CS32 Discussion
Section 1B
Week 6

TA: Zhou Ren

Template Classes

```
class Pair {
    public:
       Pair();
       Pair(int firstValue,
            int secondValue);
       void setFirst(int newValue);
       void setSecond(int newValue);
       int getFirst() const;
       int getSecond() const;
    private:
       int m_first;
       int m_second;
};
```

- This class works only with integers.
- Can we make a "generic" Pair class? (Note that typedef does not do the job for us.)

Template Classes

```
template<typename T>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            T secondValue);
       void setFirst(T newValue);
       void setSecond(T newValue);
       T getFirst() const;
       T getSecond() const;
    private:
       T m_first;
       T m_second;
};
```

- Here we go.

```
Pair<int> p1;
Pair<char> p2;
```

Template Classes

```
template<typename T, U>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            U secondValue);
       void setFirst(T newValue);
       void setSecond(U newValue);
       T getFirst() const;
       U getSecond() const;
    private:
       T m_first;
       U m_second;
};
```

- More than one type:

```
Pair<int, int> p1;
Pair<string, int> p2;
```

Template Classes

```
template<typename T>
void Pair<T>::setFirst(T newValue)
{
    m_first = newValue;
}
```

- Member functions should be edited as well.

Template Specialization

```
template<>
class Pair<char> {
    public:
       Pair();
       Pair(char firstValue,
            char secondValue);
       void setFirst(char newValue);
       void setSecond(char newValue);
       char getFirst() const;
       char getSecond() const;
       void uppercase();
    private:
       char m_first;
       char m_second;
};
```

- Make an exception.

```
Pair<char> p1;
Pair<int> p2;

p1.uppercase(); (O)
P2.uppercase(); (X)
```

Template Functions

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

- Pretty much the same trick.
- Call the function without <>. The types are automatically detected.

```
int x = 2, y = 3;
swap(x, y);

char j = 'c', k = 'm';
swap(j, k);
```

Note

```
// From Prof. Smallberg's slide
template<typename T>
T minimum(const T& a, const T& b)
{
    if (a < b)
        return a;
    else
        return b;
}
```

- When you are not changing the values of the parameters, make them const references to avoid potential computational cost.

STL

- **S**tandard **T**emplate **L**ibrary
  - Library of commonly used data structures.
    - `vector` (array)
    - `set` (binary search tree – will learn it soon)
    - `list` (doubly linked list)
    - `map`
    - `stack`
    - `queue`

STL

- A few common functions:
  - `.size() .empty()`
- For a container that is neither stack nor queue:
  - `.insert() .erase() .swap() .clear()`
- For list/vector:
  - `.push_back() .pop_back()`
- For set/map:
  - `.find() .count()`
- ... and you've seen stacks and queues.

STL Example

```
#include <list>
using namespace std;
int main()
{
    list<int> a;
    for (int i = 0; i < 10; i++)
        a.push_back(i);
    cout << a.size() << endl;  // prints 10
}
```

STL Example

```
#include <vector>
using namespace std;
int main()
{
    vector<int> a;
    for (int i = 0; i < 10; i++)
        a.push_back(i);
    cout << a.size() << endl;  // prints 10
}
```

STL Iterators

- Suppose I want to iterate through elements in a container:
- For an array, you would do:

```
int arr[100];
...
for (int i = 0; i < 100; i++)
{
    cout << arr[i] << endl;
}
```

- But how do we do this for a list or a set?

STL Iterators

- "abstract" way of traversing through elements
- `structure<data type>::iterator` -- pointer to an element in a container
- `.begin()` gives you the "first" element in the container
- .end() indicates that the iteration is complete

```
list<int> l;
for (list<int>::iterator it = l.begin(); it != l.end(); it++)
{
   cout << *it << " ";   // Note that '*'!!
}
```

STL Iterators

- Use **const_iterator** when the container is constant!

```
void func(const list<int> &l)
{
  for (list<int>::const_iterator it = l.begin(); it != l.end(); it++)
  {
     cout << *it << " ";
  }
}
```

STL Iterators

- If you need to iterate in the reverse direction, you can optionally use **rbegin()** and **rend()**:

```
void func(const list<int> &l)
{
  for (list<int>::const_iterator it = l.rbegin(); it != l.rend(); it++)
  {
      cout << *it;       // Note that '*'!!
  }
}
```

- Note that you're still using it++ to "advance" the iterator.

STL Iterators

- Iterators are used to call some important functions like insert() and erase():

```
list<int> myList;
myList.push_back(0);   // 0
myList.push_back(1);   // 0 1

list<int>::iterator it = myList.begin();
it++;
myList.insert(it, 30); // 0 30 1, it still points to 1.
myList.erase(it);      // 0 30
```

Quick Note on `erase()`

- Suppose you're given a structure and would like to remove all elements that satisfy a certain condition:

```
for (list<int>::iterator it = l.begin(); it != l.end(); it++)
{
  if (*it == 10)
  {
     l.erase(it);   // remove the element pointed by it
  }
}
```

- What is the problem here?

Quick Note on erase()

- Suppose you're given a structure and would like to remove all elements that satisfy a certain condition:

```
for (list<int>::iterator it = l.begin(); it != l.end();)
{
  if (*it == 10)
  {
     it = l.erase(it);   // remove the element pointed by it
  }
  else
     it++;
}
```

- erase() returns an iterator for the next element.

STL

- You don't have to memorize names of member functions for each – you can just look things up when you need to.

  e.g. http://www.cplusplus.com/reference/stl/
- But **do** remember:
  - what data structure each container implements
  - how to use iterators