CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Lambdas and Functions

Mauro Bianco

Advanced C++ for HPC

A special syntax

- Creates a function on the fly

```
auto f = [](int a, int b) { return a + b; };
int x = f(45, 77);
vector<int> v={1,2,3,4,5};
for_each(v.begin(), v.end(), [](int & v){v++});
```

- Mental model

```
auto f = [](int a, int b) { return a + b; };

Becomes:
struct <some_name> {
    auto operator()(int a, int b) const {
      return a+b;
    }
};
```

Captures

- Provide state

```
int t = 8;
auto f = [t](int a, int b) { return t*(a + b); };
int x = f(45, 77);
```

- Mental model

```
auto f = [t](int a, int b) { return t*(a + b); };

Becomes:
struct <some_name> {
    int t = 8;
    auto operator()(int a, int b) const {
        return t*(a+b);
    }
};
```

A special syntax

```
int t = 8;
auto f = [t](int a, int b) mutable { t++; return t*(a + b); };
int x = f(45, 77);
Assert(t==8);
```

- Mental model

```
auto f = [t](int a, int b) mutable { return t*(a + b); };

Becomes:
struct <some_name> {
   mutable int t = 8;
   auto operator()(int a, int b) const {
      t++;
      return t*(a+b);
   }
};
```

Generic Lambdas

- C++14: argument types are deduced

```
auto f = [](auto a, auto b) { return a + b); };
auto x = f(45, 4.67);
```

- Mental model

```
auto f = [](auto a, auto b) { return a + b; };

Becomes:
struct <some_name> {
   template <class T, class U>
   auto operator()(T a, U b) const {
      return a+b;
   }
};
```

Renaming

- C++14 on

```
bool sum = true;
auto f = [flag=sum](int a, int b) {
            return flag?(a + b):(a-b);
         };
```

```
struct <some_name> {
   int flag;

   <some_name>(bool x): flag(x) {}

   auto operator()(int a, int b) const {
      return flag?(a+b):(a-b);
   }
};
```

Capture list

- Every variable can be captured in different ways

```
bool sum = true;
int x = 42;
auto f = [sum, &x](int a, int b) {
             x = !sum?(a + b):(a-b);
             return sum?(a + b):(a-b);
};
```

```
struct <some_name> {
   bool sum;
   int& x;

   <some_name>(bool sum, int& x)
       : sum(sum), x(x) {}

   auto operator()(int a, int b) const {
       ...
   }
};
```

Shortcuts: Capture all by Reference

```
bool sum = true;
int x = 42;
auto f = [&](int a, int b) /*mutable*/ {
             sum = ((x>0) == !sum);
             x=7;
             return sum?(a + b):(a-b);
         };
```

```
struct <some_name> {
   bool& sum;
   int& x;

   <some_name>(bool sum, int x)
       : sum(sum), x(x) {}

   auto operator()(int a, int b) const {
       ...
   }
};
```

Shortcuts: Capture all by Value

```
bool sum = true;
int x = 42;
auto f = [=](int a, int b) mutable {
            sum = ((x>0) == !sum);
            x=7;
            return sum?(a + b):(a-b);
         }; assert(x==42);
```

```
struct <some_name> {
   mutable bool sum;
   mutable int x;

   <some_name>(bool sum, int x)
       : sum(sum), x(x) {}

   auto operator()(int a, int b) const {
      ...
   }
};
```

Capture All Except…

```
bool sum = true;
int x = 42;
auto f = [=, &x](int a, int b) mutable {
             sum = ((x>0) == !sum);
             x=7;
             return sum?(a + b):(a-b);
         };
```

Explicit Return

```
auto copy_complex = [](complex c) -> complex {return c;};
complex copy = copy_complex(complex{4.,2.});

auto sum_inplace = [](complex &c, complex d)
                   -> auto& {c += d; return c;};
```

Lambdas and function pointers

- Lambdas can be converted in function pointers!

```
void run(int (*f)(int, int)) {
    assert(f);
    f(6,4);
}

int main() {
    run([](int a, int b) {return a+b;});
}
```

Capturing in a Member Function

```
struct A {
  int a;

  void operator()() {
    auto f = [=]() {int a=0; a++; std::cout << a << “\n”;};
    f();
  }

  void alternate() {
    auto f = [=]() {a++; std::cout << a << “\n”;};
    f();
  }

  void alternate2() {
    auto f = [=]() {this->a++; std::cout << this->a << “\n”;};
    f();
  }

  void alternate3() {
    int a = 5;
    auto x = [=] () { std::cout << this->a << a << “\n”;};
    a = 3;
    x();
  }
};
```

Shadowing a

Data member of A

This is captured by value!

This would shadows the data member

Capturing in a Member Function (C++17)

Copies ***this**
The class must be copy constructible

```
struct A {
  int a;

  void operator()() {
    auto f = [*this]() {
                 a++;
                 std::cout << a << “\n”;};
    f();
  }
```

std::function

```
float foo(int a, int b) {
    return static_cast<float>(a+b);
}

struct A {
    float operator()(int a, int b) {
        return static_cast<float>(a+
    }
};

int main() {
    std::function<float(int,int)> f = [](int a,int b)
        {
            return static_cast<float>(a+b);
        };
    f(3,4);

    f = foo;
    f(3,4);

    f = A();
    f(3,4);
}
```

Every invocation is a virtual function call

Running the function

Re-targeting to stand-alone function

Retargeting to member operator()

std:mem_fn

```
struct A {
    template<typename T>
    void display_thing(T i) {
        std::cout << "number: " << i << '\n';
    }
}


int main() {
    A a;

    auto print_num = std::mem_fn(&A::display_thing<int>);
    print_num(a, 42);

    std::unique_ptr<A> b{new A};
    print_num(b, 42);
}
```

Bind and Placeholders

```
int foo(int a, int b) {return a - b;}

int main() {
    using namespace std::placeholders;

    auto x = std::bind(foo, _1, 4);
    x(7);
}
```

Equivalent to
`foo(7, 4)`

```
int main() {
    using namespace std::placeholders;

    std::bind(foo, _1, 4)(6); // 2
    std::bind(foo, _1, _1)(6); // 0
    std::bind(foo, _2, _2)(6,8); // 0
    std::bind(foo, _2, _1)(6,4); // -2
}
```

Dealing with References

```
int bar(int &a, int &b) {return a + b++;}

int main() {
    using namespace std::placeholders;

    int x = 4;
    int y = 6;
    std::bind(bar, _1, std::ref(x))(y);
}
```

Dealing with Const References

```
int foo(int const &a, int const& b) {return a + b;}


int main() {
    using namespace std::placeholders;

    int x = 4;
    SHOW(std::bind(foo, _1, std::cref(x))(6));

}
```

The Target Method

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

void run(int (*f)(int, int)) {
    assert(f);
    f(6,4);
}

int main() {
    run([](int a, int b) {return a+b;});
    run(std::bind(foo, _1, _2));
    std::function<int(int,int)> my_f = foo;
    run(my_f);
    run(*my_f.target<int(*)(int,int)>());

    auto wrongf = my_f.target<int(*)(float &)>();
    assert(wrongf == nullptr);
}
```

With member functions

```
struct A {
    int v;
    A(int v) : v(v) {}

    static int member(int, int) {return 80;}
    int member2(int, int) {return v;}
};

int main() {
    std::function<int(int,int)> member1 = A::member;
    to_run = (member1.target < int(*)(int,int)>());
    run(*to_run);

    A a(42);
    std::function<int(A*,int,int)> member2 = &A::member2;
    SHOW(member2(&a, 3,4));

    function<int(int,int)> member3 = bind(&A::member2, &a, _1, _2);
    SHOW((member3(3,4)));
}
```

Best Practices

- Lambdas are good (think about mental model)
- Bind is as efficient as calling the functions
  - But cannot be converted to function pointers
- std::functions have runtime overhead