Chapter 3

CS 3358
Spring 2015
Jill Seaman

Sections 3.1, 3.2, 3.3, 3.4

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 numValues)`
  - `sortFloatArray (float a[]; int numValues)`
  - `sortCharArray (char a[]; int numValues)`
- the code to define these would all be identical.

Generic programming

- Writing functions and classes that are type-independent is called generic programming.
- These functions and classes will have an 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);
  ```
- Note: vector is a templated class.

Templates

- C++ provides templates to implement generic 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 the appropriate type for the type variable.
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;
}
```

Notes about the previous example

- The header: template <class Object>
  - `class` is a keyword. You could also use `typename`:
    template <typename Object>
  - Object is the type 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.

Notes about the example

- Normal syntax to call the templated function includes the type in angled brackets: `<int>
  - swap<int> (x,y);
  - swap<string> (a,b);
- It’s not necessary to specify the type when the compiler is capable of figuring it out from context.
  - swap (x,y);
  - swap (a,b);

How function templates work

- The compiler will not use (compile) the pattern unless/until it encounters a call to the function.
  - At that point, the compiler substitutes your type for the type variable, and then compiles the newly generated function as if you’d written the function that way yourself.
- What happens if I instantiate the same template multiple different ways (with more than 1 type)?
  - It is just function overloading, you get two or more functions with the same name, but with different parameter types!
Class Templates

- Template classes work similarly to template functions with the following exceptions:
  - The compiler will never guess at type argument for a template class, you must always use `<...>` during object declaration.
  - Classes cannot be “overloaded”, but the compiler will permit you to instantiate the same template class in multiple ways.
  - Each distinct instantiation results in a completely distinct class!
    (with its own copy of the static data members, for example).
  - The member functions in a template class are template functions (their definitions require a template header)

Simple example, class template 
MemoryCell (formerly IntCell)

```cpp
#include <iostream>
using namespace std;

int main() { 
    MemoryCell<int> m;
    m.write(5);
    cout << "Cell contents are " << m.read() << endl;
    
    MemoryCell<string> s;
    s.write("five");
    cout << "Cell contents are " << s.read() << endl;
} 
```

Output:
Cell contents are 5
Cell contents are five

Example 2, class template 
vector: class decl

```cpp
// A barebones vector ADT
// T: must have default constructor and operator=

template <typename T>
class vector {
public:
    vector(int initial_capacity=8);
    void push_back(T);
    T pop_back();
    T operator[](int k);
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
};
```

Note: not ALL types should be replaced by the type variable T
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(void) {
    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 (declaring an object of) 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.
  
Could have used pop_back, it works too. But...