CS368 Lecture 14

Wednesday 3 December 2014

Reminders:

- P5 due Friday
- Reading: Chapter 10

Last class:

- `getline` functions
- (Function) Templates

Today:

- Templates Review
- Class Templates
- STL intro
    - Containers
    - Iterators

Templates (Review)

Parameterizing types

One code for many types (recall Java generics)

Function templates

- Used when a function can take in different types in its parameter list
- All those types must conform to interface used in the function body

Example declarations and definitions:

```
template <typename T>
void foo(T& someParameter);


template <typename U>
const U& foo2(const vector<U>& vec) { … }


template <typename T, typename U>
void foo3(T& arg1, U& arg2);
```

Function Templates: Example

```
template <typename T>
void swap(T &a, T &b){
    T temp = a;
    a = b;
    b = temp;
}
```

```
template <typename T>
void printReverse(const vector<T> & list);
```

```
template <typename someType>
const someType& getMiddle(const vector<someType>& list);
// note the return type. Why?
```

```
int main(){
    Fraction f1(2, 3);
    Fraction f2(3, 4);
    swap(f1, f2);

    vector<Fraction> v4;
    v4.push_back(Fraction(2,3));
    v4.push_back(Fraction(1,7));
    printReverse(v4);
    return 0;
}
```

How does function templating work?

- Function templates are not actual functions, just patterns (“templates”)
- Compiler does the work (“real” generics, unlike Java).
- Compiler generates separate instances of the function for each distinct type it is invoked with.
- Debugging caveat

```
void printReverse(const vector<int> & list){
    int vSize = list.size();
    for (int i=vSize-1; i>0; i--)
        cout << list[i]<< ", " ;
    cout << list[0] << endl;
}

void printReverse(const vector<string> & list){
    int vSize = list.size();
    for (int i=vSize-1; i>0; i--)
        cout << list[i]<< ", " ;
    cout << list[0] << endl;
}

void printReverse(const vector<Fraction> & list){
    int vSize = list.size();
    for (int i=vSize-1; i>0; i--)
        cout << list[i]<< ", " ;
    cout << list[0] << endl;
}

const someType& returnMiddle(const vector<int> & list){
    int midIndex = list.size() / 2;
    return list[midIndex];
}

const someType& returnMiddle(const vector<string> & list){
    int midIndex = list.size() / 2;
    return list[midIndex];
}
```

Note: No `template` keyword.

Class Templates

Members of classes that could be any of a number of types

Recall `Stack<Integer>` from Java

`Pair` example

- Why are we using references in our parameters and return types?
- All in one `.h` file (see 7.4.1 for details on how to split this)

Example `main.cpp`:

```
int main() {

    Pair<int> p1 = Pair<int>(2,6);
    Pair<double> p2 = Pair<double>(2.55, 6.43);
    Pair<string> p3 = Pair<string>("Stevie", "Sam");

    p1.print();
    if (p1.same())
        cout << "same" << endl;
    else
        cout << "not same" << endl;


    p2.print();
    cout << p2.getFirst() << endl;

    p3.print();

    return 0;
}
```

More on Templated Classes

Templated class with one of the parameters as a primitive:

```
template <typename Object, int size>
class Stack{ … };

Stack<string, 20> toDoList;
```

Templated class with two different parameterized types:

```
template <typename KeyType, typename ValueType>
class Dictionary { … };

Dictionary<string, Date> birthdays;
Dictionary<int, string> zipCodeList;
```

Default types for template parameters:

```
template <typename Object = char, int size = 4096>
Class Buffer{ … };

Buffer <int> b1;  // same as Buffer<int, 4096>
Buffer <> b2      // same as Buffer<char, 4096>
```

Aliases in C++11

A way to do “templated typedefs”

```
template <typename T>
using aliasName = definition
```

E.g. : *aliasName* might be `Vector`, and *definition* might be `Matrix<N, 1>`

Recall that to compile with newer C++ standard, you pass g++ the switch –
`std=c++0x`

Standard Template Library (STL)

Reference: http://www.sgi.com/tech/stl/ (bookmark it!)

STL is a library of **containers**, **algorithms**, and **iterators.** It provides powerful reusable components implementing common data structures and algorithms.

STL makes heavy use of **pointers**, **templates**, **iterators**, **operator overloading** (basically, almost everything we've learnt so far)

STL samples (from Keith Schwarz):

Create a list of random numbers, sort it, and print it in *four* lines of code!:

```
vector<int> myVector(NUM_INTS);
generate(myVector.begin(), myVector.end(), rand);
sort(myVector.begin(), myVector.end());
copy(myVector.begin(), myVector.end(),
    ostream_iterator<int>(cout, "\n"));
```

Open a file and print its contents in two lines of code:

```
ifstream input("my-file.txt");
copy(istreambuf_iterator<char>(input),
    istreambuf_iterator<char>(),
    ostreambuf_iterator<char>(cout));
```

Convert a string to upper case in one line of code:

```
transform(s.begin(), s.end(), s.begin(), ::toupper);
```

Containers

Sequence (list-type) containers:

- `vector` → `stack (adapter, derived, additional members)`
- `deque` → `queue, priority_queue (adapter class, derived)`
- `list` (doubly-linked) and `forward_list` (singly-linked)

Associative containers

- `set`
- `multiset`
- `map`
- `multimap`

  These are all ***ordered*** – for the unordered versions, prefix `unordered_` to the container name.

  For the ***hash*** versions, prefix `hash_` to the container name.

All containers implement at least a (proper) copy constructor, (proper) destructor, some kind of add operation (e.g. `void insert()`), `void clear()`, `int size()`, and `bool empty()`. Sequence containers also implement

Keep in mind:

- These are templated classes
- Make sure object type in container supports a set of basic operations: copy constructor, `operator=`, and comparisons such as `operator<` and `operator==`

STL Containers: An Example

```
#include<vector>
#include<set>
#include <map>

using namespace std;

int main() {
    vector<int> vec(10, 0);
    set<string> sset;
    map<string, double> scores;

    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(7);
    vec.push_back(4);
    cout << "The third element in vec is " << vec[2] << endl;
    vec.erase(vec.begin() + 12); // delete the 13th element
    cout << "The 13th element in vec is " << vec[12] << endl;

    sset.insert(“Hello”);
    sset.insert(“World”);
    sset.insert(“from”);
    sset.insert(“Madison”);

    scores[“Alice”] = 91.5;
    scores[“Bob”] = 85.9;
    string student = "Bob";
    cout << student << "'s score: "<< scores[student] << endl;

    return 0;
}
```

STL Iterators

What are iterators and what are they good for?

- Objects pointing to other objects (in containers)
- Interface between a container and an algorithm

Getting an iterator from a container:

- `iterator begin();`
- `const_iterator begin() const;`
- `iterator end();`
- `const_iterator end() const;`

Remember, iterator types are specific to the object they "point" to (just like Java).

(Some) Iterator operations:

- `++iter`
- `*iter`
- `operator==`

An example:

```
list<double> L;
L.push_back(1.2);
L.push_front(3.4);
L.insert(L.begin(), 5.6);
L.insert(L.end(), 7.8);

list<double>::const_iterator iter;
for (iter = L.begin(); iter != L.end(); ++iter)
    cout << *iter << " ";
cout << endl;
```

Iterator Kinds and Operations

All iterators support ++ (both prefix and postfix), *, ==

```
forward_list
```

Bidirectional Iterators support --

```
list, set, map
```

Random Access Iterators support +=k

```
vector, deque, array
```

Input/Output Iterators