw
    sout << hour << ":" << setw(2) << minute;
    return sout.str(); //str() returns the string
    // from the stream
}
```

ostringstream: allows you to create a string by "outputting" to it using << and i/o manipulators.
fill(ch): sets the padding character used with setw

9

13.3 Defining an instance of the class

- `ClassName` variable;

```
Time t1;
```

like declaring a structure variable

- This defines `t1` to contain an object of type `Time` (with hour and minute members).
- Then access public members of class with dot notation:

```
t1.setHour(3);
t1.setMinute(41);
t1.addMinute();
```

calls to member functions

10

Using the Time class

```
int main() {
    Time t;
    t.setHour(12);
    t.setMinute(58);
    cout << t.display() << endl;
    t.addMinute();
    cout << t.display() << endl;
    t.addMinute();
    cout << t.display() << endl;
}
```

Output:

```
12:58
12:59
1:00
```

Note: the program includes the code from slides 3, 8, 9, and 10 (and any #includes needed). See AllTime.cpp in Time.zip

11

13.1 Procedural Programming

A style of programming in which:

- Data is stored in variables
 - Perhaps using arrays and structs.
- Program is a collection of functions that perform operations over the variables
 - Good example: PA2 inventory program
- Variables are passed to the functions as arguments
- Focus is on organizing and implementing the **functions**.

12

Procedural Programming: Problem

- It is not uncommon for
 - program specifications to change
 - representations of data to be changed for internal improvements.
- As procedural programs become larger and more complex, it is difficult to make changes.
 - A change to a given variable or data structure requires changes to all of the functions operating over that variable or data structure.
- Example: use vectors or linked lists instead of arrays for the inventory

13

Object Oriented Programming: Solution

- An object (instance of a class) contains
 - data (like fields of a struct)
 - functions that operate over that data
- Code outside the object can access the data **only** through the object's functions.
- If the representation of the data inside the object needs to change:
 - Only the object's function definitions must be redefined to adapt to the changes.
 - The code outside the object does not need to change, it accesses the object in the same way.

14

Object Oriented Programming: Concepts

- **Encapsulation**: combining data and code into a single object.
- **Data hiding** (or **Information hiding**) is the ability to hide the details of data representation from the code outside of the object.
- **Interface**: the mechanism that code outside the object uses to interact with the object.
 - The object's (public) functions
 - Specifically, outside code needs to "know" only the function prototypes (not the function bodies).

15

Object Oriented Programming: Real World Example

- In order to drive a car, you need to understand only its interface:
 - ignition switch
 - gas pedal, brake pedal
 - steering wheel
 - gear shifter
- You don't need to understand how the steering works internally.
- You can operate any car with the same interface.

16

Classes and Objects

- A class is like a blueprint for an object.
 - a detailed description of an object.
 - used to make many objects.
 - these objects are called **instances** of the class.
- For example, the string class in C++.

- Make an instance (or two):

```
string cityName1="Austin", cityName2="Dallas";
```

- use the object's functions to work with the objects:

```
int size = cityName1.length();  
cityName2.append(" Cowboys");
```

17

13.5 Separating Specs from Implementation

- Class declarations are usually stored in their own "header files" (Time.h)
 - called the specification file
- Member function definitions are stored in a separate file (Time.cpp)
 - called the class implementation file
- Main function and standalone functions go in a third file (Driver.cpp)

See the Multi-file
Development Lecture
and TimeDemo.zip

18

13.6 Inline member functions

- Member functions can be defined
 - after the class declaration (normally) OR
 - inline: in class declaration
- Inline is appropriate for short function bodies:

```
class Time {  
private:  
    int hour;  
    int minute;  
    void addHour(); // not inlined  
public:  
    int getHour() const { return hour; }  
    int getMinute() const { return minute; }  
    void setHour(int h) { hour = h; }  
    void setMinute(int m) { minute = m; }  
    string display() const; //not inlined  
    void addMinute(); //not inlined  
};
```

19

13.7 Constructors

- A constructor is a member function with the same name as the class.
- It is called automatically when an object is created
- It performs initialization of the new object
- It has no return type

```
class Time  
{  
private:  
    int hour;  
    int minute;  
    void addHour();  
public:  
    Time(); // Constructor prototype  
    ...
```

20

Constructor Definition

- Note no return type, prefixed with Class:::

```
// file Time.cpp
#include <sstream>
#include <iomanip>
using namespace std;

#include "Time.h"

Time::Time() { // initializes hour and minute
    hour = 12;
    minute = 0;
}
void Time::setHour(int hr) {
    hour = hr;
}
void Time::setMinute(int min) {
    minute = min;
}
```

21

Constructor "call"

- From main:

```
//using Time class (Driver.cpp)
#include<iostream>
#include "time.h"
using namespace std;

int main() {
    Time t; //Constructor called implicitly here

    cout << t.display() <<endl;
    t.addMinute();
    cout << t.display() << endl;
}
```

Output: 12:00
12:01

22

13.8 Passing Arguments to Constructors

- To create a constructor that takes arguments:
 - Indicate the parameters in the prototype:

```
class Time
{
public:
    Time(int,int); // Constructor prototype
...
}
```

- Use the parameters in the definition:

```
Time::Time(int hr, int min) {
    hour = hr;
    minute = min;
}
```

23

Passing Arguments to Constructors

- Pass arguments to the constructor when you create an object (in the declaration):

```
int main() {
    Time t (12, 59);
    cout << t.display() <<endl;
}
```

Output:
12:59

24

Default Constructors

- A default constructor is a constructor that takes no arguments (like Time()).
- If you write a class with NO constructors, the compiler will include a default constructor for you, one that does (almost) nothing.
- The original version of the Time class did not define a constructor, so the compiler provided a constructor for it.

25

Classes with no Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.
 - C++ will NOT automatically generate a constructor with no arguments unless your class has NO constructors at all.
- When there are constructors, but no default constructor, you **must** pass the required arguments to the constructor when creating an object.

26

13.9 Destructors

- Member function that is automatically called when an object is destroyed.
- Destructor name is ~classname, e.g., ~Time
- Has no return type; takes no arguments.
- Only one destructor per class (it cannot be overloaded, cannot take arguments).
- If the class dynamically allocates memory, the destructor should release (delete) it

27

Destructors

- Example: Inventory class, with dynamically allocated array:

```
struct Product {
    string productName; // product description
    string locator; // used to find product
    int quantity; // number of copies in inventory
    double price; // selling price of the product
};

class Inventory {
private:
    Product *products; //dynamically allocated array
    int count;
public:
    Inventory (int);
    ~Inventory(); //destructor
    bool addItem(Product);
    int removeItem(String);
    void showInventory();
}

```

Inventory.h

28

Destructors

- Example: member function definitions for constructor and destructor:

```
#include "Inventory.h" Inventory.cpp

Inventory::Inventory(int size){
    products = new Product[size]; //dynamic allocation
    count = 0;
}

Inventory::~Inventory() {
    delete [] products;
}
```

29

Destructors

- Example: driver creates and destroys an Inventory

```
int main() {
    Inventory storeProducts(100); //calls constructor
    //do stuff with storeProducts here
} //end of main, storeProducts object destroyed here,
// calls its destructor (deletes products array)
```

- When is an object destroyed?
 - at the end of its scope OR
 - when it is deleted (if it's dynamically allocated)

30

13.10 Overloaded Constructors

- Recall: when 2 or more functions have the same name they are *overloaded*.
- A class can have more than one constructor function
 - They have the same name, so they are overloaded
- Overloaded functions must have different parameter lists:

```
class Time
{
    private:
        int hour;
        int minute;
    public:
        Time();
        Time(int);
        Time(int,int);
    ...
}
```

31

Overloaded Constructors

- definitions:

```
#include "Time.h"

Time::Time() {
    hour = 12;
    minute = 0;
}
Time::Time(int hr) {
    hour = hr;
    minute = 0;
}
Time::Time(int hr, int min) {
    hour = hr;
    minute = min;
}
```

32

Overloaded Constructor “call”

- From main:

```
int main() {  
    Time t1;  
    Time t2(2);  
    Time t3(4,50);  
  
    cout << t1.display() <<endl;  
    cout << t2.display() <<endl;  
    cout << t3.display() << endl;  
}
```

Output:

```
12:00  
2:00  
4:50
```

33

13.12 Arrays of Objects

- An array can contain objects (the element type can be a Class):

```
int main() {  
    Time recentCalls[10]; //times of last 10 phone calls  
}
```

- The default constructor (Time()) is used to initialize each element of the array when it is created.
- This array is initialized to 10 Time objects, each set to 12:00.
- To invoke a constructor that takes arguments, you must use an initializer list . . .

34

Arrays of Objects initializer lists

- Each initializer takes the form of a function call:

```
int main() {  
    Time recentCalls[7] = {Time(1),  
                           Time(2,13),  
                           Time(3,24),  
                           Time(4),  
                           Time(4,50)};  
}
```

- If there are fewer initializers in the list than elements in the array, the default constructor will be called for all the remaining elements.
- This array is initialized to 7 Time objects, set to 1:00, 2:13, 3:24, 4:00, 4:50, 12:00 and 12:00.

35

Accessing Objects in an Array

- Objects in an array are referenced using subscripts
- Member functions are referenced using dot notation
- Must access the specific object in the array BEFORE calling the member function:

```
recentCalls[2].setMinute(30);  
cout << recentCalls[4].display() << endl;
```

- Processing array elements in a loop:

```
for (int i=0; i<7; i++)  
    cout << recentCalls[i].display() << " ";  
cout << endl;
```

36