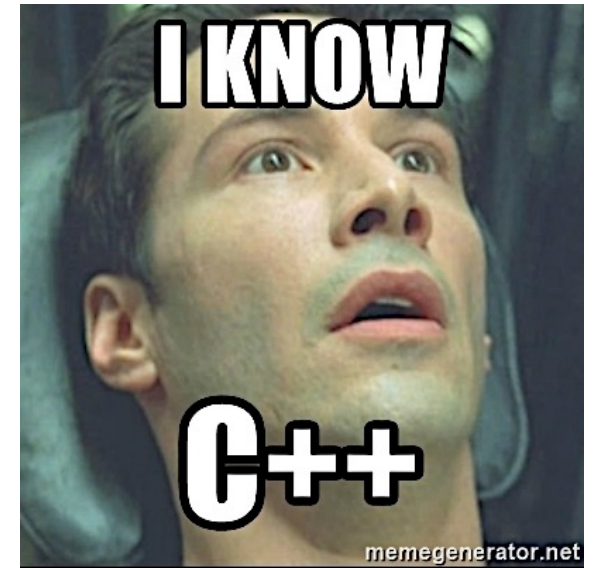
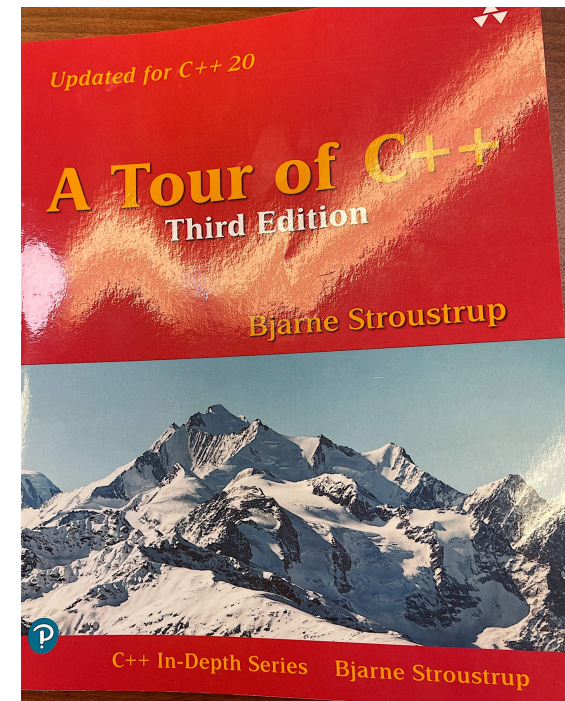


Lecture 17: C++ part II



Suggested Literature



RECAP: Hello World (new C++)

C++ offers >1 way to do almost anything

```
int main ()
{
    fmt::print("Hello World!\n");

    // Positional Arguments
    fmt::print("{1} {0}.\n", "World", "Hello");

    fmt::print(
        "Hello, {name}! The answer is {number}. Goodbye, {name}\n",
        fmt::arg("name", "World"), fmt::arg("number", 42));
}
```

```
whovian@phy504:~/host$ nano hello3.cpp
whovian@phy504:~/host$ g++ hello3.cpp -o hello3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./hello3
Hello World!
Hello World.
Hello, World! The answer is 42. Goodbye, World
```

C++23
introduces
std::print

Standard based
on the

[FMT library](#)

(which I will
use a lot here)

RECAP: Functions C vs C++

```
#include <string>
#include <iostream>

void Print(int x, double y)
{
    std::cout << "Print1: " <<
        x+y << std::endl;
}

void Print(double x, int y)
{
    std::cout << "Print2: " <<
        x-y << std::endl;
}

int main()
{
    Print(1, 2.0);
    Print(1.0, 2);
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ -std=c++20 functions2.cpp -o func2
whovian@phy504:~/host/CPP$ ./func2
Print1: 3
Print2: -1
```

Functions in C++ allow naming overloading.

C++ lets you specify >1 function with the same name in the same scope

Overloading works on types
Argument order is imp.

f(int, double) and
f(double, int) are different

RECAP: Functions C vs C++

```
std::tuple<double, char, std::string>
get_student(const int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
        case 1:
            return {2.9, 'C', "Milhouse Van Houten"};
        case 2:
            return {1.7, 'D', "Ralph Wiggum"};
    }

    // C++ error handling (we will revisit them later)
    throw std::invalid_argument("id");
}

int main()
{ // Many ways to access the return type of get_students

    // 3: (Best) with C++17 structured binding:
    const auto [ gpa, grade, name ] = get_student(1);

    fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
        1, gpa, grade, name);
}
```

Many ways to receive
the elements of a tuple

Options change in
subtle but key features

Provide the compiler
information for users
to use **auto** keyword
(a C++ superpower)

← Bad example if C++14
but ok if C++17!

Functions C vs C++

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

// (1) Does the optional arg have a logical default?
// (2) Is the function implemented the same when the
//      optional arg is given and when it's not?
// From SO: stackoverflow.com/a/56388368/2472169
// if YES and YES, use default argument
// if NO and YES, then use std::optional<T>
// if NO and NO, then use overloading

using fptr = int (*)(int, int);

int add(int x, int y, fptr sum = nullptr) {
    return (sum == nullptr) ? x + y : sum(x, y);
}

int sum(int x, int y) { return x + y; }

int main()
{
    fmt::print("Default arg: {0} \n", add(2, 3));
    fmt::print("Default arg: {0} \n", add(2, 3, &sum));
}
```

Functions can have optional arguments and return values.

How?

1. default values
2. overloading
3. [std::optional<T>](#)

```
whovian@phy504:~/host$ g++ optional3.cpp -o opt3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./opt3
Default arg: 5
Default arg: 5
```

Functions C vs C++

```
using fptr = int (*)(int, int);

int add(int x, int y) { // here we avoid if/else struct
    return x + y;
}

int add(int x, int y, fptr sum) {
    return sum(x, y);
}

int add(int x, int y, std::nullptr_t) {
    // nullptr_t = type of nullptr (C++ version of NULL)
    return x + y;
}

int sum(int x, int y) { return x + y;}

int main()
{
    fmt::print("Overloading: {0} \n", add(2, 3));
    fmt::print("Overloading: {0} \n", add(2, 3, &sum));
    fmt::print("Overloading: {0} \n", add(2, 3, nullptr));
}
```

Functions can have optional arguments and return values.

How?

1. default values
2. overloading
3. [std::optional<T>](#)

```
whovian@phy504:~/host$ g++ optional4.cpp -o opt4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./opt4
Overloading: 5
Overloading: 5
Overloading: 5
```

Functions C vs C++

```
#include <cstdint>
#include <optional>
#include <fmt/core.h>

// (1) Does the optional arg have a logical default?
// (2) Is the function implemented the same when the
//      optional arg is given and when it's not?
// From SO: stackoverflow.com/a/56388368/2472169
// if YES and YES, use default argument
// if NO and YES, then use std::optional<T>
// if NO and NO, then use overloading

using fptr = int (*)(int, int);

int add(int x, int y, std::optional<fptr> sum) {
    return (sum.has_value() == 0 || sum == nullptr) ?
        x + y : sum.value()(x,y);
}

int sum(int x, int y) { return x + y;}

int main()
{
    // interesting here - you must always give 3 args
    fmt::print("std::optional: {0} \n", add(2, 3, nullptr));
    fmt::print("std::optional: {0} \n", add(2, 3, {}));
    fmt::print("std::optional: {0} \n", add(2, 3, std::nullopt));
    fmt::print("std::optional: {0} \n", add(2, 3, &sum));
}
```

```
whovian@phy504:~/host$ g++ optional5.cpp -o opt5 -std=c++20 -lfmt
whovian@phy504:~/host$ ./opt5
std::optional: 5
std::optional: 5
std::optional: 5
std::optional: 5
```

Functions can have
optional arguments
and return values.

How?

1. default values
2. overloading
3. [std::optional<T>](#)

```

#include <vector>
#include <optional>
#include <fmt/ranges.h>

std::vector<int> filter(
    const std::vector<int>& data,
    std::optional<int> min = std::nullopt,
    std::optional<int> max = std::nullopt)
{
    std::vector<int> result;
    for (int x : data) {
        if (min.has_value() &&
            x < min.value()) {
            continue;
        }
        if (max.has_value() &&
            x > max.value()) {
            continue;
        }
        result.push_back(x);
    }
    return result;
}

int main() {
    std::vector<int> data = {-5, -1, 0, 1, 5, 10};

    // unambiguously: no minimum
    fmt::print("{}\n",
        fmt::join(filter(data), ", "));

    // unambiguously: min is 0
    fmt::print("{}\n",
        fmt::join(filter(data, 0), ", "));
}

```

Functions C vs C++

Why not default value in this case?

There is a fundamental difference between not providing **min** and **max** and having a default **min/max**

Question: is no value a meaningful state?

Does the absence itself must be preserved and observed as data?

Functions C vs C++

This code is slightly inefficient, as the `.value()` member function provides **Checked Access (CA)**.

What is **CA**? `.value()` throws a `std::bad_optional_access` exception if empty

```
using fptr = int (*)(int, int);

int add(int x, int y, std::optional<fptr> sum) {
    return (sum.has_value() == 0 || sum == nullptr) ?
        x + y : sum.value()(x,y);
}
```

C++ offers both **Checked and Unchecked Access** to the underlying data the C++ STL abstraction holds

Functions C vs C++

Unchecked Access: usually done (not always) via **operators**. In this case, we will use the **operator ***

```
int add(int x, int y, std::optional<fptr> sum) {  
    return (sum.has_value() == 0 || sum == nullptr) ?  
        x + y : (*sum)(x,y);  
}
```

Why not just stick to C and use a C **void*** pointer and an integer flag to select the options?

No right answer. You may stick to C and be happy.

What C++ offers is (when well coded) a stronger type safety and checks (at compiler and runtime)

Functions C vs C++

The return type may also be
optional

In this case, what if the
student is not found?

In C it is possible to recreate
the functionality by
returning an integer (error
code) and having arg as
Student double pointer

```
#include <vector>
#include <optional>
#include <string>
#include <fmt/format.h>

struct Student {
    std::string name;
    double      grade;
};

std::optional<Student> find_student(
    const std::vector<Student> &students,
    const std::string &name)
{
    for (auto [n, g] : students) {
        if (n == name) {
            return Student{n, g};
        }
    }
    return std::nullopt;
}

int main() {
    std::vector<Student> students = {
        {"Alice", 9.5},
        {"Bob", 7.2}
    };

    auto result = find_student(students, "Bob");
    if (result.has_value()) {
        fmt::print("Found: {} with grade {}\n",
            result.value().name,
            result.value().grade);
    }

    auto missing = find_student(students, "Dave");
    if (!missing.has_value()) {
        fmt::print("Student not found\n");
    }
}
```

```
#include <cstdlib>
#include <optional>
#include <cmath>
#include <fmt/core.h>
```

C++ standard is
much richer than C

```
double f(double x)
{
    fmt::print("We called double f(double) \n");
    // cbrt = cubic root
    // tgamma = gamma function (C++17 has many special functions!)
    return std::cbrt(x) +
           std::sqrt(std::abs(x)) +
           std::tgamma(x);
}
```

```
double f(double* x)
{
    fmt::print("We called double f(double*) \n");
    if (x != nullptr)
        return *x;
    else
        return -1.;
}
```

```
int main()
{
    // var type is super important in C++.
    double r1 = f(NULL);
    double r2 = f(2.0);
    double r3 = f(nullptr);
}
```

Functions C vs C++

Why **nullptr**? Func.
overloading: C++ is
pedantic with types
(no implicit casts).

NULL is an **int**, but
sometimes **int***

```
whovian@phy504:~/host$ g++ null.cpp -o null -std=c++20 -lfmt
null.cpp: In function 'int main()':
null.cpp:28:16: error: call of overloaded 'f(NULL)' is ambiguous
    28 |     double r1 = f(NULL);
        |                  ~~~~~~
null.cpp:6:8: note: candidate: 'double f(double)'
     6 | double f(double x)
        |         ^
null.cpp:16:8: note: candidate: 'double f(double*)'
    16 | double f(double* x)
        |         ^
```

Functions C vs C++

Why **nullptr**?

Function overloading:
C++ must be pedantic
with types

Don't assume
functions in C and
C++ behave the same

```
#include <cstdlib>
#include <optional>
#include <cmath>
#include <fmt/core.h>

double f(double x)
{
    fmt::print("We called double f(double) \n");
    // cbrt = cubic root
    // tgamma = gamma function (C++17 has many special functions!)
    return std::cbrt(x) +
           std::sqrt(std::abs(x)) +
           std::tgamma(x);
}

double f(double* x)
{
    fmt::print("We called double f(double*) \n");
    if (x != nullptr)
        return *x;
    else
        return -1.;
}

int main()
{
    // var type is super important in C++. Be careful
    double r2 = f(2.0);
    double r3 = f(nullptr);
}
```

```
whovian@phy504:~/host$ g++ null2.cpp -o null2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./null2
We called double f(double)
We called double f(double*)
```

Maps

C++ Maps

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

using std::map;    // better way to avoid using std::
using std::string; // don't pollute your code

int main ()
{
    // map<key type, value type> = initialize list
    // init the map with some key, values
    // We learned that we can use {} to init vars in C++
    map<string, double> mymap = {{"Vivian", 37},
                                {"Marcelo", 38},
                                {"Vania", 58},
                                {"Isabella", 25}};

    // Similar to Python, we can access some functions with .
    // These are member functions that modify/access elements
    // associated with a particular variable.
    mymap.insert({"Gilberto", 56});

    mymap["Beyonce"] = 41; // some types have operators

    // Operator [] is sometimes deemed unsafe for two reasons.
    // key doesn't exist: the code adds it to the map
    // key does exist: [] modifies it while insert doesn't
    fmt::print("(1) Rihanna is {} years old\n", mymap["Rihanna"]);

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

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

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps.cpp -o maps1 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps1
(1) Rihanna is 0 years old
(2) Rihanna is 0 years old
(3) Rihanna is 35 years old
```

Maps allow us to
recover values from,
e.g., string keys

C++ has a special type
for strings (std::string)

Notice the existence
of *member functions*.

E.g., x.**insert**(args)

C++ Maps

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

using std::map;    // better way to avoid using std::
using std::string; // don't pollute your code

int main ()
{
    // map<key type, value type> = initialize list
    // init the map with some key, values
    // We learned that we can use {} to init vars in C++
    map<string, double> mymap = {{"Vivian", 37},
                                {"Marcelo", 38},
                                {"Vania", 58},
                                {"Isabella", 25}};

    // Similar to Python, we can access some functions with .
    // These are member functions that modify/access elements
    // associated with a particular variable.
    mymap.insert({"Gilberto", 56});

    mymap["Beyonce"] = 41; // some types have operators

    // Operator [] is sometimes deemed unsafe for two reasons.
    // key doesn't exist: the code adds it to the map
    // key does exist: [] modifies it while insert doesn't
    fmt::print("(1) Rihanna is {} years old\n", mymap["Rihanna"]);

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

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

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps.cpp -o maps1 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps1
(1) Rihanna is 0 years old
(2) Rihanna is 0 years old
(3) Rihanna is 35 years old
```

Notice again that
maps (container) have
type as arguments.

map<int,int>

(key are integers) is
not the same as

map<string,int>

(key are C++ strings)

C++ Maps

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

using std::map;    // better way to avoid using std::
using std::string; // don't pollute your code

int main ()
{
    // map<key type, value type> = initialize list
    // init the map with some key, values
    // We learned that we can use {} to init vars in C++
    map<string, double> mymap = {{"Vivian", 37},
                                {"Marcelo", 38},
                                {"Vania", 58},
                                {"Isabella", 25}};

    // Similar to Python, we can access some functions with .
    // These are member functions that modify/access elements
    // associated with a particular variable.
    mymap.insert({"Gilberto", 56});

    mymap["Beyonce"] = 41; // some types have operators

    // Operator [] is sometimes deemed unsafe for two reasons.
    // key doesn't exist: the code adds it to the map
    // key does exist: [] modifies it while insert doesn't
    fmt::print("(1) Rihanna is {} years old\n", mymap["Rihanna"]);

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

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

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps.cpp -o maps1 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps1
(1) Rihanna is 0 years old
(2) Rihanna is 0 years old
(3) Rihanna is 35 years old
```

We can interact with maps (containers) via

1. *member functions*,
e.g., `x.insert(args)`
2. *Operators*,
e.g., operator `[]`
3. *Free functions* (we will see it later)

C++ Maps

C++ offers different ways to access / modify elems.

1. operator `[]`
2. mem func `.at()`
3. mem func `.insert()`
4. Iterators (~pointers)

Similar but not equivalent

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

using std::map;    // better way to avoid using std::
using std::string; // don't pollute your code

int main ()
{
    // map<key type, value type> = initialize list
    // init the map with some key,values
    // We learned that we can use {} to init vars in C++
    map<string, int> mymap = {{"Vivian", 37},
                             {"Marcelo", 38},
                             {"Vania", 58},
                             {"Isabella", 25}};

    // Operator [] is sometimes deemed unsafe for two reasons.
    // key doesn't exist: the code adds it to the map
    // key does exists: [] modifies it while insert doesn't

    // .at() throws an exception if key does not exists.
    // Sometimes you want that strictness in your code, where you
    // are adamant to control when you want to insert new keys
    //(with .insert()) vs when you want to access known keys
    fmt::print("Rihanna is {0} years old\n", mymap.at("Rihanna"));

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps2.cpp -o maps2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps2
terminate called after throwing an instance of 'std::out_of_range'
what(): map::at
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted
```

C++ Maps

Maps sort the keys

So, inserting new keys is expensive, but access is cheap

(opposite behavior: std::unordered_map)

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

using std::map; using std::string;

int main ()
{
    map<string, int> mymap = {{ "Vivian", 37},
                              {"Marcelo", 38},
                              {"Vania", 58},
                              {"Beyonce", 41},
                              {"Rihanna", 35}};

    // How to get the number of keys?
    fmt::print("Number of elements: {0}\n", mymap.size());

    // How to loop over map elements. May ways to do that
    for (auto it = mymap.begin(); it != mymap.end(); it++)
    { // iteration type is long (use auto!!!)
        fmt::print("Key: {0}, Val: {1}\n", it->first, it->second);
    }
    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps3.cpp -o maps3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps3
Number of elements: 5
Key: Beyonce, Val: 41
Key: Marcelo, Val: 38
Key: Rihanna, Val: 35
Key: Vania, Val: 58
Key: Vivian, Val: 37
```

C++ Maps

```
// How to get the number of keys?  
fmt::print("Number of elements: {0}\n", mymap.size());  
  
// How to loop over map elements. May ways to do that  
for (auto it = mymap.begin(); it != mymap.end(); it++)  
{ // iteration type is long (use auto!!!)  
    fmt::print("Key: {0}, Val: {1}\n", it->first, it->second);  
}  
return 0;  
}
```

In C, we learned that you can't use pointer arithmetic for non-contiguous containers (e.g., linked lists).

Remember what **it++** does if it is an array of int and it is an int pointer (it would jump in mem in mem by sizeof(int)).

Because **it++** is not allowed on linked lists, we had to use **while loops** (never **for loops**)

Clearly, **it** looks like a pointer, but it **can't be** a simple C pointer. C++ must somehow allow operators like ++ to do much more complex computations (yes – that is true!)

C++ Maps

This looks like a pointer. Almost, not exactly

```
// How to loop over map elements. May ways to do that
for (auto it = mymap.begin(); it != mymap.end(); it++)
{ // iteration type is long (use auto!!!)
    fmt::print("Key: {0}, Val: {1}\n", it->first, it->second);
}
return 0;
}
```

it even uses arrows as pointers

They are iterators. Iterators are “*like*” pointers with some advantages (biggest one is allowing ++ operator on non-contiguous containers). But they are more verbose

The keyword **auto** here is hiding the following

```
// How to loop over map elements. Many ways to do that
for (map<string,int>::iterator it = mymap.begin(); it != mymap.end(); it++) {
    // iteration type is long (use auto!!!)
    fmt::print("Key: {}, Val: {}\n", it->first, it->second);
}
```

C++ Maps

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

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

```
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) Structure binding again!
```

```
    { // Second Way (C++17)
```

```
        fmt::print("Key: {0}, Val: {1}\n", key, val);
```

```
    }
```

```
    return 0;
```

```
}
```

```
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
```

C++ is compatible
with Python style
loops

C++ provides many
solutions to each
problem!

C++ Maps

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

```
using std::map;
using std::string;
using std::for_each;
```

```
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 quite
powerful

We will discuss
them in detail later

C++ Maps

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

```
using std::map;
using std::string;
using std::tuple;
using std::get;
```

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

    for (auto [key, val] : mymap)
    {
        fmt::print("Key: {0}, Age: {1}, Surname: {2}\n",
                    key, get<int>(val), get<string>(val));
    }

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps6.cpp -o maps6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps6
Key: Rihanna, Age: 35, Surname: Fenty
Key: Vivian, Age: 37, Surname: Miranda
```

Maps can be combined with tuples to allow multiple values of different types

For multiple values of the same type, C++ offers [std::multimap](#)

C++ Maps

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

using std::map; using std::string;

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

    // Many ways to find if key exists
    auto x = "Beyonce";

    // C++20
    if (mymap.contains(x) == true) { // C++ bool
        fmt::print("key {0} exists\n", x);
    } else {
        fmt::print("key {0} does not exist\n", x);
    }

    if (mymap.count(x) > 0) {
        fmt::print("key {0} exists\n", x);
    } else {
        fmt::print("key {0} does not exist\n", x);
    }

    auto y = mymap.find(x);

    if (y != mymap.end()) {
        fmt::print("key {0} exists\n", x);
    } else {
        fmt::print("key {0} does not exist\n", x);
    }

    return 0;
}
```

```
whovian@phy504:~/host$ g++ maps7.cpp -o maps7 -std=c++20 -lfmt
whovian@phy504:~/host$ ./maps7
key Beyonce does not exist
key Beyonce does not exist
key Beyonce does not exist
```

How to find if a particular
key exists in C++ maps?

Again, many options!

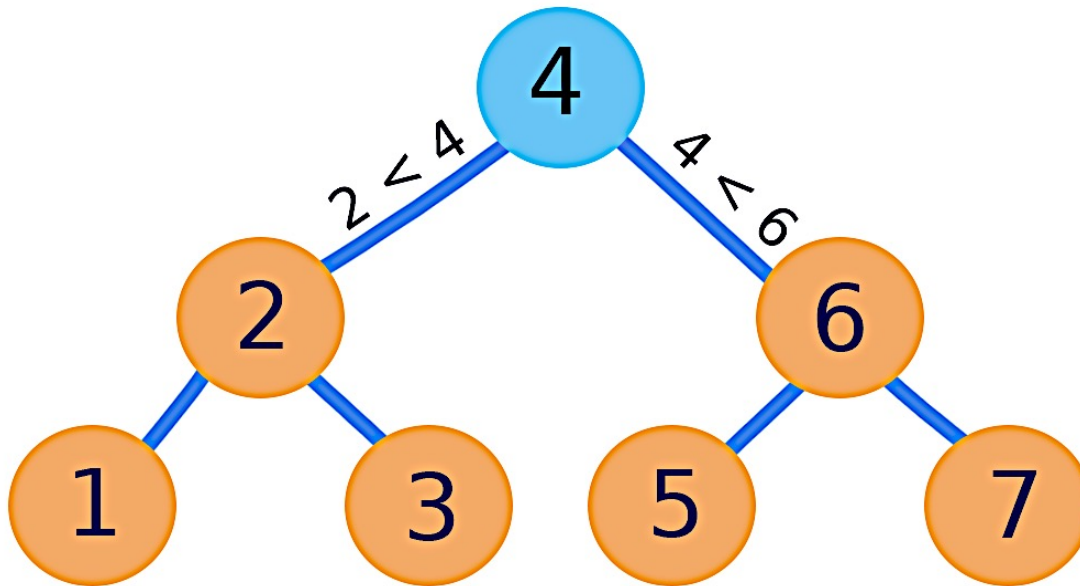
.count() returns only 0 or 1
(num of instances of a key)

.find() returns {key, value}
pair if it finds the key

C++ maps (how do they work)?

std::map: This container is implemented as a binary search tree → its elements are always sorted.

Why? What do we mean by sorted?



All elements to
the right (left) of
a node are
greater (smaller)
than the node

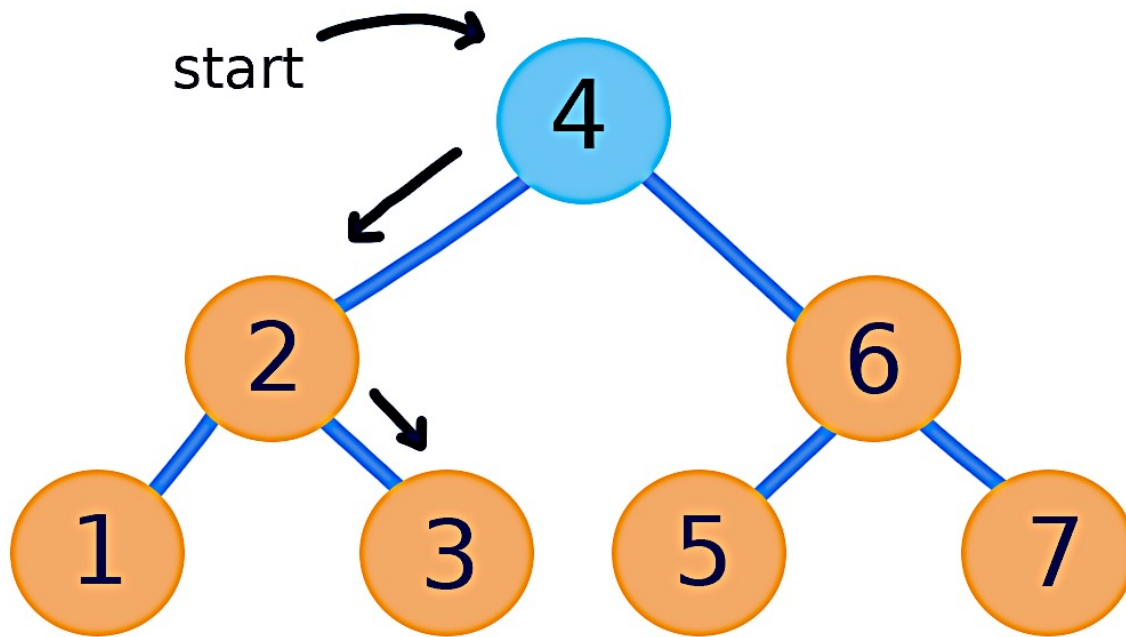
In Order Traversal: 1 2 3 4 5 6 7

C++ maps (how do they work)?

How do you search for an element in a BST?

$\sim O(\log(N))$ in a well-balanced tree

Search for 3



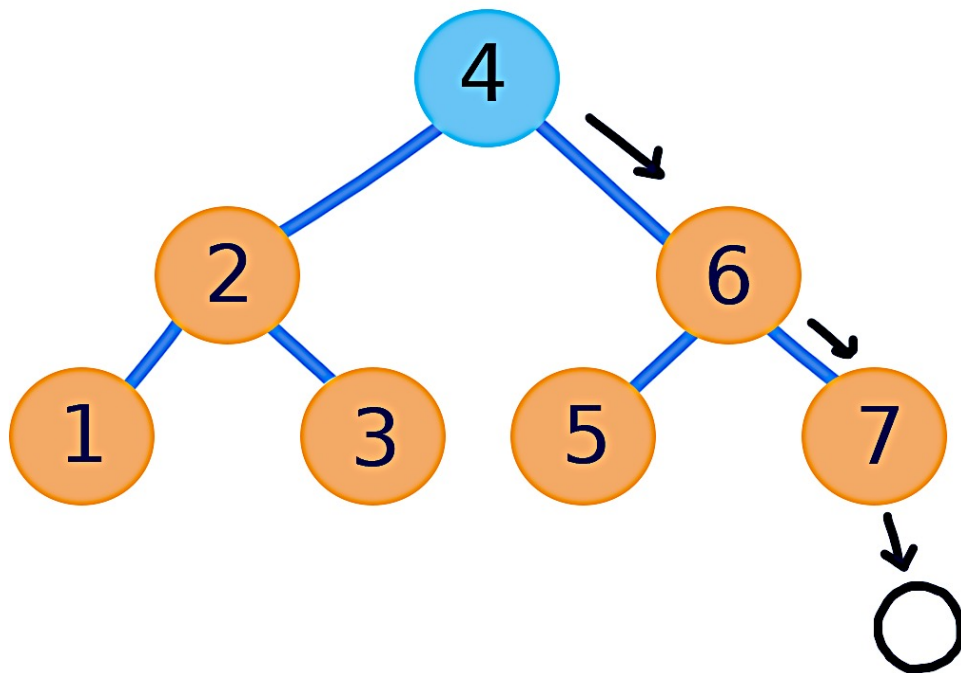
1. Begin at the tree's *root node*
2. If the value is smaller than the current node, move left
3. If the value is larger than the current node, move right

C++ maps (how do they work)?

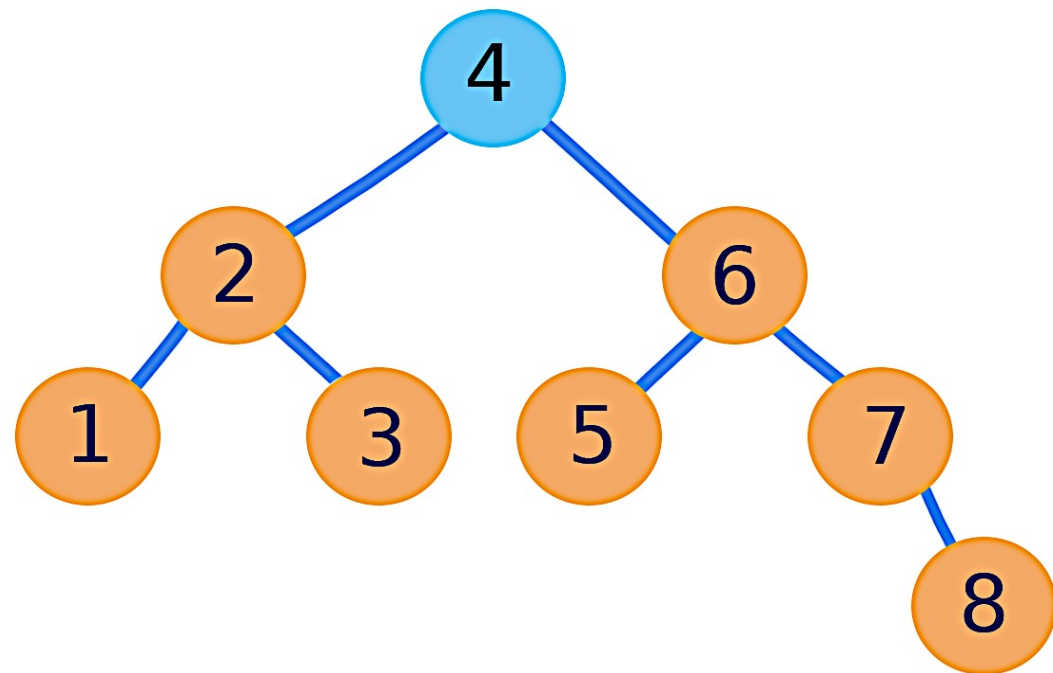
How do you insert new elements in a BST?

E.g., insert num 8 ($\sim O(\log(N))$ in a well-balanced tree)

1. find the new node's location



2. create the new node



C++ maps (how do they work)?

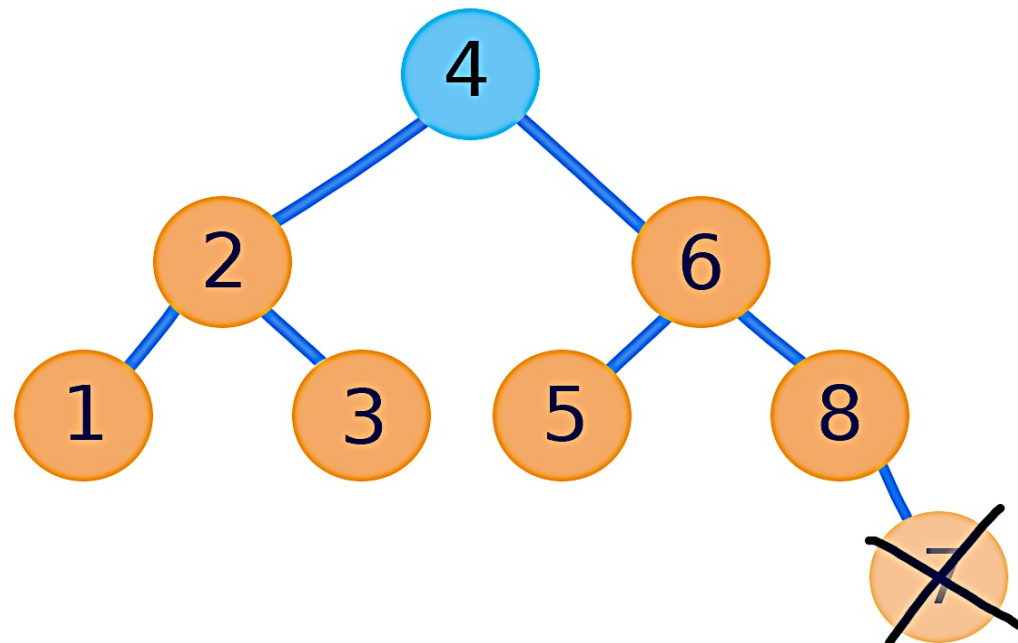
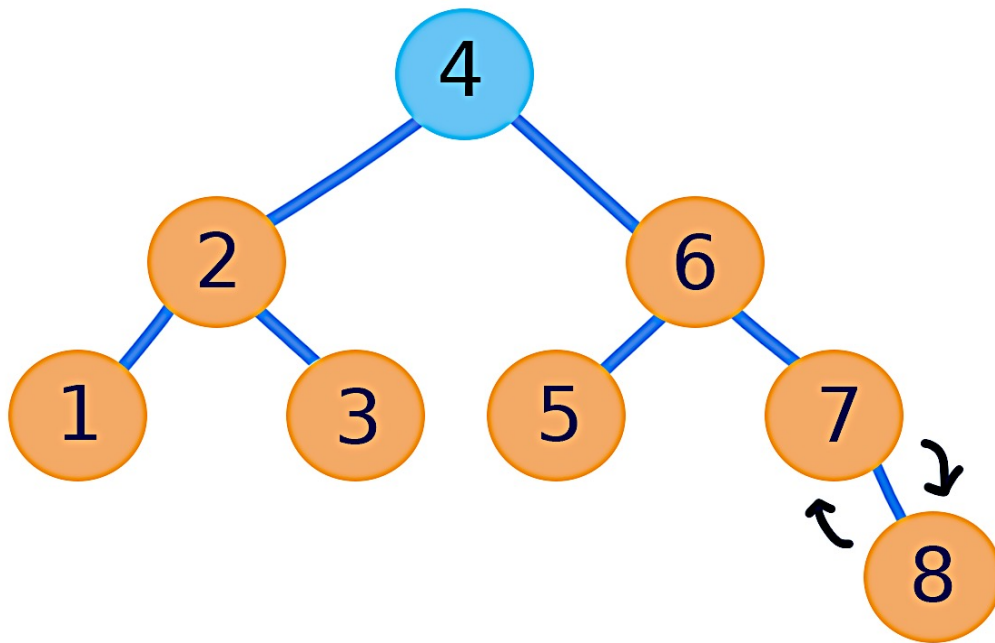
How do you remove a new element in a BST?

Depends on its location. Easier: one-child remove

Remove 7 (One-Child Remove)

1. swap the node and its child

2. remove the target node (now a leaf)



C++ maps (how do they work)?

How do you remove a new element in a BST?

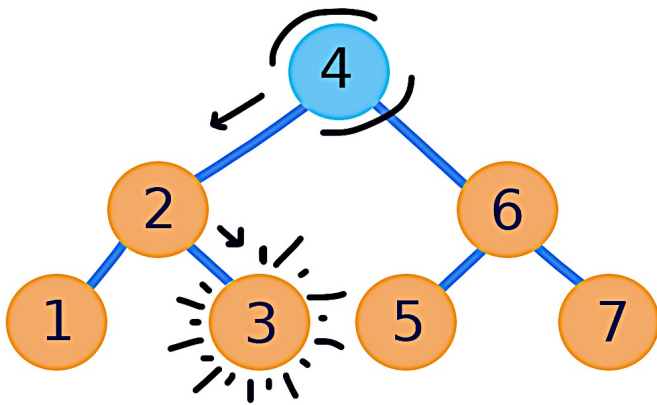
Depends on its location. Harder: two-child remove

Need to find IOP: **In-Order Predecessor**

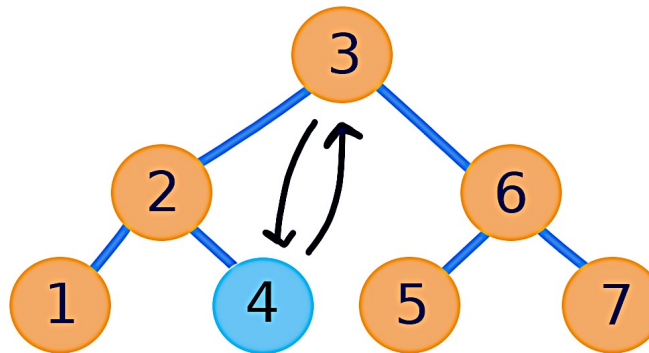
IOP = largest element on the left branch

Remove 4 (Two-Child Remove)

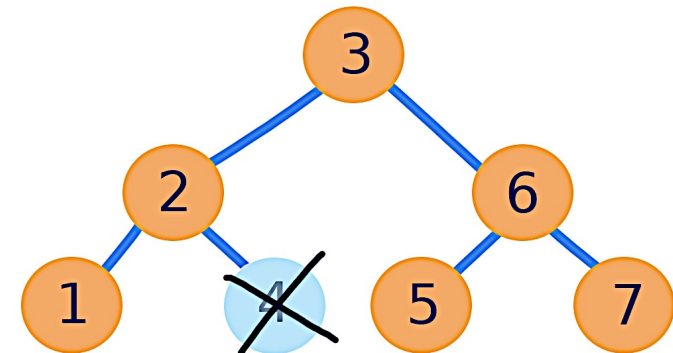
1. find IOP



2. swap IOP and target node



3. delete target, as it is now a leaf



Homework I

All functions should be defined inside a namespace with your name

- Generalize the following code to be able to handle the sum of three numbers. The final code should still be able to handle null pointer via overloading
- Add a custom print message on each function to make sure your test cases on the main function are selecting the correct corresponding overloading function

```
using fptr = int (*)(int, int);

int add(int x, int y) { // here we avoid if/else struct
    return x + y;
}

int add(int x, int y, fptr sum) {
    return sum(x, y);
}

int add(int x, int y, std::nullptr_t) {
    // nullptr_t = type of nullptr (C++ version of NULL)
    return x + y;
}

int sum(int x, int y) { return x + y;}

int main()
{
    fmt::print("Overloading: {0} \n", add(2, 3));
    fmt::print("Overloading: {0} \n", add(2, 3, &sum));
    fmt::print("Overloading: {0} \n", add(2, 3, nullptr));
}
```

Homework II (can be impl. in C)

A Binary Tree can be implemented using the following C **struct**.

We will soon implement it using a C++ class

```
typedef struct TreeNode {  
    int val;  
    struct TreeNode* left;  
    struct TreeNode* right;  
} TreeNode;
```

(1) Create a function that just allocate a new Node

```
TreeNode* newNode(int val)
```

(2) A valid Binary Search Tree (BST) is such that

*For every node, the value of all the nodes in its left subtree must be **less than** the node's value, and the value of all the nodes in its right subtree must be **greater than** the node's value.*

Using function recursion, create a function that validate a BST

```
bool isValid(TreeNode* node, long minVal, long maxVal)
```

Homework III (can be impl. in C)

- (1) Given an integer n , return the number of structurally unique BST with exactly n nodes of unique values from 1 to n .

Key Insight: To build BSTs with n nodes , we must:

1. Choose a root: Pick any value i from 1 to n to be the root
2. Left subtree: All numbers $< i \rightarrow$ total $i-1$ nodes.
3. Right subtree: All numbers $> i \rightarrow$ total $n-i$ nodes.
4. $n(\text{BSTs})$ with i as the root $= \text{numTrees}(i-1) \times \text{numTrees}(n-i)$

The numbers are known as Catalan Numbers.

Homework IV (can be impl. in C)

A balanced BST is a Binary Tree in which the number of left nodes is similar to the number of right nodes.

(1) To convert a sorted integer array into a height-balanced Binary Search Tree (BST).

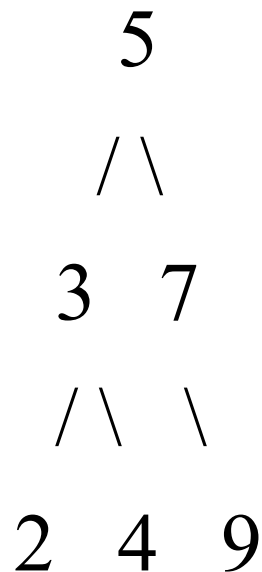
Tip: use a divide-and-conquer approach and function recursion

(2) Use the function you developed on HW II to test that the BST you created is a valid BST.

Homework V (can be impl. in C)

In-order traversal is one of the standard ways to visit all the nodes in a binary tree.

Example:



In-order traversal:

1. Go to the left subtree of 5 → at 3
2. Go to the left subtree of 3 → at 2 (no left child)
3. Visit 2
4. Back to 3, visit it
5. Go to the right subtree of 3 → visit 4
6. Back to 5, visit it
7. Go to the right subtree of 5 → at 7, visit it
8. Go to the right of 7 → visit 9

In-order output: 2 3 4 5 7 9

Using this concept & function recursion: write a C function that find the kth smallest element in a BST

Additional Reading: Logging

```
#include <fmt/core.h>
#include <spdlog/spdlog.h>
#include <tuple>
#include <string>

std::tuple<double, char, std::string> get_student(int id)
{
    switch (id)
    {
        case 0:
            return {3.8, 'A', "Lisa Simpson"};
    }
    std::string msg =
        fmt::format("id error on function {0}", __FUNCTION__);
    throw std::invalid_argument(msg);

    fmt::print("This won't run!!");
}

int main()
{
    spdlog::info("Testing spdlog library");
    spdlog::info("Hello, {}!", "World");

    try {
        const auto [gpa, grade, name] = get_student(3);
    }
    catch (std::exception& ex) {
        spdlog::critical(ex.what());
        exit(1);
    }
    catch (...) {
        spdlog::critical("Unknown exception");
        exit(1);
    }
    return 0;
}
```

Instead of **fmt::print** / **std::cout**, I use a professional logging library in C++ named **spdlog**

spdlog has an option to disable exceptions. This avoids the loophole shown on the last slide.

```
whovian@phy504:~/host$ g++ tuples8.cpp -o tuple8 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple8
[2023-03-23 22:53:59.008] [info] Testing spdlog library
[2023-03-23 22:53:59.015] [info] Hello, World!
[2023-03-23 22:53:59.018] [critical] id error on function get_student
```