CSCI 104
Templates
Mark Redekopp
David Kempe
Overview

• C++ Templates allow alternate versions of the same code to be generated for various data types
FUNCTION TEMPLATES
How To's

• Example reproduced from: http://www.cplusplus.com/doc/tutorial/templates/

• Consider a max() function to return the max of two int's

• But what about two double's or two strings

• Define a generic function for any type, T

• Can then call it for any type, T, or let compiler try to implicitly figure out T

```c
int max(int a, int b)
{
    if(a > b) return a;
    else return b;
}

double max(double a, double b)
{
    if(a > b) return a;
    else return b;
}

template<typename T>
T max(const T& a, const T& b)
{
    if(a > b) return a;
    else return b;
}

int main()
{
    int x = max<int>(5, 9); //or x = max(5, 9); // implicit max<int> call
double y = max<double>(3.4, 4.7);
    // y = max(3.4, 4.7);

    return 0;
}
```

Non-Templated = Multiple code copies

Templated = One copy of code
CLASS TEMPLATES
Templates

• We’ve built a list to store integers
• But what if we want a list of double’s or string’s or other objects
• We would have to define the same code but with different types
  – What a waste!
• Enter C++ Templates
  – Allows the one set of code to work for any type the programmer wants
  – The type of data becomes a parameter

```cpp
#ifndef LIST_INT_H
#define LIST_INT_H
struct IntItem {
    int val; IntItem* next;
};
class ListInt{
    public:
        ListInt(); // Constructor
        ~ListInt(); // Destructor
        void push_back(int newval); ...
    private:
        IntItem* head;
};
#endif
```

```cpp
#ifndef LIST_DBL_H
#define LIST_DBL_H
struct DoubleItem {
    double val; DoubleItem* next;
};
class ListDouble{
    public:
        ListDouble(); // Constructor
        ~ListDouble(); // Destructor
        void push_back(double newval); ...
    private:
        DoubleItem* head;
};
#endif
```
• Allows the type of variable in a class or function to be a parameter specified by the programmer
• Compiler will generate separate class/struct code versions for any type desired (i.e. instantiated as an object)
  – LList<int> my_int_list causes an ‘int’ version of the code to be generated by the compiler
  – LList<double> my_dbl_list causes a ‘double’ version of the code to be generated by the compiler

```cpp
// declaring templatized code
template <typename T>
struct Item {
    T val;
    Item<T>* next;
};

template <typename T>
class LList {
    public:
        LList();  // Constructor
        ~LList();  // Destructor
        void push_back(T newval); ...
    private:
        Item<T>* head_;
};

// Using templatized code
// (instantiating templatized objects)
int main()
{
    LList<int> my_int_list;
    LList<double> my_dbl_list;
    my_int_list.push_back(5);
    my_dbl_list.push_back(5.5125);
    double x = my_dbl_list.pop_front();
    int y = my_int_list.pop_front();
    return 0;
}
```
Templates

• Writing a template
  – Precede class with:
    template <typename T>
  Or
    template <class T>
  – Use T or other identifier where you want a generic type
  – Precede the definition of each function with template <typename T>
  – In the scope portion of the class member function, add <T>
  – Since Item and LList are now templated, you can never use Item and LList alone
    • You must use Item<T> or LList<T>

```cpp
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
    T val;
    Item<T>* next;
};

template <typename T>
class LList{
public:
    LList();  // Constructor
    ~LList();  // Destructor
    void push_back(T newval);
    T& at(int loc);
private:
    Item<T>* head_;
};

template<typename T>
LList<T>::LList() {
    head_ = NULL;
}

template<typename T>
LList<T>::~LList() {
}

template<typename T>
void LList<T>::push_back(T newval) {
   ...
}
#endif
```
Exercise

• Recall that maps/dictionaries store key,value pairs
  – Example: Map student names to their GPA

• How many key,value type pairs are there?
  – string, int
  – int, double
  – Etc.

• Would be nice to create a generic data structure

• Define a Pair template with two generic type data members
 Templates

• Usually we want you to write the class definition in a separate header file (.h file) and the implementation in a .cpp file

• **Key Fact:** Templated classes must have the implementation **IN THE HEADER FILE!**

• **Corollary:** Since we don't compile .h files, you cannot compile a templated class separately

• Why? Because the compiler would have no idea what type of data to generate code for and thus what code to generate

```cpp
#ifndef LIST_H
#define LIST_H

template<typename T>
struct Item {
    T val; Item<T>* next;
};

template<typename T>
class LList{
    public:
        LList(); // Constructor
        ~LList(); // Destructor
        void push_back(T newval);
    private:
        Item<T>* head_;
};
#endif

#include "List.h"

template<typename T>
LList<T>::push_back(T newval) {
    if(head_ = NULL) {
        head_ = new Item<T>;
        // how much memory does an Item
        // require?
    }
}
```
Templates

- The compiler will generate code for the type of data in the file where it is instantiated with a certain type

```cpp
#include "List.h"

int main()
{
    LList<int> my_int_list;
    LList<double> my_dbl_list;
    my_int_list.push_back(5);
    my_dbl_list.push_back(5.5125);
    double x = my_dbl_list.pop_front();
    int y = my_int_list.pop_front();
    return 0;
}

// Compiler will generate code for LList<int> when compiling main.cpp
```

```cpp
#ifndef LIST_H
#define LIST_H

template <typename T>
struct Item {
    T val; Item<T>* next;
};

template <typename T>
class LList{
public:
    LList(); // Constructor
    ~LList(); // Destructor
    void push_back(T newval);
    T& at(int loc);
private:
    Item<T>* head_;
};

template<typename T>
LList<T>::LList() {
    head_ = NULL;
}

template<typename T>
LList<T>::~LList() {
}

template<typename T>
void LList<T>::push_back(T newval) {
    ...
}
#endif
```
The devil in the details

C++ TEMPLATE ODDITIES
For various reasons the compiler may have difficulty resolving members of a templated base class

When accessing members of a templated base class provide the full scope or precede the member with `this->`
"typename" & Nested members

- For various reasons the compiler may have difficulty resolving nested types of a templated class whose template argument is still generic (i.e. T vs. int)
- Precede the nested type with the keyword 'typename'

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

template <typename T>
class Stack {
    public:
        void push(const T& newval) {
            data.push_back(newval);
        }
        T& top();
    private:
        std::vector<T> data;
};

template <typename T>
T& Stack<T>::top() {
    vector<T>::iterator it = data.end();
    typename vector<T>::iterator it = data.end();
    return *(it-1);
}

int main() {
    Stack<int> s1;
    s1.push(1); s1.push(2); s1.push(3);
    cout << s1.top() << endl;
    return 0;
}
```
What the "functor"

It's an object, it's a function...it's both rolled into one!
Who you gonna call?

- Functions are "called" by using parentheses () after the function name and passing some arguments
- Objects use the . or -> operator to access methods of an object
- Calling an object doesn't make sense
  - You call functions not objects
  - Or can you?

```cpp
class ObjA {
public:
    ObjA();
    void action();
};

int main()
{
    ObjA a;
    ObjA *aptr = new ObjA;
    // This makes sense:
    a.action();
    aptr->action();
    // This doesn't make sense
    a();
    // a is already constructed, so
    // it can't be a constructor call
    // So is it illegal?

    return 0;
}
```
**Operator()**

- Calling an object does make sense when you realize that () is an operator that can be overloaded

- For most operators their number of arguments is implied
  - operator+ takes an LHS and RHS
  - operator-> takes no args

- You can overload operator() to take any number of arguments of your choosing

```cpp
class ObjA {
public:
    ObjA();
    void action();
    void operator()() {
        cout << "I'm a functor!";
        cout << endl;
    }
    void operator() (int &x) {
        return ++x;
    }
};

int main()
{
    ObjA a;
    int y = 5;
    // This does make sense!!
    a();
    // prints "I'm a functor!"
    a(y);
    // y is now 6
    return 0;
}
```
Functors: What are they good for?

- I'd like to use a certain class as a key in a map or set
- Maps/set require the key to have...
  - A less-than operator
- Guess I can't use ObjA
  - Or can I?

```cpp
class ObjA {
public:
    ObjA(...);
    void action();
    int getX() { return x; }
    string getY() { return y; }
private:
    int x; string y;
};

int main()
{
    // I'd like to use ObjA as a key
    // Can I?
    map<ObjA, double> mymap;

    ObjA(5, "hi") a;
    mymap[a] = 6.7;
    return 0;
}
```

objc.h – Someone else wrote it
Functors: What are they good for?

- Map template takes in a third template parameter which is called a "Compare" object
- It will use this type and assume it has a functor [i.e. operator()] defined which can take two key types and compare them

```cpp
struct ObjAComparer
{
    bool operator()(const ObjA& lhs,
                    const ObjA& rhs)
    { return lhs.getX() < rhs.getX(); }
};

int main()
{
    // Now we can use ObjA as a key!!!!
    map<ObjA, double, ObjAComparer> mymap;

    ObjA(5,"hi") a;
    mymap[a] = 6.7;
    return 0;
}
```

```cpp
class ObjA {
    public:
        ObjA(...);
        void action();
        int getX() { return x; } // x
        string getY() { return y; } // y
    private:
        int x; string y;
};
```

```cpp
obja.h – Someone else wrote it
```
More Uses

- Functors can act as a user-defined "function" that can be passed as an argument and then called on other data items
- Below is a modified `count_if` template function (from STL `<algorithm>`) that counts how many items in a container meet some condition

```cpp
template <class InputIterator, class Cond>
int count_if (InputIterator first, InputIterator last, Cond pred)
{
    int ret = 0;
    for( ; first != last; ++first){
        if ( pred( *first ) )
            ++ret;
    }
    return ret;
}
```
More Uses

- Functors can act as a user-defined "function" that can be passed as an argument and then called on other data items.
- You need to define your functor struct [with the operator()], declare one and pass it to the function.

```cpp
struct NegCond {
    bool operator(int val) { return val < 0; }
};

int main()
{
    std::vector<int> myvec;
    // myvector: -5 -4 -3 -2 -1 0 1 2 3 4
    for (int i=-5; i<5; i++)
        myvec.push_back(i);
    NegCond c;
    int mycnt = count_if (myvec.begin(), myvec.end(), c);
    cout << "myvec contains " << mycnt;
    cout << " negative values." << endl;
    return 0;
}
```
Final Word

- Functors are all over the place in C++ and STL
- Look for them and use them where needed
- References
Practice

• SlowMap
  – wget http://ee.usc.edu/~redekopp/cs104/slowmap.cpp

• Write a functor so you can use a set of string*'s and ensure that no duplicate strings are put in the set
  – strset