Lecture Notes 5

Templates

- Template – The main mechanism for generic programming in C++
- Limits of Overloading – Overloading is powerful, but requires explicit definition for every type
  - Example:

```
int Compare(const std::string &l, const std::string &r){
    if(l < r){
        return -1;
    }
    if(l > r){
        return 1;
    }
    return 0;
}

int Compare(const int &l, const int &r){
    if(l < r){
        return -1;
    }
    if(l > r){
        return 1;
    }
    return 0;
}
```

- Function Template – Formula for generic programming
  - Example:

```
template <typename T>
int Compare(const T &l, const T &r){
    if(l < r){
        return -1;
    }
    if(l > r){
        return 1;
    }
    return 0;
}
```

- Template Parameter List – The list of types between the < and >
- Function Template Instantiation – Invoking a concrete version of the function
  - Example:

```
int main(){
    int I1 = 3, I2 = 1;
    double D1 = 3.14, D2 = 3.14;
    std::string S1 = "ABC", S2 = "DEF";

    // Invokes int version of Compare
    std::cout<<Compare(I1,I2)<<std::endl;
    // Invokes double version of Compare
    std::cout<<Compare(D1,D2)<<std::endl;
    // Invokes std::string version of Compare
    std::cout<<Compare(S1,S2)<<std::endl;
    return 0;
}
```

- Template Type Parameters – Types are specified in the Template Parameter List, preceded with typename (or class for older style)
  - Example:

```
template <typename T, typename U> T foo(T &t, U&u);
template <class T, class U> U bar(const T&t, const U&u);
```

- Nontype Template Parameters – Allows for value type parameters
  - Example:

```
template<typename T, unsigned M, unsigned N>
int CompareLength(const T (&l)[M], const T (&r)[N]){
    if(M < N){
        return -1;
    }
    if(M > N){
        return 1;
    }
    return 0;
}
```

- Class Template –A generic class that allow parameterization of types, vector, list, map are STL examples
  - Example:

```
template<typename T>
class Stack{
    public:
        Stack(int sz);
        ~Stack();
        bool Push(const T &val);
        bool Pop(T &val);
        bool Empty() const;
        bool Full() const;
    private:
        int Size;
        int Count;
        T *Base;
};

template<typename T>
Stack<T>::Stack(int sz){
    Size = sz;
    Count = 0;
    Base = new T[Size];
}

template<typename T>
Stack<T>::~Stack(){
    delete [] Base;
}

template<typename T>
bool Stack<T>::Push(const T &val){
    if(Count == Size){
        return false;
    }
    Base[Count] = val;
    return true;
}

template<typename T>
bool Stack<T>::Pop(T &val){
    if(!Count){
        return false;
    }
    Count--;
    val = Base[Count];
    return true;
}

template<typename T>
bool Stack<T>::Empty(){
    return !Count;
}

template<typename T>
bool Stack<T>::Full(){
    return Count == Size;
}
```