Lecture 6

Boost Library 9

Kenny Erleben

Department of Computer Science
University of Copenhagen
What is it about?

Adds functionality and normalizes syntax for creating function objects on the fly
Function Object

A function object looks like

template<typename T>
class MyFancyFunctor
{
public:

    void operator()( T const & value) const 
    {
        ...
    }
};

We use it like

typedef MyFancyFunctor<Dodah> functor_type;
std::vector<Dodah> data;
...
for_each(data.begin(), data.end(), functor_type());
The First Problem

Say we have a function

```cpp
void my_fancy_function(Dodah const & value) {
  ...
}
```

So how do we use something like `std::for_each` with our function?

```cpp
for_each(data.begin(), data.end(), ? my_fancy_function(?));
```
The First Problem

Or maybe we have a method

class Hodah {
    public:
        void my_fancy_method(Dodah const & value) {
            ...
        }
};

So how do we

for_each(data.begin(), data.end(), ? Hodah::my_fancy_method(?));
The First Problem

We might not have a functor object
The Naïve Solution

So we would write

```cpp
class A { };
void plain(A const & a){....}
```

and

```cpp
std::list<A> lots;
for(std::list<A>::iterator it = lots.begin(); it != lots.end(); it++)
    plain(*it);
```

Why is this “ugly”? 
The Naïve Solution

Well we could optimize it

```cpp
std::list<A> lots;
std::list<A>::iterator it = lots.begin();
std::list<A>::iterator end = lots.end();
for(;it != end;++it)
    plain( *it );
```

- Yeah, we got the performance
- But `std::for_each` already do all of these optimizations for us
- Oh there is a bunch of text ⇒ increase chance of a typo
- What if I wanted to use a vector container instead?
- Besides the above leads to repetitive code, is this clever?

Conclusion it makes sense to use `std::for_each`, so let us write our functor.
For this case the functor looks like

```cpp
struct DoThe DamnThing {
    void operator()(A const & a){ plain(a); }
};
```

and now we can write

```cpp
std::for_each(lots.begin(), lots.end(), DoTheDamnThing());
```

Nice, we got a **one-liner** instead of four lines

```cpp
std::list<A>::iterator it = lots.begin();
std::list<A>::iterator end = lots.end();
for(;it != end;++it)
    plain(*it);
```

Are we happy?
In a typical simulator one often find

```cpp
class Tetrahedron {
    public:
        void compute_elastic_forces() { .... }
};

class Node {
    public:
        void compute_external_forces() { ... }
        void position_update() { ... }
        void velocity_update() { ... }
};
```
Staying on top of things

In the simulation loop one would write

```cpp
std::for_each(nodes.begin(), nodes.end(), ?velocity_update?);
std::for_each(nodes.begin(), nodes.end(), ?compute_external_forces?);
std::for_each(tetrahedra.begin(), tetrahedra.end(), ?compute_elastical_forces?);
std::for_each(nodes.begin(), nodes.end(), ?position_update?);
```

- Imagine having to write “wrapper” functor classes for all this
- Yikes we would just scatter the “logic” of our implementation

See real-life example:
“OpenTissue/t4mesh/util/t4mesh_persson_strang_generator.h”
Functions and function pointers

Given

```c
int f(int a, int b){ return a + b; }
```

Then

```c
bind(f, 1, 2)
```

Corresponds to a “nullary” function object

```c
class NullaryBindWrapper
{
    public:
        int operator() { return f(1,2); }
};
```
Functions and function pointers

Given

```cpp
int g(int a, int b, int c) { return a + b + c; }
```

Then

```cpp
bind(g, 1, 2, 3)()
```

is equivalent to

```cpp
class Wrapper {
    public:
        int operator() { return g(1, 2, 3); }
    }

Wrapper w;
w();
```

That is $g(1, 2, 3)$. 

Functions and function pointers

It is possible to selectively bind only some of the arguments.

\[
\text{bind}(f, \_1, 5)(x)
\]

is equivalent to

\[
f(x, 5);
\]

here \_1 is a place-holder argument that means “substitute with the first input argument”

For comparison, with STL

\[
\text{std::bind2nd(std::ptr_fun(f), 5)(x)};
\]

Yrgh!!! Boost bind is just so pretty...
Functions and function pointers

bind covers `std::bind1st` as well:

\[
\begin{align*}
\text{std::bind1st} & \left( \text{std::ptr\_fun}(f), 5 \right) (x); & \quad \text{// } f(5, x) \\
\text{bind} & (f, 5, _1) (x); & \quad \text{// } f(5, x)
\end{align*}
\]

Besides

- bind can handle functions with more than two arguments
- its argument substitution mechanism is more general

\[
\begin{align*}
\text{bind} & (f, _2, _1) (x, y); & \quad \text{// } f(y, x) \\
\text{bind} & (g, _1, 9, _1) (x); & \quad \text{// } g(x, 9, x) \\
\text{bind} & (g, _3, _3, _3) (x, y, z); & \quad \text{// } g(z, z, z) \\
\text{bind} & (g, _1, _1, _1) (x, y, z); & \quad \text{// } g(x, x, x)
\end{align*}
\]
Functions and function pointers

Arguments are copied and held internally

```cpp
int i = 5;
bind(f, i, _1);  // copy of i stored in function object
```

This might not be a good idea, say

```cpp
class Huge
{
  ...
  char m_big_chunk[1024*1024*100];
}

f(Huge & h, int i){...}

Huge first;
bind(f, first, 2);
```
Functions and function pointers

Solution, use

- boost::ref
- boost::cref

to make the function object store a reference to an object

```cpp
int i = 5;
bind(f, ref(i), _1);
bind(f, cref(42), _1);
```

No copying!
Function objects

Also

- bind is not limited to functions
- it accepts arbitrary function objects

For instance

```cpp
struct F {
    int operator()(int a, int b) { return a - b; }
    bool operator()(long a, long b) { return a == b; }
};
F f;
int x = 104;
bind<int>(f, _1, _1)(x); // f(x, x), i.e. zero
```
Function objects

When the function object exposes

a nested type named result_type

the explicit return type can be omitted:

```cpp
int x = 8;
bind(std::less<int>(), _1, 9)(x); // x < 9
```

Note: the ability to omit the return type is not available on all compilers.
Pointers to members

Pointers to
- member functions
- data members
are not function objects (do not support \texttt{operator()}). For convenience,
- \texttt{bind} accepts member pointers as its first argument
- \textbf{behavior is as if} \texttt{boost::mem_fn} \textbf{has been used to convert the member pointer into a function object}

The expression

\begin{verbatim}
bind(&X::f, args)
\end{verbatim}

is equivalent to

\begin{verbatim}
bind<R>(mem_fn(&X::f), args)
\end{verbatim}

where \( R \) is the return type of \( X::f \) (for member functions) or the type of the member (for data members.)
Pointers to members

Example:

```cpp
struct X {
    bool f(int a);
};

X x;

shared_ptr<X> p(new X);

int i = 5;

bind(&X::f, ref(x), _1)(i);       // x.f(i)
bind(&X::f, &x, _1)(i);           // (&x)->f(i)
bind(&X::f, x, _1)(i);            // (internal copy of x).f(i)
bind(&X::f, p, _1)(i);            // (internal copy of p)->f(i)
```
Function composition

Arguments may be

- nested bind expressions

So

\[
\text{bind}(f, \text{bind}(g, _1))(x); \quad // \quad f(g(x))
\]

Note

- The inner bind expressions are evaluated, in unspecified order, before the outer bind when the function object is called

- the results of the evaluation are then substituted in their place when the outer bind is evaluated

In the example above,

- when the function object is called with the argument list \((x)\), \text{bind}(g, _1)(x) is evaluated first, yielding \(g(x)\)

- and then \text{bind}(f, g(x))(x) is evaluated, yielding the final result \(f(g(x))\)
What about templates?

On my M$ compiler

template<typename T>
void plain(T t){};
...
    std::list<A> lots;
...
    std::for_each( lots.begin(), lots.end(), boost::bind( plain, _1) );

results in

d:\work\...\src\bind_fun.h(64) : error C2896: 'boost::_bi::bind_t<R,boost::'
d:\work\...\src\bind_fun.h(51) : see declaration of 'plain'
d:\work\...\src\bind_fun.h(64) : error C2784: 'boost::_bi::bind_t<R,boost::'
    D:\work\third.party\boost_1_33_1\boost\bind\bind_hpp(1616) : see declaration
d:\work\...\src\bind_fun.h(64) : error C2780: 'boost::_bi::bind_t<R,boost::'
    D:\work\third.party\boost_1_33_1\boost\bind\bind_mf_cc.hpp(222) : see
What about templates?

Yikes, template functions can not be bound?

```cpp
std::for_each(
    lots.begin(),
    lots.end(),
    boost::bind( plain<A>, _1)
);
```

Unless we help the compiler.
What about templates?

Now consider

```cpp
class A
{
public:
    template<typename T>
    void doit(T t){....}
};
```

and

```cpp
std::for_each(
    lots.begin(), lots.end(),
    boost::bind( &A::doit<A>, _1)
);
```

Guess what happens?
More Cool Stuff

Now what if we wrote

```cpp
class A
{
    public:
    void doit(){...}
};
```

And say we have

```cpp
std::list<A *> B;
```

Now how would we go about invoking `doit` on all elements in `B`?

```cpp
std::list<A*>::iterator it = B.begin();
std::list<A*>::iterator end = B.end();
for(;it != end; ++it)
    (*it)->doit();
```

Oh no even more ugly `*`-syntax, but with bind

```cpp
for_each(B.begin(), B.end(), bind(&A::doit, _1));
```
More Cool Stuff

And it can even figure out

```cpp
std::list<boost::shared_ptr<A> > B;
```
One-liners in short

Experience
- It takes discipline to start using them (I know we are still trying)
- Some code gets really easy to write
- Some code get really easy to read
- Other stuff just gets too complicated or totally unreadable
- It is easy to impress your friend with one-lines

Some benefits
- bind is up for STL
- It replaces all the even more ugly STL constructs: mem_fn, bind1st,....

Drawbacks
- Damn hard to debug
- Not always obvious how to use (pitfall: faster to write naïve implementation...or is it?)