Overloading Operators and Templates

Week 6

Gaddis: 14.5 and 16.2-16.4

CS 5301
Spring 2014

Jill Seaman

Example class: Time

class declaration with functions defined inline

We will use this for operator overloading examples:

```
class Time {              //new data type
  private:
    int hour;
    int minute;
  public:
    Time()  { hour = 12; minute = 0; }
    Time(int hr,int min) { hour = hr; minute = min; }
    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; }
};
```


Operator Overloading

- Operators such as =, +, <, and others can be defined to work for objects of a user-defined class
- The name of the function defining the over-loaded operator is `operator` followed by the operator symbol:
  `operator+` to define the + operator, and
  `operator=` to define the = operator
- Just like a regular member function:
  - Prototype goes in the class declaration
  - Function definition goes in implementation file

Overloaded Operator Prototype

- Prototype:

```
int operator-(const Time &right);
```

return type — function name — parameter for object on right side of operator

- Pass by constant reference
  - Does NOT copy the argument as pass-by-value does
  - But does not allow the function to change its value
  - (so it's like pass by value without the copying).

Invoking an Overloaded Operator

- Operator can be invoked (called) as a regular member function:

```
int minutes = object1.operator-(object2);
```

- It can also be invoked using the more conventional syntax for operators:

```
int minutes = object1 - object2;
```

This is the main reason to overload operators, so you can use this syntax for objects of your class

- Both call the same function (operator-), from the perspective of object1 (on the lefthand side).

Example: minus for Time objects

Subtraction

```
class Time {
   private:
     int hour, minute;
   public:
     int operator- (const Time &right);
};

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

//in a driver:
Time time1(12,20), time2(4,40);
int minutesDiff = time2 - time1;
cout << minutesDiff << endl;
```

Output: 260

Overloading == and < for Time

```
bool Time::operator== (Time right) {
   if (hour == right.hour &&
       minute == right.minute)
      return true;
   else
      return false;
}

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

//in a driver:
Time time1(12,20), time2(12,21);
if (time1<time2) cout << "correct" << endl;
if (time1==time2) cout << "correct again"<< endl;
```


Overloading + for Time

```
class Time {
  private:
    int hour, minute;
  public:
    Time operator+ (Time right);
};
Time Time::operator+ (Time right) { //Note: 12%12 = 0
  int totalMin = (hour%12)*60 + (right.hour%12)*60
               + minute + right.minute;
  int h = totalMin / 60;
  if (h==0) h = 12;       //convert 0:xx to 12:xx
  Time result(h, totalMin % 60);
  return result;
}
//in a driver:
  Time t1(12,5);
  Time t2(2,50);
  Time t3 = t1+t2;
  t3.display();
```

Output: 2:55

The this pointer

- this: a predefined pointer that can be used in a class's member function definitions
- this always points to the instance (object) of the class whose function is being executed.
- Use this to access member vars that may be hidden by parameters with the same name:

```
Time::Time(int hour, int minute) {
   // Time *this; implicit decl
  this->hour = hour;
  this->minute = minute;
}
```

- Or return *this from a function.

Overloading Prefix ++ for Time

```
class Time {
  private:
    int hour, minute;
  public:
    Time operator++ ();
};
Time Time::operator++ (Time right) { //Note: 12%12 = 0
  if (minute == 59) {
    minute = 0;
    if (hour == 12)
      hour = 0;
  } else {
    minute++;
  }
  return *this;  //this points to the calling instance
}
//in a driver:
  Time t1(12,55);
  Time t2 = ++t1;
  t1.display(); cout << " ";  t2.display();
```

Output: 12:56 12:56

Problem with default definition of operator=

- C++ provides a default definition of operator=
- It uses member-wise assignment (like the default copy constructor)
- If you use it with IntCell (see week5), you will end up with a shared value:

```
IntCell object1(5);
IntCell object2(0);
object2 = object1;
// object2.storedValue = object1.storedValue

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

Output: 13
13

Overload = for IntCell

Now = for IntCell will not use member-wise assignment

```
class IntCell {
   private:
      int *value;
   public:
      IntCell(const IntCell &obj);
      IntCell(int);
      ~IntCell();
      int read() const;
      void write(int);
      void operator= (IntCell rhs);
};

void IntCell::operator= (IntCell rhs) {
   write(rhs.read());
}

//in a driver:
IntCell object1(5), object2(0);
object2 = object1;
object2.write(13);
cout << object1.read() << endl;
```

Output: 5

Templates: Type independence

- Many functions, like finding the maximum of an array, do not depend on the data type of the elements.
- We would like to re-use the same code regardless of the item type...
- **without** having to maintain duplicate copies:
  - maxIntArray (int a[]; int size)
  - maxFloatArray (float a[]; int size)
  - maxCharArray (char a[]; int size)

Generic programming

- Writing functions and classes that are type-independent is called generic programming.
- These functions and classes will have one (or more) extra parameter to represent the specific type of the components.
- When the stand-alone function is called the programmer provides the specific type:

```
max<string> students(array,size);
```


Templates

- C++ provides templates to implement generic stand-alone functions and classes.
- A function template is not a function, it is a design or pattern for a function.
- The function template makes a function when the compiler encounters a call to the function.
  - Like a macro, it substitutes appropriate type

Example function template
swap

```
template <class T>
void swap (T &lhs, T &rhs) {
   T tmp = lhs;
   lhs = rhs;
   rhs = tmp;
}
int main() {
  int x = 5;
  int y = 7;
  string a = “hello”;
  string b = “there”;
  swap <int> (x, y);     //int replaces Object
  swap <string> (a, b); //string replaces Object
  cout << x << “  “ << y << endl;
  cout << a << “  “ << b << endl;
}
```

Output:

```
7 5
there  hello
```

Notes about the example

- The header: template <class T>
  - class is a keyword. You could also use typename: template <typename T>
- T is the parameter name. You can call it whatever you like.
  - it is often capitalized (because it is a type)
  - names like T and U are often used
- The parameter name (T in this case) can be replaced ONLY by a type.

Simple example, class template
MemoryCell

```
template <class T>
class MemoryCell        {
   private:
      T storedValue;       //stores the memory cell contents

   public:
      // Construct a MemoryCell.
      MemoryCell ( T initVal)
      { storedValue = initVal; }

      // public methods
      T read ()
      { return storedValue; }
      void write (T x)
      { storedValue = x; }

};
```


Simple example, class template
MemoryCell

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

int main() {
   MemoryCell<int> m;
   m.write(5);
   cout  << "Cell contents are " << m.read() << endl;
   MemoryCell<string> m1;
   m1.write("abc");
   cout  << "Cell contents are " << m1.read() << endl;
}
```

Output:

```
Cell contents are 5
Cell contents are abc
```


Example 2, class template
vector: class decl

```
// A barebones vector ADT

template <typename T>
class vector {
private:
    T* data;           //stores data in dynamically allocated array
    int length;        //number of elements in vector
    int capacity;      //size of array, to know when to expand
    void expand();     //to increase capacity as needed
public:
    vector(int initial_capacity);
    ~vector();
    void push_back(T);     //add a T to the end
    T pop_back();          //remove a T from the end and return
    T operator[](int k);   // access the T in the kth position
};
```

Note: not ALL types should be replaced by the type variable T

This is NOT the same as SimpleVector in the Gaddis book.

Example 2, class template

vector, function definitions

```
template <typename T>
vector<T>::vector(int init_cap) {
    capacity = init_cap;
    data = new T[capacity];
    length = 0;
}
template <typename T>
void vector<T>::push_back(T x) {
    if (capacity == length)
       expand();
    data[length] = x;
    length ++;
}
template <typename T>
T vector<T>::pop_back() {
    assert (length > 0);
    length--;
    return data[length];
}
```


Example 2, class template

vector, function definitions

```
template <typename T>
T vector<T>::operator[](int k) {
   assert (k>=0 && k<length);
   return data[k];
}
template <typename T>
void vector<T>::expand() {
    capacity *= 2;
    T* new_data = new T[capacity];
    for (int k = 0; k < length; k += 1)
       new_data[k] = data[k];
    delete[] data;
    data = new_data;
}
template <typename T>
void vector<T>::~vector() {
   delete [] data;
}
```


Simple example, class template

using vector

```
int main() {
   vector<string> m(2);
   m.push_back("As");
   m.push_back("Ks");
   m.push_back("Qs");
   m.push_back("Js");
   for (int i=0; i<4; i++) {
      cout << m[i] << endl;
   }
}
```

Output:

```
As
Ks
Qs
Js
```


Class Templates and .h files

- Template classes cannot be compiled separately
  - Machine code is generated for a template class only when the class is instantiated (used).
    - When you compile a template (class declarations + functions definitions) it will not generate machine code.
  - When a file using (instantiating) a template class is compiled, it requires the **complete** definition of the template, including the function definitions.
  - Therefore, for a class template, the class declaration AND function definitions must go in the header file.
  - It is still good practice to define the functions outside of (after) the class declaration.