Operator Overloading and Templates

Week 6
Gaddis: 8.1, 14.5, 16.2-16.4

CS 5301
Spring 2015
Jill Seaman

Linear Search

- **Search**: find a given target item in an array, return the index of the item, or -1 if not found.
- **Linear Search**: Very simple search method:
  - Compare first element to target value, if not found then compare second element to target value . . .
  - Repeat until: target value is found (return its index) or we run out of items (return -1).

Linear Search in C++

```cpp
int searchList (int list[], int size, int target) {
    int position = -1; // position of target
    for (int i=0; i<size; i++)
        if (list[i] == target) // found the target!
            position = i; // record which item
    return position;
}
```

Is this algorithm correct? Does the if need an else?

How can we make it more efficient?

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).
  - Optional for overloading operators

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 class: Time

class declaration with functions defined inline

We will use this for operator overloading examples:

```cpp
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; }
};
```

Example: minus for Time objects

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

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

Output: 260

Subtraction
Overloading == and < for Time

```cpp
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

```cpp
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;
    h = h%12;               //keep it between 0 and 11
    if (h==0) h = 12;       //convert 0:xx to 12:xx
    int resultMin = totalMin % 60;
    return Time(h, resultMin);
}
//in a driver:
Time t1(12,5);
Time t2(2,50);
Time t3 = t1+t2;
t3.display();
```

Overloading Prefix ++ for Time

```cpp
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 = 1; else hour++;
    } else {
        minute++;
    }
    return *this;  //this points to the calling instance
}
//in a driver:
Time t1(12,5);
Time t2 = ++t1;
t1.display(); cout << " ";
t2.display();
```

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:

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

- Or return *this from a function.
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>(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

```c
  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.

Example, class template
vector: class decl

```
// A barebones vector ADT

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

template <typename T>
class vector {
private:
  T* data;          //stores data in dynamically allocated array
  int length;       //number of elements in vector
public:
  vector();
  vector(int);
  ~vector();
  int size() const;
  T getElementAt(int k); //access the T in the kth position
  void setElementAt(int k, T v); //set the T in the kth position
};

This is similar to the SimpleVector in the Gaddis book.
```

Example, class template
vector, function definitions

```
template <typename T>
vector<T>::vector() {
  length = 0;
  data = NULL;
}

template <typename T>
vector<T>::vector(int s) {
  length = s;
  data = new T[length];
  //note that the elements are not initialized
}

template <typename T>
vector<T>::vector() {
  delete [] data;
}
```

Example, class template
vector, function definitions

```
template <typename T>
int vector<T>::size() const {
  return length;
}

template <typename T>
T vector<T>::getElementAt(int k) {
  assert (k>=0 && k<length);
  return data[k];
}

template <typename T>
void vector<T>::setElementAt(int k, T v) {
  assert (k>=0 && k<length);
  data[k]=v;
}
```
Example, class template
using vector

```cpp
int main() {
    vector<string> m(3);
    m.setElementAt(0, "K");
    m.setElementAt(1, "Q");
    m.setElementAt(2, "J");
    for (int i = 0; i < m.size(); i++) {
        cout << m.getElementAt(i) << endl;
    }
}
```

Output:
```
K
Q
J
```

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.

Sample Problem 1

**FeetInches Modification:** Modify the FeetInches class discussed in this chapter so it overloads the following operators:
- `<=`
- `>=`
- `!=`

Demonstrate the class’s capabilities in a simple program.

Sample Problem 2

**SimpleVector Modification:** Modify the SimpleVector class template presented in this chapter to include the member functions `push_back` and `pop_back`. These functions should emulate the STL vector class member functions of the same name. (See Table 16-5.) The `push_back` function should accept an argument and insert its value at the end of the array. The `pop_back` function should accept no argument and remove the last element from the array. Both of these should change the size of the vector appropriately. Test the class with a driver program.