# 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

# Functor

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

using namespace std;

struct PrinterFunctor
{
  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(), PrinterFunctor());
    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;
}

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

# Lambdas are mapped to functors

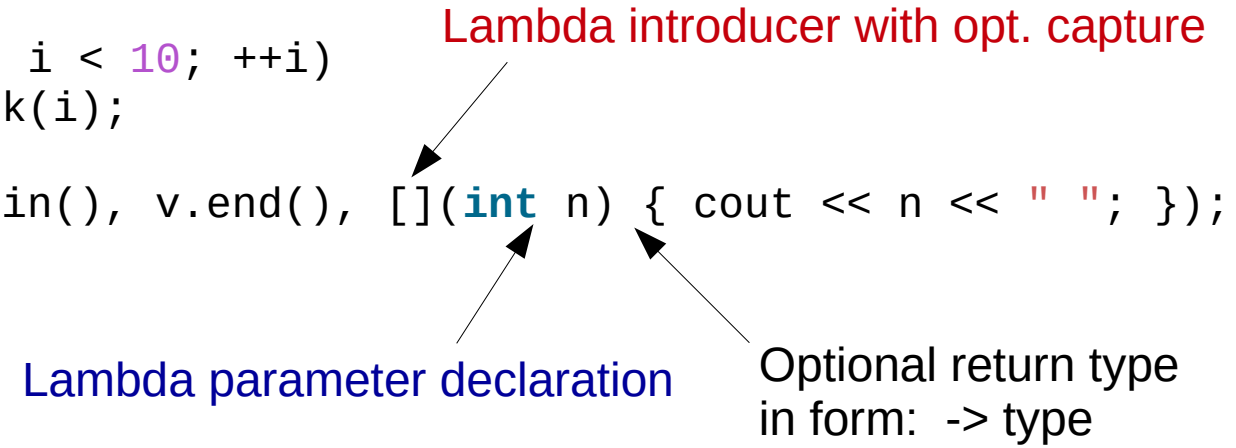
```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

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

# 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

```
#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 terminology

- Lambda expression

  [ ] (int n) { }

- Closure object
  - Runtime object created from lambda expression
  - Copy constructable (but not copy assignable)
  - Can be stored in std::function
  - May hold captured variables
- Closure class
  - The type of the closure object
  - Deleted default constructor and copy assignment operator

# Explicit return type

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

    deque<double> dd;
    deque<int>    di;

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

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

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

    return 0;
}
```

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

# 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
```

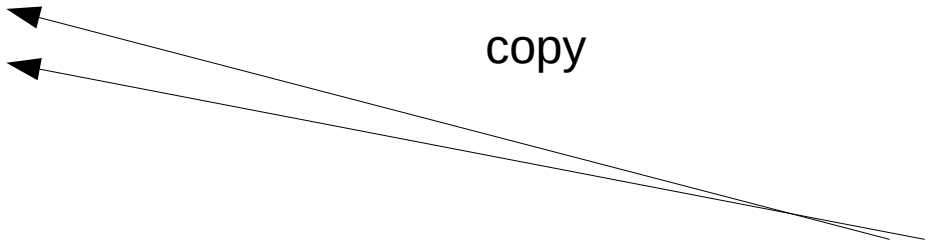
# Capture

```
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;  // read 3 6

    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

```
[x,y](int n) { return x < n && n < y; }

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 by reference

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

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x < n && n < y) sum += n; });

    cout << "sum = " << sum << endl;

    return 0;
}

2 7
sum = 18
```

# Mutable lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x-- < n && n < y++) sum += n; });

    cout << "sum = " << sum << endl;

    return 0;
}
```

```
$ g++ -std=c++11 -Wall -pedantic lambda4.cpp
lambda4.cpp: In lambda function:
lambda4.cpp:18:62: error: decrement of read-only variable ‘x’
     std::for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x--<n && n<y++) sum+=n; });
                                                              ^~
lambda4.cpp:18:73: error: increment of read-only variable ‘y’
 std::for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x--<n && n<y++) sum+=n; });
                                                                     ^~
```

# Mutable lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) mutable { if (x-- < n && n < y++)
                                                                sum += n; });
    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

# Globals are not captured

```
int   x = 0;
int   y = 0;

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

    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) mutable { if (x-- < n && n < y++)
                                                              sum += n; });
    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

$ g++ -std=c++11 -Wall -pedantic lambda4.cpp
lambda5.cpp: In function main:
lambda5.cpp:19:43: warning: capture of variable ‘x’ with non-automatic storage duration
lambda5.cpp:19:45: warning: capture of variable ‘y’ with non-automatic storage duration
```

# Globals are not captured

```
int   x = 0;
int   y = 0;

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

    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum](int n) mutable { if (x-- < n && n < y++)
                                                            sum += n; });
    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = -8
y = 15
```

# Capturing summary

- No capture

  [ ]
- By value

  [ x,y]  [=]
- By reference

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

  [ =, &x, &y ]  [ &, x, y ]
- Only automatic lifetime (local) variables could be captured
- Constness is preserved on capture
- Global variables, static members, heap storage can be used if visible 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**
- Since C++17 ***this** can be captured (by value)
- 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;  // global

struct X
{
  X(int i) : ii(i) {}
  int ii;
  void addLambda()
  {
    f = [=](int n) { if (n == ii) cout << n;  // [=] captures this if needs
                     else         cout << ii; // this->ii, indicates capturing this
                   };
  }
};

int main()
{
  {
    std::unique_ptr<X> up = std::make_unique<X>(4);
    up->addLambda();
    f(4);
  }  // object pointed by "up" destroyed here

  f(4);    // Likely aborts! The captured this points to already dead object
  return 0;
}
```

# Copying lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    cin >> x >> y;  // read 2 7


    for_each(v.begin(),v.end(),f);

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 0
x = 2
y = 7
```

# Copying lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;
    // auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    for_each(v.begin(),v.end(),f);

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

# Copying lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;
    // auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    for_each(v.begin(),v.end(),f);
    for_each(v.begin(),v.end(),f);  // 2nd time

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 88
x = 2
y = 7
```

# Copying lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // copy constr

    for_each(v.begin(),v.end(),f);  // copy constr









    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

# Copying lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // copy constr

    for_each(v.begin(),v.end(),f);  // copy constr

    // f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };  op= deleted




    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

# Copying lambda

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

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // copy constr

    for_each(v.begin(),v.end(),f);  // copy constr

    // f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };  op= deleted

    const auto& f2 = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // ref

    for_each(v.begin(),v.end(),f2);  // copy constr

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 88
x = 2
y = 7
```

# 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
```

# Conversion to function pointer

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

    deque<double> dd;
    deque<int>    di;

    double (*fp1)(int) = [](int n) -> double  { return n / 2.0; };   // no capture
    int    (*fp2)(int) = [](int n) -> int     { return n / 2.0; };   // no capture
    void   (*fp3)(double) = [](double n) { std::cout << n << " "; }; // no capture

    std::transform(v.begin(), v.end(), front_inserter(dd), fp1);
    std::transform(v.begin(), v.end(), back_inserter(di),  fp2);

    for_each(dd.begin(), dd.end(), fp3);
    cout << endl;
    for_each(di.begin(), di.end(), fp3);
    cout << endl;

    return 0;
}
```

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

# IIFE – Immediately Invoked Function Expression

```
/* 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:

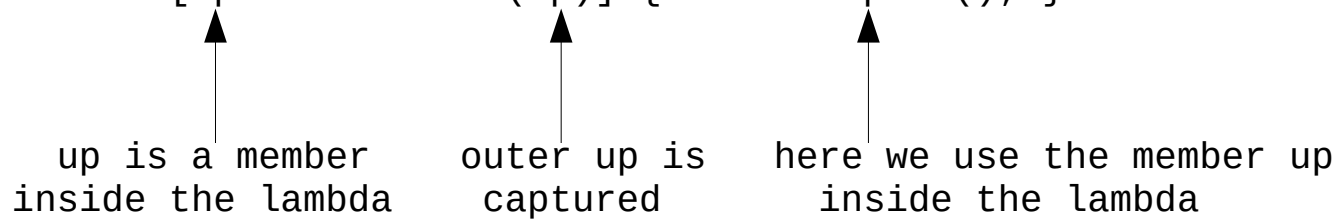
```
struct /* anonymous */
{
    template <typename T, typename U>
    auto operator()(const T& x, U& y) const // N3386 Return type deduction
    {
        return x + y;
    }
} L;
```

# Generalized lambdas in C++14

```
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::string as = “Hello”, bs = “World”;

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

    return 0;
}
```

# Init capture in C++14

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

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




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

auto func = [x = std::as_const(x)] { ... }     // make x const inside the lambda
```

# Init capture 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;
}
```

# Constexpr lambda in C++17

```
template<typename Range, typename Func, typename T>
constexpr T SimpleAccumulate(const Range& range, Func func, T init)
{
    for (auto &&elem : range)
    {
        init += func(elem);
    }
    return init;
}
int main()
{
    constexpr int t[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    constexpr int x = 2;
    constexpr int y = 7;

    constexpr auto f = [x,y](int n) { return (x < n && n < y) ? 0 : n; };
    constexpr int sum = SimpleAccumulate( t, f, 0);


    static_assert( 27 == sum );

    std::cout << "sum = " << SimpleAccumulate( t, f, 0) << '\n';
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    return 0;
}
```

# Constexpr lambda in C++17

```
template<typename Range, typename Func, typename T>
constexpr T SimpleAccumulate(const Range& range, Func func, T init)
{
    for (auto &&elem: range)
    {
        init += func(elem);
    }
    return init;
}
int main()
{
    constexpr int t[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    constexpr int x = 2;
    constexpr int y = 7;

    // constexpr auto f = [x,y](int n) { return (x < n && n < y) ? 0 : n; };
    constexpr int sum = SimpleAccumulate( t,
                              [x,y](int n) { return (x < n && n < y) ? 0 : n; };, 0);

    static_assert( 27 == sum );

    std::cout << "sum = " << SimpleAccumulate( t, f, 0) << '\n';
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    return 0;
}
```

# Capture *this in C++17

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

void my_struct::value()
{
  [=, this](){};     // C++17 error: = captures this by default, ok since C++20
  [=, *this](){};    // captures my_struct by value since C++17
  [this, *this](){}; // ok since C++20: repeating this in capture
}
```

# C++20

- Allow [=,this]
- Pack expression in init capture  [...args = std::move(args)]
- Capture for structured bindings
- Template lambdas (with concepts)
- Default constructible and assignable lambdas (if no state)
- Lambdas in unevaluated context (pl. sizeof)
- *this is captured by reference if captured implicitly by [=] or [&]
- *this captured by [=] is deprecated