Operator Overloading and Templates

Week 6
Gaddis: 8.1, 14.5, 16.2-16.4

CS 5301
Fall 2014
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;
}
```

**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)?**
#include <iostream>
using namespace std;

int searchList(int[], int, int);

int main() {
    const int SIZE = 5;
    int idNums[SIZE] = {871, 750, 988, 100, 822};
    int results, id;

    cout << "Enter the employee ID to search for: ";
    cin >> id;

    results = searchList(idNums, SIZE, id);
    if (results == -1) {
        cout << "That id number is not registered\n";
    } else {
        cout << "That id number is found at location ";
        cout << results + 1 << endl;
    }
}

Example class: Time
class declaration with functions defined inline

We will use this for operator overloading examples:

class Time {
    private:
        int hour;
        int minute;
    public:
        Time();
        Time(int hr, int min);
        void setHour(int hr);
        void setMinute(int min);
        int getHour() const;
        int getMinute() const;
        void display() const;
};

We will use this for operator overloading examples:

class Time {
    private:
        int hour;
        int minute;
    public:
        Time();
        Time(int hr, int min);
        void setHour(int hr);
        void setMinute(int min);
        int getHour() const;
        int getMinute() const;
        void display() const;
};

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);

- 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;
  ```

- 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);
};
```

```cpp
time1 = time1 - time2;
```

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

Overloading == and < for Time

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

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

//in a driver:
```cpp
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;
    if (h==0) h = 12;       //convert 0:xx to 12:xx
    Time result(h, totalMin % 60);
    return result;
}
```

//in a driver:
```cpp
Time t1(12,5), 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 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 = 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();
```

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:
  ```cpp
  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

```cpp
template <class T>
void swap (T &lhs, T &rhs) {
    T tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

```cpp
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

```cpp
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

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

*assert(e): if e is false, it causes the execution of the program to stop (exit). Requires #include<cassert>*
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;
    }
}

Output:
As
Ks
Qs
Js