Lambda Functions, Ranges, Algorithm and Functional Library

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

Autumn 2020

Function as a Parameter

```
int foo(int a, int b) {
    return a * 3 + b;
}
```

How to create a pointer to a function?

Function as a Parameter

```
int foo(int a, int b) {
    return a * 3 + b;
}
```

How to create a pointer to a function?

```
int main() {
    auto f = foo;
    std::cout << f(3, 8) << '\n';
}
```

What is the type of `f`?

Function as a Parameter

```cpp
int foo(int a, int b) {
    return a * 3 + b;
}
```

How to create a pointer to a function?

```cpp
int main() {
    auto f = foo;
    std::cout << f(3, 8) << '\n';
}
```

What is the type of `f`?

- `int(*)(int, int)`

Function Type

```
int foo(int a, int b) {
    return a * 3 + b;
}
```

The type of the `foo` function is `int(int, int)`

```
using FooType = int(int, int);
FooType *ptrToFoo = foo;
using FooPtrType = int (*)(int, int);
FooPtrType ptrToFoo2 = foo;
```

Function Type

```
int foo(int a, int b) {
    return a * 3 + b;
}
```

The type of the `foo` function is `int(int, int)`

```
using FooType = int(int, int);
FooType *ptrToFoo = foo;
using FooPtrType = int (*)(int, int);
FooPtrType ptrToFoo2 = foo;
```

The older way: `typedef int (*FooPtrT)(int, int);`

Function Type

```cpp
int foo(int a, int b) {
    return a * 3 + b;
}
```

The type of the `foo` function is `int(int, int)`

```cpp
using FooType = int(int, int);
FooType *ptrToFoo = foo;
using FooPtrType = int (*)(int, int);
FooPtrType ptrToFoo2 = foo;
```

The older way: `typedef int (*FooPtrT)(int, int);`

What is `int (*getFun(char op))(int, int);`?

Function Type

```
int foo(int a, int b) {
    return a * 3 + b;
}
```

The type of the `foo` function is `int(int, int)`

```
using FooType = int(int, int);
FooType *ptrToFoo = foo;
using FooPtrType = int (*)(int, int);
FooPtrType ptrToFoo2 = foo;
```

The older way: `typedef int (*FooPtrT)(int, int);`

What is `int (*getFun(char op))(int, int);`?

Note: Functions get automatically cast to function pointers and vice versa. Thus `foo`, `&foo`, and `*foo` behave the same if `foo` is a function.

Member Functions as a Parameters

```
struct X {
    int foo(int a, int b) { return a * 3 + b; }
};
```

How to create a pointer to a member function?

Member Functions as a Parameters

```
struct X {
    int foo(int a, int b) { return a * 3 + b; }
};
```

How to create a pointer to a member function?

```
int main() {
    X x;
    auto f = &X::foo;
}
```

- What is the type of `f`?

Member Functions as a Parameters

```
struct X {
    int foo(int a, int b) { return a * 3 + b; }
};
```

How to create a pointer to a member function?

```
int main() {
    X x;
    auto f = &X::foo;
}
```

- What is the type of `f`?
  - `int (X::*)(int, int)`
- How can we call `f`?

Member Functions as a Parameters

```
struct X {
    int foo(int a, int b) { return a * 3 + b; }
};
```

How to create a pointer to a member function?

```
int main() {
    X x;
    auto f = &X::foo;
}
```

- What is the type of `f`?
  - `int (X::*)(int, int)`
- How can we call `f`?
  - `(x.*f)(3, 8)`
  - `(ptrToX->*f)(3, 8)`
- Is the ampersand necessary?

Member Functions as a Parameters

```
struct X {
    int foo(int a, int b) { return a * 3 + b; }
};
```

How to create a pointer to a member function?

```
int main() {
    X x;
    auto f = &X::foo;
}
```

- What is the type of `f`?
  - `int (X::*)(int, int)`
- How can we call `f`?
  - `(x.*f)(3, 8)`
  - `(ptrToX->*f)(3, 8)`
- Is the ampersand necessary?
  - Yes. Rules for taking the address of a member function are different from the old C rules for plain functions.

Lambda Functions

It is often useful to be able to create a function just for one use – an anonymous function.

```
int large = std::count_if(v.begin(), v.end(),
                          [](int x) { return x > 42; });
```

- a lambda function that takes `int` and returns `bool`

Lambda Functions

It is often useful to be able to create a function just for one use – an anonymous function.

```
int large = std::count_if(v.begin(), v.end(),
                          [](int x) { return x > 42; });
```

- a lambda function that takes `int` and returns `bool`

```
[](int x) { return x > 42; }
```

- empty capture list
- does not have access to any local variables in its scope

Lambda Functions

It is often useful to be able to create a function just for one use – an anonymous function.

```
int large = std::count_if(v.begin(), v.end(),
                          [](int x) { return x > 42; });
```

- a lambda function that takes `int` and returns `bool`

```
[](int x) { return x > 42; }
```

- argument list as for normal function

Lambda Functions

It is often useful to be able to create a function just for one use – an anonymous function.

```
int large = std::count_if(v.begin(), v.end(),
                          [](int x) { return x > 42; });
```

- a lambda function that takes `int` and returns `bool`

```
[](auto x) { return x > 42; }
```

- argument list as for normal function
- including `auto` 14
    - useful if the type cannot be known beforehand
    - o r to simplify

Lambda Functions

It is often useful to be able to create a function just for one use – an anonymous function.

```
int large = std::count_if(v.begin(), v.end(),
                          [](int x) { return x > 42; });
```

- a lambda function that takes `int` and returns `bool`

```
[](std::integral auto x) { return x > 42; }
```

- argument list as for normal function
- including `auto` 14
  - useful if the type cannot be known beforehand
  - o r to simplify
- including concepts 20
- can be variadic

Lambda Functions

It is often useful to be able to create a function just for one use – an anonymous function.

```
int large = std::count_if(v.begin(), v.end(),
                          [](int x) { return x > 42; });
```

- a lambda function that takes `int` and returns `bool`

```
[](int x) { return x > 42; }
```

- normal function body
- return type deduced
  - single-expression lambdas C++11
  - all lambdas (all returns of the same type) C++14

Lambda Functions

It is often useful to be able to create a function just for one use – an anonymous function.

```
int large = std::count_if(v.begin(), v.end(),
                          [](int x) { return x > 42; });
```

- a lambda function that takes `int` and returns `bool`

```
[](int x) -> bool { return x > 42; }
```

- return type can be explicit

Lambdas With Capture

The power of lambda functions is in captures.

```
struct Bar {
  void foo(const std::vector<int> &vec, int p) {
    int cnt = std::count_if(vec.begin(), vec.end(),
                            [this, p](int x) { good(x); });
  }
  bool good(int);
};
```

- the lambda can call member functions of Bar because it captures this and p (by value)

Lambdas With Capture

The power of lambda functions is in captures.

```
struct Bar {
  void foo(const std::vector<int> &vec, int p) {
    int cnt = std::count_if(vec.begin(), vec.end(),
                             [this, p](int x) { good(x); });
  }
  bool good(int);
};
```

```
[this, p](int x) { good(x + p); }
```

- value capture: values copied to lambda

Lambdas With Capture

The power of lambda functions is in captures.

```
struct Bar {
  void foo(const std::vector<int> &vec, int p) {
    int cnt = std::count_if(vec.begin(), vec.end(),
                            [this, p](int x) { good(x); });
  }
  bool good(int);
};
```

```
[this, p](int x) mutable { good(x + p++); }
```

- value capture: values copied to lambda
- `mutable` can be used to allow modification of the copies

Lambdas With Capture

The power of lambda functions is in captures.

```
struct Bar {
  void foo(const std::vector<int> &vec, int p) {
    int cnt = std::count_if(vec.begin(), vec.end(),
                            [this, p](int x) { good(x); });
  }
  bool good(int);
};
```

```
[this, &p](int x) { good(x + p); }
```

- reference capture: p captured by reference, original can be modified
- **this** cannot be be caught by reference

Lambdas With Capture

The power of lambda functions is in captures.

```
struct Bar {
  void foo(const std::vector<int> &vec, int p) {
    int cnt = std::count_if(vec.begin(), vec.end(),
                            [this, p](int x) { good(x); });
  }
  bool good(int);
};
```

```
[=, this](int x) { good(x + p); }
```

- wildcard capture: all by value
- can be combined: `[=, &x, &y]`
- only used variables stored

Lambdas With Capture

The power of lambda functions is in captures.

```
struct Bar {
  void foo(const std::vector<int> &vec, int p) {
    int cnt = std::count_if(vec.begin(), vec.end(),
                            [this, p](int x) { good(x); });
  }
  bool good(int);
};
```

```
[&](int x) { good(x + p); }
```

- wildcard capture: all by reference

How Does Lambda Capture Work?

The compiler creates a closure object:

```
std::count_if(vec.begin(), vec.end(),
              [this](int x) { good(x); });
```

translates to something like

```
struct Lambda {
    Lambda(Bar *that) : that(that) { }
    bool operator()(int x) const {
        return that->good(x);
    }
    Bar *that;
};
std::count_if(vec.begin(), vec.end(), Lambda(this));
```

How Does Lambda Capture Work?

The compiler creates a closure object:

```
std::count_if(vec.begin(), vec.end(),
              [this](int x) { good(x); });
```

translates to something like

```
struct Lambda {
    Lambda(Bar *that) : that(that) { }
    bool operator()(int x) const {
        return that->good(x);
    }
    Bar *that;
};
std::count_if(vec.begin(), vec.end(), Lambda(this));
```

- each lambda gets a unique type
  - cannot be written by the programmer
  - is compiler specific

More On Capture

- the capture can also create values: C++14
  `[i = 0](int x) mutable { return x + i++; }`
  - a lambda with an additional variable `i` (of deduced type `int`) stored in the lambda object
  - type cannot be written explicitly (deduced as in `auto i = 0`)
  - useful for move-only objects `[ptr = std::move(ptr)]`
  - also by reference: `[&x = y]`

More On Capture

- the capture can also create values: C++14
  `[i = 0](int x) mutable { return x + i++; }`
  - a lambda with an additional variable `i` (of deduced type `int`) stored in the lambda object
  - type cannot be written explicitly (deduced as in `auto i = 0`)
  - useful for move-only objects `[ptr = std::move(ptr)]`
  - also by reference: `[&x = y]`
- the current object can be captured by value: C++17
  - `[*this]() { return foo(); }`
  - creates a copy of the object pointed to by `this` in the lambda
  - `this` inside lambda body points to the copy

More On Capture

- the capture can also create values: C++14
  `[i = 0](int x) mutable { return x + i++; }`
  - a lambda with an additional variable `i` (of deduced type `int`) stored in the lambda object
  - type cannot be written explicitly (deduced as in `auto i = 0`)
  - useful for move-only objects `[ptr = std::move(ptr)]`
  - also by reference: `[&x = y]`
- the current object can be captured by value: C++17
  - `[*this]() { return foo(); }`
  - creates a copy of the object pointed to by `this` in the lambda
  - `this` inside lambda body points to the copy
- `this` should be captured explicitly when by-value capture used:
  `[=, this]` or `[=, *this]` C++20

Templated Lambdas (`auto` Lambdas) C++14

Lambda arguments can be templates:

```
std::count_if(v.begin(), v.end(),
      [=](const auto &x) { return good(x); });
```

- creates the following templated operator in the lambda object

  ```
  template <typename Arg1T>
  bool operator()(const Arg1T& x) const {
      return that->good(x);
  }
  ```

- in gcc's support library there is bug which causes the demangler in gdb to fail when it encounters a name of an `auto` lambda
  - demangler error can be ignored and debugging resumed

Templated Lambdas C++20

Templates can be also explicitly named, possibly with **requires**

```
[]<std::input_iterator It, std::sentinel_for<It> End>
  (It from, End to) { ... }

[]<typename It, typename End>
  requires std::input_iterator<It>
        && std::sentinel_for<End, It>
  (It from, End to) { ... }
```

More on Lambdas

- beware of dangling references in capture-by-reference

```
auto getAdder(int x) {
    return [&](int y) { return x + y; };
}
```

  - solution: use capture-by-value

More on Lambdas

- beware of dangling references in capture-by-reference

```
auto getAdder(int x) {
    return [&](int y) { return x + y; };
}
```

  - solution: use capture-by-value

- capture-less lambdas can be converted to function pointers:

```
std::set_terminate([]() {
    std::cerr << "terminate called" << std::end;
    std::abort();
});
```

  - the lambda object actually defines a cast-to-function-pointer operator

More on Lambdas

- beware of dangling references in capture-by-reference

```
auto getAdder(int x) {
    return [&](int y) { return x + y; };
}
```

  - solution: use capture-by-value

- capture-less lambdas can be converted to function pointers:

```
std::set_terminate([]() {
    std::cerr << "terminate called" << std::end;
    std::abort();
});
```

  - the lambda object actually defines a cast-to-function-pointer operator

- the argument list can be missing: `[x] { return x; }`

Lambda as a Function Argument

The type of lambda is not known to the programmer, how can a function take such an argument?

Lambda as a Function Argument

The type of lambda is not known to the programmer, how can a function take such an argument?

1. templates

   ```
   template <typename Fun>
   auto foo(Fun fun) { // note: passing by value
       /* ... */
       fun(x, y);
   ```

   - `foo` has to be defined in a header file
   - the types of lambda's arguments cannot be written explicitly

Lambda as a Function Argument

The type of lambda is not known to the programmer, how can a function take such an argument?

1. templates

   ```
   template <typename Fun>
   auto foo(Fun fun) { // note: passing by value
       /* ... */
       fun(x, y);
   ```

   - `foo` has to be defined in a header file
   - the types of lambda's arguments cannot be written explicitly

2. templates with concepts C++20

   ```
   auto foo(std::invocable<int, int> auto fun) {
       /* ... */
       fun(x, y);
   ```

   - type is statically derived (as with plain templates)
   - any concept can be used – constraining return type, ...
   - also `std::predicate`

...

Lambda as a Function Argument

3. `std::function`

   ```
   auto foo(std::function<void (int, int)> fun)
   ```

   - slower (usually cannot be inlined, possibly uses virtual methods)

4. by function pointer
   - only capture-less lambdas
   - faster than `std::function`
   - slower that using templates (usually cannot be inlined)

`std::function`

a polymorphic wrapper that can hold any callable object satisfying the given signature

- basically a generalisation of a function pointer
    - can also hold a lambda or a functional object (instance of a class that defines the call operator `()`)
    - can even hold a member function of an object (**`this`** has to be passed as the first argument)
- defines a call operator `()` that forwards all arguments to the stored callable object
- can be empty (can be assigned from **`nullptr`**)
- the type of the stored object can be accessed if necessary
- defined in `<functional>`

std::function Example

```cpp
struct Foo { int get() { return x + 42; }; int x = 0; };
int bla(Foo &x) { return x.x + 16; }

int main() {
    Foo f;
    std::function<int(Foo &)> fun = &Foo::get;
    std::cerr << fun(f) << '\n'; // 42
    int i = 0;
    fun = [&](Foo &x) { return x.x + i++; };
    std::cerr << fun(f) << '\n'; // 0
    std::cerr << fun(f) << '\n'; // 1
    fun = &bla;
    std::cerr << fun(f) << '\n'; // 16
}
```

How to Use Lambdas

- do not overuse them
- lambdas should be short
- if you need to name a capture-less lambda, consider using an ordinary function instead
- if your lambda is long, use a function or a method instead
- prefer references to copies of large data in the capture list if possible
  - the generated class will be smaller
  - however, references can be dangerous, so be careful
- if the number of lambdas is higher than the number of functions/methods, you should consider refactoring your code

Partial Application, `std::bind`, `std::bind_front`

Sometimes it is useful to pass a member function together with its object, or a partially applied function to an algorithm:

- by lambda: `[this, x](auto y) { foo(x, y); }`

Partial Application, `std::bind`, `std::bind_front`

Sometimes it is useful to pass a member function together with its object, or a partially applied function to an algorithm:

- by lambda: `[this, x](auto y) { foo(x, y); }`
- using `std::bind`:

```
using namespace std::placeholders;
auto bar = std::bind(&Foo::foo, this, x, _1);
bar(5); // calls this->foo(x, 5)
```

Partial Application, `std::bind`, `std::bind_front`

Sometimes it is useful to pass a member function together with its object, or a partially applied function to an algorithm:

- by lambda: `[this, x](auto y) { foo(x, y); }`
- using `std::bind`:

  ```
  using namespace std::placeholders;
  auto bar = std::bind(&Foo::foo, this, x, _1);
  bar(5); // calls this->foo(x, 5)
  ```

  - binds function passed as first argument to some arguments, some of which can be represented by placeholders (`_1`,...,`_N`)
  - works with functions, callable objects, member functions
  - creates a callable object that accepts `N` arguments
  - return value type is unspecified

Partial Application, `std::bind`, `std::bind_front`

Sometimes it is useful to pass a member function together with its object, or a partially applied function to an algorithm:

- by lambda: `[this, x](auto y) { foo(x, y); }`
- using `std::bind`:

```
using namespace std::placeholders;
auto bar = std::bind(&Foo::foo, this, x, _1);
bar(5); // calls this->foo(x, 5)
```

  - binds function passed as first argument to some arguments, some of which can be represented by placeholders (`_1`,...,`_N`)
  - works with functions, callable objects, member functions
  - creates a callable object that accepts `N` arguments
  - return value type is unspecified
- using `std::bind_front` C++20

```
auto bar = std::bind_front(&Foo::foo, this, x);
bar(5); // calls this->foo(x, 5)
```

Note: Since C++14, all uses of bind can be written using a lambda.

Ranges C++20

Boring "Corporate" Example

- given a container of `Employees`, find the sum of all men's salaries
- to make things complicated, QA gets a 10% raise

Traditional solution:

```
int sum = 0;
for (const Employee& e : loadEmployees()) {
    if (e.gender == Gender::Female)
        continue;
    sum += e.department == "QA" ? 1.1 * e.salary : e.salary;
}
return sum;
```

Boring "Corporate" Example

- given a container of Employees, find the sum of all men's salaries
- to make things complicated, QA gets a 10% raise

Using STL algorithms:

```
std::vector<Employee> employees = loadEmployees();
employees.erase(
    std::remove_if(employees.begin(), employees.end(),
        [](const Employee& e) {
            return e.gender == Gender::Female;
        }), employees.end());
std::vector<int> s;
std::transform(employees.begin(), employees.end(),
    std::back_inserter(s), [](const Employee& e) {
        return e.department == "QA" ?
            1.1 * e.salary : e.salary;
    });
return std::accumulate(s.begin(), s.end(), 0);
```

Iterators and Algorithms in STL

- are supposed to provide higher abstraction and prevent code duplication
  - when you see `std::copy` you know what to expect
  - when you see `for` you have to put mental effort to find out what it does
  - algorithms should eliminate "off-by-one" bugs and similar
- have terrible syntax (as seen on the previous example)
  - almost in all cases we go from `begin()` to `end()`
  - in almost all cases we back-insert the result
- do not compose well (as seen on the previous example)
  - note the memory overhead of second solution

Iterators and Algorithms in STL

- are supposed to provide higher abstraction and prevent code duplication
  - when you see `std::copy` you know what to expect
  - when you see `for` you have to put mental effort to find out what it does
  - algorithms should eliminate "off-by-one" bugs and similar
- have terrible syntax (as seen on the previous example)
  - almost in all cases we go from `begin()` to `end()`
  - in almost all cases we back-insert the result
- do not compose well (as seen on the previous example)
  - note the memory overhead of second solution

Solution in C++20: concept of **ranges** and **range adaptors**

- provide abstraction
- do not lead to code duplication
- compose well and are efficient

Ranges C++20

```
std::vector numbers = {1, 2, 3, 4, 5, 6};

auto results = numbers
    | std::views::filter([](int n) { return n % 2 == 0; })
    | std::views::transform([](int n) { return n * 2; });

for (auto v : results)
    std::cout << v << " ";
```

- functional style of transformation of sequences of values
- composable
- lazy – values computed on-demand

Range C++20

- a range is an object with `begin()` and `end()` that return iterators
  - can be iterated over
  - concept `std::ranges::range`

Range C++20

- a range is an object with `begin()` and `end()` that return iterators
  - can be iterated over
  - concept `std::ranges::range`
- `end` iterator need not be the same type as `begin`
  - so-called *sentinel* – can be compared with an iterator to detect end of a range
  - `std::iterator` and `std::sentinel_for<It>` concepts
  - *note*: range-for allows use of sentinels C++17

Range C++20

- a range is an object with `begin()` and `end()` that return iterators
  - can be iterated over
  - concept `std::ranges::range`
- end iterator need not be the same type as `begin`
  - so-called *sentinel* – can be compared with an iterator to detect end of a range
  - `std::iterator` and `std::sentinel_for<It>` concepts
  - *note*: range-for allows use of sentinels C++17
- all containers are ranges

Range C++20

- unlike containers, ranges do not have to own elements
- **range iterators can be smart**
- technically, there are many concepts of ranges
  - forward
  - random access
  - …
- ranges come with range-enabled algorithms

```
std::views::filter(nums, [](int n) { return n % 2 == 0; })
nums | std::views::filter([](int n) { return n % 2 == 0; })
```

Range Adaptors C++20

- can be combined with a range using operator | to produce a new range:

```
std::vector numbers = { 1, 2, 3, 4, 5 };
auto range = numbers | std::view::transform(multiplyBy(42));
```

- operator |:
  - semantics similar to UNIX pipes: “feeds output of the left side to the right side”
  - is left associative (“read it from left to right”)

```
auto range = numbers
    | std::view::filter(isEven)
    | std::view::transform(multiplyBy(42));
```

Views = adaptors that do not own data and are lazy

Boring "Corporate" Example: Now with Ranges! C++20

- given a container of `Employees`, find sum of all men's salaries
- to make things complicated, QA gets a 10% raise

```
auto range = loadEmployees()
    | std::view::filter([](auto& e) {
        return e.gender == Gender::Man })
    | view::transform([](auto& e) {
        return e.department == "QA" ?
            1.1 * e.salary : e.salary; });
return std::accumulate(range.begin(), range.end());
```

Boring "Corporate" Example: Now with Ranges! C++20

- given a container of `Employees`, find sum of all men's salaries
- to make things complicated, QA gets a 10% raise

```
auto range = loadEmployees()
    | std::view::filter([](auto& e) {
        return e.gender == Gender::Man })
    | view::transform([](auto& e) {
        return e.department == "QA" ?
            1.1 * e.salary : e.salary; });
return std::accumulate(range.begin(), range.end());
```

- named functions instead of lambdas:

```
loadEmployees()
    | view::filter(isMan)
    | view::transform(raise({ { "QA", 1.1 } }))
);
```

Function naming leads to more readable code!

Performance of ranges C++20

- ranges are **lazy**
- only the employee under transformation occupies memory
- efficiency is roughly the same as for `for` loop implementation

Ranges in Practice

- only GCC ≥ 10 (no clang, no MSVC so far)
- "ranges are nice, but C++20 is too new"

Ranges in Practice

- only GCC $\geq$ 10 (no clang, no MSVC so far)
- "ranges are nice, but C++20 is too new"

- **range-v3**
  - reference implementation of ranges for standard library
  - open-source, `https://github.com/ericniebler/range-v3`
  - header-only, compatible with C++11 and higher
  - ready to be used in practice
  - not everything in ranges-v3 is in C++20 (e.g., `accumulate`)
- **Boost** also provides its own implementation

range-v3 overview

Documentation: `https://ericniebler.github.io/range-v3/`

- views: produce new ranges, lazy if possible:
  - `view::zip`
  - `view::zip_with`
  - `view::take`
  - `view::tail`
  - `view::tokenize`
  - `view::reverse`
  - ...
- actions: mutate container in place, not lazy:
  - `action::sort`
  - `action::unique`
  - ...
- algorithms: STL algorithms taking ranges as arguments

The Old Way – C++ Algorithm Libraries

C++ Algorithm Libraries

Recommended talk from CppCon 2018:
Jonathan Boccara: 105 STL Algorithms in Less Than an Hour
`https://www.youtube.com/watch?v=2olsGf6JIkU`

C++ Algorithm Libraries

headers `<algorithm>`, `<numeric>`

many useful algorithms on collections/iterators

- `sort`, `stable_sort`, `nth_element`, `binary_search`, `reverse`
- `all_of`, `any_of`, `node_of`
- `find`, `find_if`, `count`, `count_if`
- `copy`, `move`
- `accumulate`, `inner_product`

- especially usable with lambda functions
- beware of using them for things that can be written clearer and shorter with range-based `for`
- all that take function arguments take them using templates and by value:

```
template <typename InputIterator, typename T,
          typename BinaryOperation>
T accumulate(InputIterator first, InputIterator last,
             T init, BinaryOperation binary_op);
```

Iterators

Iterators are central concept for containers and algorithms in the standard C++ library:

- in a sense a generalisation of pointers
  - point to some place in a collection
  - can be dereferenced to obtain (a reference to) a value in the collection
  - can be incremented, optionally decremented, added to, compared
- requirements described by the *Iterator* concept and its extensions
  - concepts are a way to describe a set of types which satisfy the same public interface
  - used in C++ standard, documentation, not part of the language (yet)

Iterator Concepts I

Iterator concept:

- for type `It` to be iterator it must meet the following:
- be copy constructible, copy assignable, destructible
- lvalues need to be swappable
- if `r` is an lvalue of type `It` then `*r` and `++r` must be valid expressions and `++r` must return `It &`
  - but certain `r` might not be dereferenceable:
  - if `r` points past-the-end of a container
  - if it was invalidated by an operation on the container
  - if it is not associated with any container
- `std::iterator_traits<It>` must define the member types `value_type`, `difference_type`, `reference`, `pointer`, and `iterator_category`
  - this can be done by defining these types in `It` itself, `std::iterator_traits` is a helper to allow adding those to pointer types
  - iterator category is used to determine the capabilities of the iterator

Iterator Concepts II

InputIterator concept extends **Iterator** in the following ways:

- is comparable (`==`, `!=`)
- `*i` returns `reference`, convertible to `value_type` (if `i` is dereferenceable)
- members of object pointed to by the iterator can be accessed: `i->m`
- postfix increment `i++`, but with unspecified return type
- `*i++` is convertible to `value_type`

Iterator Concepts III

ForwardIterator adds to **InputIterator**:

- must be default constructible
- *multipass* guarantee
  - `a == b` implies `++a == ++b`
  - if `a` and `b` are equal then they must both either be non-dereferenceable or `*a` and `*b` must be references to the same object
  - assignment to a mutable **ForwardIterator** cannot invalidate it
- `reference` must be a (constant or non-constant) lvalue reference to `value_type`
- equality must be defined between all iterators to the same sequence and (since C++14) the value-initialized iterators (`It{}`)
- `i++` must return `It` and be equivalent to `It o=i; ++i; return o;`

Iterator Concepts IV

BidirectionalIterator adds to **ForwardIterator**:

- `--i`, `i--`, and `*i--` are defined (analogous to ++ but moving back)

RandomAccessIterator adds to **BidirectionalIterator**:

- let `a`, `b` be iterators, `n` a numeric value of type `difference_type`
- iterator `a` can be added to, subtracted from:
  `a += n`, `a -= n`, `a + n`, `a - n`
  with constant complexity
- two iterators can be subtracted to obtain their distance: `a - b`
- iterator can be indexed: `a[n]` which is equal to `*(a + n)`
- they can be ordered: `a < b`, `a >= b`, `...`
  - `a < b` iff `b - a > 0`

Iterator Concepts V

ContiguousIterator adds to **RandomAccessIterator** (since C++17):

- for dereferenceable iterator values `a` and `(a + n)`, `*(a + n)` equivalent to `*(std::addressof(*a) + n)`
  - where `std::addressof` obtains real address of a reference even if it overloads `operator &`

OutputIterator adds the mutability requirement to any type which meets the **Iterator** concept:

- `*i` can be assigned to
- iterators which meet **OutputIterator** are called mutable iterators

The concept an iterator adheres to is indicated by `iterator_category` which should be `typedef` to one of:

- `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag`, `random_access_iterator_tag`
- (yes, there is none yet for **ContiguousIterator**)

More on Iterators I

- inheriting from `std::iterator` is deprecated in C++17
- there are some utilities and concepts in `<iterator>`
- `std::reverse_iterator` adaptor can be used to adapt a bidirectional iterator for reverse iteration
- `std::next`, `std::prev` can be used to advance iterators in single expression
  - `++x.begin()` might not be well defined, but `std::next(x.begin())` is
  - see https://en.cppreference.com/w/cpp/iterator/next#Notes
  - can also jump over elements: `std::next(x.begin(), 4)`

More on Iterators II

- `std::back_inserter`, `std::front_inserter`, `std::inserter` are useful for filling containers:

```
std::copy_if(a.begin(), a.end(),
             std::back_inserter(v),
             []( auto &x ) { return x.good(); } );
```

- prefer container's `insert` for bulk insertion from other containers

```
v.insert(v.end(), a.begin(), a.end());
```

  - eliminates repeated resizes

More on Iterators III

- `std::istream_iterator`, `std::ostream_iterator` are useful for combining algorithms with streams:

```
std::copy(s.begin(), s.end(),
          std::ostream_iterator<int>(std::cout, " "));
```

  - both use a template argument to set which type is read from/written to the stream
  - the second (optional) argument is a separator

More on Iterators III

- `std::istream_iterator`, `std::ostream_iterator` are useful for combining algorithms with streams:

```
std::copy(s.begin(), s.end(),
          std::ostream_iterator<int>(std::cout, " "));
```

  - both use a template argument to set which type is read from/written to the stream
  - the second (optional) argument is a separator
    (well, not really; it is output after every value)
  - maybe we will have `ostream_joiner` in the future