Templates and generic programming
Chapter 3

CS 3358
Summer II 2013
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)

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);`

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 appropriate type
Example function template

```cpp
template <class Object>
void swap (Object &lhs, Object &rhs) {
    Object 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 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
- **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.

Notes about the example

- Normal syntax to call the templated function includes the type: `<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 performs the text substitution you asked for, and then compiles the newly generated function as if you’d written that function yourself.
- What happens if I instantiate the same template multiple different ways?
  - It is just function overloading, you get two or more functions with the same name, but with different arguments!
Simple example, class template
MemoryCell (formerly IntCell)

```cpp
// Object: must have zero-parameter constructor and operator=

template <class Object>
class MemoryCell
{
public:
    // Construct a MemoryCell.
    MemoryCell ( const Object & initVal = Object () )
        : storedValue (initVal ) { }

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

private:
    Object storedValue;       //stores the memory cell contents
};
```

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

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

Output:
Cell contents are 5
```

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 `<...>`
  - 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 (defs require template header)

Example 2, class template
vector: class decl

```cpp
// A barebones vector ADT
// T: must have zero-parameter 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(void); // to increase capacity as needed
};
```
Example 2, class template
vector, function definitions

```cpp
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>
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

```cpp
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 |

Could have used pop_back, it works too. But . . .

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.
```
#include <utility>
#include <vector>
using namespace std;

int main() {
    pair<int,string> p1;
    pair<int, int> p2(1, 12);
    p1 = make_pair(0,"Zero");
    cout << p1.first << "  " << p1.second << endl;
    cout << p2.first << "  " << p2.second << endl;

    //to declare a list of pairs as a vector:
    vector<pair<int,double>> listOfPairs;
    listOfPairs.push_back(make_pair(1,1.2));
```