Ch 13: Introduction to Classes

CS 2308
Spring 2013

Jill Seaman

13.1 Procedural Programming

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

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 instead of arrays for the inventory

Object Oriented Programming: Solution

- An object contains
  - data (like fields of a struct)
  - functions that operate over that data
- Code outside the object can access the data only via the object’s functions.
- If the representation of the data in the object needs to change:
  - Only the object’s functions 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.

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).

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.

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.insert(0,”Big “);
```

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:
  - variables AND
  - functions
- These are called members
- Members can be:
  - private: inaccessible outside the class
  - public: accessible outside the class.

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();
};
```

Access rules

- Used to control access to members of the class
- public: can be accessed by functions inside AND outside of the class
- private: can be called by or accessed by only functions that are members of the class (inside)
  - member variables (attributes) 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 in the calling object.

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

- These member functions won't change hour or minute.

Defining member functions

- Member function definitions usually occur outside of the class definition (in a separate file).
- 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

Accessors and mutators

- Accessor functions
  - return a value from the object (without changing it)
  - a “getter” returns the value of a member variable
- Mutator functions
  - Change the value(s) of member variable(s).
  - a “setter” changes (sets) the value of a member variable.

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++;
}
```

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 string formatted to hh:mm
   ostringstream sout;  //include <sstream>
   sout.fill('0');      //padding char for setw
   sout << hour << ":" << setw(2) << minute;
   return sout.str();
}
```

13.3 Defining an instance of the class

- ClassName variable (like a structure):

```
Time t1;
```

- This defines t1 to contain an object of type Time (the values of hour and minute are not set).
- Access public members of class with dot notation:

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

- Use dot notation OUTSIDE class only.

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;
  return 0;
}
```

Output:

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

Do not store stale data

- Why not store display string in a variable instead of composing it every time?
- Because it could become stale.
  - If the minute or hour changes, then the data in the object would be inconsistent:
  - stored display string would not match new hours and minutes.
- Don't store any data that could become stale, compute it in a member function instead.

13.4 Setters and getters: what's the point?

- Why have setters and getters that just do assignment and return values?
- Why not just make the member variables public?

- Setter functions can validate the incoming data.
  - setMinute can make sure minutes are between 0 and 59 (if not, it can report an error).
- Getter functions could act as a gatekeeper to the data or provide type conversion.

13.5 Separating Specs from Implementation

- Class declarations are usually stored in their own header files (Time.h)
  - called the specification file
  - filename is usually same as class name.
- Member function definitions are stored in a separate file (Time.cpp)
  - called the class implementation file
  - it must #include the header file,
- Any program/file using the class must include the class's header file (#include "Time.h")

13.6 Inline member functions

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

```
class Time {
  private:
    int hour;
    int minute;

  public:
    int getHour() const
    {  return hour; }
    int getMinute() const
    {  return minute; }
    void setHour(int);
    void setMinute(int);   // ... class decl cont.
```

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
...
```

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;
}
```

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;
  return 0;
}
```

Output:

```
12:00
12:01
```

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 nothing.
- The original version of the Time class did not define a constructor, so the compiler provided a “do-nothing” constructor for it.

13.8 Passing Arguments to Constructors

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

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

  - Use parameters in the definition:

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

Passing Arguments to Constructors

- Then pass arguments to the constructor when you create an object:

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

Output:

```
12:59
```

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.

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, i.e., it cannot be overloaded, cannot take arguments
- If the class allocates dynamic memory, the destructor should release (delete) it

Destructors

- Example: class decl
  An alternative way to declare the PasswordManager:

pwman.h

```
#include <string>
using namespace std;

  class PasswordManager
  {
    private:
      char *encryptedPassword;

    public:
      PasswordManager(char *encPW);  //constructor
      ~PasswordManager();            //destructor
  };
```

Destructors

- Example: member function definitions (class impl)

pwman.cpp

```
#include "pwman.h"

PasswordManager::PasswordManager(char *encPW){
   encryptedPassword = new char [strlen(encPW)+1];
   strcpy(encryptedPassword,encPW);
}

PasswordManager::~PasswordManager() {
   delete [] encryptedPassword;
}
```

Destructors

- Example: member function definitions (class impl)

```
int main() {

  char k[] = “SSS”;
  PasswordManager pm(k);     //calls constructor

  //do stuff with pm here

  return 0;
}    //end of prog, pm destroyed here, calls destructor
```

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

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
  - 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);
...
```

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;
}
```

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;
  return 0;
}
```

Output:

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

Overloaded Member Functions

- Non-constructor member functions can also be overloaded
- Must have unique parameter lists as for constructors

```
class Time
{
    private:
      int hour;
      int minute;
    public:
      Time();
      Time(int);
      Time(int,int);
      void addMinute();        //adds one minute
      void addMinute(int);     //adds minutes from arg
...
```

13.12 Arrays of Objects

- Objects can be the elements of an array:

```
int main() {

  Time  missedCalls[10];  //times of last 10 missed calls

}
```

- Default constructor is used to initialize each element when the array is defined

Arrays of Objects

- To invoke a constructor that takes arguments, you must use an initializer list:

```
int main() {

  Time  missedCalls[10] = {1,2,3,4,5,6,7,8,9,10};

}
```

- The constructor taking one argument is used to initialize each of the 10 Time objects here

Arrays of Objects

- If the constructor requires more than one argument, the initializer must take the form of a function call:

```
int main() {

  Time  missedCalls[5] = {Time(1,5),
                          Time(2,13),
                          Time(3,24),
                          Time(3,55),
                          Time(4,50)};

}
```

Arrays of Objects

- It isn't necessary to call the same constructor for each object in an array:

```
int main() {

  Time  missedCalls[7] = {1,
                          Time(2,13),
                          Time(3,24),
                          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.

Accessing Objects in an Array

- Objects in an array are referenced using subscripts
- Member functions are referenced using dot notation:

```
missedCalls[2].setMinute(30);

cout << missedCalls[4].display() << endl;
```