C++11 Lambda Expressions

Professional C++ Training

Michael Caisse

http://ciere.com/cppnow12
follow @consultciere

`std::disclaimer`

- The example code is for demonstrating a purpose
- Please do not assume styles are considered good practice
- Please, *never* `using std;` in your own code
- Please, *always* use scoping and namespaces properly

Fine print: while we often think these are reasonable guidelines to expect from slides, our lawyer made us add a disclaimer

Fine print to the fine print: not really, but we sometimes here people say, "but you had `using std;` in your slides"

Part I

Lambda Expressions

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

ciere.com

Some Motivation - Old School

```
vector<int>::const_iterator iter     = cardinal.begin();
vector<int>::const_iterator iter_end = cardinal.end();

int total_elements = 1;
while( iter != iter_end )
{
   total_elements *= *iter;
   ++iter;
}
```

ciere.com

Some Motivation - Functor

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          product<int>(total_elements) );
```

```
template <typename T>
struct product
{
   product( T & storage ) : value(storage) {}

   template< typename V>
   void operator()( V & v )
   {
      value *= v;
   }

   T & value;
};
```

ciere.com

Some Motivation - Functor

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          product<int>(total_elements) );


template <typename T>
struct product
{
   product( T & storage ) : value(storage) {}

   template< typename V>
   void operator()( V & v )
   {
      value *= v;
   }

   T & value;
};
```

Some Motivation - Phoenix

```
// Boost.Phoenix
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          phx::ref(total_elements) *= _1 );
```

ciere.com

Some Motivation - Lambda Expression

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          [&total_elements](int i){total_elements *= i;} );
```

Some Motivation - Lambda Expression

Before:

```
vector<int>::const_iterator iter     = cardinal.begin();
vector<int>::const_iterator iter_end = cardinal.end();

int total_elements = 1;
while( iter != iter_end )
{
   total_elements *= *iter;
   ++iter;
}
```

After:

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          [&total_elements](int i){total_elements *= i;} );
```

Functors / Lambda comparison

```
struct mod
{
    mod(int m) : modulus(m) {}
    int operator()(int v){ return v % modulus; }
    int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Functors / Lambda comparison

```
struct mod
{
   mod(int m) : modulus(m) {}
   int operator()(int v){ return v % modulus; }
   int modulus;
};

int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [my_mod](int v) ->int
           { return v % my_mod; } );
```

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

ciere.com

Lambda Expression Parts - Introduction

$\underbrace{\texttt{[my\_mod]}}_{\textit{introducer capture}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$

ciere.com

Lambda Expression Parts - Introduction

$\underbrace{\texttt{[my\_mod]}}_{\substack{\textbf{introducer}\\ \textit{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[}\textcolor{red}{\texttt{\textbf{my\_mod}}}\texttt{]}}_{\substack{\textit{introducer}\\ \textbf{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\textit{introducer capture}}\ \underbrace{\textcolor{red}{\mathbf{\texttt{(int v\_)}}}}_{\textbf{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\substack{\textit{introducer}\\ \textit{capture}}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textbf{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$$

ciere.com

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod]}}_{\textit{introducer capture}}\ \underbrace{\texttt{(int v\_)}}_{\textit{parameters}}\ \underbrace{\texttt{->int}}_{\textit{return type}}\ \underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textbf{statement}}$$

Lambda Expression Parts - Introduction

$\underbrace{\texttt{[my\_mod]}}_{\textit{introducer}}\ \underbrace{\texttt{(int v\_) ->int}}_{\textbf{declarator}}\underbrace{\texttt{\{return v\_ \% my\_mod;\}}}_{\textit{statement}}$

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod](int v\_)->int\{return v\_ \% my\_mod;\}}}_{\textit{lambda expression}}$$

$\Downarrow$

closure object

- Evaluation of the expression results in a temporary called a closure object
- A closure object is unnamed
- A closure object behaves like a function object

Lambda Expression Parts - Introduction

$$\underbrace{\texttt{[my\_mod](int v\_)->int\{return v\_ \% my\_mod;\}}}_{\textit{lambda expression}}$$

$$\Downarrow$$

closure object

- Evaluation of the expression results in a temporary called a closure object
- A closure object is unnamed
- A closure object behaves like a function object

`[&]()->rt{...}` – introducer

```
[](){ cout << "foo" << endl; }
```

`[&]()->rt{...}` – introducer

$$\underbrace{\texttt{[]}}_{\textbf{introducer}}\;\underbrace{\texttt{()}}_{\textit{parameters}}\;\underbrace{\texttt{\{cout << "foo" <<endl;\}}}_{\textit{statement}}$$

We start off a lambda expression with the introducer

ciere.com

`[&]()->rt{...}` – introducer

$$\underbrace{\texttt{[](){cout << "foo" <<endl;}}}_{\Downarrow}$$

closure object

`[&]()->rt{...}` – introducer

How can we call this nullary *function object*-like temporary?

```
[](){ cout << "foo" << endl; } ();
```

Output

```
foo
```

ciere.com

`[&]()->rt{...}` – introducer

How can we call this nullary *function object*-like temporary?

```
[](){ cout << "foo" << endl; } ();
```

Output

```
foo
```

[&]()->rt{...} – parameter

```
[](int v){cout << v << "*6=" << v*6 << endl;} (7);
```

Output

```
7*6=42
```

[&]()->rt{...} – parameter

```
[](int v){cout << v << "*6=" << v*6 << endl;} (7);
```

Output

```
7*6=42
```

[&]()->rt{...} – parameter

```
int i = 7;

[](int & v){ v *= 6; } (i);

cout << "the correct value is: " << i << endl;
```

Output

```
the correct value is:  42
```

`[&]()->rt{...}` – parameter

```
int i = 7;

[](int & v){ v *= 6; } (i);

cout << "the correct value is: " << i << endl;
```

Output

```
the correct value is:  42
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int const & v){ v *= 6; } (j);

cout << "the correct value is: " << j << endl;
```

Compile error

```
error: assignment of read-only reference 'v'
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int const & v){ v *= 6; } (j);

cout << "the correct value is: " << j << endl;
```

Compile error

```
error: assignment of read-only reference 'v'
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int v)
{v *= 6; cout << "v: " << v << endl;} (j);
```

Output

```
v: 42
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int v)
{v *= 6; cout << "v: " << v << endl;} (j);
```

Output

```
v: 42
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int & v, int j){ v *= j; } (j,6);

cout << "j: " << j << endl;
```

Notice that the lambda's parameters do not affect the namespace.

Output

```
j: 42
```

ciere.com

[&]()->rt{...} – parameter

```
int j = 7;

[](int & v, int j){ v *= j; } (j,6);

cout << "j: " << j << endl;
```

Notice that the lambda's parameters do not affect the namespace.

Output

```
j: 42
```

ciere.com

`[&]()->rt{...}` – parameter

```
[](){ cout << "foo" << endl; } ();
```

same as

```
[]{ cout << "foo" << endl; } ();
```

Lambda expression without a declarator acts as if it were ()

`[&]()->rt{...}` – parameter

```
[](){ cout << "foo" << endl; } ();
```

same as

```
[]{ cout << "foo" << endl; } ();
```

Lambda expression without a declarator acts as if it were **()**

ciere.com

[&]()->rt{...} – capture

We commonly want to capture state or access values outside our *function objects*.

With a function object we use the constructor to populate state.

```
struct mod
{
   mod(int m_ ) : modulus( m_ ) {}
   int operator()(int v_){ return v_ % modulus; }
   int modulus;
};


int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           mod(my_mod) );
```

[&]()->rt{...} – capture

Lambda expressions provide an optional capture.

```
[my_mod](int v_) ->int { return v_ % my_mod; }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[&](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[=](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

[&]()->rt{...} – capture

Lambda expressions provide an optional capture.

```
[identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[&identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[foo,&bar,gorp](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

[&]()->rt{...} – capture

Lambda expressions provide an optional capture.

```
[&,identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Lambda expressions provide an optional capture.

```
[=,&identifier](){ ... }
```

We can capture by:

- Default all by reference
- Default all by value
- List of specific identifer(s) by value or reference and/or this
- Default and specific identifiers and/or this

ciere.com

`[&]()->rt{...}` – capture

Capture default all by reference

```
int total_elements = 1;
for_each( cardinal.begin(), cardinal.end(),
          [&](int i){ total_elements *= i; } );
```

[&]()->rt{...} – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
      [&]{ return stuff.size() >= 8; } );
```

Output

0 1 2 3 4 5 6 7

ciere.com

[&]()->rt{...} – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
      [&]{ return stuff.size() >= 8; } );
```

Output

```
0 1 2 3 4 5 6 7
```

[&]()->rt{...} – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
        [&]{ int sum=0;
             for_each( stuff.begin(), stuff.end(),
                       [&](int i){ sum += i; } );
             return sum >= 10;
           }
    );
```

Output

0 1 2 3 4

`[&]()->rt{...}` – capture

```
template< typename T >
void fill( vector<int> & v, T done )
{
    int i = 0;
    while( !done() )
    {
        v.push_back( i++ );
    }
}

vector<int> stuff;
fill( stuff,
        [&]{ int sum=0;
             for_each( stuff.begin(), stuff.end(),
                       [&](int i){ sum += i; } );
             return sum >= 10;
           }
    );
```

Output

```
0 1 2 3 4
```

[=]()->rt{...} – capture

Capture default all by value

```
int my_mod = 8;
transform( in.begin(), in.end(), out.begin(),
           [=](int v){ return v % my_mod; } );
```

[=]()->rt{...} – capture

Where is the value captured?

```
int v = 42;
auto func = [=]{ cout << v << endl; };
v = 8;
func();
```

[=]()->rt{...} – capture

Where is the value captured?

```
int v = 42;
auto func = [=]{ cout << v << endl; };
v = 8;
func();
```

At the time of evaluation

Output

```
42
```

ciere.com

[=] ()->rt{...} – capture

```
int i = 10;
auto two_i = [=]{ i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Compile error

```
error: assignment of member
'capture_test()::<lambda()>::i' in read-only
object
```

[=] ()->rt{...} – capture

```
int i = 10;
auto two_i = [=]{ i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Compile error

```
error: assignment of member
'capture_test()::<lambda()>::i' in read-only
object
```

[=]()->rt{...} – capture

Lambda closure objects have a public `inline` function call operator that:

- Matches the parameters of the lambda expression
- Matches the return type of the lambda expression
- Is declared **`const`**

Make mutable:

```
int i = 10;
auto two_i = [=]() mutable { i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Output

```
2i:20 i:10
```

`[=]()->rt{...}` – capture

Lambda closure objects have a public `inline` function call operator that:

- Matches the parameters of the lambda expression
- Matches the return type of the lambda expression
- Is declared **`const`**

Make mutable:

```
int i = 10;
auto two_i = [=]() mutable { i *= 2; return i; };
cout << "2i:" << two_i() << " i:" << i << endl;
```

Output

```
2i:20 i:10
```

[=,&identifer]()->rt{...} – capture

```
class gorp
{
    vector<int> values;
    int m_;

public:
    gorp(int mod) : m_(mod) {}
    gorp& put(int v){ values.push_back(v); return *this; }

    int extras()
    {
        int count = 0;
        for_each( values.begin(), values.end(),
                  [=,&count](int v){ count += v % m_; } );

        return count;
    }
};

gorp g(4);
g.put(3).put(7).put(8);
cout << "extras: " << g.extras();
```

[=,&identifer]()->rt{...} – capture

Capture default by value and count by reference

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

[=,&identifer]()->rt{...} – capture

Capture count by reference, accumulate, return

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

ciere.com

`[=,&identifer]()->rt{...}` – capture

How did we get **`m_`**?

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

[=,&identifer]()->rt{...} – capture

Implicit capture of **this** by value

```
class gorp
{
  vector<int> values;
  int m_;

public:
  int extras()
  {
     int count = 0;
     for_each( values.begin(), values.end(),
               [=,&count](int v){ count += v % m_; } );

     return count;
  }
};
```

[=,&identifer]()->rt{...} – capture

```
class gorp
{
   vector<int> values;
   int m_;

public:

   int extras()
   {
      int count = 0;
      for_each( values.begin(), values.end(),
                [=,&count](int v){ count += v % m_; } );

      return count;
   }
};

gorp g(4);
g.put(3).put(7).put(8);
cout << "extras: " << g.extras();
```

```
extras:  6
```

[=]()->rt{...} – capture

Will this compile? If so, what is the result?

```
struct foo
{
   foo() : i(0) {}
   void amazing(){ [=]{ i=8; }(); }

   int i;
};

foo f;
f.amazing();
cout << "f.i : " << f.i;
```

`[=]()->rt{...}` – capture

`this` implicity captured. **`mutable`** not required.

```
struct foo
{
   foo() : i(0) {}
   void amazing(){ [=]{ i=8; }(); }

   int i;
};
```

```
foo f;
f.amazing();
cout << "f.i : " << f.i;
```

Output

```
f.i : 8
```

ciere.com

Capture restrictions:

- Identifiers must only be listed once
- Default by value, explicit identifiers by reference
- Default by reference, explicit identifiers by value

```
[i,j,&z](){...}  // ok
[&a,b](){...}    // ok
[z,&i,z](){...}  // bad, z listed twice
```

[=,&identifer]()->rt{...} – capture

Capture restrictions:

- Identifiers must only be listed once
- Default by value, explicit identifiers by reference
- Default by reference, explicit identifiers by value

```
[=,&j,&z](){...}   // ok
[=,this](){...}    // bad, no this with default =
[=,&i,z](){...}    // bad, z by value
```

Capture restrictions:

- Identifiers must only be listed once
- Default by value, explicit identifiers by reference
- Default by reference, explicit identifiers by value

```
[&,j,z](){...}      // ok
[&,this](){...}     // ok
[&,i,&z](){...}     // bad, z by reference
```

`[=]()->rt{...}` – capture

Scope of capture:

- Captured entity must be defined or captured in the immediate enclosing lambda expression or function

`[=]()->rt{...}` – capture

```
int i = 8;
{
    int j = 2;
    auto f = [=]{ cout << i/j; };
    f();
}
```

Output

4

[=] ()->rt{...} – capture

```
int i = 8;
{
    int j = 2;
    auto f = [=]{ cout << i/j; };
    f();
}
```

Output

4

[=] ()->rt{...} – capture

```
int i = 8;
auto f =
    [=]()
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Output

4

ciere.com

`[=]()->rt{...}` – capture

```
int i = 8;
auto f =
    [=]()
    {
       int j = 2;
       auto m = [=]{ cout << i/j; };
       m();
    };

f();
```

Output

4

`[=]()->rt{...}` – capture

```
int i = 8;
auto f =
    [i]()
    {
       int j = 2;
       auto m = [=]{ cout << i/j; };
       m();
    };

f();
```

Output

4

`[=]()->rt{...}` – capture

```
int i = 8;
auto f =
   [i]()
   {
      int j = 2;
      auto m = [=]{ cout << i/j; };
      m();
   };

f();
```

Output

4

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [](
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Compile error

`error: 'i' is not captured`

ciere.com

[=] ()->rt{...} – capture

```
int i = 8;
auto f =
    [] ()
    {
        int j = 2;
        auto m = [=]{ cout << i/j; };
        m();
    };

f();
```

Compile error

`error: 'i' is not captured`

ciere.com

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [=]()
    {
        int j = 2;
        auto m = [&]{ i /= j; };
        m();
        cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Compile error

```
error: assignment of read-only location
'...()::<lambda()>::<lambda()>::i'
```

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [=]()
    {
        int j = 2;
        auto m = [&]{ i /= j; };
        m();
        cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Compile error

error: assignment of read-only location '...()::<lambda()>::<lambda()>::i'

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [i]() mutable
    {
        int j = 2;
        auto m = [&i,j]()mutable{ i /= j; };
        m();
        cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Output

```
inner: 4 outer: 8
```

ciere.com

[=]()->rt{...} – capture

```
int i = 8;
auto f =
    [i]() mutable
    {
       int j = 2;
       auto m = [&i,j]()mutable{ i /= j; };
       m();
       cout << "inner: " << i;
    };

f();
cout << " outer: " << i;
```

Output

```
inner: 4 outer: 8
```

ciere.com

[=]()->rt{...} – capture

```
int i=1,  j=2, k=3;
auto f =
    [i,&j,&k]() mutable
    {
        auto m =
            [&i,j,&k]() mutable
            {
                i=4; j=5; k=6;
            };

        m();
        cout << i << j << k;
    };

f();
cout << " : " << i << j << k;
```

Output

```
426 :  126
```

[=]()->rt{...} – capture

```
int i=1,  j=2, k=3;
auto f =
    [i,&j,&k]() mutable
    {
        auto m =
            [&i,j,&k]() mutable
            {
                i=4; j=5; k=6;
            };

        m();
        cout << i << j << k;
    };

f();
cout << " : " << i << j << k;
```

Output

```
426 :  126
```

[=]()->rt{...} – capture

- Closure object has implicity-declared copy constructor / destructor.

```
struct trace
{
 trace() : i(0)              { cout << "construct\n"; }
 trace(trace const &)        { cout << "copy construct\n"; }
 ~trace()                    { cout << "destroy\n"; }
 trace& operator=(trace&)    { cout << "assign\n"; return *this;}

 int i;
};
```

[=]()->rt{...} – capture

- Closure object has implicity-declared copy constructor / destructor.

```
struct trace
{
 trace() : i(0)             { cout << "construct\n"; }
 trace(trace const &)       { cout << "copy construct\n"; }
 ~trace()                   { cout << "destroy\n"; }
 trace& operator=(trace&)   { cout << "assign\n"; return *this;}

 int i;
};
```

ciere.com

[=]()->rt{...} – capture

Closure object has implicity-declared copy constructor / destructor.

```
{
    trace t;
    int i = 8;

    // t not used so not captured
    auto m1 = [=](){ return i/2; };
}
```

Output

```
construct
destroy
```

ciere.com

[=]()->rt{...} – capture

Closure object has implicity-declared copy constructor / destructor.

```
{
   trace t;
   int i = 8;

   // t not used so not captured
   auto m1 = [=](){ return i/2; };
}
```

Output

```
construct
destroy
```

ciere.com

[=] ()->rt{...} – capture

```
{
   trace t;

   // capture t by value
   auto m1 = [=](){ int i=t.i; };

   cout << "-- make copy --" << endl;
   auto m2 = m1;
}
```

Output

```
construct
copy construct
- make copy -
copy construct
destroy
destroy
destroy
```

`[=]()->rt{...}` – capture

```
{
    trace t;

    // capture t by value
    auto m1 = [=](){ int i=t.i; };

    cout << "-- make copy --" << endl;
    auto m2 = m1;
}
```

Output

```
construct
copy construct
- make copy -
copy construct
destroy
destroy
destroy
```

`[&]()->rt{...}` – return type

If the return type is omited from the lambda expression

and the statement has a return such as:

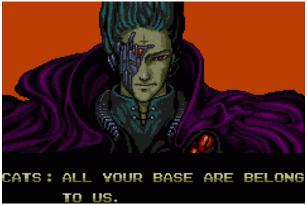
```
{ ...  return expression; }
```

then it is the type of the returned expression after:

- lvalue-to-rvalue conversion
- array-to-pointer conversion
- function-to-pointer conversion

Otherwise, the type is **`void`**

lambda<T>




```
for_each( your_base.begin(), your_base.end(),
          [&my_base](base_t & b)
          { my_base.push_back( move(b) ); } );

// all your base are belong to us
```

http://cierecloud.com/cppnow/

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

Storing / Passing Lambda Objects

Seen two ways so far:

- `tempalte<typename T> void foo(T f)`
- `auto f = []{};`

Function pointer

If the lambda expression has no capture it can be converted to a function pointer with the same signature.

```
typedef int(*f_type)(int);

f_type f = [](int i){ return i+20; };

cout << f(8);
```

Output

```
28
```

ciere.com

Function pointer

If the lambda expression has no capture it can be converted to a function pointer with the same signature.

```
typedef int(*f_type)(int);

f_type f = [](int i){ return i+20; };

cout << f(8);
```

Output

```
28
```

ciere.com

function<R(Args...)>

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

std::function< R (A1, A2, A3...) > f;

ciere.com

function<R(Args...)>

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

std::function< R (A1, A2, A3...) > f;

function<R(Args...)>

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

std::function< R (A1, A2, A3...) > f;

ciere.com

function<R(Args...)>

Polymorphic wrapper for function objects applies to anything that can be called:

- Function pointers
- Member function pointers
- Functors (including closure objects)

Function declarator syntax

std::function< R (A1, A2, A3...) > f;

function<R(Args...)>

Type	Old School Define	std::function
Free	`int(*callback)(int,int)`	`function< int(int,int) >`
Member	`int (object_t::*callback)(int,int)`	`function< int(int,int) >`
Functor	`object_t callback`	`function< int(int,int) >`

function<R(Args...)>

Function pointers

```
int my_free_function(std::string s)
{
   return s.size();
}

std::function< int(std::string) > f;
f = my_free_function;

int size = f("cierecloud.com/cppnow");
```

ciere.com

function<R(Args...)>

Member function pointers

```
struct my_struct
{
   my_struct( std::string const & s) : s_(s) {}
   int size() const { return s_.size(); }
   std::string s_;
};

my_struct mine("cierecloud.com/cppnow");
std::function< int() > f;

f = std::bind( &my_struct::size, std::ref(mine) );
int size = f();
```

function<R(Args...)>

Functors

```
struct my_functor
{
   my_functor( std::string const & s) : s_(s) {}
   int operator()() const
   {
      return s_.size();
   }

   std::string s_;
};
```

```
my_functor mine("cierecloud.com/cppnow");
std::function< int() > f;

f = std::ref(mine);
int size = f();
```

function<R(Args...)>

Closure Objects

```
std::function< int(std::string const &) > f;

f = [](std::string const & s){ return s.size(); };
int size = f("cierecloud.com/cppnow");
```

ciere.com

Fun with function

```
std::function<int(int)> f1;
std::function<int(int)> f2 =
    [&](int i)
    {
        cout << i << " ";
        if(i>5) { return f1(i-2); }
    };

f1 =
    [&](int i)
    {
        cout << i << " ";
        return f2(++i);
    };

f1(10);
```

Output

10 11 9 10 8 9 7 8 6 7 5 6 4 5

Fun with function

```
std::function<int(int)> f1;
std::function<int(int)> f2 =
    [&](int i)
    {
        cout << i << " ";
        if(i>5) { return f1(i-2); }
    };

f1 =
    [&](int i)
    {
        cout << i << " ";
        return f2(++i);
    };

f1(10);
```

Output

10 11 9 10 8 9 7 8 6 7 5 6 4 5

More fun with function

```
std::function<int(int)> fact;

fact =
    [&fact](int n)->int
        {
            if(n==0){ return 1; }
            else
            {
                return (n * fact(n-1));
            }
        } ;

cout << "factorial(4) : " << fact(4) << endl;
```

Output

```
factorial(4) :  24
```

More fun with function

```
std::function<int(int)> fact;

fact =
    [&fact](int n)->int
        {
            if(n==0){ return 1; }
            else
            {
                return (n * fact(n-1));
            }
        } ;

cout << "factorial(4) : " << fact(4) << endl;
```

Output

```
factorial(4) : 24
```

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

Exercise

- Class that can queue callable "things"
- The callable "thing" takes an `int` argument
- The callable "thing" returns an `int`
- The class will have a method:

```
int run( int init )
```

- When `run` is called:
  - Call each of the queued items
  - `init` will be the initial state of the first call
  - The result of each call feeds the input of the next call
  - The result of final call will be the return value of `run`

`http://cierecloud.com/cppnow`

ciere.com

Exercise

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

struct machine
{
   template< typename T >
   void add( T f )
   {
      to_do.push_back(f);
   }

   int run( int v )
   {
      std::for_each( to_do.begin(), to_do.end(),
                     [&v]( std::function<int(int)> f )
                     { v = f(v); } );
      return v;
   }

   std::vector< std::function<int(int)> > to_do;
};

int foo(int i){ return i+4; }

int main()
{
   machine m;
   m.add( [](int i){ return i*3; } );
   m.add( foo );
   m.add( [](int i){ return i/5; } );

   std::cout << "run(7) : " << m.run(7) << std::endl;
   return 1;
}
```

Outline

- Introduction
- Expression Parts
- Storing
- Exercise
- Use Cases

Where can we use them?

Lambda expression cannot appear in an unevaluated operand.

- typeid
- sizeof
- noexcept
- decltype

ciere.com

Some thoughts

Make Stepanov happy, revisit standard algorithms.

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

ciere.com

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

- Standard alogrithms
- Callbacks
- Runtime policies
- Locality of expression
- `std::bind`

Some thoughts

```
template< typename T >
void foo(T f)
{
   // amazing profit making function
   money_maker( f(8) );
}

int bar( std::string v, int m ){ return m * v.size(); }


std::string profit_item("socks");

foo( std::bind( bar, profit_item, _1 ) );

foo( [profit_item](int m){ return bar(profit_item, m);} );
```

Cannot do Currying

Some thoughts

```
template< typename T >
void foo(T f)
{
   // amazing profit making function
   money_maker( f(8) );
}

int bar( std::string v, int m ){ return m * v.size(); }


std::string profit_item("socks");

foo( std::bind( bar, profit_item, _1 ) );

foo( [profit_item](int m){ return bar(profit_item, m);} );
```

Cannot do Currying

Your Thoughts

How are you going to use lambdas?

Slides Available

`http://ciere.com/cppnow12/`