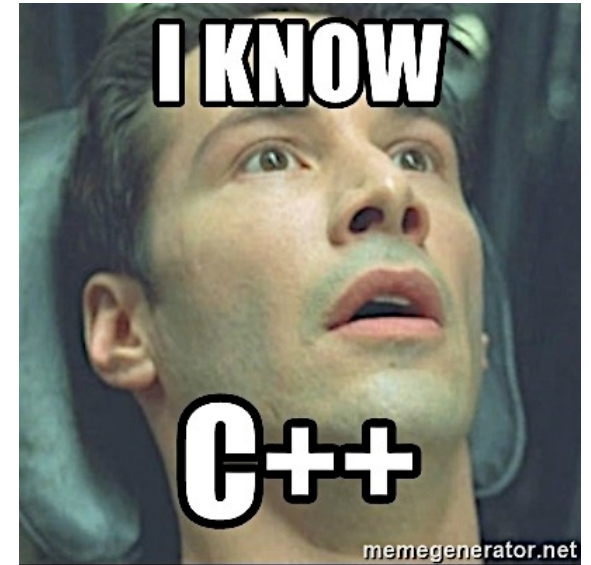
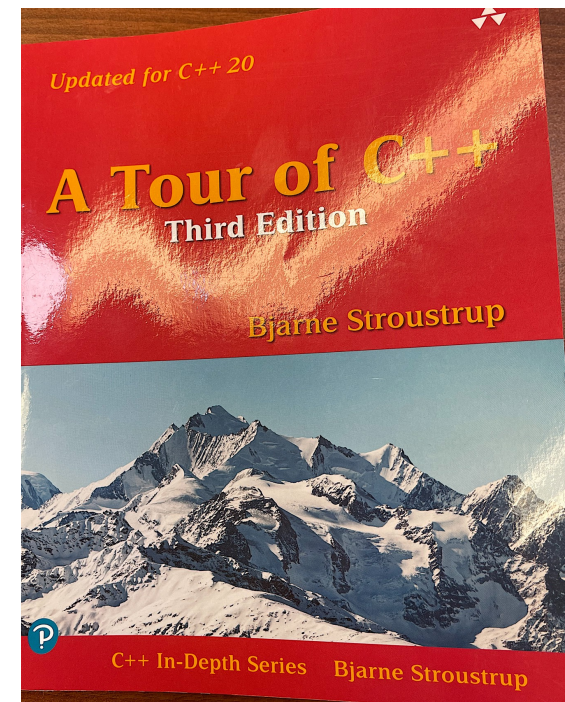


Lecture 19: C++ part IV



Suggested Literature



More Lists

C++ Lists

Algorithms don't
have permission to
delete elements

But they do return the
position where they
stop

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

    auto ptr = unique_copy(lt.begin(), lt.end(), lt2.begin());
    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");

    // now we will show that ptr iterator points
    // to the last element of the unique_copy
    for (auto x = lt2.begin(); x != ptr; x++) {
        print("{0}, ", *x); // x is an iterator (restricted version
                            // of a pointer)
    }
    print("\n\n");

    // How to permanently erase the last elements?
    lt2.erase(ptr, lt2.end());

    for (auto x : lt2)
        print("{0}, ", x);
    print("\n\n");
}
```

```
whovian@phy504:~/host$ g++ list6.cpp -o list6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./list6
5, 6, 7, 8, 10, 16, 25,
5, 6, 7, 8, 10, 16, 25, 1, 1, 1,
5, 6, 7, 8, 10, 16, 25,
5, 6, 7, 8, 10, 16, 25,
```

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

```
int main ()
```

```
{
```

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

```
    auto it1 = lt.begin();
```

```
    while (it1 != lt.end()) {
```

```
        fmt::print("{0}, ", *it1);
```

```
        it1++; // Key difference to C. Even considering that
                // memory in linked lists is NOT contiguous (elements
                // are not close in memory, the command
                // it1++ is ok. The iterator will go to the
                // correct next element of the linked list
```

```
    }
```

```
    fmt::print("\n");
```

```
    auto it2 = lt.begin();
```

```
    std::advance (it2,3);
```

```
    while (it2 != lt.end()) {
```

```
        fmt::print("{0}, ", *it2);
```

```
        it2++;
```

```
    }
```

```
    fmt::print("\n");
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host/Lecture18$ g++ list7.cpp -o list7 -std=c++20 -lfmt
```

```
whovian@phy504:~/host/Lecture18$ ./list7
```

```
25, 7, 5, 16, 8, 10, 13,
```

```
16, 8, 10, 13,
```

C++ Lists

Can I advance
to position N
without having
to loop over
iterators
explicitly?

Yes, with
std::advance

C++ (23) Lists

```
#include <list>
#include <print>
#include <ranges>

int main() {
    std::list<int> xs{100, 200, 300};

    xs.assign_range(
        std::views::iota(1, 6)
        | std::views::transform([](int x) { return x * x; })
    );

    for (int x : xs) {
        std::print("{} ", x);
    }
    std::print("\n");
}
```

assign_range replaces the entire contents of the list with whatever the given range produces.

C++23
introduces
advanced range
concepts that
resemble bash
piping

```
whovian@whovianphy504:~/host/Lecture18$ nano list13.cpp
whovian@whovianphy504:~/host/Lecture18$ g++ -std=c++26 list13.cpp -o lt13
whovian@whovianphy504:~/host/Lecture18$ ./lt13
1 4 9 16 25
```

Vectors

C++ Vectors

Memory in vectors
is **contiguous**

There is no
vector::push_front
(Vectors are not a
double linked list)

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

int main ()
{
    std::vector<int> lt = {7, 5, 16, 8, 10};
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    lt.push_back(13); // add to the back of the list
    // There is no push_front on vectors
    for (auto x : lt)
        fmt::print("{0}, ", x);
    fmt::print("\n");

    // find elements in the list with the STL algorithm
    auto it = std::find(lt.begin(), lt.end(), 16);
    if (it != lt.end())
        lt.insert(it, 42); // inserts value before pos

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

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ vector.cpp -o vector1 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector1
7, 5, 16, 8, 10,
7, 5, 16, 8, 10, 13,
7, 5, 42, 16, 8, 10, 13,
```

C++ Vectors

Memory in vectors is
contiguous

So totally ok to use
operator[] or **.at()**

(**.at()** is safer: checks if
out of bounds)

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

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

    fmt::print("First element {0}\n", lt[0]);
    fmt::print("First element {0}\n", lt.at(0));

    const auto sz = lt.size();
    fmt::print("Size of the vector: {0}\n", sz);

    // This an interesting concept (capacity)
    // vector sometimes holds more memory that its size
    const auto cap = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap);

    // Why? to allow fast push back
    lt.push_back(100);

    const auto sz2 = lt.size();
    fmt::print("Size of the vector: {0}\n", sz2);

    const auto cap2 = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap2);

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

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ vector2.cpp -o vector2 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector2
First element 7
First element 7
Size of the vector: 5
Capacity of the vector: 5
Size of the vector: 6
Capacity of the vector: 10
7, 5, 16, 8, 10, 100,
```

C++ Vectors

Capacity is usually larger than size (vector holds more memory than currently needed)

Why? To allow fast future expansion via **vector::push_back**

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

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

    fmt::print("First element {0}\n", lt[0]);
    fmt::print("First element {0}\n", lt.at(0));

    const auto sz = lt.size();
    fmt::print("Size of the vector: {0}\n", sz);

    // This an interesting concept (capacity)
    // vector sometimes holds more memory that its size
    const auto cap = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap);

    // Why? to allow fast push back
    lt.push_back(100);

    const auto sz2 = lt.size();
    fmt::print("Size of the vector: {0}\n", sz2);

    const auto cap2 = lt.capacity();
    fmt::print("Capacity of the vector: {0}\n", cap2);

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

    return 0;
}
```

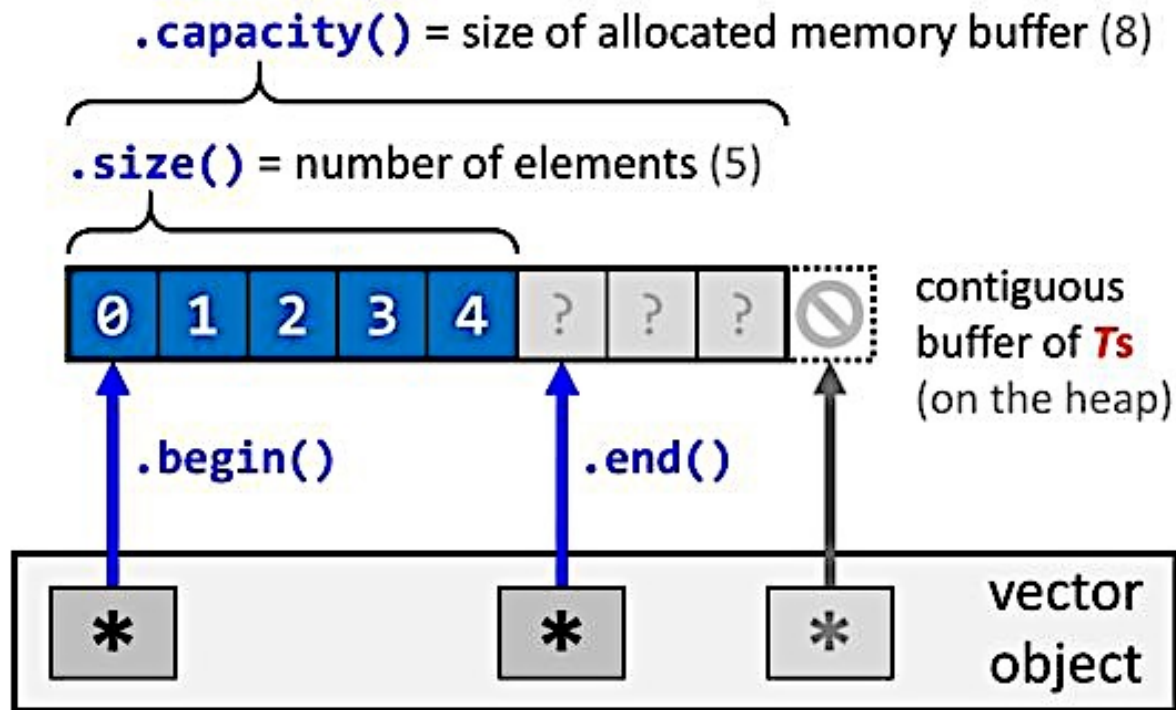
```
whovian@phy504:~/host/CPP$ g++ vector2.cpp -o vector2 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector2
First element 7
First element 7
Size of the vector: 5
Capacity of the vector: 5
Size of the vector: 6
Capacity of the vector: 10
7, 5, 16, 8, 10, 100,
```

C++ Vectors

Capacity is usually larger than size

.push_back() is cheap as long as its size < capacity

`std::vector<T>` Size vs. Capacity



What happens when calling **.push_back()** with **size = capacity**?

Slower: the entire vector needs to move to a new location with extra capacity

C++ Vectors

What happens when calling

.push_back()

with **sz = capacity?**

The entire vector needs to move to a new location with extra capacity

```
#include <algorithm>
#include <fmt/core.h>
#include <vector>
#include <chrono>
constexpr int bf = 250'000'000;

int main ()
{
    { // just creating local stack
        std::vector<int> lt;
        // change the capacity without changing its size
        lt.reserve(bf+1);

        auto t0 = std::chrono::high_resolution_clock::now();

        for (int i=0; i<bf; i++)
            lt.push_back(i);

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;

        fmt::print("Time to executie the loop (capacity"
            " pre-allocated): {0}\n", fs.count());
    }
    {
        std::vector<int> lt;

        auto t0 = std::chrono::high_resolution_clock::now();

        for (int i=0; i<bf; i++)
            lt.push_back(i);

        auto t1 = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> fs = t1 - t0;

        fmt::print("Time to executie the loop (capacity NOT"
            " pre-allocated): {0} sec\n", fs.count());
    }
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ nano vector3.cpp
whovian@phy504:~/host/CPP$ g++ vector3.cpp -o vector3 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./vector3
Time to executie the loop (capacity pre-allocated): 23.943483455
Time to executie the loop (capacity NOT pre-allocated): 25.488765179 sec
```



```
#include <vector>
```

```
constexpr int sum_list() {  
    std::vector<int> xs{1, 2, 3};  
    xs.insert(xs.begin(), 0);  
    xs.push_back(4);
```

```
    int sum = 0;  
    for (int x : xs) {  
        sum += x;  
    }  
    return sum;  
}
```

```
static_assert(sum_list() == 9);
```

```
int main() {}
```

C++ (20) Vector

As we seen before, C++
can perform operations
at compile time

(**constexpr**)

C++20 expanded the
number of such
operations, including
std::vector operations

```
whovian@whovianphy504:~/host/Lecture18$ nano vector9.cpp
```

```
whovian@whovianphy504:~/host/Lecture18$ g++ -std=c++26 vector9.cpp -o vt9
```

```
vector9.cpp:15:26: error: static assertion failed
```

```
15 | static_assert(sum_list() == 9);
```

```
vector9.cpp:15:26: note: the comparison reduces to '(10 == 9)'
```

C++ (20) ranges

Next class is all about ranges.

Problem: copy the keys of a map to a vector

Old vs New C++ solutions.

```
#include<vector>
#include<print>
#include<map>
#include<ranges>
#include<string>
using std::map; using std::vector;
using std::string; using std::views::keys;
using std::ranges::to; using std::ranges::views::zip;
int main() {
    map<string,int>mymap = {{ "Vivian", 37},
                           {"Rihanna", 35}};

    // How to copy the keys of the map to a vector?

    // Old C++98 style (no auto in C++98)
    std::vector<std::string> k1;
    typedef map<string,int>::iterator miter;
    for (miter it = mymap.begin(); it != mymap.end(); ++it)
        k1.push_back(it->first);

    // C++17 (structure binding)
    vector<string> k2;
    for (auto [x, _] : mymap) k2.push_back(x);

    // C++20 (views and ranges)
    vector<string> k3 = mymap | keys | to<vector>();

    // Lets print the results (C++23 style)
    for (auto [x, y, z] : zip(k1, k2, k3))
        std::println("{0}, {1}, {2}", x, y, z);
}
```

```
whovian@phy504:~/host/Lecture19$ nano vectormap.cpp
whovian@phy504:~/host/Lecture19$ ./vmap
Rihanna, Rihanna, Rihanna
Vivian, Vivian, Vivian
```

Linear Algebra in C++ with STL Vectors

```
#include <print>
#include <vector>
#include <mdspan>
using std::print; using std::vector;
int main ()
{
    vector<int> lt;
    lt.resize(9); // changing vector size

    // wrap this vector as a 3 x 3 matrix
    auto ms = std::mdspan(lt.data(), 3, 3);

    // size_t = unsigned integer
    for(size_t i=0; i != ms.extent(0); i++) {
        for(size_t j=0; j != ms.extent(1); j++) {
            ms[i,j] = i + j; // note the 2D Numpy indexing
        }
    }
    for(size_t i=0; i != ms.extent(0); i++) {
        for(size_t j=0; j != ms.extent(1); j++) {
            print("a({0},{1}) = {2}\n", i, j, ms[i,j]);
        }
    }
    return 0;
}
```

Can we stride a 1D
vector to be a matrix

(or a tensor)?

In C++23, we can!

Works with **gcc-16**

```
whovian@phy504:~/host/Lecture19$ nano vector4.cpp
whovian@phy504:~/host/Lecture19$ /opt/gcc-latest/bin/g++ vector4.cpp -o v4 -std=c++26
whovian@phy504:~/host/Lecture19$ ./v4
a(0,0) = 0
a(0,1) = 1
a(0,2) = 2
a(1,0) = 1
a(1,1) = 2
a(1,2) = 3
a(2,0) = 2
a(2,1) = 3
a(2,2) = 4
```

Linear Algebra in C++ with STL Vectors

```
#include <print>
#include <vector>
#include <mdspan>
#include <numeric>
using std::print; using std::vector;
int main() {
    constexpr int N = 3;
    vector<int> a_data(N * N);
    vector<int> b_data(N * N);
    vector<int> c_data(N * N, 0);

    std::iota(a_data.begin(), a_data.end(), 0);
    std::iota(b_data.begin(), b_data.end(), 1);

    auto A = std::mdspan(a_data.data(), N, N);
    auto B = std::mdspan(b_data.data(), N, N);
    auto C = std::mdspan(c_data.data(), N, N);

    // matrix multiply: C = A * B
    for (size_t i = 0; i != A.extent(0); i++) {
        for (size_t j = 0; j != B.extent(1); j++) {
            for (size_t k = 0; k != A.extent(1); k++) {
                C[i,j] += A[i,k] * B[k,j];
            }
        }
    }

    print("\nC = A * B:\n");
    for (size_t i = 0; i != C.extent(0); i++) {
        for (size_t j = 0; j != C.extent(1); j++) {
            print("{:4d} ", C[i,j]);
        }
        print("\n");
    }
    return 0;
}
```

Can we stride a 1D
vector to be a matrix
(or a tensor)?

In C++23, we can!

Perfect for Matrix
Multiplication

```
whovian@phy504:~/host/Lecture19$ nano vector4v2.cpp
whovian@phy504:~/host/Lecture19$ /opt/gcc-latest/bin/g++ vector4v2.cpp -o v4v2 -std=c++26
whovian@phy504:~/host/Lecture19$ ./v4v2

C = A * B:
18  21  24
54  66  78
90 111 132
```

Linear Algebra in C++

STL only implemented Linear Algebra (higher order Blas functions) in **C++26**

cppreference.com

Search

Page Discussion

C++ Standard library headers

Standard library header `<linalg>` (C++26)

This header is part of the `numeric` library.

Source <https://en.cppreference.com/w/cpp/header/linalg.html>

No compiler has implemented it yet
(*nvidia compiler seems to have a partial implementation)

So, how did C++ users do LA before C++26?
External Libraries like Armadillo (Additional
reading)

Linear Algebra in C++

C++ has some awesome LA libraries.

My favorite is: [Armadillo lib](#)



Armadillo

C++ library for linear algebra & scientific computing

[About](#) [Documentation](#) [Questions](#) [Speed](#) [Contact](#) [Download](#)

- Armadillo is a high quality linear algebra library (matrix maths) for the C++ language, aiming towards a good balance between speed and ease of use
- Provides high-level syntax and [functionality](#) deliberately similar to Matlab
- Useful for algorithm development directly in C++, or quick conversion of research code into production environments
- Provides efficient classes for vectors, matrices and cubes; dense and sparse matrices are supported
- Integer, floating point and complex numbers are supported
- A sophisticated expression evaluator (based on template meta-programming) automatically combines several operations to increase speed and efficiency
- Dynamic evaluation automatically chooses optimal code paths based on detected matrix structures
- Various matrix decompositions (eigen, SVD, QR, etc) are provided through integration with [LAPACK](#), or one of its high performance drop-in replacements (eg. [MKL](#) or [OpenBLAS](#))
- Can automatically use OpenMP multi-threading (parallelisation) to speed up computationally expensive operations
- Distributed under the permissive [Apache 2.0 license](#), useful for both open-source and proprietary (closed-source) software
- Can be used for machine learning, pattern recognition, computer vision, signal processing, bioinformatics, statistics, finance, etc
- [download latest version](#) | [git repo](#) | [browse documentation](#)

Supported by:



Additional Reading: Linear Algebra in C++

```
#include <iostream>
#include <armadillo>

using namespace std;

int main()
{
    // Matrix 5 x 5, fill with random numbers
    arma::mat A(5, 5, arma::fill::randn);
    arma::mat B(5, 5, arma::fill::randn);

    arma::mat C = A * B;
    cout << C << endl;

    arma::vec d = C.col(0); // first column of C matrix
    cout << d << endl;

    // Two ways of computing the inverse matrix
    arma::mat D = C.i();
    cout << C * D << endl;

    arma::mat E = inv(C, arma::inv_opts::no_ugly);
    cout << C * E << endl;

    // Computing Eigenvalues
    arma::mat F = C.t()*C; // generate symmetric matrix
    arma::vec eigval = arma::eig_sym( F );
    cout << eigval << endl;

    return 0;
}
```

C++ STL didn't have a
LA library prior to
C++26. A much better
solution: **Armadillo lib**

Armadillo wraps lower-
level linear algebra
libraries (good practice)

Additional Reading: Linear Algebra in C++
C++ STL doesn't have high performance library for linear algebra. Much better solution: **Armadillo lib**

```
whovian@whovianphy504:~/host/Lecture19$ g++ vector5.cpp -o v5 -std=c++26 -larmadillo -lblas -llapack
whovian@whovianphy504:~/host/Lecture19$ ./v5
-0.9735  0.7091  0.4922 -0.8623 -1.1253
-3.3394 -5.8915  4.2455 -5.2301  0.5084
 0.8657  0.2248 -2.9582  1.8600 -0.4311
 0.2310  0.6728 -0.6978 -0.0737 -0.0110
-3.4597 -3.5987  7.0374 -6.7701 -1.9558

-0.9735
-3.3394
 0.8657
 0.2310
-3.4597

 1.0000e+00 -1.1102e-16 -2.2204e-16 -1.6653e-16 0
-4.2186e-16  1.0000e+00 -5.5511e-16  1.6376e-15 2.7756e-17
 3.3610e-17  2.7756e-17  1.0000e+00 -8.3267e-17 -8.3267e-17
 6.8850e-17  2.0383e-17  9.6277e-17  1.0000e+00 -3.5128e-17
-1.7868e-16  1.1102e-16 -4.4409e-16 -1.1102e-16  1.0000e+00

 1.0000e+00 -1.1102e-16 -2.2204e-16 -1.6653e-16 0
-4.2186e-16  1.0000e+00 -5.5511e-16  1.6376e-15 2.7756e-17
 3.3610e-17  2.7756e-17  1.0000e+00 -8.3267e-17 -8.3267e-17
 6.8850e-17  2.0383e-17  9.6277e-17  1.0000e+00 -3.5128e-17
-1.7868e-16  1.1102e-16 -4.4409e-16 -1.1102e-16  1.0000e+00

2.5922e-01
6.2891e-01
2.7567e+00
1.2318e+01
2.1754e+02
```

Last Container of Phy504: Unordered maps

C++ unordered maps

The API is
similar, but
behind the
scenes, they
are quite
different

```
#include <map>
#include <unordered_map>
#include <fmt/core.h>

int main() {
    using namespace std::string_literals; // suffix s C++14 feature

    std::unordered_map<int, std::string> uMap;
    uMap.insert({3, "Three"s}); // the s suffix tells the compiler
                                // to convert string literal to std::string

    uMap[1] = "One"s; // Inefficient but correct. Operator [] will
                     // create the key pair for you

    uMap[2] = "Two"; // Compiler knows the second element of the pair
                    // is an std::string so the "s" suffix is optional

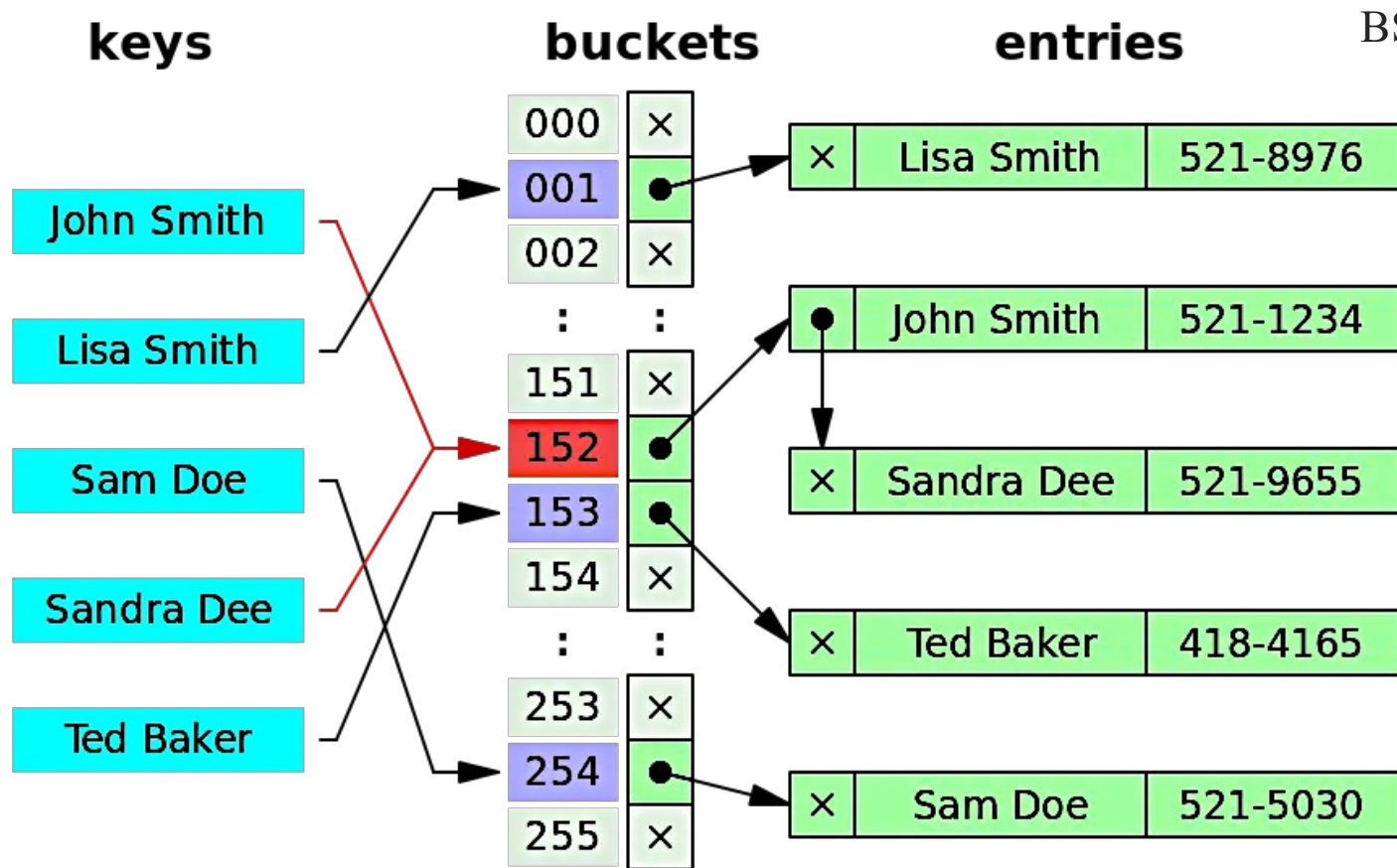
    // "s" suffix would not be optional in C++ constructions with auto
    // auto x = "one"s // creates a std::string x
    // auto x = "one"  // creates const char* x
    // One of the many quirks of C being accepted as subset of c++!

    // C++ allows python style for loop styles
    for (const auto pair : uMap) {
        fmt::print("{0}, {1}\n", pair.first, pair.second);
    }
    return 0;
}
```

```
whovian@phy504:~/host/Lecture18$ g++ maps.cpp -o maps -lfmt
whovian@phy504:~/host/Lecture18$ ./maps
2, Two
1, One
3, Three
```

C++ unordered maps (behind the scenes)

Unordered maps use a **hash function** instead of a BST



<https://jbseg.medium.com/c-unordered-map-under-the-hood-9540cec4553a>

Keys are hashed to an index in the array, aka a bucket. Each bucket is the head of a linked list.

C++ unordered maps (behind the scenes)

Each bucket is the head of a linked list

Bucket array:

[0] → NULL

[1] → NULL

[2] → [John Smith, 521-1234] → [Lisa Smith, 521-8976] → NULL

[3] → [Sam Doe, 555-4321] → NULL

[4] → NULL

Why? Because we need to take into account collision
(otherwise we could assign a single pair to each bucket)

E.g, John/Lisa hash to the same bucket, 152. Thus,
they both get added to the linked list for bucket 152.

C++ unordered maps (behind the scenes)

Unordered maps use a **hash function** instead of BST

Properties of Hash functions

We try very hard to make **uniform** hash functions to minimize the number of hash collisions.

A **hash collision** is when two objects x, y hash to the same value (i.e. $H(x) = H(y)$).

In the table we generated earlier William and Kate have a hash collision.

Name	Age	Sex	Hash
William	21	M	5
Kate	19	F	5
Bob	33	M	1
Rose	26	F	0

C++ unordered maps (behind the scenes)

Unordered maps use a **hash function** instead of BST

```
#include <string>
#include <unordered_map>
#include <fmt/core.h>
```

```
int bucket_size = 100;
```

Concept of a functor will be
taught in Lectures 6-8

```
int hash(std::string key) {
    int hashed_key = std::hash<std::string>{}(key); // note the extra {} - functor
    return hashed_key % bucket_size;
}
```

```
int main () {
    fmt::print("{0}\n", hash("Vivian Miranda"));
    fmt::print("{0}\n", hash("Phy504"));
    return 0;
}
```

Keys are hashed to an index in the array, aka a bucket.

```
whovian@phy504:~/host/Lecture18$ g++ hash.cpp -std=c++23 -o hash -lfmt
whovian@phy504:~/host/Lecture18$ ./hash
80
73
```

C++ unordered maps (behind the scenes)

Keys are hashed to an index in the array, a.k.a a bucket.

```
#include <string>
#include <unordered_map>
#include <fmt/core.h>
#include <list>
#include <vector>

// std::vector<type X> = C++ array of type X (this lecture)

// std::list<type X> = linked list of type X (this lecture)

// std::pair<type X, type Y> is a pair of
//                               one object of type X
//                               one object of type Y (previous lecture)

// A bucket is linked lists of pairs of (string, int)
// The string of the pair is the original key.
// The int of the pair is the value
// The hash of the key determines which bucket (vector index) to use
std::vector<std::list<std::pair<std::string, int>>> buckets;
int bucket_size = 100;
```

C++ unordered maps (behind the scenes)

How do you code to find elements in a hash table?

```
using mypair = std::pair<std::string,int>;
using mylist = std::list<mypair>;
using myvec  = std::vector<mylist>;
myvec buckets;
int bucket_size = 100;
```

using is an excellent syntax sugar for types (more powerful than C typedef)

```
// new concept: iterator (they are like a pointers)
mylist::iterator find(std::string key) {
    int hashed_key    = std::hash<std::string>{}(key);
    int bucket_index = hashed_key % bucket_size;
```

Buckets[bucket_index]
is a linked list

```
// old C++ of iteration on all elements linked list
```

```
for (mylist::iterator i = buckets[bucket_index].begin(); i != buckets[bucket_index].end(); ++i) {
    if((*i).first == key) {          // find if the linked list contain the key
        return i;
    }
}
```

```
// if the map does not have the key, return the
// reference to the last linked list element
return buckets[bucket_index].end();
}
```

The mysterious part (yet) in C++ is how iterators can do the **operator ++** in a linked list

C++ unordered maps (behind the scenes)

How do you code to find elements in a hash table?

auto keyword here can be used for a little syntax sugar

```
// new concept: iterator (they are like a pointers)
mylist::iterator find(std::string key) {
    int hashed_key    = std::hash<std::string>{}(key);
    int bucket_index  = hashed_key % bucket_size;

    auto& bucket = buckets[hashed_key]; // & = reference - no copies
    for (auto i = bucket.begin(); i != bucket.end(); ++i) {
        if((*i).first == key) { // find if the linked list contain the key
            return i;
        }
    }
    // if the map does not have the key, return the
    // reference to the last linked list element
    return bucket.end();
}
```

C++ unordered maps (behind the scenes)

How do you code to add elements in a hash table?

```
void insert(std::string key, int val) {  
    int hashed_key = std::hash<std::string>{}(key);  
  
    auto it = find(key); last slide  
    if(it != buckets[hashed_key].end()) {  
        // if key already exists, replace val  
        it->second = val;  
        return;  
    }  
  
    // if it is a new key - add to the linked list  
    buckets[hashed_key].push_back({key, val});  
    total_elements++; // keep track of total elements  
    return;  
}
```

Homework I

Given the vector below, print all its elements in five different ways

```
std::vector<int> lt = {7, 5, 16, 8, 10};
```

- (a) Print using C style **for loop** and operator []
- (b) Print using C style **for loop** and member function .at()
- (c) Print using C style **for loop** & C++ iterators
- (d) Print using C++ Python Style
- (e) Print using C++ STL algorithm **for_each()**

Homework II

Below, I show my simple implementation of Pascal's Triangle

```
#include <vector>
#include <fmt/core.h>
#include <algorithm>
#include <vector>
using vec = std::vector<int>;
using iter = std::vector<int>::iterator;

vec NextRow(vec row) {
    vec res;
    int prev = 0;
    for (auto x : row) {
        res.push_back(prev + x);
        prev = x;
    }
    res.push_back(prev);
    return res;
}

int main () {
    vec pascal = {1};
    for (int i=0; i<4; i++) {
        pascal = NextRow(pascal);
        for (auto x : pascal) {
            fmt::print("{0} ", x);
        }
        fmt::print("\n");
    }
}
```

Change this implementation to

- (1) Use iterators
- (2) Do not use **.push_back()** (you can extract the size of the row!)
- (3) Although not efficient, redo this code replacing STL vector with STL list (both (1) and (2) with lists)

I hope you can see that with C++ abstraction, it was not hard to go from vector to list.

Homework III

STL Algorithm + C++ random library is fantastic for generating a vector of random integers (w/o repetition) as shown below

Using STL algorithms, sort this vector (you sort manually by implementing [bubble-sort](https://www.geeksforgeeks.org/dsa/bubble-sort-algorithm/) – see source code in the link <https://www.geeksforgeeks.org/dsa/bubble-sort-algorithm/>)

```
#include <iostream>
#include <vector>
#include <numeric>    // For std::iota
#include <algorithm>  // For std::shuffle
#include <random>     // For std::random_device and std::mt19937

int main() {
    std::vector<int> v(10);

    // Fill the vector with sequential integers [0, 1, 2, ..., N-1].
    std::iota(v.begin(), v.end(), 0);

    // Create a random number generator seeded with a random device.
    std::random_device rd;
    std::mt19937 gen(rd());

    // Shuffle the vector to randomize the order of the numbers.
    std::shuffle(v.begin(), v.end(), gen);
```

Homework IV

Given two sorted STL vectors **v1** and **v2** of the same size
return the median of the two sorted arrays.

Assume the size is odd and use **std::iota** to generate them.

- (1) Solve this exercise by merging the two STL vectors. Merge them thinking like a C program (defining a third STL vector with sufficient size)
- (2) Solve this exercise by merging the two STL vectors, but this time, think like a C++ program and use the STL algorithm **std::merge** (research about this function)
- (3) Solve without merging the two STL vectors

Homework V

```
#include <fmt/core.h>
#include <array>
#include <numeric>    // For std::iota
#include <random>      // For std::random_device and std::mt19937
#include <algorithm>   // For std::ranges::generate

auto dice() {
    // static: variables retain memory between calls
    static std::uniform_int_distribution<int> distr{1, 6};
    static std::random_device device;
    static std::mt19937 engine {device()};
    return distr(engine);
}

int main() {
    constexpr int size = 10; // constexpr = compile time const

    std::array<int, size> v; // Arrays: size is known at compile time

    std::ranges::generate(v, dice); // modern C++20
    for (auto x : v) {
        fmt::print("{0} ", x);
    }
    fmt::print("\n");
}
```

<https://en.cppreference.com/w/cpp/numeric/random>

```
whovian@phy504:~/host/Lecture18$ g++ vector8.cpp -o vector8 -std=c++20 -lfmt
whovian@phy504:~/host/Lecture18$ ./vector8
6 2 4 5 6 1 3 3 6 4
```

C++ offers support for random numbers. Here, we simulate a dice roll and then fill an array.

- (a) Modify this code to fill the array from a normal distribution mean zero and sigma one. Use **`std::normal_distribution`**
- (b) Research about **`std::ranges::generate_n`** rewrite the code to roll the dice again, but only on the last 5 elements.

Homework VI

- (1) Using C++ random number generators, fill two matrices with a Gaussian Normal distribution (mean zero and sigma = 1)
- (2) Using `std::mdspan`, convert this pseudo-code into C++ to compute the determinant of a matrix. (You get ½ point if you assume it is a 3x3 matrix)

DETERMINANT via Gaussian Elimination with Partial Pivoting

Input: an $N \times N$ matrix A

Output: $\det(A)$

1. Copy A into a working matrix W (we will modify it in place)
2. Set $\det = 1$
Set $\text{swaps} = 0$
3. For each column $\text{col} = 0, 1, \dots, N-1$:
 - a. FIND PIVOT: search rows $\text{col}, \text{col}+1, \dots, N-1$
find the row with the largest absolute value in column col
call that row "pivot"
 - b. SWAP ROWS: if $\text{pivot} \neq \text{col}$
swap all elements of row col and row pivot in W
increment swaps
 - c. CHECK ZERO: if $W[\text{col}, \text{col}] == 0$
 $\det = 0$, stop early
 - d. ACCUMULATE: multiply $\det$ by $W[\text{col}, \text{col}]$
 - e. ELIMINATE BELOW: for each row below the diagonal ($\text{row} = \text{col}+1, \dots, N-1$)
compute $\text{factor} = W[\text{row}, \text{col}] / W[\text{col}, \text{col}]$
for each column $j = \text{col}+1, \dots, N-1$
 $W[\text{row}, j] = W[\text{row}, j] - \text{factor} * W[\text{col}, j]$
4. If swaps is odd, flip the sign of $\det$
5. Return $\det$