Operator Overloading and Templates

Week 6

Gaddis: 8.1, 14.5, 16.2-16.4

CS 5301
Fall 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++

**first attempt**

```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?
Is this algorithm efficient (does it do unnecessary work)?

**second attempt**

```cpp
int searchList (int list[], int size, int value) {
    int index=0; //index to process the array
    int position = -1; //position of target
    bool found = false; //flag, true when target is found
    while (index < size && !found) {
        if (list[index] == value) //found the target!
            found = true; //set the flag
            position = index; //record which item
        index++; //increment loop index
    }
    return position;
}
```

Is this algorithm correct?
Is this algorithm efficient (does it do unnecessary work)?
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:
  ```cpp
  int operator-(const Time &right);
  ```
- 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:
  ```cpp
  int minutes = object1.operator-(object2);
  ```
- It can also be invoked using the more conventional syntax for operators:
  ```cpp
  int minutes = object1 - object2;
  ```
- Both call the same function (operator-), from the perspective of object1 (on the lefthand side).

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

Output: 260
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);
}
```

```cpp
//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);
};
```

```cpp
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 min = totalMin % 60;
    Time result(h, min);
    return result;
}
```

```cpp
//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:

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

- Or return `*this` from a function.
Overloading Postfix ++ for Time

class Time {
private:
    int hour, minute;
public:
    Time operator++ (int);
};
Time Time::operator++ (int) {
    Time temp(hour,minute);  //save this to return it
    if (minute == 59) {
        minute = 0;
        if (hour == 12) hour = 1; else hour++;
    } else {
        minute++;
    }
    return temp;  //this points to the calling instance
}

//in a driver:
Time t1(12,55);
Time t2 = t1++;
t1.display(); cout << “ “; t2.display();

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

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 T
    swap <string> (a, b); //string replaces T
    cout << x << "  " << y << endl;
    cout << a << "  " << b << endl;
}

Output:
7 5
there hello

Notes about the function template 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

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 getElementAt(int k); //access the T in the kth position
};

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

Notes about the function template example

Example 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];
}

assert(e): if e is false, it causes the execution of the program to stop (exit). Requires #include<cassert>
Example class template
vector, function definitions

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

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.getElementAt(i) << endl;
    }
}
```

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.