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
Function Templates

• 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

```cpp
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);
}
```

Non-Templated = Multiple code copies

Templated = One copy of code
Motivating Example

• 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
```
Templates

- 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(const 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;
}
```
Template Mechanics (2)

• Writing a template
  – Precede class with:
    ```cpp
template <typename T>
Or
template <class T>
```
  (in this context there is ABSOLUTELY no difference or implication for using `typename` vs. `class`)
  – 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>
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
Another Example

• A pair struct:

```cpp
template<typename T1, typename T2>
struct pair {
    T1 first;
    T2 second;
    pair(const T1& f, const T2& s);
};

template<typename T1, typename T2>
pair<T1,T2>::pair(
    const T1& f,
    const T2& s);
    : first(f), second(s)
{ }
```
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(const T& newval);
    private:
        Item<T>* head_;
};
#endif
```

```cpp
#include "List.h"

template<typename T>
LList<T>::push_back(const 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.

**Main.cpp**

```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;
}
```

**List.h**

```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(const 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(const T& newval)
{ ... }
#endif
```
The devil in the details

C++ TEMPLATE ODDITIES
Templates & Inheritance

- 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->`

```cpp
#include "llist.h"
template <typename T>
class Stack : private LList<T>{
    public:
        Stack(); // Constructor
        void push(const T& newval);
        T const & top() const;
    }

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

template<typename T>
void Stack<T>::push(const T& newval)
{
    // call inherited push_front()
    push_front(newval); // may not compile
    LList<T>::push_front(newval); // works
    this->push_front(newval); // works
}

template<typename T>
void Stack<T>::push(const T& newval)
{
    // assume head is a protected member
    if(head) return head->val; // may not work
    if(LList<T>::head) // works
        return LList<T>::head->val;
    if(this->head) // works
        return this->head->val;
}
"typename" & Nested members

• For various reasons the compiler will 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' when you are
  – Not in the scope of the templated class AND
  – The template type is still generic

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

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

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

int main()
{
    Stack<int> s1;
    vector<int>::iterator it;
s1.push(1); s1.push(2); s1.push(3);
cout << s1.top() << endl;
return 0;
}
```

When the template type is still generic and you scope a nested type, precede with `typename`
When the template type is specific there is no need to use `typename`
It's an object, it's a function...it's both rolled into one!

WHAT THE "FUNCTOR"
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
• **Def.** A **functor** or **function object** is a class/struct that defines an operator()

```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!"

    // This also makes sense !!
    a(y);
    // y is now 6
    return 0;
}
```
The purpose of functors is to genericize code so that the same template of code can be customized.

Suppose I have a container of data and want to count how many elements meet a certain criteria but the criteria may change (negative values, even values, etc.).

- Seems like a lot of work to keep repeating the same generic code.

Is there a way to "genericize" the code?

```cpp
int count_if_neg (vector<int>::iterator first, vector<int>::iterator last)
{
    int ret = 0;
    for( ; first != last; ++first){
        if ( *first < 0 )
            ++ret;
    }
    return ret;
}

int count_if_even (vector<int>::iterator first, vector<int>::iterator last)
{
    int ret = 0;
    for( ; first != last; ++first){
        if ( *first % 2 == 0 )
            ++ret;
    }
    return ret;
}
```
With Function Pointers

- We could make the `count_if` routine generic by passing in a function pointer (yes there are pointers to functions)
  - But the criteria may change generic behavior
- Function pointer types:
  - `bool (*funcPtr)(int);`
  - This declares a pointer named `funcPtr` which can point to any function that returns a `bool` and takes an `int` argument

```cpp
bool isNeg(int x) { return x < 0; }
bool isEven(int x) { return x % 2 == 0; }

int count_if (vector<int>::iterator first, vector<int>::iterator last, bool (*funcPtr)(int) )
{ int ret = 0;
  for( ; first != last; ++first){
    if ( funcPtr(*first) )
      ++ret;
  }
  return ret;
}

int main()
{
  vector<int> v;
  // fill data somehow
  int neg = count_if(v.begin(), v.end(), isNeg);
  int even = count_if(v.begin(), v.end(), isEven);
  return 0;
}
```
With Functors

- We could also make the `count_if` routine generic by making it a template and use a functor object.

```cpp
struct isNeg {
    bool operator()(int x) { return x < 0; };
}

struct isEven {
    bool operator()(int x) { return x % 2 == 0; };
}

template <typename Comp>
int count_if(vector<int>::iterator first, vector<int>::iterator last, Comp c) {
    int ret = 0;
    for( ; first != last; ++first){
        if ( c(*first) )
            ++ret;
    }
    return ret;
}

int main() {
    vector<int> v;  isNeg c1;  isEven c2;
    // fill data somehow
    int neg = count_if(v.begin(), v.end(), c1);
    int even = count_if(v.begin(), v.end(), c2);
    return 0;
}
```
std::count_if

- 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;
}

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

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

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

```cpp
class Pt {
public:
    Pt(...);
    void action() { /* do stuff */ }
    int getX() { return x; }
    int getY() { return y; }
private:
    int x, y;
};

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

    Pt p1(4,5);
    mymap[p1] = 6.7;
    return 0;
}
```
Functors for Maps and Sets

- Map template takes in a third template parameter which is called a "Compare" functor.
- It will use this type and assume it has a functor [i.e. operator()] defined which can take two key types and compare them.
- In the map implementation:
  - It will never do \( \text{if}(k1 < k2) \) or \( \text{if}(k1 > k2) \)
  - But instead use the comparator: \( \text{if}(c(k1,k2)) \) or \( \text{if}(c(k2,k1)) \)

```cpp
class Pt {
public:
    Pt(...);
    void action() { /* do stuff */ }
    int getX() { return x; }
    int getY() { return y; }
private:
    int x, y;
};

struct PtComparer {
    bool operator()(const Pt& lhs, const Pt& rhs) {
        return (lhs.getX() < rhs.getX()) ||
               (lhs.getX() == rhs.getX() &&
                lhs.getY() < rhs.getY());
    }
};

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

    Pt a(4, 5), b(3, 7);
    mymap[a] = 6.7; mymap[b] = 2.1;
    return 0;
}
```
Warm Up: Functor Exercise

Write a single function to find max by different criteria

```cpp
template <typename T>
    T mymax(const T& a, const T& b)
{
    if(a > b) return a;
    return b;
}

struct SizeComp {
    bool operator()(const vector<int>& a, const vector<int>& b) const {

    }
};

struct SumComp {
    bool operator()(const vector<int>& a, const vector<int>& b) const {

    }
};
```

Warm Up: Functor Exercise Solution

Write a single function to find max by different criteria

```cpp
template <typename T, typename comp>
T mymax(const T& a, const T& b, comp test)
{
    if(test(a, b)) return a;
    return b;
}

struct SizeComp {
    bool operator()(const vector<int>& a, const vector<int>& b) const {
        return a.size() > b.size();
    }
};

struct SumComp {
    bool operator()(const vector<int>& a, const vector<int>& b) const {
        int asum = std::accumulate(a.begin(),a.end(),0);
        int bsum = std::accumulate(b.begin(),b.end(),0);
        return asum > bsum;
    }
};
```
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