

# Advanced C++

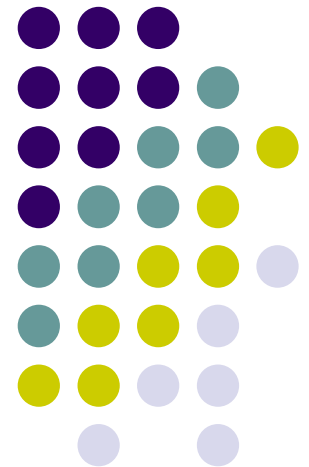
July 2, 2009

---

Mike Spertus

[mike\\_spertus@symantec.com](mailto:mike_spertus@symantec.com)

YIM: spertus



# Useful books for this class and after



- Scott Meyer, *Effective C++*, 3<sup>rd</sup> Edition
  - Make sure you get the 3<sup>rd</sup> Edition, which uses templates
  - Last week's lecture referred to this
  - *Effective STL* is also good
- Alexandrescu, *Modern C++ Design*
  - Implements Design Patterns with template metaprograms
  - This book will be the last part of the course
- Vandervoorde and Josuttis, *C++ Templates, the Complete Guide*
  - Everything about templates by the template specialist on the C++ standards committee, clearly written
- Herlihy and Shavit, *The Art of Multiprocessor Programming*
  - Actually a Java book, but best reference to get seriously good at concurrency

# How do I take the address of operator\*?



- Suppose, I have a function apply defined as follows

```
int f(int (*)(int i, int j), int i, int j)
{
    return f(i, j);
}
f(&max<int>, 3, 5) == 5 // calls max(3, 5)
```
- If I want to use the product of two ints instead of max, I can't take the address of `operator*<int>`
- Fortunately, the `<functional>` header supplies corresponding functions you can use

```
f(std::multiplies<int>, 3, 5) == 15;
```



# TR1

- Short for ISO/IEC TR 19768, C++ Library Extensions
- Not officially a standard, a non-normative "Tech Report" that was used as a testbed for new library proposals in 2005
- Nearly everything in TR1 will be included in C++0X
- Boost supplies a TR1 implementation as well as many compiler vendors
- Everything is in the `std::tr1` namespace



# Using `std::tr1::bind`

- Bind lets you reduce or reorder the number of arguments in a function.

- For example, suppose `f` is declared as

```
int f(int, int)
```

Then

```
bind(f, 3, _1)
```

is function of one variable that calls `f`:

```
bind(f, 3, _1)(7)
```

is the same as

```
f(3, 7)
```

- Similarly, `bind(f, _2, _1)(x, y)` is the same as `f(y, x)`
- Placeholders can be repeated:

```
bind(std::multiplies<int>, _1, _1)(7) == 49
```

- For full examples, see

<http://cspp51044.cs.uchicago.edu/2009/Lectures/lecture-12/data/bind.cpp>



# Using Boost lambda

- Lambda is like bind, only you can make expressions out of the placeholder

```
for_each(v.begin(), v.end(), cout << _1*_1 << '\n')
```

prints the squares of the elements of `v`

- Lambda expressions also work with multiple arguments: E.g., `_1 + _2` is a lambda function taking two arguments
- Unfortunately, lambda sometimes behaves bizarrely

```
for_each(v.begin(), v.end(), cout << '\n' << _1*_1)
```

only prints one newline because `cout << '\n'` is only evaluated once. Correct way is

```
for_each(v.begin(), v.end(), cout << var('\n') << _1*_1)
```

- Since lambda is so badly behaved, only extremely simple uses (if any) are suitable for production code, but it can be useful and edifying
- For full examples, see For full examples, see <http://cspp51044.cs.uchicago.edu/2009/Lectures/lecture-12/data/lambda.cpp>
- C++0X will have real lambda functions



# Using `std::tr1::function`

- We can declare a pointer to a function like

```
int (*fp) (int, double);
```

- However, you can't assign a functor to `fp`

```
int f(int, double);  
struct functor {  
    int operator() (int, double);  
};  
fp = f; // OK  
Functor fun;  
fp = fun; // Error: not a function
```



# TR1 Function to the rescue

- TR1's function object is like a function pointer that can also take anything that is callable with the right arguments

```
#include<boost/tr1/functional.hpp>
using namespace std::tr1;
function<int(int, double)> fp;
fp = f; // OK
fp = fun; // OK
```



# Other TR1 features

- Reference wrappers
- Shared\_ptr
- type\_traits
- random numbers (different from C++0X)
- Special functions (statistics. Different from C++0X)
- Tuples, additional containers
- Regular expressions

# SFINAE



- "Substitution Failure is Not an Error"
- [http://en.wikipedia.org/wiki/Substitution\\_failure\\_is\\_not\\_an\\_error](http://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error)

# enable\_if



- Boost provides a way to select template overloads based on SFINAE called `enable_if`
- [http://www.boost.org/doc/libs/1\\_38\\_0/libs/utility/enable\\_if.html](http://www.boost.org/doc/libs/1_38_0/libs/utility/enable_if.html)



## HW 12-1

- We discussed in the lecture how `tr1::function` objects lets you reference functors (i.e., objects that define `operator()`) as well as actual functions. This exercise shows that `tr1::function` is valuable even if you are only using functions and not functors.
- Try compiling the program on the next slide with and without `USE_TR1_FUNCTION` defined.

# HW 12-1 (continued)



```
#include <boost/tr1/functional.hpp>
using namespace std::tr1;
struct Base {};
struct Derived : public Base {
    Derived(const Base *bp) : Base(*bp) {}
};

Base *restrict(Derived *dp) { return dp; }
Derived *extend(Base *bp)
    { return new Derived(bp); }

int main()
{
    Base b;
    Derived d(&b);
#ifdef USE_TR1_FUNCTION
    function<Base *(Derived *)> f;
#else
    Base *(*f)(Derived *);
#endif
    f = restrict;
    f(&d);
    f = extend;
    return 0;
}
```



## HW 12-1 (continued)

- Is the result the same?
- Based on this, explain when you might want to use `tr1::function` objects even if you are only using actual functions and not functors.

# HW 12-2



- We want to write a program to compute the distance of a vector from the origin (i.e., the square root of the sum of the squares of the elements).
- We will do several versions on the following slides



## HW 12-2: Part 1

- This part is meant to be a review of the approach we used last quarter. Make the program on the next slide work by defining a global `square(double)` function and replacing the \_\_\_\_\_ with `square`.

# HW 12-2: Part 1 (continued)



```
#include <boost/tr1/functional.hpp>
#include <numeric>
#include <algorithm>
#include <vector>
#include <cmath>
#include <iostream>
using namespace std;
using namespace std::tr1;

int main()
{
    vector<double> v;
    v.push_back(7);
    v.push_back(24);
    vector<double> w;
    transform(v.begin(), v.end(), back_inserter(w), _____);
    cout << sqrt(accumulate(w.begin(), w.end(), 0.0))
         << endl;
    return 0;
}
```



## HW 12-2: Part 2

- In part 1, it was awkward to have to have to define a separate global squaring function. In this part, make the above program work by replacing the \_\_\_\_\_ with an expression using `std::tr1::bind` and `std::multiplies`. **Important:** The only changes you make to the above program should be to the \_\_\_\_\_ area.



## HW 12-2: Part 3

- In Part 2, we still had to calculate the length in two steps (transform and accumulate). We'd like to do it in one step.
- replacing the \_\_\_\_\_ with an expression using `std::tr1::bind`, `std::plus`, and `std::multiplies`.

# HW 12-2: Part 3



- ```
#include <boost/bind.hpp>
#include <numeric>
#include <algorithm>
#include <vector>
#include <cmath>
#include <functional>
#include <iostream>
using namespace std;
using namespace boost;
int main()
{
    vector<double> v;
    v.push_back(7);
    v.push_back(24);
    cout << sqrt(accumulate(v.begin(), v.end(), 0.0, _____))
         << endl;
    return 0;
}
```



## HW 12-2: Part 4

- Do part 3 with `boost::lambda` instead of `std::tr1::bind`, `std::multiplies` and `std::plus`.