Templates and generic programming

CS 5301
Fall 2013

Jill Seaman

Gaddis: sections 16.2-16.4
Main + Savitch: sections 6.1-6.2
Weiss: chapter 3

Type independence

- Many algorithms like search, sort, or swap do not depend on the type of the elements/items.
- We would like to re-use the same code regardless of the item type...
- **without** having to maintain duplicate copies:
  - sortIntArray (int a[]; int size)
  - sortFloatArray (float a[]; int size)
  - sortCharArray (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, or class is instantiated, the programmer provides the specific type:

```
vector<string> students (20);
vector<double> dailySales (365);
```


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 Object>
void swap (Object &lhs, Object &rhs) {
   Object 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 Object>
  - class is a keyword. You could also use typename: template <typename Object>
- Object 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 (Object in this case) can be replaced ONLY by a type.

Simple example, class template
MemoryCell

```
template <class Object>
class MemoryCell        {
   private:
      Object storedValue;       //stores the memory cell contents

   public:
      // Construct a MemoryCell.
      MemoryCell ( Object initVal)
      { storedValue = initVal; }

      // public methods
      Object read ()
      { return storedValue; }
      void write (Object x)
      { storedValue = x; }

};
```


Simple example, class template
MemoryCell

```
#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);
    void push_back(T);    //add a T to the end
    T pop_back();         //remove a T from the end and return
    T operator[](int k);  // access the T in the kth position
};
```


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];
}
```


Example 2, class template
vector, function definitions

```
template <typename T>
T vector<T>::operator[](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;
}
```


Simple 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[i] << endl;
   }
}
```

Output:

```
As
Ks
Qs
Js
```

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.