More About Classes

Gaddis Ch. 14, 13.3

CS 2308 :: Spring 2016
Molly O'Neil
Instance & Static Members

- **Instance variable**: a member variable in a class. Each object (instance) of the class has its own copy (i.e. separate storage space).

- **Static variable**: one variable (storage location) shared among all instances (objects) of a class
  - Exist for the lifetime of the program (including before any objects of the class have been instantiated)

- **Static member function**: a member function that can be called as a standalone function (without reference to any specific object of the class)
  - Can be called before any objects of the class exist
  - Function can access only static member variables (Normal member functions can access static member variables, too)
class Tree {
private:
    static int treeCount; // static member variable
    int numLeaves;       // instance variable

public:
    Tree(int);
    int getNumLeaves() const;
    int getTreeCount() const;
};

// Outside of class declaration
int Tree::treeCount = 0;

Tree::Tree(int leaves) {
    treeCount++;
    numLeaves = leaves;
}

int Tree::getNumLeaves() const { return numLeaves; }

int Tree::getTreeCount() const {
    return treeCount;
}
Demo of Static Variable

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

int main() {
    Tree oak(306);
    Tree elm(635);
    Tree pine(951);

    cout << "We have " << pine.getTreeCount() << " trees" << " in our program." << endl;
    cout << "The pine tree has " << pine.getNumLeaves() << " leaves." << endl;
}
```

What's the output?

What's the output if we replace `pine.getTreeCount()` with `elm.getTreeCount()`?
Instance vs. Static Variables

- There are 3 instances of the Tree class, but only one treeCount variable

```
treeCount [static]
  3
  ↓
  oak
    numLeaves 306

  ↓
  elm
    numLeaves 635

  ↓
  pine
    numLeaves 951
```
Tree Class w/ Static Member Function

```cpp
class Tree {
private:
    static int treeCount;  // static member variable
    int numLeaves;         // instance variable

public:
    Tree(int);            // No const keyword!
    int getNumLeaves() const;
    static int getTreeCount();
};
```

// Outside of class declaration

```cpp
int Tree::getTreeCount() {
    return treeCount;       // "static" keyword is not repeated here
}
```

- Static member functions cannot be declared `const`
- "const" means "doesn't modify the state of any instance variables"; static functions can't access instance variables!
Demo of Static Member Function

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

int main() {
    cout << "We have " << Tree::getTreeCount() << " trees"
         << " in our program." << endl;

    Tree oak(306);
    Tree elm(635);
    Tree pine(951);

    cout << "Now we have " << pine.getTreeCount() << " trees"
         << " in our program." << endl;
    cout << "The pine tree has " << pine.getNumLeaves()
         << " leaves." << endl;
}
```

We can now call `getTreeCount()` independent of any object

Can still access static function through an object, too... but this results in less intuitive code!
Pointers to Objects

• Just like pointers to structures, we can define pointers to objects

```cpp
Time t1(12, 20, true);
Time *tPtr;

tPtr = &t1;
```

• We can access public members of the object using the structure pointer operator (→)

  • Unlike structs, for objects, the public members are likely to be member functions, not variables

```cpp
tPtr->addMinute();
cout << tPtr->toString() << endl;
```

Output:

```
12:21 AM
```

12:21 AM
Dynamic Allocation of Objects

• Objects can be dynamically allocated with `new`

```cpp
Time *tPtr = new Time(12, 20, true);
...
delete tPtr;
```

Passes arguments to the constructor

• Arrays of objects can also be dynamically allocated

```cpp
Time *tList = new Time[100];
tList[0].addMinute();
...
delete [] tList;
```

Must use (and have) a default constructor!
Deleting Dynamically Allocated Objects

- Recall our MP3Player example from the OOP lecture:

```cpp
class MP3Player {
    private:
        Song *songs;
        int maxSongs;
        double maxMB;

    public:
        MP3Player(int, double);
        ~MP3Player();
        double getMaxMB() const;
};

MP3Player::MP3Player(int numSongs, double numMB) {
    songs = new Song[numSongs];  // dynamic allocation
    maxSongs = numSongs;
    maxMB = numMB;
}

MP3Player::~MP3Player() {
    delete [] songs;
}
```
Deleting Dynamically Allocated Objects

- Where is the songs array deallocated?

```cpp
#include "MP3Player.h"

int main() {
    MP3Player mp3(50, 25.0);
    cout << mp3.getMaxMB();
    return 0;
}
```

```cpp
#include "MP3Player.h"

int main() {
    MP3Player *mp3;
    mp3 = new MP3Player(50, 25.0);
    cout << mp3->getMaxMB();
    cout << endl;
    delete mp3;
    return 0;
}
```

mp3.~MP3Player() is called here, when mp3 goes out of scope. The destructor de-allocates mp3.songs, then mp3 is destroyed.

mp3->~MP3Player() is called here, when mp3 is deleted. The destructor de-allocates mp3->songs, then mp3 is de-allocated.
The **this** Pointer

- **this**: A pre-defined `pointer` available within a class (inside the member function definitions)
  - Always points to the instance (object) of the class whose function is being executed

- You can use **this** to access member variables that may be hidden by parameters with the same name:

```cpp
void Time::setHour(int hour) {
    if(hour < 1 || hour > 12)
        this->hour = 0;
    else
        this->hour = hour;
}
```

This means:

Set the **member variable** called `hour` in **THIS** object to the value of the **local variable** called `hour`
Constructor Initialization Lists

• We have seen constructors that initialize class member variables as follows:

```cpp
Time::Time() {
    hour = 12;
    minute = 0;
    isAM = true;
}

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

• There is another syntax to perform this initialization, called an **initialization list**

```cpp
Time::Time() : hour(12), minute(0), isAM(true) { }

Time::Time(int hr, int min, bool am) : hour(hr), minute(min), isAM(am) { }
```
Constructor Initialization Lists

• The initialization list goes with the **constructor definition**, not the declaration
  • If the definition is in the class declaration (i.e., inlined), the initialization list goes there, too

• Not all of the class member variables must be initialized in the list, and the constructor body does not have to be empty, e.g.:

```cpp
Time::Time(int hr, int min, bool am) : hour(hr), minute(min) {
    isAM = am;
}
```

• So what's the difference?
  • Often, nothing (except maybe minor performance advantage from initialization list). Use whichever you prefer!
  • Sometimes an initialization list is required
When Are Initialization Lists Required?

- To initialize non-static const data members
- To initialize reference members
- To initialize member objects that do not have a default constructor
- To initialize a class member with the same name as a parameter (without using the `this` pointer)

```cpp
class Example {
    const int x;
    int &y;
    SomeClass c;  // SomeClass has no default constructor,
                   // constructor requires a single int param
    int m;

public:
    Example(int ax, int &ay, SomeClass ac, int m) :
        x(ax), y(ay), c(10), m(m) { }
};
```
Memberwise Assignment of Objects

• Can use the = operator to
  • Assign (copy) one object to another
  • Initialize an object with another object's data

• Just like = operator with structs, this will copy member variable to variable. For example:

```c
object2 = object1;
```

means: copy all member values from `object1` and assign the copied values to the corresponding member variables of `object2`

• Member-wise assignment also used at initialization:

```c
Time t2 = t1;
```
Memberwise Assignment Demo

Time t1(10, 20, true);
Time t2(12, 40, false);

cout << "t1: " << t1.toString() << endl;
cout << "t2: " << t2.toString() << endl;
cout << endl;

t2 = t1;

cout << "t1: " << t1.toString() << endl;
cout << "t2: " << t2.toString() << endl;

Output:

<table>
<thead>
<tr>
<th>t1: 10:20 AM</th>
<th>t2 = t1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2: 12:40 PM</td>
<td>is equivalent to:</td>
</tr>
<tr>
<td>t1: 10:20 AM</td>
<td>t2.hour = t1.hour;</td>
</tr>
<tr>
<td>t2: 10:20 AM</td>
<td>t2.minute = t1.minute;</td>
</tr>
<tr>
<td></td>
<td>t2.isAM = t1.isAM;</td>
</tr>
</tbody>
</table>
Copy Constructors

• Assignment and initialization use the same operator (\(=\)), but they are two different things:
  • Object assignment calls the object's \texttt{operator=} function (more on this later...)
  • Object initialization with the \texttt{=} operator calls the object's \textit{copy constructor}

• A \textit{copy constructor} is a specialized constructor called automatically when an object is initialized using an object of the same class

\begin{verbatim}
Time t1;
Time t2(t1);
Time t3 = t1;
\end{verbatim}

\textit{Both call the copy constructor}
Copy Constructors

• The copy constructor is also implicitly called when you pass an object as an argument to a function
  • Just like other types, objects are passed by value in C++, i.e. copied into the function's local parameter variable

• As with default constructors and destructors, the compiler will provide a default copy constructor
  • The default copy constructor performs memberwise assignment
  • In many cases, this works just fine. However...
Uh-Oh...

• **What if the object contains a pointer?**

```cpp
class DynamicInt  {
    private:
        int *val;  // pointer to an int

    public:
        DynamicInt(int);
        ~DynamicInt();

        int getVal() const;
        void setVal(int);
};
```
#include "DynamicInt.h"

DynamicInt::DynamicInt(int v) {
    val = new int;
    *val = v;
}

DynamicInt::~DynamicInt() {
    delete val;
}

int DynamicInt::getVal() const {
    return *val;
}

void DynamicInt::setVal(int v) {
    *val = v;
}
Uh-Oh...

What happens if we try member-wise initialization of an object with a pointer to dynamically-allocated memory?

```cpp
DynamicInt i1(5);
DynamicInt i2 = i1;

i1.setVal(10);

cout << i1.getVal() << endl;
cout << i2.getVal() << endl;
```

What's the output?

- 10
- 5
- 10
- 10

?
What Happened?!

• Pointer variables are storage locations that store addresses. In `DynamicInt`, the member variable that is copied on memberwise assignment is an int-pointer.

• The new object receives a copy of the address of the first object's allocation, not a new allocation

• There's still only one integer variable on the heap (dynamic memory), and both `i1` and `i2` have a pointer to it

• This is called a **shallow copy**: Pointers are duplicated, but the locations they point at are **not** duplicated
Defining Your Own Copy Constructor

• Prototype of programmer-defined copy constructor (public member function):

```cpp
DynamicInt(const DynamicInt &);  // goes in class declaration
```

• Definition of programmer-defined copy constructor:

```cpp
DynamicInt::DynamicInt(const DynamicInt &rhs) {
    val = new int;
    *val = rhs.getVal(); // or *(rhs.val)
    // or even: setValue(rhs.getVal());
}
```
Copy Constructor Parameter

DynamicInt::DynamicInt(const DynamicInt &rhs) { ... }

- **Why the *reference parameter*?**
  - Because if the parameter was pass-by-value, it would implicitly call the copy constructor to initialize the member function's parameter variable... which would call the copy constructor to initialize the parameter... which would call the copy constructor... oops!

- **Why the *const keyword*?**
  - Because we're forced to pass by reference, without it we would be giving the copy constructor the ability to modify the argument object's member variables
Deep Copy Constructor

• A **deep copy** allocates new storage space for all dynamically-allocated member variables (pointers) and copies over the *value*, rather than just the pointer

• Assuming the programmer-defined copy constructor from 2 slides ago...

```cpp
DynamicInt i1(5);
DynamicInt i2 = i1;  // now calls OUR copy constructor

i1.setVal(10);

cout << i1.getVal() << endl;
cout << i2.getVal() << endl;
```

Output:

```
10
5
```
When Is Copy Constructor Called?

- Copy constructor is invoked...
  - When we initialize an object from another object:
    ```
    DynamicInt i2 = i1;
    DynamicInt i3(i1);
    ```
  - Implicitly by the compiler when we pass an object by value to a function parameter:
    ```
    void someFunction(DynamicInt obj);
    ```
  - Implicitly by the compiler when we return an object by value from a function:
    ```
    DynamicInt someFunction(int x);
    ```
What About Assignment?

• The copy constructor is **NOT** invoked when we assign one *existing* object to another *existing* object

```cpp
Time t1(10, 20, true);
Time t2(12, 40, false);

// This invokes the assignment operator (=)

// The compiler provides a default one for all classes

// The default implements a shallow (memberwise) copy

// Classes that allocate memory have a problem...
```
Let's Draw a Picture...

DynamicInt i1(3);
DynamicInt i2(7);

i2 = i1;
i1.val = 1;

• We have a similar problem as with a shallow copy constructor, but it's even worse, because now we have a memory leak, too!

• We need to overload the default behavior of the = operator
Operator Overloading

• Operators (e.g, =, +, <, ...) can be overloaded to work for objects of a programmer-defined class

• To overload an operator for a class, define a member function named `operator` followed by the operator symbol you want to overload:
  - `operator+` to define the `+` operator,
  - `operator=` to define the `=` operator, etc.

• Just like a regular member function:
  • Prototype goes in the class declaration
  • Function definition goes in the implementation file
Operator Overloading

• Prototype in class declaration (public member function):

```cpp
void operator=(const DynamicInt &rhs);
```

• `operator=` is the function name

• The operator function is defined from the perspective of the object on the left hand side of the assignment statement
  
  • The reference parameter, DynamicInt &rhs, is the right hand side object in the assignment statement
  
  • For the statement: `i1 = i2`
    
    • Inside the `operator=` function definition, `val` refers to `i1's val` member variable
    
    • `rhs.val` refers to `i2's val` member variable
Calling an Overloaded Operator

• The operator function is called on the object on the left hand side of the operator

• It can be called like a normal member function (if you really like writing strange-looking code...):

```cpp
DynamicInt i1(5);
DynamicInt i2(10);

i1.operator=(i2);
```

• It can also be called using the more conventional operator syntax:

```cpp
i1 = i2;
```

• Both call the same `operator=` member function, from the perspective of `i1`
Overloading = for DynamicInt

```cpp
class DynamicInt {
    private:
        int *val;   // pointer to an int

    public:
        DynamicInt(int);                 // constructor
        DynamicInt(const DynamicInt &);  // copy constructor
        ~DynamicInt();                   // destructor
        int getVal() const;
        void setVal(int);
        void operator=(const DynamicInt &);
};

void DynamicInt::operator=(const DynamicInt &rhs) {
    setVal(rhs.getVal());
}

DynamicInt i1(3);
DynamicInt i2(7);

i2 = i1;

i1.setVal(1);    // i2's *val is now unchanged!
```
Overloading Other Operators

• The general reason you want to overload operators is to allow nicer syntax for common class operations
  • For example, we may want to overload the − operator of our Time class to return the number of minutes between two times
  • Overloading >> and << is another common use, to allow you to print out a class object or set its member variables from user input (but we won’t do this in this class)
  • However, there's nothing to stop you from doing weird and crazy things like turning the = operator into a print-to-console operation, or making the * operator divide...
• You can overload almost all of the C++ operators, including ++, −−, &&, ||, [], <<, etc.
Operators for Our Time Class

(I've re-defined Time as a 12-hour clock with no AM/PM, to shorten the code)

class Time {
  private:
    int hour;
    int minute;
    void addHour();

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

    int operator-(const Time &);
    bool operator==(const Time &);
    bool operator<(const Time &);
    Time operator+(const Time &);

    void addMinute();
    string toString() const;
};
Time::operator-

- time1 - time2 is the number of minutes that separate the two times
- We can choose the return type: in this case, an integer

```cpp
int Time::operator-(const Time &rhs) {
    return (hour % 12) * 60 + minute -           // 12 % 12 = 0
              ((rhs.hour % 12) * 60 + rhs.minute);
}
```

```cpp
int main() {
    Time t1(12, 20);
    Time t2(4, 40);

    int minutesApart = t2 - t1;

    cout << minutesApart << endl;
}
```

Output: 260
bool Time::operator==(const Time &rhs) {
    if(hour == rhs.hour && minute == rhs.minute)
        return true;
    return false;
}

bool Time::operator<(const Time &rhs) {
    if(hour == rhs.hour)
        return (minute < rhs.minute);
    return (hour % 12) < (rhs.hour % 12);
}

int main() {
    Time t1(12, 20);
    Time t2(12, 21);

    if(t1 < t2)
        cout << "correct" << endl;
    t1.addMinute();
    if(t1 == t2)
        cout << "correct again" << endl;
}
Time::operator+

- time1 + time2 yields another time (e.g., 3:00 AM + 2:00 hours = 5:00 AM)
  - Addition operator should return a new Time object

```cpp
Time Time::operator+(const Time &rhs) {
    int totalMin = (hour % 12) * 60 + minute +
                   (rhs.hour % 12) * 60 + rhs.minute;
    int totalHours = totalMin / 60; // int div, rounds down
    totalHours %= 12; // must be between 0 and 11
    if(totalHours == 0)
        totalHours = 12; // convert 0:XX to 12:XX
    totalMin %= 60; // get remainder minutes

    Time result(totalHours, totalMin); // create new Time obj
    return result;
}
```

```cpp
int main() {
    Time t1(12, 5);
    Time t2(2, 50);
    Time t3 = t1 + t2;
    cout << t3.toString() << endl;
}
```

**Output:**

```
2:55
```
When An Object Must Return Itself

- Sometimes, an object must return itself as the result of a binary operation, like assignment:

  \[ v1 = v2 = v3; \quad \text{is equivalent to} \quad v1 = (v2 = v3); \]

- Associativity of = operator is right to left
  - Value of \( (v2 = v3) \) is \( v2 \) (the left-hand operand)
  - This means \( v1 = v2 = v3 \) is equivalent to
    \[
    v2 = v3; \\
    v1 = v2;
    \]

- To make \( v1 = v2 = v3 \) work correctly, the assignment operator must also return the post-assignment value of the left-hand operand!
Returning *this

- We overloaded `operator=` for DynamicInt, but we had a `void` return type!

```cpp
class DynamicInt {
  private:
    int *val; // pointer to an int
  public:
    int getVal() const;
    void setVal(int);
    DynamicInt operator=(const DynamicInt &);
};
```

```cpp
DynamicInt DynamicInt::operator=(const DynamicInt &rhs) {
  setVal(rhs.getVal());
  return *this;
}
```

this is a pointer, so must de-reference it to return the DynamicInt object