Classes and Objects

Week 5

Gaddis: 13.1-13.12
14.3-14.4

CS 5301
Spring 2014

Jill Seaman

The Class

- A class in C++ is similar to a structure.
- A class contains:
  - variables (members) AND
  - functions (member functions or 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; }
};
```

Using Time class in a driver

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

- These member functions won'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;
int getMinute() const;
```

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

```
void setHour(int hr);
void setMinute(int min);
```

Example class: Time (version 2)

class declaration with functions defined outside

```
class Time {             //new data type
  private:
    int hour;
    int minute;
  public:
    void setHour(int);
    void setMinute(int);
    int getHour() const;
    int getMinute() const;
    void display() const;
};
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::display() const {
   cout << hour << ":" << minute;
}
```

The member functions can be defined outside the class

Don't forget the class name and scope resolution operator (::)

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;
};
```

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;
}
```

Time class, separate files

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;
}
```

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 (and use)

- 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;
};
```

```
//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
```

Constructor Definition

- Note no return type, prefixed with Class::

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;
}
...
```

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

This class declaration uses inlined function definitions

```
//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.display();
    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 copies field-to-field (member-wise assignment).
- 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 object contains a pointer?

```
class IntCell
{
   private:
      int *storedValue;    //ptr to int

   public:
      IntCell (int initialValue);
      ~IntCell();
      int read () const;
      void write (int x);
};
```

IntCell Implementation

```
#include “IntCell.h”

IntCell::IntCell (int initialValue) {
  storedValue = new int;      //dynamic memory allocation
  *storedValue = initialValue;
}

IntCell::~IntCell() {
  delete storedValue;
}

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

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

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;
```

What is output?

```
5
13
```

or

```
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.read();   //or *(obj.storedValue)
}
```

- Copy constructor takes a **reference** parameter to an object of the class
  - otherwise it would use the copy constructor to initialize the obj parameter, which would call the copy constructor: this is an infinite loop

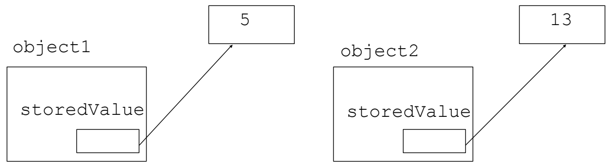
Programmer-Defined Copy Constructor

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
```


5
13
object1
object2
storedValue
storedValue