Lambda functions

- Terminology
- How it is compiled
- Capture by value and reference
- Mutable lambdas
- Use of this
- Init capture and generalized lambdas in C++14
- Constexpr lambda and capture *this and C++17

Lambda

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

using namespace std;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
    cout << endl;
    return 0;
}

$ g++ l.cpp
$ ./a.out
0 1 2 3 4 5 6 7 8 9
```

Lambda

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

using namespace std;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
    cout << endl;
    return 0;
}
```

Lambda introducer with opt. capture

Lambda parameter declaration

Optional return type
in form: -> type

```
$ g++ l.cpp
$ ./a.out
0 1 2 3 4 5 6 7 8 9
```

Lambdas are mapped to functors

```
[](int n) { cout << n << " "; }


struct LambdaFunctor
{
    void operator() (int n) const { cout << n << " "; }
};

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), LambdaFunctor());
    cout << endl;
    return 0;
}
```

Lambda terminology

- Lambda expression

  [ ] (int n) { }

- Closure
  - Runtime object created from lambda expression
  - May hold captured variables
  - Assignable
  - Can be stored in std::function
- Closure class
  - The type of the closure object

Can contain multiple statements

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    deque<double> d;

    transform(v.begin(), v.end(), front_inserter(d),
            [](int n) -> double  { return n / 2.0; } );

    for_each(d.begin(), d.end(), [](double n) { cout << n << " "; });

    cout << endl;
    return 0;
}
```

```
4.5 4 3.5 3 2.5 2 1.5 1 0.5 0
```

Can contain multiple statements

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

using namespace std;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), [](int n) {
             cout << n ;
             if ( n % 2 )
                 cout << ":odd ";
             else
                 cout << ":even ";
                        });
    cout << endl;
    return 0;
}
0:even 1:odd 2:even 3:odd 4:even 5:odd 6:even 7:odd 8:even 9:odd
```

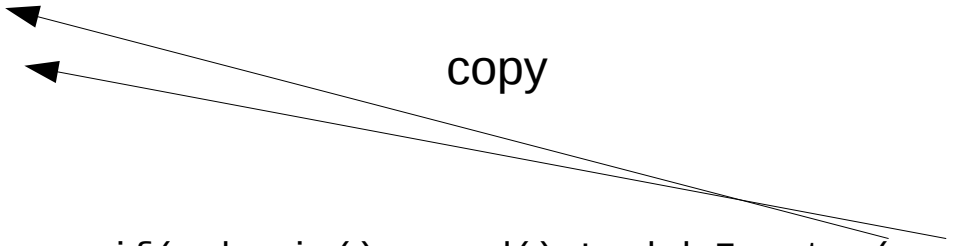
Capture

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int x = 0;
    int y = 0;
    cin >> x >> y;

    v.erase( remove_if(v.begin(),v.end(),[x,y](int n) { return x < n && n < y; }),
             v.end()
           );

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });

    cout << endl;
    return 0;
}

3 6
0 1 2 3 6 7 8 9
```

Capture by value

Capture by value

```
struct LambdaFunctor
{
public:
    LambdaFunctor(int a, int b) : m_a(a), m_b(b) { }
    bool operator()(int n) const { return m_a < n && n < m_b; }
private:
    int m_a;
    int m_b;
};

// ...

 v.erase( remove_if(v.begin(),v.end(),LambdaFunctor(x,y)), v.end());
```



The x and y parameters are copied and being stored in the function object. We cannot modify the captured values because the operator() in functor is const. It is a "real" copy, therefore the modification of x and y is not reflected inside the lambda.

[=] capture all by value

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int x = 0;
    int y = 0;
    cin >> x >> y;

    v.erase( remove_if(v.begin(),v.end(),[=](int n) { return x < n && n < y; }),
             v.end()
           );

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });

    cout << endl;
    return 0;
}

3 6
0 1 2 3 6 7 8 9
```

Capture by reference

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int prev = 0;
    for_each(v.begin(), v.end(), [&prev](int& r) mutable {
                         const int oldr = r;
                         r *= prev;
                         prev = oldr;
                       });

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });

    cout << "prev = " << prev << endl;
    return 0;
}
```

```
0 0 2 6 12 20 30 42 56 72 prev = 9
```

Capturing

- No capture

  [ ]
- By value

  [x,y] [=]
- By reference

  [&x, &y] [&]
- Mixed:

  [=, &x, &y] [&, x, y]
- Global variables, static members can be used but they are not captured

Capturing *this*

```
struct X
{
    int s;
    vector<int> v;
    void print() const
    {
        for_each(v.begin(), v.end(), [](int n) { cout << n*s << " "; });
    }
};

int main()
{
    X x;
    x.s = 2;
    for(int i = 0; i < 10; ++i)
        x.v.push_back(i);

    x.print();
    return 0;
}
```

```
$ g++ l10.cpp
l10.cpp: In lambda function:
l10.cpp:15:60: error: ‘this’ was not captured for this lambda function
```

Capturing *this*

```
struct X
{
    int s;
    vector<int> v;
    void print() const
    {
        for_each(v.begin(), v.end(), [this](int n) { cout << n*s << " "; });
    }
};

int main()
{
    X x;
    x.s = 2;
    for(int i = 0; i < 10; ++i)
        x.v.push_back(i);

    x.print();
    return 0;
}
```

```
0 2 4 6 8 10 12 14 16 18
```

Capturing *this*

```
struct X
{
    int s;
    vector<int> v;
    void print() const
    {
        int s = 9;
        for_each(v.begin(), v.end(), [this,s](int n) { cout << n*s << " "
                                                       << this->s << " "; });
    }
};

int main()
{
    X x;
    x.s = 2;
    for(int i = 0; i < 10; ++i)
        x.v.push_back(i);

    x.print();
    return 0;
}
```

```
0 2 9 2 18 2 27 2 36 2 45 2 54 2 63 2 72 2 81 2
```

Capturing *this*

- The this not captured by default
- The this is always captured by value
- [=] implicitly captures this
- Capturing this can be dangerous
  - Storing a non-smart pointer
  - Lifetime may already finished when lambda function is called

Capturing *this*

```
std::function<void (int)> f;

struct X
{
  X(int i) : ii(i) {}
  int ii;
  void addLambda()
  {
    f = [=](int n) { if (n == ii) cout << n;
                     else         cout << ii;
                   };
  }
};

int main()
{
  {
    std::unique_ptr<X> up = std::make_unique<X>(4);
    up->addLambda();
    f(4);
  }
  f(4);    // Likely aborts!
  return 0;
}
```

Lambda can be stored

```
int ii = 5;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);
    int x = 0;
    int y = 10;
    auto f = [=](int n) { if (x < n-ii && n-ii < y)
                              cout << n << " ";
                          };
    for_each(v.begin(), v.end(), f);
    cout << endl;

    ii = 1;
    for_each(v.begin(), v.end(), f);
    cout << endl;
    return 0;
}

$ ./a.out
6 7 8 9
2 3 4 5 6 7 8 9
```

Nullary lambdas

```
int main()
{
    vector<int> v;
    int i = 0;

    generate_n(back_inserter(v), 10, [&] { return i++; } );

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Lambda can be stored

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

void doit(const vector<int>& v, const function<void (int)>& f)
{
    for_each(v.begin(), v.end(), f);
    cout << endl;
}

int main()
{
    vector<int> v;
    int i = 0;

    generate_n(back_inserter(v), 10, [&] { return i++; } );

    doit(v, [](int n) { cout << n << " "; });

    const function<void (int)>& ff = [](int n) { cout << n << " "; };
    doit(v, ff);

    return 0;
}
```

Initialization trick

```
/* const */ int i = some_default_value; // can't do it const
                                        // since value depends
if(someConditionIstrue)                 // on some condition.
{
    // Do some operations and calculate the value of i;
    i = // some calculated value;
}
int x = i; // use i

// But unfortunately in this case there is no way to guarantee
// it is used as a constant, so now if some one comes and does
i = 10; // This is valid


const int i = [&]{

    int i = some_default_value;

    if(someConditionIstrue)
    {
        // Do some operations and calculate the value of i;
        i = // some calculated value;
    }
    return i;

} (); // note: () invokes the lambda!
```

Generalized lambdas in C++14

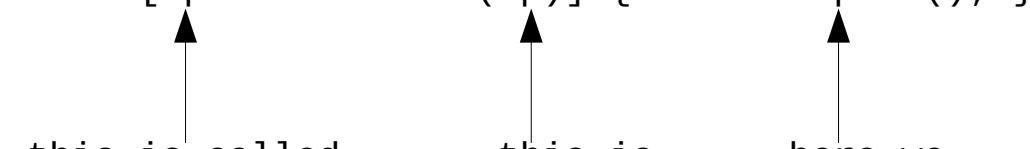
```
auto L = [](const auto& x, auto& y){ return x + y; };


means:


struct /* anonymous */
{
    template <typename T, typename U>
    auto operator()(const T& x, U& y) const // N3386 Return type deduction
    {
        return x + y;
    }
} L;
```

Example

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

int main()
{
    auto my_lambda = [](auto &a, auto &b) { return a < b; };

    float af = 1.5, bf = 2.0;
    int ai = 3, bi = 1;

    std::cout << "Float: "   << my_lambda(af, bf) << std::endl;
    std::cout << "Integer: " << my_lambda(ai, bi) << std::endl;

    return 0;
}
```

Init capture in C++14

```
auto up = std::make_unique<X>();

auto func = [up = std::move(up)] { return up->f(); }
```



Or:

```
auto func = [up = std::make_unique<X>()] { return up->f(); }

auto f = [](auto x) { return func(func2(x)); }

class UnnamedFunctor
{
public:
  template <typename T>
  auto operator()(T x) cont
  {
    return func(func2(x));
  }
};
```

Example

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

int main()
{
  std::vector<int> v;
  int i = 0;

  auto f = [cnt = 0](int n) mutable { std::cout << ++cnt << ":" << n << " "; };

  std::generate_n( std::back_inserter(v), 10, [&] { return i++; } );

  std::for_each( v.begin(), v.end(), f);
  std::for_each( v.begin(), v.end(), f);
  return 0;
}
```

```
1:0 2:1 3:2 4:3 5:4 6:5 7:6 8:7 9:8 10:9 1:0 2:1 3:2 4:3 5:4 6:5 7:6 8:7 9:8 10:9
```

Constexpr lambda in C++17

```
#include <iostream>

int main()
{
    constexpr auto multi = [](int a, int b){ return a * b; };

    static_assert(multi(3,7) == 21, "3x7 == 21");
    static_assert(multi(4,5) == 15, "4x5 != 15");

    return 0;
}
```

Capture *this in C++17

```
struct my_struct
{
  int x;
  int y;
  void value();
};

void my_struct::value()
{
  [=, this](){};     // error: = captures this by default
  [=, *this](){};    // captures my_struct by value since C++17
  [this, *this](){}; // error: repeating this in capture
}
```