CS368 Lecture 13

Wednesday 26 November 2014

Reminders:

- P5 due Friday next week
- Reading: Chapter 7

Last class:

- Manipulators
- C strings

Today:

- `getline` functions
- Templates

Using `getline()`

`getline()` the member method

`getline()` the freestanding function

Templates

Parameterizing types

One code for many types (recall Java generics)

Function templates vs. Class templates

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);
```

Recall that you can declare a function, then write code invoking it, then define it.

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`