Operator Overloading
& Templates

Week 6
Gaddis: 14.5, 16.2-16.4

CS 5301
Spring 2018
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:
  
  ```
  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:
  ```
  int minutes = object1.operator-(object2);
  ```

- They 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` (`object2` is the argument).
**Example: minus for Time objects**

- I decide I want `time1-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);
};
```

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

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

**Output: 260**

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

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

**Output: 2:55**

**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);
}
```

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

**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
```

```cpp
template <class T>
void mySwap (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";
    mySwap <int> (x, y);   //int replaces T
    mySwap <string> (a, b); //string replaces T
    cout << x << " " << y << endl;
    cout << a << " " << b << endl;
}
```

Output:

```
7 5
there hello
```

Notes about C++ templates

- The **template prefix**: `template <class T>`
  - `class` is a keyword. You could also use `typename`:
    ```cpp
    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
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 s, T item) {
    arraySize = s;
    if (arraySize > 0)
      aptr = new T[arraySize];
    for (int count = 0; count < arraySize; count++)
      *(aptr + count) = item;
  }
  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);
  }
  ~SimpleVector() {
    if (arraySize > 0)
      delete [] aptr;
  }
  T getElement(int position) {
    assert (0 <= position && position < arraySize);
    return aptr[position];
  }
  void setElement(int position, T item) {
    assert (0 <= position && position < arraySize);
    aptr[position] = item;
  }
};

Example class template
constructor, copy constructor
template <class T>
SimpleVector<T>::SimpleVector(int s, T item) {
  arraySize = s;
  if (arraySize > 0)
    aptr = new T[arraySize];
  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
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
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.

SimpleVector Modification

Add these functions to SimpleVector:

- push_back() Accepts as an argument a value to be inserted after the last element. (Pushed onto the back of the vector). Makes aptr array bigger.
- pop_back() Removes the last element from the vector. Makes aptr array smaller.
- Hint: both of these operations require allocating a new array of a different size and copying elements from the old array to the new one.