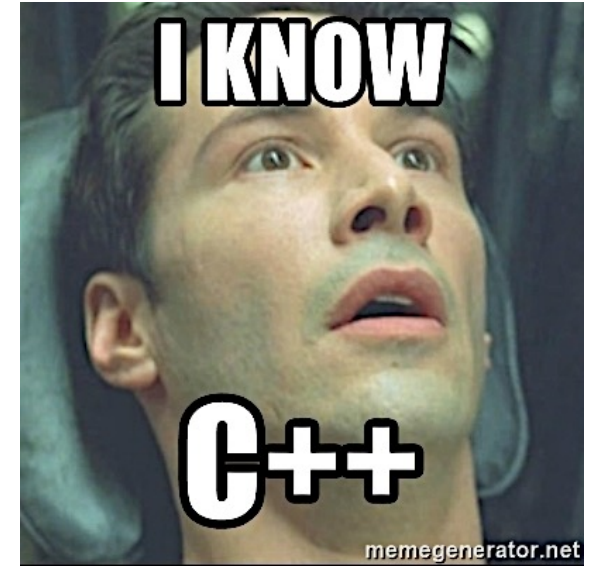
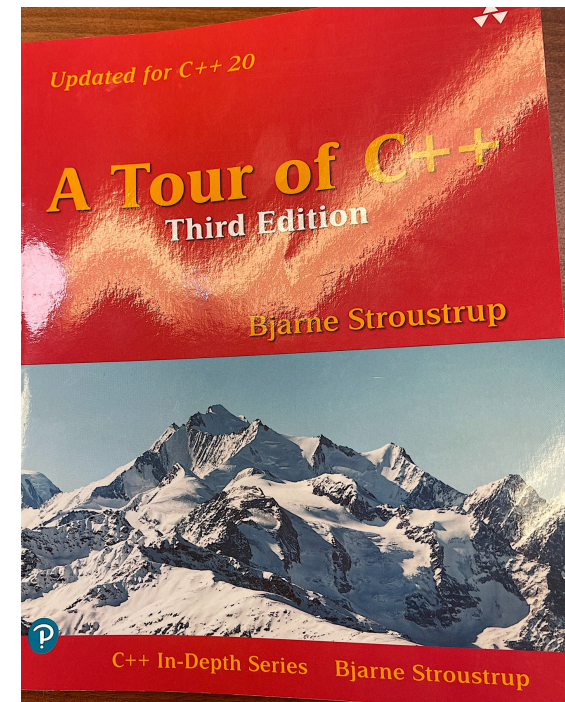


Lecture 18: C++ part III



Suggested Literature



RECAP: C++ Maps

```
int main ()
{
    map<string, int> mymap = {{"Vivian", 37},
                             {"Rihanna", 35}};

    // C++ allows for map elements to be accessed via
    // Python style of loops in two different ways

    // First way: x is a {key, value} pair
    for (auto x : mymap)
    {
        // x is not a pointer. We access elements
        // first and second via "." and not "->"
        fmt::print("Key: {0}, Val: {1}\n", x.first, x.second);
    }

    fmt::print("\n");

    for (auto [key, val] : mymap)
    { // Second Way (C++17)
        fmt::print("Key: {0}, Val: {1}\n", key, val);
    }
}
```

C++ is
compatible with
Python style
loops

C++ provides
many solutions
to each problem!

```
whovian@phy504:~/host$ g++ maps4.cpp -o maps4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps4
Key: Rihanna, Val: 35
Key: Vivian, Val: 37

Key: Rihanna, Val: 35
Key: Vivian, Val: 37
```

RECAP: C++ Maps

```
int main ()
{
    map<string, int> mymap = {"Vivian", 37},
                             {"Rihanna", 35}};

    // Lambdas are anonymous functions in C++
    auto pf = [](auto x)
    {
        fmt::print("Key: {0}, Val: {1}\n", x.first, x.second);
    };
    // STL provides many useful algorithms
    for_each(mymap.begin(), mymap.end(), pf);

    fmt::print("\n");

    // you can also write a lambda func in-place (Python style)
    for_each(mymap.begin(), mymap.end(), [](auto x){
        fmt::print("Key: {0}, Val: {1}\n", x.first, x.second);
    });

    return 0;
}
```

whovian@phy504:~/host\$ g++ maps5.cpp -o maps5 -std=c++20 -lfmt
whovian@phy504:~/host\$./maps5
Key: Rihanna, Val: 35
Key: Vivian, Val: 37

```
Key: Rihanna, Val: 35  
Key: Vivian, Val: 37
```

Preview of
STL
algorithms and
Lambda
functions

Lambdas are
insanely
powerful

C++ Maps

What if we want to
one (just one)
value of different
types

std::any (C++17)

```
#include <map>
#include <string>
#include <any>
#include <fmt/core.h>
```

```
using std::map;
using std::string;
using std::any;
```

```
int main ()
{
```

```
    map<string, any> mymap = {{"Vivian", string("Miranda")},
                              {"Rihanna", 35},
                              {"Beyonce", 5000000000.0}};
```

```
    string x = std::any_cast<string>(mymap.at("Vivian"));
    fmt::print("Key: {0}, Val: {1} (surname)\n", "Vivian", x);
```

```
    int y = std::any_cast<int>(mymap.at("Rihanna"));
    fmt::print("Key: {0}, Val: {1} (age)\n", "Rihanna", y);
```

```
    double z = std::any_cast<double>(mymap.at("Beyonce"));
    fmt::print("Key: {0}, Val: {1} (fortune)\n", "Beyonce", z);
```

```
    return 0;
```

```
}
```

String literals are **const char***, that cannot be cast to C++ strings. We need to create a C++ string from **const char*** (using **constructor**)

```
whovian@phy504:~/host$ g++ maps8.cpp -o maps8 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps8
Key: Vivian, Val: Miranda (surname)
Key: Rihanna, Val: 35 (age)
Key: Beyonce, Val: 5000000000 (fortune)
```

C++ Maps

```
#include <map>
#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
```

```
using std::map;
using std::string;
using boost::to_upper_copy;
```

```
int main ()
{
    // When I use maps I prefer to have keys all
    // uppercase. The Boost Library (informal
    // extension of STL) save us with two functions:
    // boost::to_upper_copy<std::string>
    // boost::to_upper

    map<string, int> mymap =
        {{to_upper_copy<string>("Vivian"), 37},
         {to_upper_copy<string>("Marcelo"), 38},
         {to_upper_copy<string>("Vania"), 58},
         {to_upper_copy<string>("Isabella"), 25},
         {to_upper_copy<string>("Rihanna"), 35}};

    fmt::print("{0} is {1} years old\n",
               to_upper_copy<string>("Rihanna"),
               mymap.at(to_upper_copy<string>("Rihanna")));

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps9.cpp -o maps9 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps9
RIHANNA is 35 years old
```

I prefer to have
uppercase keys (avoid
confusion)

STL has only a
function to uppercase
a single char

Boost Library comes
to help us (super
useful library)

C++ Maps

maps::emplace() inserts the key (and increase map size) if it is not set

“if it is not set” creates an optimization problem: if it is set, do you create the temporary object? Yes (bad)

C++17 solves that problem with

map::try_emplace

```
#include <iostream>
#include <map>
#include <string>
#include <vector>
#include <chrono>
using namespace std;
using namespace chrono;

int main() {
    const int N = 500'000, KEYS = 100;

    // emplace: ALWAYS builds the vector
    map<int, vector<string>> m1;
    auto t1 = high_resolution_clock::now();
    for (int i = 0; i < N; ++i)
        m1.emplace(i % KEYS, vector<string>(100, "super expensive"));
    auto d1 =
        duration_cast<milliseconds>(high_resolution_clock::now() - t1);

    // try_emplace: only builds the vector when key is NEW
    map<int, vector<string>> m2;
    auto t2 = high_resolution_clock::now();
    for (int i = 0; i < N; ++i)
        m2.try_emplace(i % KEYS, 100, "super expensive");
    auto d2 =
        duration_cast<milliseconds>(high_resolution_clock::now() - t2);

    cout << "emplace:      " << d1.count() << " ms\n";
    cout << "try_emplace: " << d2.count() << " ms\n";
}
```

```
whovian@whovianphy504:~/host/Lecture17$ g++ maps12.cpp -o m12 --std=c++23
whovian@whovianphy504:~/host/Lecture17$ ./m12
emplace:      5082 ms
try_emplace: 129 ms
```

C++ obsession: temporary object creation

C++ introduces complicated types, and creating temporary objects of those types may be expensive

This can be a problem in any language

Every language pays for unnecessary copies

C avoids it with pointers: pass the address, not the data (via pointers)

C++ goes further: don't even create the object if you don't need it

This is why C++ has **map::emplace**, **map::try_emplace**, and other advanced features we will learn

C++ obsession: temporary object creation

`emplace()`



`vector<string>(100)`
CONSTRUCTED!

"key already exists"

DISCARDED

**DOES THIS
500,000 TIMES**

`try_emplace()`



"key already exists?"

SKIP. DONE.

**no construction
no allocation
no destruction**

368 ms → 3 ms

C++: "DON'T BUILD IT IF YOU DON'T NEED IT"

C++23 introduced flat maps

API is similar to map
It is the performance
and trade-offs under
the hood that changes

```
#include <flat_map>
#include <print>
#include <string>

using std::flat_map;
using std::string;

int main() {
    // flat_map<key type, value type> = initializer list
    // Like std::map, the keys stay sorted.
    flat_map<string, double> mymap = {
        {"Vivian", 37},
        {"Marcelo", 38},
        {"Vania", 58},
        {"Isabella", 25}
    };

    // Member functions that modify/access elements
    mymap.insert({"Gilberto", 56});

    mymap["Beyonce"] = 41;

    // operator[] can be risky:
    // 1) if the key does not exist, it inserts it with a default value
    // 2) if the key does exist, it gives mutable access to the value
    std::print("(1) Rihanna is {} years old\n", mymap["Rihanna"]);

    mymap.insert({"Rihanna", 35}); // nothing happens: key already exists
    std::print("(2) Rihanna is {} years old\n", mymap["Rihanna"]);

    mymap["Rihanna"] = 35;
    std::print("(3) Rihanna is {} years old\n", mymap["Rihanna"]);

    std::print("\nAll entries:\n");
    for (const auto& [name, age] : mymap) {
        std::print("  {} -> {}\n", name, age);
    }

    return 0;
}
```

```
whovian@whovianphy504:~/host/Lecture17$ g++ flat_maps.cpp -o fm -std=c++23
whovian@whovianphy504:~/host/Lecture17$ ./fm
```

```
(1) Rihanna is 0 years old
(2) Rihanna is 0 years old
(3) Rihanna is 35 years old
```

```
All entries:
Beyonce -> 41
Gilberto -> 56
Isabella -> 25
Marcelo -> 38
Rihanna -> 35
Vania -> 58
Vivian -> 37
```

C++23 introduced flat maps

Comparison Summary

Feature 	<code>std::map</code>	<code>std::flat_map</code> (C++23)
Implementation	Balanced Binary Tree (Red-Black Tree)	Sorted Sequence Containers (e.g., <code>std::vector</code>)
Memory Layout	Node-based (Fragmented)	Contiguous (Cache-friendly)
Lookup	$O(\log n)$ (Pointer chasing)	$O(\log n)$ (Binary search)
Insertion/Erasure	$O(\log n)$	$O(n)$ (Requires shifting elements)
Iteration	Slower (Pointer traversing)	Faster (Linear memory access)
Iterator Stability	Highly Stable	Invalidated on insertion/deletion

C++23 introduced range operators

C++23 introduce range operators.

E.g., **std::from_range**

allows you to create/extend a map from the elements of a contiguous container

```
#include <map>
#include <print>
#include <ranges>
#include <string>
#include <utility>
#include <vector>

using std::pair; using std::vector;
using std::map; using std::string;
using std::from_range;

int main() {
    vector<pair<int, string>> init{
        {2, "two"},
        {1, "one"}
    };

    // passed to the map constructors to indicate
    // the contained member is range constructed.
    map<int, string> m(from_range, init);

    vector<pair<int, string>> more{
        {4, "four"},
        {3, "three"}
    };

    m.insert_range(more);

    for (auto [k, v] : m) {
        std::print("{} -> {}\n", k, v);
    }
}
```

```
whovian@whovianphy504:~/host/Lecture17$ g++ maps10.cpp -o m10 --std=c++23
```

```
whovian@whovianphy504:~/host/Lecture17$ ./m10
```

```
1 -> one
2 -> two
3 -> three
4 -> four
```

```

#include <map>

// C++26 made maps constexpr
// friendly
|
constexpr int lookup_sum() {
    std::map<int, int> m;
    m.emplace(1, 10);
    m.emplace(2, 20);
    m.emplace(3, 30);
    return m.at(2) + m.at(3);
}

static_assert(lookup_sum() == 50);

int main() {}

```

maps::emplace() inserts the key (and increase map size) if it is not set

C++11 introduced the notion of **constexpr**

C++26 made **STL maps** **constexpr**

constexpr = if the information is available at compilation time. The operation can be performed during compilation

List

C++ Lists

C++ has an implementation of **doubled linked lists** in the STL library. Notice that containers in C++ have “member functions” that are called with “.” (dot)

```
#include <algorithm>
#include <fmt/core.h>
#include <list>

int main ()
{
    std::list<int> lt = {7, 5, 16, 8, 10};
    for (auto x : lt) // single line loop {} not needed
        fmt::print("{0}, ", x);
    fmt::print("\n");
```

```
lt.push_back(13);    // add to the back of the list
lt.push_front(25);   // add to the front of the list
```

```
for (auto x : lt)
    fmt::print("{0}, ", x);
fmt::print("\n");
```

```
whovian@phy504:~/host$ g++ list.cpp -o list -std=c++20 -lfmt
whovian@phy504:~/host$ ./list
7, 5, 16, 8, 10,
25, 7, 5, 16, 8, 10, 13,
```

C++ Lists

STL also provides algorithms to manipulate lists

```
// find elements in the list with the STL algorithm
// remember that memory in linked lists is not contiguous
auto it = std::find(lt.begin(), lt.end(), 16);
if (it != lt.end())
    lt.insert(it, 42); // .insert() inserts value before pos
for (auto x : lt)
    fmt::print("{0}, ", x);
fmt::print("\n");

lt.push_front(13);
lt.sort();
for (auto x : lt)
    fmt::print("{0}, ", x);
fmt::print("\n");

lt.unique(); // remove adjacent duplicates
for (auto x : lt)
    fmt::print("{0}, ", x);
fmt::print("\n");

return 0;
}
```

```
whovian@phy504:~/host$ g++ list.cpp -o list -std=c++20 -lfmt
whovian@phy504:~/host$ ./list
7, 5, 16, 8, 10,
25, 7, 5, 16, 8, 10, 13,
25, 7, 5, 42, 16, 8, 10, 13,
5, 7, 8, 10, 13, 13, 16, 25, 42,
5, 7, 8, 10, 13, 16, 25, 42,
```

C++ Lists

You can quickly
reverse linked
lists with the
member
function

list::reverse()

```
#include <algorithm>
#include <fmt/core.h>
#include <list>

int main ()
{
    std::list<int> lt = {25, 7, 5, 16, 8, 10, 13};
    for (auto x : lt) // single line loop {} not needed
        fmt::print("{0}, ", x);
    fmt::print("\n");

    lt.pop_back();    // remove the last elem of the list
    lt.pop_front();   // remove the first elem of the list
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    // easy access to the first and last elem of the list
    fmt::print("{0}, {1}\n", lt.front(), lt.back());

    lt.reverse();
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    return 0;
}
```

```
whovian@phy504:~/host$ g++ list2.cpp -o list2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list2
25, 7, 5, 16, 8, 10, 13,
7, 5, 16, 8, 10,
7, 10
10, 8, 16, 5, 7,
```

C++ Lists

Iterators superpowers (the ++ being a valid operator), and the **list::end()** member function allow us to go over every element of a linked list with for loops!

```
#include <algorithm>
#include <fmt/core.h>
#include <list>

int main ()
{
    std::list<int> lt = {25, 7, 5, 16, 8, 10, 13};

    for(auto it = lt.begin(); it != lt.end(); it++) {
        // notice that the loop ends with it not equal .end()
        // NOT it less than .end()
        fmt::print("{0}, ", *it);
    }
    fmt::print("\n");
    return 0;
}
```



```
whovian@phy504:~/host/Lecture18$ g++ list8.cpp -o list8 -std=c++20 -lfmt
whovian@phy504:~/host/Lecture18$ ./list8
25, 7, 5, 16, 8, 10, 13,
```

C++ Lists

Some view **for loops** as not being the “C++ way”

What would be the C++ way? STL algorithms

```
#include <algorithm>
#include <fmt/core.h>
#include <list>

using std::list; using std::for_each;

int main () {
    list<int> lt = {25, 7, 5, 16, 8, 10, 13};

    for_each(lt.begin(), lt.end(),
        [](auto n) {fmt::print("{0}, ", n);}); // lambda
                                                // function
    fmt::print("\n");
    return 0;
}
```

```
whovian@phy504:~/host/Lecture18$ g++ list9.cpp -o list9 -std=c++20 -lfmt
whovian@phy504:~/host/Lecture18$ ./list9
25, 7, 5, 16, 8, 10, 13,
```

C++ Lists

Some view **for loops** as not being the “C++ way”

C++20: algorithms even more powerful w/ ranges

```
#include <algorithm>
#include <fmt/core.h>
#include <list>
#include <ranges>

using std::list; using std::ranges::for_each;
using std::ranges::views::filter; using std::ranges::views::transform;

int main () {
    list<int> lt = {25, 7, 5, 16, 8, 10, 13};

    // resembles the pipe operation in bash
    for_each(lt |
        filter([](auto n){ return n % 2 == 1; }) | // loop
        transform([](auto n){ return n * n; }),    // check if is odd
        [](auto n) {fmt::print("{0}, ", n); });    // square it
                                                    // print

    fmt::print("\n");
    return 0;
}
```

```
whovian@phy504:~/host/Lecture18$ g++ list10.cpp -o list10 -std=c++20 -lfmt
whovian@phy504:~/host/Lecture18$ ./list10
625, 49, 25, 169,
```

C++ Lists

```
#include <algorithm>
#include <fmt/core.h>
#include <list>

int main () {
    std::list<int> lt = {25, 7, 5, 16, 8, 10, 13};

    auto t1 = lt.begin();
    std::advance(t1, 2); // now t1 points to 5

    t1 = lt.erase(t1); // member function takes as an
                        // argument the iterator at the
                        // position you want to delete
    // changes in the container size render iter t1
    // invalid. This is why .erase() returns a new iter
    std::for_each(lt.begin(), lt.end(),
        [](auto n) {fmt::print("{0}, ", n);});
    fmt::print("\n");
    return 0;
}
```

whovian@phy504:~/host/Lecture18\$ g++ list11.cpp -o list11 -std=c++20 -lfmt
whovian@phy504:~/host/Lecture18\$./list11
25, 7, 16, 8, 10, 13,

How do you
erase a list
element?

member
function

list::erase()

C++ Lists

How do you append a linked list to another w/o knowing their sizes? **inserters** + STL algorithms

```
#include <fmt/core.h>
#include <list>
#include <algorithm>
using std::list; using fmt::print;
using std::unique_copy; using std::back_inserter;
using std::front_inserter; using std::inserter;

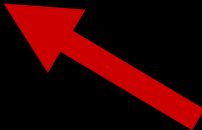
int main ()
{
    list<int> lt = {25, 7, 5, 25, 16, 10, 8, 10, 7, 6};
    lt.sort();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    // suppose we want to copy the list but without duplicates
    list<int> lt2 = {100, 101, 102};
    unique_copy(lt.begin(), lt.end(), back_inserter(lt2));
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");
}
```

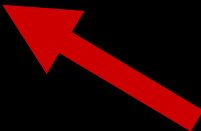
whovian@phy504:~/host\$ g++ list3.cpp -o list3 -std=c++20 -lfmt
whovian@phy504:~/host\$./list3
5, 6, 7, 7, 8, 10, 10, 16, 25, 25,
100, 101, 102, 5, 6, 7, 8, 10, 16, 25,



```
list<int> lt3 = {100, 101, 102};
unique_copy(lt.begin(), lt.end(), front_inserter(lt3));
for (auto x : lt3)
    print("{0}, ", x);
print("\n\n");
```



```
list<int> lt4 = {100, 101, 102};
unique_copy(lt.begin(), lt.end(), inserter(lt4, lt4.begin()));
for (auto x : lt4)
    print("{0}, ", x);
print("\n\n");
```



```
list<int> lt5 = {100, 101, 102, 103, 104};
auto x = std::find(lt5.begin(), lt5.end(), 102);
unique_copy(lt.begin(), lt.end(), inserter(lt5, x));
for (auto x : lt5)
    print("{0}, ", x);
print("\n\n");
```

inserter
+ STL
algorithms

```
whovian@phy504:~/host$ g++ list3.cpp -o list3 -std=c++20 -lfmt
```

```
whovian@phy504:~/host$ ./list3
```

```
5, 6, 7, 7, 8, 10, 10, 16, 25, 25,
```

```
100, 101, 102, 5, 6, 7, 8, 10, 16, 25,
```

```
25, 16, 10, 8, 7, 6, 5, 100, 101, 102,
```

```
5, 6, 7, 8, 10, 16, 25, 100, 101, 102,
```

```
100, 101, 5, 6, 7, 8, 10, 16, 25, 102, 103, 104,
```

C++ Lists

How to copy a linked list
w/o knowing its size?

STL algorithm offers
tools to manipulate list
inserters saved us from
memory corruption (lt2,
lt3, lt4, lt5 not large
enough)

```
#include <fmt/core.h>
#include <list>
#include <algorithm>
using std::list; using fmt::print;
using std::unique_copy; using std::back_inserter;
using std::front_inserter; using std::inserter;

int main ()
{
    list<int> lt = {25, 7, 5, 25, 16, 10, 8, 10, 7, 6};
    lt.sort();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    // suppose we want to copy the list but without duplicates.
    list<int> lt2 = {100, 101, 102};
    unique_copy(lt.begin(), lt.end(), back_inserter(lt2));
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt3 = {100, 101, 102};
    unique_copy(lt.begin(), lt.end(), front_inserter(lt3));
    for (auto x : lt3)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt4 = {100, 101, 102};
    unique_copy(lt.begin(), lt.end(), inserter(lt4, lt4.begin()));
    for (auto x : lt4)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt5 = {100, 101, 102, 103, 104};
    auto x = std::find(lt5.begin(), lt5.end(), 102);
    unique_copy(lt.begin(), lt.end(), inserter(lt5, x));
    for (auto x : lt5)
        print("{0}, ", x);
    print("\n\n");
}
```

inserter(list, pos)
inserts before pos

```
whovian@phy504:~/host$ g++ list3.cpp -o list3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list3
5, 6, 7, 7, 8, 10, 10, 16, 25, 25,
100, 101, 102, 5, 6, 7, 8, 10, 16, 25,
25, 16, 10, 8, 7, 6, 5, 100, 101, 102,
5, 6, 7, 8, 10, 16, 25, 100, 101, 102,
100, 101, 5, 6, 7, 8, 10, 16, 25, 102, 103, 104,
```

C++ Lists

```
#include <fmt/core.h>
#include <list>
#include <algorithm>
```

```
using std::list;
using fmt::print;
using std::reverse_copy;
using std::back_inserter;
using std::inserter;
```

```
int main ()
```

```
{
    list<int> lt = {25, 7, 5, 25, 16, 10, 8, 10, 7, 6};
    lt.sort();
    lt.unique();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt2 = {};
    reverse_copy(lt.begin(), lt.end(), back_inserter(lt2));
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    list<int> lt3 = {100, 101, 102, 103, 104};
    auto x = std::find(lt3.begin(), lt3.end(), 102);
    reverse_copy(lt.begin(), lt.end(), inserter(lt3, x));
    for (auto x : lt3)
        print("{0}, ", x);
    print("\n\n");
}
```

```
whovian@phy504:~/host$ g++ list5.cpp -o list5 -std=c++20 -lfmt
```

```
whovian@phy504:~/host$ ./list5
```

```
5, 6, 7, 8, 10, 16, 25,
```

```
25, 16, 10, 8, 7, 6, 5,
```

```
100, 101, 25, 16, 10, 8, 7, 6, 5, 102, 103, 104,
```

**Algorithms don't have
permission to delete
elements**

STL algorithm offers
powerful abstract
tools to manipulate
doubled linked lists

inserters saved us
from memory
corruption

C++ Lists

STL algorithm
offers powerful
abstract tools to
manipulate **doubled
linked lists**

inserters saved us
from corruption
(lt2, lt3, lt4, lt5 not
large enough)

```
#include <fmt/core.h>
#include <list>
#include <algorithm>

using std::list;
using fmt::print;
using std::unique_copy;

int main ()
{
    list<int> lt  = {25, 07, 05, 25, 16, 10, 8, 10, 07, 06};
    list<int> lt2 = {01, 01, 01, 01, 01, 01, 1, 01, 01, 01};

    lt.sort();
    for (auto x : lt)
        print("{0}, ", x);
    print("\n\n");

    // no need for inserters (lt2 has enough capacity)
    unique_copy(lt.begin(), lt.end(), lt2.begin());
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    return 0;
}
```

```
whovian@phy504:~/host$ g++ list4.cpp -o list4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list4
5, 6, 7, 7, 8, 10, 10, 16, 25, 25,
5, 6, 7, 8, 10, 16, 25, 1, 1, 1,
```

**Algorithms don't have
permission to delete
elements**