Classes and Objects

Week 5

Gaddis: 13.2-13.12
14.3-14.4

CS 5301
Fall 2015

Jill Seaman

The Class

- A class in C++ is similar to a structure.
- A class contains members:
  - variables AND
  - functions (often called methods)
    (these manipulate the member variables).
- Members can be:
  - private: inaccessible outside the class
    (this is the default)
  - public: accessible outside the class.

Example class: Time

class declaration with functions defined inline

```
class Time {            //new data type
  private:
    int hour;
    int minute;
  public:
    void setHour(int hr)    { hour = hr; }
    void setMinute(int min) { minute = min; }
    int getHour() const    { return hour; }
    int getMinute() const { return minute; }
    void display() const   { cout << hour << ":" << minute; }
};
int main()
{
   Time t1, t2;

   t1.setHour(6);
   t1.setMinute(30);
   cout << t1.getHour() << endl;

   t2.setHour(9);
   t2.setMinute(20);
   t2.display();
   cout << endl;
}
```

Output:

```
6
9:20
```

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   { return hour; }
int getMinute() const { return minute; }
void display() const  { cout << hour << ":" << minute; }
```

- These member functions don’t change hour or minute.

Accessors and mutators

- Accessor functions
  - return a value from the object (without changing it)
  - a “getter” returns the value of a member variable

```
int getHour() const   { return hour; }
int getMinute() const { return minute; }
```

- Mutator functions
  - Change the value(s) of member variable(s).
  - a “setter” changes (sets) the value of a member variable.

```
void setHour(int hr)    { hour = hr; }
void setMinute(int min) { minute = min; }
```

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)

```
int main()
{
   Time t1;

   t1.setHour(6);
   t1.setMinute(30);
   cout << t1.hour << endl; //Error, hour is private

};
```

Separation of Interface 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”)

Time class, separate files

Time.h

```
// models a 12 hour clock
class Time {

private:
    int hour;
    int minute;

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

    void display() const;
};
```

Time.cpp

```
#include <iostream>
#include "Time.h"
using namespace std;

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

void Time::setMinute(int min) {
  minute = min;
}

int Time::getHour() const {
  return hour;
}

int Time::getMinute() const {
  return minute;
}

void Time::display() const {
   cout << hour << “:” << minute;
}
```

Time class, separate files

Driver.cpp

```
//Example using Time class
#include<iostream>
#include "Time.h"
using namespace std;

int main() {
    Time t;
    t.setHour(12);
    t.setMinute(58);
    t.display();
    cout <<endl;
    t.setMinute(59);
    t.display();
    cout  << endl;
}
```

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
- It can be overloaded: more than one constructor function, each with different parameter lists.
- A constructor with no parameters is the **default** constructor.
- If your class defines **no** constructors, C++ will provide a default constructor automatically.

Constructor Declaration+Definition

- Note no return type, same name as class:

```
// models a 12 hour clock
class Time {

private:
    int hour;
    int minute;

public:
    Time();
    Time(int,int);

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

    void display() const;
};
```

Time.cpp

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

#include "Time.h"

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

Constructor Use

- Called from the object declaration

```
//Example using Time class
#include<iostream>
#include "Time.h"
using namespace std;

int main() {
    Time t;
    t.display();
    cout <<endl;

    Time t1(10,30);
    t1.display();
    cout  << endl;
}
```

Output:

```
12:0
10:30
```

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.

```
class Time
{
   public:
      Time();     // Constructor prototype
      ~Time();    // Destructor prototype       …
```

Composition

- When one class contains another as a member:

Calls.h

```
#include "Time.h"
class Calls
{
  private:
    Time calls[10];  // times of last 10 phone calls
    // array is initialized using default constructor
  public:
    void set(int,Time);
    void displayAll();
}
```

Calls.cpp

```
#include "Calls.h"
#include <iostream>
using namespace std;

void Calls::set(int i, Time t) {
   calls[i] = t;
}
void Calls::displayAll () {
   for (int i=0; i<10; i++) {
      calls[i].display();     //calls member function
      cout << "  ";
   }
}
```

Composition

- Driver for Calls

```
//Example using Calls and Time classes
#include<iostream>
#include "Calls.h"    //this includes "Time.h"
using namespace std;

int main() {
    Calls callTimes;
    Time t1(4,30);
    callTimes.set(0,t1);
    Time t2(11,42);
    callTimes.set(1,t2);

    callTimes.displayAll();
    cout  << endl;
}
```

Output:

```
4:30  11:42  12:0  12:0  12:0  12:0  12:0  12:0  12:0  12:0
```

Pointers to Objects

- We can define pointers to objects, just like pointers to structures

```
Time t1(12,20);
Time *timePtr;
timePtr = &t1;
```

- We can access public members of the object using the structure pointer operator (->)

```
timePtr->setMinute(21);
cout << timePtr->display() << endl;
```

```
Output:
12:21
```

Dynamically Allocating Objects

- Objects can be dynamically allocated with new:

```
Time *tptr;
tptr = new Time(12,20);
...
delete tptr;
```

You can pass arguments to a constructor using this syntax.

- Arrays of objects can also be dynamically allocated:

```
Time *tptr;
tptr = new Time[100];
tptr[0].setMinute(11);
...
delete [] tptr;
```

It can use only the default constructor to initialize the elements in the new array.

Copy Constructors

- Special constructor used when a newly created object is initialized using another object of the **same class**.

```
Time t1;
Time t2 = t1;
Time t3 (t1);
```

Both of the last two use the copy constructor

- The **default copy** constructor, provided by the compiler, copies member-to-member.
- Default copy constructor works fine in most cases
- You can re-define it for your class as needed.

IntCell declaration

- Problem with the default copy constructor: what if the class contains a pointer member?

```
class IntCell
{
   public:
      IntCell (int);
      int read () const;
      void write (int );

   private:
      int *storedValue;
};
```

```
IntCell::IntCell (int initialValue)
{ storedValue = new int;
  *storedValue = initialValue; }

int IntCell::read () const
{ return *storedValue; }

void IntCell::write (int x)
{ *storedValue = x; }
```

Note: dynamic memory allocation

Problem with member-wise assignment

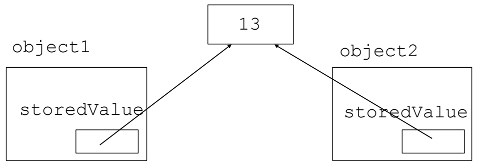
- What we get from member-wise assignment in objects containing dynamic memory (ptrs):

```
IntCell object1(5);
IntCell object2 = object1; // calls copy constructor

  //object2.storedValue=object1.storedValue

object2.write(13);
cout << object1.read() << endl;
cout << object2.read() << endl;
```

Output:
```
13
13
```



Programmer-Defined Copy Constructor

- Prototype and definition of copy constructor:

```
IntCell(const IntCell &obj);
```

Add to class declaration

```
IntCell::IntCell(const IntCell &obj) {
   storedValue = new int;
   *storedValue = *(obj.storedValue)
}
```

- Copy constructor takes a **reference** parameter to an object of the class
  - This is required.

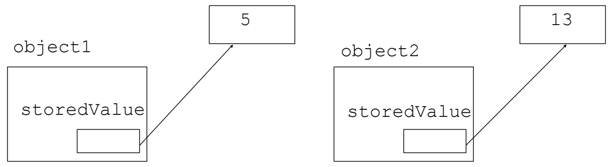
Programmer-Defined Copy Constructor

Each object now points to separate dynamic memory:

```
IntCell object1(5);
IntCell object2 = object1;   //now calls MY copy constr

object2.write(13);
cout << object1.read() << endl;
cout << object2.read() << endl;
```

Output:
```
5
13
```



Sample Problem 1

Car class: Write a class named Car that has the following member variables:

- `yearModel` An int that holds the car's year model.
- `make` A string that holds the make of the car.
- `speed` An int that holds the car's current speed.

The class should have the following member functions.

- Constructor. Accepts the car's year model and make as arguments. Assigns 0 to the speed variable.
- Accessors. Functions to get the values stored in member variables.
- accelerate. The accelerate function should add 5 to the speed member variable

Demonstrate the class in a driver program.

Sample Problem 2

Number Array Class: Design a class that has an array of floating-point numbers. The constructor should accept an integer argument and dynamically allocate the array to hold that many numbers. The destructor should free the memory held by the array. In addition, there should be member functions to perform the following operations:

- Store a number in a specified position of the array
- Retrieve a number from a position of the array
- Return the average of the values stored in the array
- Include a copy constructor!