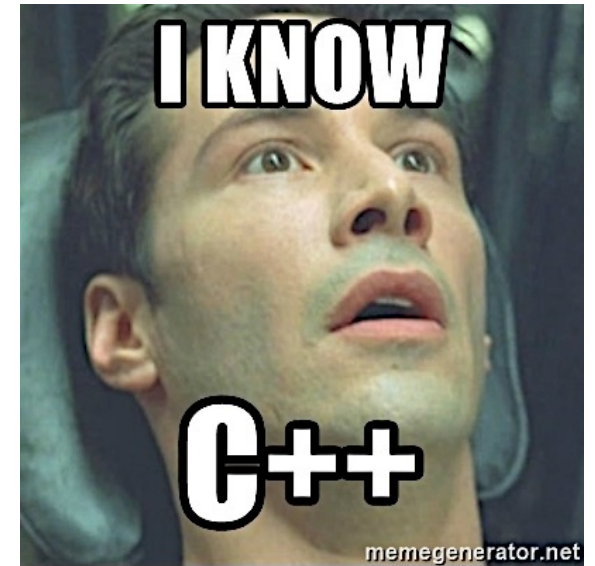
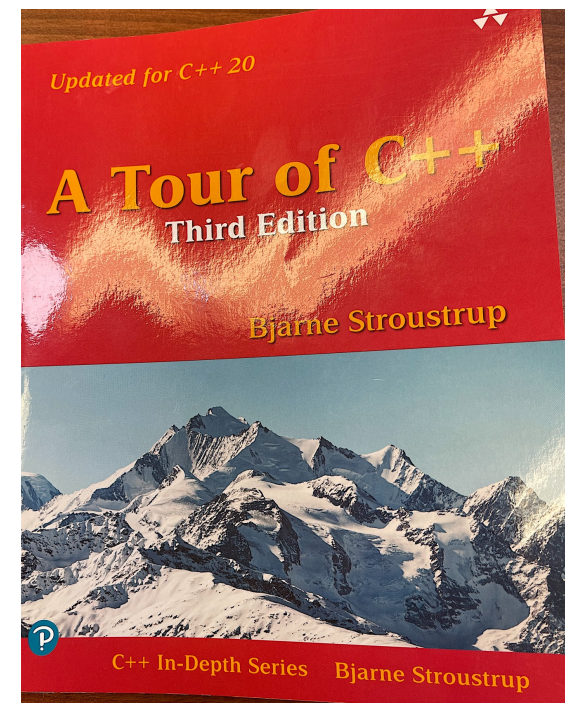


Lecture 21: C++ part VI



Suggested Literature



l-value references

l-value references

```
#include <iostream>

using namespace std;

void Print(double x)
{ // passed by value
  cout << "Print: " << x << endl;
  x = 2;
}

int main()
{
  double x = 10;

  Print(x);
  cout << "Main: " << x << endl;

  return 0;
}
```

In C, args **passed by value**.

Without pointers, where the address of a variable is passed by value, there is no way in C that function arguments can modify variables defined on **main()**

```
whovian@phy504:~/host/CPP$ g++ ref1.cpp -o ref1 -std=c++20
whovian@phy504:~/host/CPP$ ./ref1
Print: 10
Main: 10
```

l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

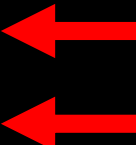
In C++, arguments can be passed by **l-value reference**

References are not pointers

No need for ***x** dereference operator (like in pointers)

Cannot change the address the function argument references

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10
Print2: 10
Main: 2
```



l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    return 0;
}
```

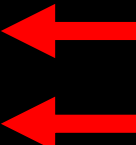
What is an **l-value** (C++03 def.)?

Variables that makes sense at the left side of an assignment

x = 2;

r-values (C++03 def.) can't be
*on the left side of an
assignment*

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10
Print2: 10
Main: 2
```



l-value references

```
#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    // passed by reference (no copy)
    // not exactly a pointer (cannot change
    // x to be address of another variable)
    // + no need to use the *x operator

    cout << "Print2: " << x << endl;
    x = 2;
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

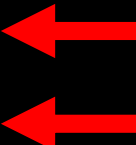
    return 0;
}
```

l-value reference is a reference to an **l-value** object. In **C++**, arguments can be passed by **l-value reference**

Why is this relevant?

Some classes are expensive to copy (some don't allow copy)

```
whovian@phy504:~/host/CPP$ g++ ref2.cpp -o ref2 -std=c++20
whovian@phy504:~/host/CPP$ ./ref2
Print: 10
Main: 10
Print2: 10
Main: 2
```



```

#include <iostream>

using namespace std;

void Print(double x) { // passed by value
    cout << "Print: " << x << endl;
    x = 2;
}

void Print2(double& x) {
    cout << "Print2: " << x << endl;
    x = 2;
}

void Print3(const double& x) {
    cout << "Print3: " << x << endl;
    // x = 2; (not allowed)
}

int main() {
    double x = 10;

    Print(x);
    cout << "Main: " << x << endl;

    Print2(x);
    cout << "Main: " << x << endl;

    Print3(x);
    cout << "Main: " << x << endl;

    return 0;
}

```

l-value references

l-value references are compatible with const correctness

Copy vs reference not relevant in speed for intrinsic types (e.g., **int, double, char, long double, long int..**)

```

whovian@phy504:~/host/CPP$ g++ ref3.cpp -o ref3 -std=c++20
whovian@phy504:~/host/CPP$ ./ref3
Print: 10
Main: 10
Print2: 10
Main: 2
Print3: 2
Main: 2

```

l-value references

```
#include <iostream>

using namespace std;

int main ()
{
    int my_super_long_name_variable = 2;
    cout << "my_super_long_name_variable = " <<
        my_super_long_name_variable << endl;

    int& x = my_super_long_name_variable;
    x = 3;

    cout << "my_super_long_name_variable = " <<
        my_super_long_name_variable << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ ref8.cpp -o ref8 -std=c++20
whovian@phy504:~/host/CPP$ ./ref8
my_super_long_name_variable = 2
my_super_long_name_variable = 3
```

l-value references can create multiple variables that affect the same address (similar to a pointer)

It serves as an alias (useful if the original var name is cumbersome)

```

#include <iostream>

using namespace std;

void Print(const double& x) {
    // compiler behind the scenes creates an
    // unnamed l-value temporary. This is
    // the so-called materialization (C++17 term)

    cout << "Print: " << x << endl;
}

void Print(const string& x) {
    cout << "Print: " << x << endl;
}

int main()
{
    Print(10);

    Print(double(15));

    Print(string("Hello World"));

    return 0;
}

```

An interesting example

r-values can bind to **const l-value references**. How?

The **const** keyword prevents the l-value reference from being on the left side of any assignment!

```

whovian@phy504:~/host/CPP$ g++ ref9.cpp -o ref9 -std=c++20
whovian@phy504:~/host/CPP$ ./ref9
Print: 10
Print: 15
Print: Hello World

```

```

#include <iostream>

using namespace std;

void Print(const double& x) {
    // compiler behind the scenes creates an
    // unnamed l-value temporary. This is
    // the so-called materialization (C++17 term)

    cout << "Print: " << x << endl;
}

void Print(const string& x) {
    cout << "Print: " << x << endl;
}

int main()
{
    Print(10);

    Print(double(15));

    Print(string("Hello World"));

    return 0;
}

```

An interesting example

Compiler creates a temporary unnamed restricted l-type variable from the materialization (C++17 word) of an **r-value**

```

whovian@phy504:~/host/CPP$ g++ ref9.cpp -o ref9 -std=c++20
whovian@phy504:~/host/CPP$ ./ref9
Print: 10
Print: 15
Print: Hello World

```

Lambda Functions

```

#include <iostream>
using namespace std;

int main () {
    // in C++11, 
    // no auto keyword on lambda's arguments
    auto pf = [](int x, int y) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };
    pf(2, 3);

    // C++>=14, that ceased to be a problem
    auto pf2 = [](auto x, auto y) {
        cout << x << " + " << y <<
            " = " << x+y << endl;
    };
    pf2(12.0, 20.2);

    // in C++11, lambdas was not compatible
    // with default arguments. In C++>=14,
    // they are compatible with default args

    // but default args with auto doesn't work
    auto pf3 = [](auto x, double y = 4.0) {
        cout << x << " + " << y <<
            " = " << x+y << endl;
    };
    pf3(1.0, 3.1);
    pf3(1.0);
}

```

C++ Lambdas

Anonymous functions

They are super useful in
STL algorithms

They are great for
readability of code

Think of them as holding
small "disposable" one-
time-use utility code

```

whovian@phy504:~/host/Phy504Docker/_CPP$ g++ lambda1.cpp -o lambda1 -std=c++20
whovian@phy504:~/host/Phy504Docker/_CPP$ ./lambda1
2 - 3 = -1
12 + 20.2 = 32.2
1 + 3.1 = 4.1
1 + 4 = 5

```

C++ Lambdas

```
#include <iostream>
using namespace std;
```

```
int main ()
```

```
{
```

```
    int y = 10;
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2.cpp -o lambda2 -std=c++20
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda2
```

```
20 - 10 = 10
```

```
30 - 10 = 20
```

```
y = 3
```

```
    // capture by value
```

```
    auto pf = [y](int x) {
```

```
        cout << x << " - " << y <<
            " = " << x-y << endl;
```

```
        // y is also read-only
```

```
        // y = 10; doesn't compile
```

```
    };
```

```
    pf(20);
```

```
    // capture by l-value reference
```

```
    auto pf2 = [&y](int x) {
```

```
        cout << x << " - " << y <<
            " = " << x-y << endl;
```

```
        y = 3;
```

```
    };
```

```
    pf2(30);
```

```
    cout << "y = " << y << endl;
```

```
}
```

Lambdas can “capture”
existing objects in their
scope (value or reference)

Capture by value makes the
variable read-only

You can override this
behavior with the **mutable**
keyword

C++ Lambdas

```
#include <iostream>
using namespace std;
```

```
int main ()
```

```
{
```

```
    int y = 10;
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2v2.cpp -o lambda2v2 -std=c++20
```

```
lambda2v2.cpp: In lambda function:
```

```
lambda2v2.cpp:13:7: error: assignment of read-only variable 'y'
```

```
    13 |         y = 10; // doesn't compile
```

```
        |         ~^~~~~
```

```
    // capture by value
```

```
    auto pf = [y](int x) {
```

```
        cout << x << " - " << y <<
            " = " << x-y << endl;
```

```
    // y is also read-only
```

```
    y = 10; // doesn't compile
```

```
};
```

```
pf(20);
```

```
    // capture by l-value reference
```

```
    auto pf2 = [&y](int x) {
```

```
        cout << x << " - " << y <<
            " = " << x-y << endl;
```

```
    y = 3;
```

```
};
```

```
pf2(30);
```

```
cout << "y = " << y << endl;
```

```
}
```

Lambdas can “capture”
existing objects in their
scope (value or reference)

Capture by value makes the
variable read-only

You can override this
behavior with the **mutable**
keyword

```
#include <iostream>
using namespace std;
```

```
int main ()
```

```
{
```

```
    int y = 10;
```

```
    // capture by value
```

```
    auto pf = [y](int x) mutable {
```

```
        cout << x << " - " << y <<
```

```
            " = " << x-y << endl;
```

```
        // y is also read-only
```

```
        y = 200; // 100% ok
```

```
        cout << x << " - " << y <<
```

```
            " = " << x-y << endl;
```

```
    };
```

```
    pf(20);
```

```
    // capture by l-value reference
```

```
    auto pf2 = [&y](int x) {
```

```
        cout << x << " - " << y <<
```

```
            " = " << x-y << endl;
```

```
        y = 3;
```

```
    };
```

```
    pf2(30);
```

```
    cout << "y = " << y << endl;
```

```
}
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2v3.cpp -o lambda2v3 -std=c++20
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda2v3
```

```
20 - 10 = 10
```

```
20 - 200 = -180
```

```
30 - 10 = 20
```

```
y = 3
```

C++ Lambdas

Lambdas can “capture”
existing objects in their
scope (value or reference)

Capture by value makes the
variable read-only

You can override this
behavior with the **mutable**
keyword

C++ Lambdas

```
#include <iostream>
using namespace std;

int main ()
{
    int y = 10;

    // capture by l-value reference
    auto pf = [&y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };

    pf(30);

    // Be very careful with capture
    // by l-value reference
    y = 0; // affects the next call
    pf(30);
}
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda2v4.cpp -o lambda2v4 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda2v4
30 - 10 = 20
30 - 0 = 30
```

Be careful with capture by
l-value reference

Any change to the “global”
variable in the scope up to
the point you call the
function affects the local
variable

C++ Lambdas

```
whovian@phy504:~/host/CPP$ g++ lambda2v4v2.cpp -o lambda2v4v2 -std=c++20
whovian@phy504:~/host/CPP$ ./lambda2v4v2
30 - 10 = 20
30 - 10 = 20
```

```
#include <iostream>
using namespace std;

int main ()
{
    int y = 10;

    auto pf = [y](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };

    pf(30);
    y = 0;
    pf(30);
}
```

What about pass-by value?

Completely different
behavior!

Copy happens at the line
**where the lambda is
defined!**

```
#include <iostream>
using namespace std;
```

```
int main ()
{
```

```
    int y = 10;
    int z = 20;
```

```
    // capture by value
```

```
    auto pf = [=](int x) {
        cout << x << " - " << y <<
            " + " << z << " = " <<
            x-y+z << endl;
        // y is also read-only
        // y = 10; doesn't compile
    };
    pf(20);
```

```
    // capture by l-value reference
```

```
    auto pf2 = [&](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        y = 3;
        z = 2;
    };
    pf2(30);
    cout << "y = " << y << endl;
    cout << "z = " << z << endl;
}
```

C++ Lambdas

```
whovian@phy504:~/host/Phy504Docker/Cpp$ g++ lambda3.cpp -o lambda3 -std=c++20
whovian@phy504:~/host/Phy504Docker/Cpp$ ./lambda3
20 - 10 + 20 = 30
30 - 10 = 20
y = 3
z = 2
```

Although not advisable,
lambdas can capture all the
objects it sees in the scope,
either by value [=] or by l-
value reference [&]

```
#include <iostream>
using namespace std;
```

```
int main ()
{
    int y = 10;
    int z = 20;

    // capture by value
    auto pf = [=](int x) mutable {
        cout << x << " - " << y <<
            " + " << z << " = " <<
            x-y+z << endl;
        y = 110;
        z = 200;
        cout << x << " - " << y <<
            " + " << z << " = " <<
            x-y+z << endl;
    };
    pf(20);

    // capture by l-value reference
    auto pf2 = [&](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
        y = 3;
        z = 2;
    };
    pf2(30);
    cout << "y = " << y << endl;
    cout << "z = " << z << endl;
}
```

C++ Lambdas

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda3v2.cpp -o lambda3v2 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda3v2
20 - 10 + 20 = 30
20 - 110 + 200 = 110
30 - 10 = 20
y = 3
z = 2
```

Although not advisable, lambdas can capture all the objects it sees in the scope, either by value [=] or l-value references [&]

[=] + **mutable** implies variables are not read-only (same behavior as w/ single capture)

C++ Lambdas

```
#include <iostream>
using namespace std;

int factory(int i) {
    return i * 10;
}

int main () {
    int z = 10;

    auto pf = [y = z+2](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };
    pf(20);

    auto pf2 = [y = factory(2)](int x) {
        cout << x << " - " << y <<
            " = " << x-y << endl;
    };
    pf2(20);
}
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda4.cpp -o lambda4 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda4
20 - 12 = 8
20 - 20 = 0
```

C++14 introduced the **generalized lambda capture**. It allows you to:

1. specify the name of a data member in the closure class generated from the lambda
2. specify an expression initializing that data member.

```

#include <iostream>
using namespace std;
int main () {
    int z = 10;
    // z = z * 10 defines a local variable
    // z with value equal to the captured
    // "global" z times 10 (local z = 100)
    auto pf3 = [&r = z, z = z * 10](int x) {
        cout << "r: " << r << endl;
        ++r; // "global" captured z is now 11
        cout << "r: " << r << endl;
        cout << "z: " << z << endl;
        cout << endl;
        return r + x + z;
    };

    auto res = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
    // every call of pf3 changes the global z,
    // But the local z is always 100 (it was
    // copied when the pf3 was defined)
    auto res2 = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
}

```

C++ Lambdas

C++14 introduced
the **generalized
lambda capture**.

Be careful, with terrible
design, you can make
your code hard to
understand (even with
simple calls to lambda
functions)

C++ Lambdas

```
#include <iostream>
using namespace std;
int main () {
    int z = 10;
    // z = z * 10 defines a local variable
    // z with value equal to the captured
    // "global" z times 10 (local z = 100)
    auto pf3 = [&r = z, z = z * 10](int x) {
        cout << "r: " << r << endl;
        ++r; // "global" captured z is now 11
        cout << "r: " << r << endl;
        cout << "z: " << z << endl;
        cout << endl;
        return r + x + z;
    };

    auto res = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
    // every call of pf3 changes the global z,
    // But the local z is always 100 (it was
    // copied when the pf3 was defined)
    auto res2 = pf3(20);
    cout << "pf3: " << pf3(20) << endl;
    cout << endl;
}
```

```
whovian@phy504:~/host/CPP$ ./lambda5
r: 10
r: 11
z: 100

pf3: r: 11
r: 12
z: 100

132

r: 12
r: 13
z: 100

pf3: r: 13
r: 14
z: 100

134
```

```

#include <iostream>
#include <utility>

using namespace std;

int main () {
    int z = 10;

    // const ref to avoid copy
    auto pf = [z = std::cref(z)](int x) {
        cout << "z: " << z << endl;
        return z;
    };

    cout << "pf: " << pf(20) << endl;
    z++;
    cout << "pf: " << pf(20) << endl;

    return 1;
}

```

```

whovian@whovianphy504:~/host/Lecture19$ g++ lambda5v2.cpp -o lambda5v2 -lfmt
whovian@whovianphy504:~/host/Lecture19$ ./lambda5v2
pf: z: 10
10
pf: z: 11
11

```

C++ Lambdas

Can we capture a variable by constant reference? YES

It avoids a potentially expensive copy (e.g., a large matrix) but ensures the global variable cannot be modified.

C++ Lambdas

```
whovian@phy504:~/host/Phy504Docker/_CPP$ g++ lambda6.cpp -o lambda6 -std=c++20
whovian@phy504:~/host/Phy504Docker/_CPP$ ./lambda6
200
```

```
#include <iostream>
using namespace std;

auto get_my_lambda(int y)
{
    return [y](int x){ return x * y; };
}

int main ()
{
    auto pf = get_my_lambda(10);
    cout << pf(20) << endl;
}
```

C++14: lambda functions can be returned from normal functions

Use the **auto** keyword to specify the types

C++ Lambdas + containers

```
#include <array>
#include <functional> // std::function
#include <fmt/core.h>
#include <fmt/ranges.h>
#include <ranges>
#include <string>
using std::array; using std::function;
using std::string; using std::views::zip;
int main() {
    // fun to introduce new container: array are like
    // vectors, but where you know at compile time the
    // size (which you add as integer inside <> template)
    array<string,3> v1{"add","sub","mul"};
    array<function<int(int,int)>,3> v2 = {
        [](int a, int b) { return a + b; },
        [](int a, int b) { return a - b; },
        [](int a, int b) { return a * b; },
    };
    array<double,2> v3{2, 3};

    for (int i = 0; i < v1.size(); ++i) { // C++98 style
        auto [x, y] = v3; // because array size is known during
                          // compilation, structure binding works
        fmt::println("{}({},{})={}", v1[i], x, y, v2[i](x,y));
    }
    fmt::println("");
    // const l-value reference (avoid copy)
    for (const auto& [name, op] : zip(v1,v2)) { // C++20 style
        const auto& [x, y] = v3; // no copies
        fmt::println("{}({},{})={}", name, x, y, op(x, y));
    }
}
```

High-level abstraction of
C++ containers

std::function is a type-erased wrapper. Can warp any callable entity

```
whovian@phy504:~/host/Lecture21$ nano lambda8.cpp
whovian@phy504:~/host/Lecture21$ g++ lambda8.cpp -o l8 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture21$ ./l8
add(2,3)=5
sub(2,3)=-1
mul(2,3)=6

add(2,3)=5
sub(2,3)=-1
mul(2,3)=6
```

More STL Algorithm and Ranges on STL vectors

Many of them we saw in examples in the
previous three classes

Algorithms that take args by l-value ref

```
#include <random>
#include <algorithm>
#include <fmt/ranges.h>
#include <chrono>
#include <ranges>
using std::views::take; using std::ranges::generate;
using std::uniform_real_distribution; using std::vector;
constexpr int sz = 5000;
int main () {
    std::random_device rd; // gen a seed from hardware
    std::mt19937 mt(rd()); // Mersenne Twister 32- bit gen
                           // uniformly distributed integers

    // C++ Class (will understand that in a few lectures)
    // that gen a real distribution from (0, sz) using
    // operator () - see below - GIVEN the Mersenne uniform
    // distributed integers, which needs as arg dist(mt).
    uniform_real_distribution<double> dist(0, sz);

    vector<double> v1(sz); // pre-allocated vector
    // std::ranges::generator need the lambda arg to be
    // l-value ref as it is going to populate the vector
    generate(v1, [&]() { return dist(mt); });

    std::ranges::sort(v1); // C++20 sort

    // print the first 10 elements (formatter to 2 digits)
    // preview of the next lecture on IO
    fmt::println("{:<.2f}", fmt::join(v1 | take(10), ", "));
}
```

```
whovian@phy504:~/host/Lecture21$ g++ algo1.cpp -o algo1 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture21$ ./algo1
0.86, 1.68, 3.61, 4.26, 4.31, 5.25, 7.57, 9.24, 9.46, 9.77
```

ranges::generate
needs to populate the
input std::vector

So, the lambda
function needs an
l-value reference
to the vector input
element

Algorithms that take args by l-value ref

```
#include <algorithm>
#include <ranges>
#include <fmt/ranges.h>
#include <vector>
using std::views::iota; using std::for_each;
using std::vector; using std::ranges::to;
int main () {
    vector v1 = iota(1, 10) | to<vector>();
    fmt::println("{} ", fmt::join(v1, ", "));

    // argument by l-value reference
    // capture by value (fun computes 3*n)
    // instead of returning n, it modifies it
    const int a = 2;
    auto pf = [b = a + 1](int& n) { n = b*n; };

    // Old C++98 style algorithm
    for_each(v1.begin(), v1.end(), pf);
    fmt::println("{} ", fmt::join(v1, ", "));

    // New C++20 style algorithm
    std::ranges::for_each(v1, pf);
    fmt::println("{} ", fmt::join(v1, ", "));
}
```

```
whovian@phy504:~/host/Lecture21$ nano algo2.cpp
whovian@phy504:~/host/Lecture21$ g++ algo2.cpp -o algo2 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture21$ ./algo2
1, 2, 3, 4, 5, 6, 7, 8, 9
3, 6, 9, 12, 15, 18, 21, 24, 27
9, 18, 27, 36, 45, 54, 63, 72, 81
```

ranges::for_each
needs to change the
input std::vector

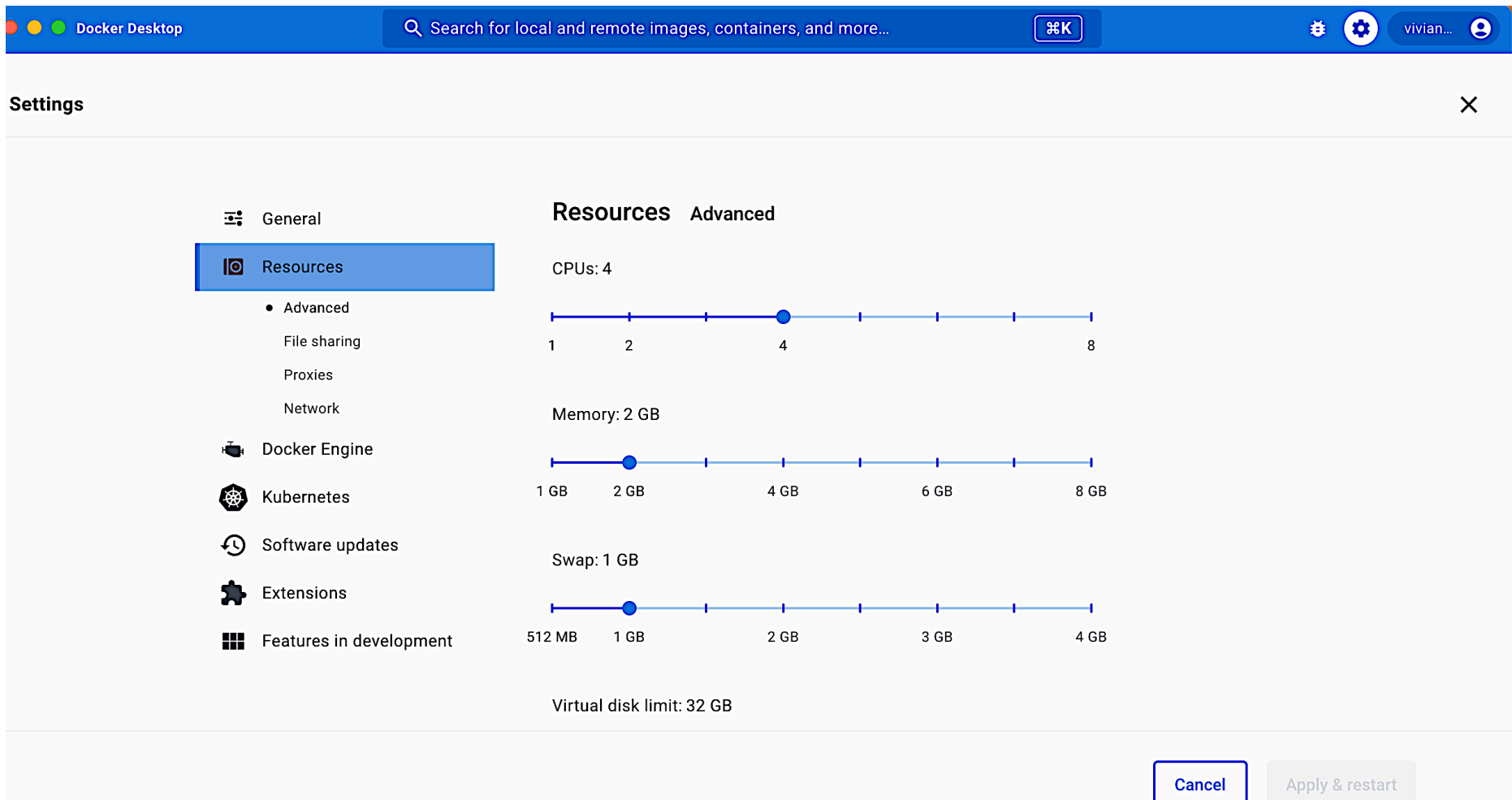
So, the lambda
function needs an
l-value reference
to the vector input
element

Parallel STL Algorithms

Parallel STL algorithms

Algorithms can run in parallel (C++17).

To see the speed up, you need to provide more cores to the docker container (here, I provide four cores)



C++ Algorithms

Algorithms can run
in parallel (C++17)

GCC uses intel
threading building
blocks (**-ltbb**) under
the hood for
execution

```
#include <random>
#include <algorithm>
#include <fmt/ranges.h>
#include <chrono>
#include <ranges>
#include <execution>
using std::views::take; using std::ranges::generate;
using std::uniform_real_distribution; using std::vector;
constexpr int sz = 10'000'000; // big number
int main () {
    std::random_device rd;
    std::mt19937 mt(rd());
    uniform_real_distribution<double> dist(0, sz);
    vector<double> v1(sz);
    generate(v1, [&]() { return dist(mt); });
    vector<double> v2(sz);
    generate(v2, [&]() { return dist(mt); });

    // serial version
    auto t0 = std::chrono::high_resolution_clock::now();
    std::ranges::sort(v1); // C++20 sort
    auto t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fs = t1 - t0;
    fmt::println("Time to apply sort (1-core) = {0}", fs.count());

    // parallel version (has the old C++98 API)
    // it seems future C++ standards may solve this oversight
    auto t2 = std::chrono::high_resolution_clock::now();
    std::sort(std::execution::par, v2.begin(), v2.end());
    auto t3 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fp = t3 - t2;
    fmt::println("Time to apply sort (parallel) = {0}\n", fp.count());
}
```

```
whovian@phy504:~/host/Lecture21$ nano algo3.cpp
whovian@phy504:~/host/Lecture21$ g++ algo3.cpp -o algo3 -std=c++26 -lfmt -ltbb
whovian@phy504:~/host/Lecture21$ ./algo3
Time to apply sort (1-core) = 15.005798785
Time to apply sort (parallel) = 0.894429917
```

C++ Algorithms

Algorithms can run in
parallel (C++17)

GCC uses intel
threading building
blocks (**-ltbb**) under
the hood for
execution

```
#include <random>
#include <algorithm>
#include <fmt/ranges.h>
#include <chrono>
#include <ranges>
#include <execution>
using std::views::take; using std::ranges::generate;
using std::uniform_real_distribution; using std::vector;
constexpr int sz = 30'000'000; // big number
int main () {
    std::random_device rd;
    std::mt19937 mt(rd());
    uniform_real_distribution<double> dist(0, sz);
    vector<double> v1(sz);
    generate(v1, [&]() { return dist(mt); });
    vector<double> v2(sz);
    generate(v2, [&]() { return dist(mt); });

    auto pf = [](double& n) { n++; }; // l-value ref

    // serial version
    auto t0 = std::chrono::high_resolution_clock::now();
    std::ranges::for_each(v1, pf);
    auto t1 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fs = t1 - t0;
    fmt::println("Time to apply for_each (1-core) = {0}",
                fs.count());

    // parallel version (has the old C++98 API)
    // it seems future C++ standards may solve this oversight
    auto t2 = std::chrono::high_resolution_clock::now();
    std::for_each(std::execution::par, v1.begin(), v1.end(), pf);
    auto t3 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> fp = t3 - t2;
    fmt::println("Time to apply for_each (parallel) = {0}\n",
                fp.count());
}
```

```
whovian@phy504:~/host/Lecture21$ nano algo4.cpp
whovian@phy504:~/host/Lecture21$ g++ algo4.cpp -o algo4 -std=c++26 -lfmt -ltbb
whovian@phy504:~/host/Lecture21$ ./algo4
Time to apply for_each (1-core) = 0.861803125
Time to apply for_each (parallel) = 0.036629792
```

Homework I

- Given a `std::map<string,int>`, create a complete example where you, in a single line, create a second map where all the values in the second map are the squares of the values in the first map.

Tip: use `std::views::transform`, `std::ranges::to` and a lambda function that squares each element.

Homework II

- Given a `std::vector<int>` with 6 elements, create a complete example where you use `std::vector<int>::iterator .begin()` and `.end()` + the old **C++98** `std::transform` style to
 - (1) double the value of every vector element.
(Tip: you will need to a second vector to hold the results)
 - (2) double the first three elements
 - (3) double the last three elements
- Redo (1) using `std::views::take` and the new range-based `std::ranges::transform` to double
 - the first three (you will need `std::views::take`)
 - the last three (you will need `std::views::drop`)

Homework III

- Using pipping and `std::ranges::to` copy all elements of a `std::vector<int>` to an `std::list<int>` (create a complete example).
- Redo (1), but in the end, use `fmt::join` to print all elements of the list separated by a comma. Use `fmt::print`
- Redo (1), but use Python style loop “`for x in z`” (range-based `C++11` loops) to print all the elements of the list (separate them by `\n`). Use `C++98 std::cout`
- Redo (1), but use old C++98 `std::list<int>::iterators` and C++98 streams (`std::cout`) to print all the list elements

Homework IV

Given two STL vectors **v1** and **v2** of the same size.

- (1) Create vectors **v1** and **v2** with a size of 30. Fill them with random numbers from a uniform distribution between 1 and 10 using **std::ranges::generate**.
- (2) Use **std::views::transform** to create a new view v3 that contains $v1[i] * v2[i]$ for each element. Materialize the result into a **vector<int>** using **std::ranges::to<vector>()**.
- (3) Use **std::views::zip** combined with **std::views::transform** to compute **v1[i] + v2[i]**. Print the first 20 elements using **std::views::take(20)** and **fmt::join**.
- (4) Given **v1**, use **std::views::filter** to create a view containing only the even elements. Then use **std::views::transform** to square them. Materialize the result and print its size.
- (5) Use **std::views::drop(N/2)** and **std::views::take(10)** together with **fmt::println** to print the 10 elements starting from the middle of **v1**.

Homework V

Running averages with `views::adjacent`. Create a `vector<double>` of 90 noisy samples (`Uniform, min=0, max=1`). Use `std::views::adjacent<5>` combined with `std::views::transform` to compute a 3-point moving average as a lazy view.

- (1) Materialize the average of the entire vector
- (2) Materialize only the last 10 averages using `views::drop` + `views::take`.

Homework VI

- Estimate π by counting random (x,y) points in $[0,1]^2$ that fall inside the unit circle. The ratio (points inside) / (total points) approaches $\pi/4$ as N grows, so $\pi \sim 4 * \text{inside} / N$.

Hints:

- Use `std::views::iota(0, N)` to generate N indices.
- Pipe through `std::views::transform` to produce (x,y) pairs using a `uniform_real_distribution<double>(0,1)`.
- Pipe through `std::views::filter` to keep only points where $x*x + y*y \leq 1.0$.
- Count the surviving points with `std::ranges::distance` (this is what forces the lazy pipeline to actually evaluate).
- Run for $N = 10^4, 10^6, 10^8$ and observe convergence. Expected statistical error scales as $1/\sqrt{N}$.

Homework VII

- Create a map of **<string, lambda functions>** whose object is named mymap so that **mymap['add'](x,y)** add two numbers, **mymap['mult'](x,y)** (create a complete example to show that).

Homework VIII

- Write a function `compose` that takes two lambdas `f` and `g` and returns a new lambda that computes $f(g(x))$. Use it to compose `square` and `add3` into $(x+3)^2$ and x^2+3 . Then apply it to a vector using `std::views::transform`.

Homework IX *(not going to be on the exam)*

(1) Using C-style loops, write a C++ program that prints the first 100 Pythagorean triples.

The [range-v3](http://ericniebler.com) library goes beyond what “ranges” in the C++ standard can do. The author, Eric Niebler (who works at Nvidia), has an interesting C++ blog (<http://ericniebler.com>). Here is his solution to the same problem. Try to run this program.

```
// Define an infinite range containing all the Pythagorean triples:
auto triples = ranges::views::for_each(ranges::views::iota(1), [](int z) {
    return ranges::views::for_each(ranges::views::iota(1, z + 1), [=](int x) {
        return ranges::views::for_each(ranges::views::iota(x, z + 1), [=](int y) {
            return ranges::yield_if(x*x+y*y == z*z, make_tuple(x,y,z));
        });
    });
});
// Display the first 100 triples
RANGES_FOR(auto triple, triples | ranges::views::take(100))
{
    auto [x, y, z] = triple;
    fmt::println("{} {} {}", x, y, z);
}
```

```
whovian@phy504:~/host/Lecture21$ g++ HW21_SOLUTIONS.cpp -o hw21 -std=c++26 -I /vcpkg/installed/x64-linux/include/ -lfmt -Wno-deprecated-literal-operator
whovian@phy504:~/host/Lecture21$ ./hw21
```

PS: Including -O2 or -O3 optimization compiler flags made a big diff in runtime

Additional reading

At least the way I think about it is:

- `std::mt19937` is a very simple and reliable random generator. It is self-contained and will live entirely in your process, not calling out to the OS or anything else. The implementation is *mandated* by the standard, and at least in boost, it used the same code everywhere, derived from the original `mt19937` paper. This code is very stable and it's cross-platform. You can be pretty confident that initializing it, querying from it, etc. is going to compile to similar code on any platform that you compile it on, and that you will get similar performance.
- `std::random_device` by contrast is pretty opaque. You don't really know exactly what it is, what it's going to do, or how efficient it will be. You don't even know if it can actually be acquired -- it might throw an exception when you attempt to create it. It might make a system call whenever you read from it, so it may have much worse performance in terms of cycles-per-byte than `std::mt19937`. You know that it doesn't require a seed. You're not usually supposed to pull tons and tons of data from it, just use it to generate seeds. Sometimes, it acts as a nice interface to cryptographic APIs, but it's not actually required to do that and sadly sometimes it doesn't. It might correspond to `/dev/random` on unix, it might correspond to `/dev/urandom/`. It might correspond to some MSVC crypto API (visual studio), or it might just be a fixed constant (mingw). If you cross-compile for some phone, who knows what it will do. (And even when you do get `/dev/random`, you still have the problem that performance may not be *consistent* -- it may appear to work great, until the entropy pool runs out, and then it runs slow as a dog.)

The way I think about it is, `std::random_device` is supposed to be like an improved version of seeding with `time(NULL)` -- that's a low bar, because `time(NULL)` is a pretty crappy seed all things considered. I usually use it where I would have used `time(NULL)` to generate a seed, back in the day. I don't really consider it all that useful outside of that.

On C++ random number generation

`std::mt19937`

VS

`std::random_device`

Share Edit Follow Flag

edited Mar 25, 2023 at 18:43

answered Sep 2, 2016 at 23:12



Chris Beck

16.4k ● 4 ● 60 ● 92