Operator Overloading, Lists and Templates

Week 6
Gaddis: 14.5, 16.2-16.4

CS 5301
Fall 2016
Jill Seaman

Operator Overloading

- Operators such as =, +, <, … can be defined to work for objects of a programmer-defined class
- The function names are operator followed by the operator symbol:
  - operator+ to define the + operator, and
  - operator= to define the = operator
- Otherwise they are like normal member functions:
  - Prototype goes in the class declaration
  - Function definition goes in implementation file

Overloaded Operator Prototype

- Prototype:
  ```cpp
  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 functions can be invoked (called) as a regular member function:
  ```cpp
  int minutes = object1.operator-(object2);
  ```
- They can also be invoked using the more conventional syntax for operators:
  ```cpp
  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 (object2 is the argument).
Example: minus for Time objects

I decide I want \( \text{time1} - \text{time2} \) to be an int, equal to the number of minutes between the times.

```cpp
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;
```
```}
```

Subtraction

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 totalMin % 60);
    return result;
}
```

// in a driver:
Time t1(12,5);
Time t2(2,50);
Time t3 = t1+t2;
t3.display();
```

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 timel(12,20), time2(12,21);
if (timel<time2) cout << “correct” << endl;
if (timel==time2) cout << “correct again”<< endl;
```

Overloading == and < for Time

The List Type

- A List is an ordered collection of items of some type \( T \):
  - each element has a position in the list
  - duplicate elements are allowed
- List is not a C++ data type. It is abstract/conceptual. It can be implemented in various ways (using arrays, STL vectors, linked lists…)
- We will implement using arrays, but we want to be able to use the same code for a List of int, a List of string, a List of float, etc.
The List Type: operations

- Common operations for lists:
  - create an empty list
  - `void add(T item)`: add item to end of the list.
  - `int size()`: returns the number of items in the list.
  - `void set(int pos, T item)`: change value of item in position pos to item (if `pos>=0` and `pos<size`).
  - `T get(int pos)`: return the item at position pos (if `pos>=0` and `pos<size`).
  - `bool isEmpty()`: true if the list is empty
  - `bool contains(T item)`: true if item is in the list
  - `void remove(T item)`: removes item from the list.

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;
}

Notes about C++ templates

- The template prefix: `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 implementation of the List abstract data type

template <class T>
class SimpleVector {
private:
    T *aptr;          // To point to the allocated array
    int arraySize;    // Number of elements in the array
public:
    SimpleVector()        { aptr = NULL; arraySize = 0;}
    SimpleVector(int, T);     
    SimpleVector(const SimpleVector &);
    ~SimpleVector();
    int size() const { return arraySize; }
    T getElement(int position);      
    void setElement(int position, T item);
};

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

Example class template

constructor, copy constructor

template <class T>
SimpleVector<T>::SimpleVector(int s, T item) {
    arraySize = s;
    if (arraySize > 0)
        aptr = new T [s];
    for (int count = 0; count < arraySize; count++)
        *(aptr + count) = item;
}

template <class T>
SimpleVector<T>::SimpleVector(const SimpleVector &obj) {
    arraySize = obj.arraySize;
    if (arraySize > 0)
        aptr = new T [arraySize];
    for(int count = 0; count < arraySize; count++)
        *(aptr + count) = *(obj.aptr + count);
Example class template
destructor, getElement, setElement

```c++
template <class T>
SimpleVector<T>::~SimpleVector() {
    if (arraySize > 0)
        delete[] aptr;
}

template <class T>
T SimpleVector<T>::getElement(int position) {
    assert (0 <= position && position < arraySize);
    return aptr[position];
}

template <class T>
void SimpleVector<T>::setElement(int position, T item) {
    assert (0 <= position && position < arraySize);
    aptr[position] = item;
}
```

Example class template
using vector

```c++
int main() {
    SimpleVector<string> strV(2,"");
    strV.setElement(0, "one");
    strV.setElement(1, "two");
    SimpleVector<int> intV(2,0);
    intV.setElement(0,1);
    intV.setElement(1,2);
    for (int i=0; i<2; i++) {
        cout << strV.getElement(i) << endl;
        cout << intV.getElement(i) << endl;
    }
}
```

Output:
```
one
1
two
2
```

Class Templates and .h files

- Template classes cannot be compiled separately
- 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.