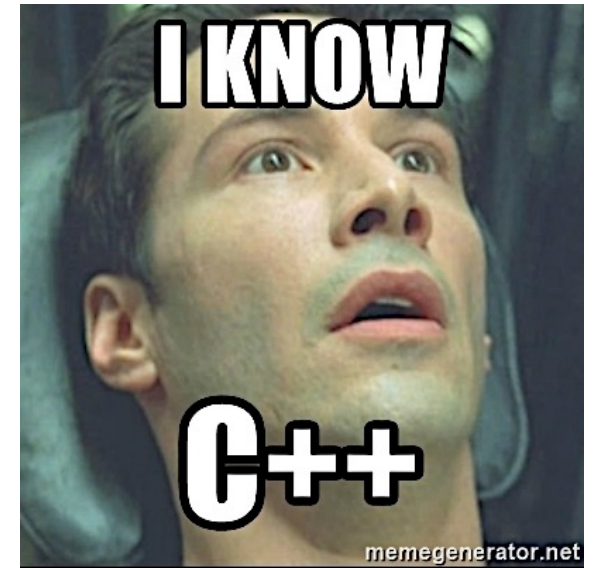
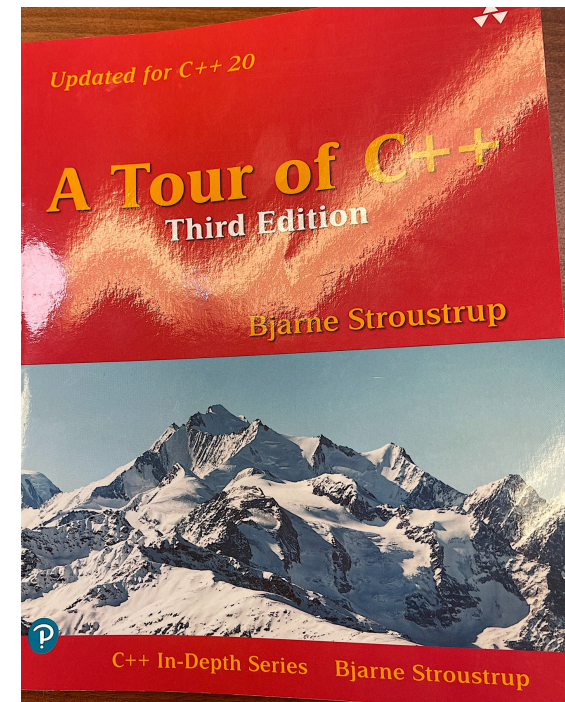


Lecture 22: C++ part VII



Suggested Literature



RECAP: 1-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 in **main()**

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

RECAP: 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;
}
```

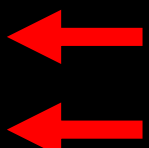
In C++, arguments can be passed by **l-value reference**

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
```



RECAP: C++ Lambdas

```
#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
```

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

More (C++14/20/26)

Advanced Lambda Functions

Lambda suffixes

We learned last class
that the keyword
mutable can be used to
allow variables captured
by value to be modified
inside the lambda

```
#include <fmt/core.h>
#include <fmt/ranges.h>
#include <list>
#include <map>
#include <ranges>
#include <vector>
using std::views::iota; using std::views::take;

int main() {
    double x = 10;

    auto pf = [&x]() { ++x; return x; };

    for (auto i : iota(10) | take(3) )
        fmt::println("x = {}", pf());
    fmt::println("x inside main = {}", x);

    double y = 0;
    auto pf2 = [y]() mutable {
        // now you can modify y, but the new values are
        // NOT accessible outside the lambda (to do that
        // you need to pass the capture by l-value reference
        for (auto i : iota(3) | take(3)) {
            y++;
            fmt::println("y = {}", y);
        }
        return y;
    };

    fmt::println("y inside pf2 = {}", pf2());
    fmt::println("y inside main = {}", y);
}
```

```
whovian@phy504:~/host/Lecture22$ g++ lambda9.cpp -o l9 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture22$ ./l9
```

```
x = 11
x = 12
x = 13
x inside main = 13
y = 1
y = 2
y = 3
y inside pf2 = 3
y inside main = 0
```

```

#include <fmt/core.h>
#include <cstdlib> // calls functions form C standard library
                  // wrap within std:: namespace

int main() {
    {
        // Generic programming in C involves void pointers
        auto pf = [](const void* x, const void* y) {
            return *(int*)x - *(int*)y;
        };
        // In this case, both will work. Why? Captureless lambda
        // implicit converts to C function pointer. So, why static?
        // Static gives you enforcement, at the definition of the lambda
        int arr[] = {5, 2, 8, 1, 9};
        std::qsort(arr, 5, sizeof(int), pf);
        for (int x : arr) fmt::println("{} ", x);
    }
    fmt::println("Example 2");
    {
        // here with static guaranteed conversion to C pointer
        auto pf = [](const void* x, const void* y) static {
            return *(int*)x - *(int*)y;
        };
        int arr[] = {5, 2, 8, 1, 9};
        std::qsort(arr, 5, sizeof(int), pf);
        for (int x : arr) fmt::println("{} ", x);
    }
}

```

Lambda suffixes

Other suffixes.

static can be used
when you promise
the compiler that
there are no
captures.

Can be converted to
a C function pointer

```

whovian@phy504:~/host/Lecture22$ /opt/gcc-latest/bin/g++ lambda10.cpp -o l10 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture22$ ./l10
1
2
5
8
9
Example 2
1
2
5
8
9

```

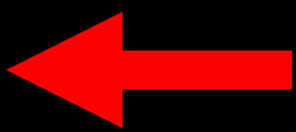
Lambda suffixes

Error on the first **pf** if you add a capture

```
whovian@phy504:~/host/Lecture22$ /opt/gcc-latest/bin/g++ lambda11.cpp -o l11 -std=c++26 -lfmt
lambda11.cpp: In function 'int main()':
lambda11.cpp:13:37: error: cannot convert 'main()::<lambda(const void*, const void*)>' to '___compar_fn_t' {aka 'int (*)(const void*, const void*)'}
   13 |         std::qsort(arr, 5, sizeof(int), pf);
      |                                     ^~
      |                                     |
      |                               main()::<lambda(const void*, const void*)>
In file included from /opt/gcc-latest/include/c++/16.0.1/cstdlib:83,
                 from /opt/gcc-latest/include/c++/16.0.1/stdlib.h:36,
                 from /usr/local/include/fmt/format.h:50,
                 from /usr/local/include/fmt/core.h:5,
                 from lambda11.cpp:1:
/usr/include/stdlib.h:999:34: note: initializing argument 4 of 'void qsort(void*, size_t, size_t, ___compar_fn_t)'
   999 |         ___compar_fn_t __compar) __nonnull ((1, 4));
      |         ~~~~~^~~~~~
```

Error on the second **pf** if you add a capture

```
whovian@phy504:~/host/Lecture22$ /opt/gcc-latest/bin/g++ lambda12.cpp -o l12 -std=c++26 -lfmt
lambda12.cpp: In function 'int main()':
lambda12.cpp:17:50: error: 'static' lambda specifier with lambda capture
   17 |         auto pf = [&x](const void* x, const void* y) static {
      |                                     ~~~~~
lambda12.cpp: In lambda function:
lambda12.cpp:17:32: error: lambda parameter 'x' previously declared as a capture
   17 |         auto pf = [&x](const void* x, const void* y) static {
      |                        ~~~~~^
lambda12.cpp: In function 'int main()':
lambda12.cpp:21:37: error: cannot convert 'main()::<lambda(const void*, const void*)>' to '___compar_fn_t' {aka 'int (*)(const void*, const void*)'}
   21 |         std::qsort(arr, 5, sizeof(int), pf);
      |                                     ^~
      |                                     |
      |                               main()::<lambda(const void*, const void*)>
In file included from /opt/gcc-latest/include/c++/16.0.1/cstdlib:83,
                 from /opt/gcc-latest/include/c++/16.0.1/stdlib.h:36,
                 from /usr/local/include/fmt/format.h:50,
                 from /usr/local/include/fmt/core.h:5,
                 from lambda12.cpp:1:
/usr/include/stdlib.h:999:34: note: initializing argument 4 of 'void qsort(void*, size_t, size_t, ___compar_fn_t)'
   999 |         ___compar_fn_t __compar) __nonnull ((1, 4));
      |         ~~~~~^~~~~~
```



Template Lambda

We learned that
functions can have
Types as arguments
(these are templates).

How to create a generic
lambda?

Part I (**C++14**)

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

int main() {
    // add suffix s which converts string literals to C++.
    // Because we are doing generic programming, we can't
    // rely on "automatic" casting like
    // std::string name = {"Hello"}. This only works because
    // you manually write std::string (and as we will see,
    // std::string has constructors that accept string literals.
    using namespace std::string_literals; // enables "hello"s

    // C++14
    // Generic lambda: auto accepts ANYTHING: can be bad
    auto pf = [](auto x, auto y) { return x + y; };

    fmt::println("int + int: {}", pf(1, 2));
    fmt::println("int + double: {}", pf(1, 2.5)); // casting ok.
    fmt::println("str + str: {}", pf("hi "s, "there"s));
    // interesting example. What will be the return type?
    // Actually it would fail as C++ does not know how to add
    // these types (complex with an int)
    //std::complex<double> x(1.0, 2.0);
    //fmt::println("int + complex: {}", pf(1, x));
}
```

```
whovian@phy504:~/host/Lecture22$ nano lambda13.cpp
whovian@phy504:~/host/Lecture22$ g++ lambda13.cpp -o l13 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture22$ ./l13
int + int: 3
int + double: 3.5
str + str: hi there
```

Template Lambda

We learned that
functions can have
Types as arguments
(these are templates).

How to create a
generic lambda?

Part II (**C++20**)

```
#include <vector>
#include <list>
#include <string>
#include <fmt/core.h>
#include <fmt/ranges.h>
using std::string;
int main() {
    using namespace std::string_literals;

    // Classical Template: typename keyword tells
    // the compiler that the argument within <> is
    // a type (template parameters can also be
    // integers like in STL arrays (see below)
    // std::array<double, 2> x = {1.,2.}
    // We enforce the types of both args to be the same
    auto pf = []<typename T>(T x, T y) { return x + y; };

    fmt::println("int + int: {}", pf(1,2));
    fmt::println("dbl + dbl: {}", pf(1.5,2.5));
    fmt::println("str + str: {}", pf("hi "s,"there"s));
    // In the examples above compiler was able to
    // figure out the types from the arguments (because
    // we use the string literals. Otherwise we would
    // have to write <std::string> explicitly which is
    // quite draconian (needs to be .operator()<string>
    fmt::println("str + str: {}",
                pf.operator()<string>("Hi","there"));
    // This is a limitation of calling any templated operator()
    // on an object, not just lambdas. Named functions like
    // f<string>("Hi", "there") work fine.
}
```

```
whovian@phy504:~/host/Lecture22$ g++ lambda14.cpp -o l14 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture22$ ./l14
int + int: 3
dbl + dbl: 4
str + str: hi there
str + str: Hithere
```

```

#include <vector>
#include <list>
#include <string>
#include <type_traits> // std::is_same_v
#include <fmt/ranges.h>
using std::vector; using std::string;
using std::is_same_v;
int main() {
    using namespace std::string_literals;

    // function that acts on types (printing their names!)
    // -> you manually specify the return type
    auto pf0 = []<typename T>() -> std::string {
        // C++ has if/else constructs but that test types!
        if constexpr (is_same_v<T,int>) return "int"s;
        if constexpr (is_same_v<T,double>) return "double"s;
        if constexpr (is_same_v<T,string>) return "string"s;
        return typeid(T).name(); // fallback on C++ typeid(T).name
                                // this function return type names
                                // but not human-readable
    };

    // argument by l-value reference: avoid copy
    // I am capturing pf0 by l-value reference
    auto pf = [&pf0]<typename T>(const vector<T>& v) {
        // again: a bit draconian the way to call pf0 but it works!
        fmt::println("vector<{}> = {}", pf0.operator()<T>(),
                    fmt::join(v, ", "));
    };

    pf(vector{1, 2, 3});
    pf(vector{1.1, 2.2, 3.3});
    pf(vector{"alpha"s, "beta"s});
}

```

```

whovian@phy504:~/host/Lecture22$ nano lambda15.cpp
whovian@phy504:~/host/Lecture22$ g++ lambda15.cpp -o l15 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture22$ ./l15
vector<int> = 1, 2, 3
vector<double> = 1.1, 2.2, 3.3
vector<string> = alpha, beta

```

Template Lambda

Classic template
shines when you want
to use a generic
container as an
argument in a lambda

Part II (C++20)

It takes time to master
templates

Strings

C++ Strings

C++ offers
more powerful
string
manipulation
compared to C
(via **STL** or
Boost Library)

```
#include <string>
#include <fmt/core.h>
// adapted from
// geeksforgeeks.org/c-string-class-and-its-applications/
using std::string; using fmt::print;

int main ()
{
    string str1("Vivian Miranda");    // init by string literal
    string str2 = {"Vivian Miranda"}; // init by string literal
    string str3{"Vivian Miranda"};    // init by string literal

    string str4(str1); // init by another string

    string str5(5, '#'); // init by char with number of occurrence

    // init by part of another string
    string str6(str1, 5, 5); // from 5th index (second parameter)
                           // 5 characters (third parameter)

    // init by part of another string : iterator version
    string str7(str1.begin(), str1.begin() + 5);

    print("{0}, {1}, {2}, {3} \n", str1, str2, str3, str4);
    print("{0}\n", str5);
    print("{0}\n", str6);
    print("{0}\n", str7);

    return 0;
}
```

```
whovian@phy504:~/host$ g++ strings.cpp -o string1 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string1
Vivian Miranda, Vivian Miranda, Vivian Miranda, Vivian Miranda
#####
n Mir
Vivia
```

C++ Strings

Diff. between **.at()** member function and **operator[]** is safety. The former checks if of bounds and throws an exception if not

```
#include <string>
#include <fmt/core.h>
// adaptd from
// geeksforgeeks.org/c-string-class-and-its-applications/
using std::string; using fmt::print;

int main ()
{
    string str1("Vivian Miranda");    // init by string literal

    string str2 = str1; // (assignment)

    const int len = str2.length();
    print("Length of str1 string is: {0}\n", len);

    print("str2: {0} \n", str2);
    str2.clear(); // deletes all character from string
    print("str2: {0} \n", str2);

    char ch1 = str1.at(2); // safer version as throws out_of_range
                          // exception if arg is out of range
    char ch2 = str1[3];
    print("ch1: {0}; ch2: {1} \n", ch1, ch2);

    char ch3 = str1.front(); // Same as str1[0]
    char ch4 = str1.back();  // Same as str1[str1.length()-1]
    print("ch3: {0}; ch4: {1} \n", ch3, ch4);

    return 0;
}
```

```
whovian@phy504:~/host$ g++ strings2.cpp -o string2 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string2
Length of str1 string is: 14
str2: Vivian Miranda
str2:
ch1: v; ch2: i
ch3: V; ch4: a
```

C++ Strings

Many string

STL member

functions

(**.something()**)

that alter the

content of a

string.

```
#include <string>
#include <fmt/core.h>
// adapted from
// geeksforgeeks.org/c-string-class-and-its-applications/
using std::string; using fmt::print;

int main () {
    string str1("Vivian Miranda"); // init by string literal
    string str2 = str1;
    print("str1: \"{0}\"\\n", str1);

    str2.erase(7, 4); // erase from 7th char, 4 chars
    print("str2: \"{0}\"\\n", str2);

    // erase iterator version
    string str3 = str1;
    str3.erase(str3.begin() + 3, str3.end() - 3);
    print("str3: \"{0}\"\\n", str3);

    string str4 = str1;
    str4.replace(0, 6, "ViV ");
    print("str4: \"{0}\"\\n", str4);

    return 0;
}
```

```
whovian@phy504:~/host$ g++ strings4.cpp -o string4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string4
str1: "Vivian Miranda"
str2: "Vivian nda"
str3: "Vivnda"
str4: "ViV   Miranda"
```

C++ Strings

```
#include <string>
#include <ctype.h>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
using std::string; using fmt::print;
using boost::to_upper; using boost::to_upper_copy;

int main ()
{
    string str1("Vivian Miranda"); // init by string literal
    print("str1: \"{0}\"\n", str1);

    string str2 = str1;
    for (auto& x : str2) // STL way to uppercase strings (ugly!)
    { // auto& = pass by reference (new concept!)
        x = toupper(x); // C function
    }
    print("str2: \"{0}\"\n", str2);

    // BOOST Library offers much better functionality
    string str3 = str1;
    boost::to_upper(str3);
    print("str3: \"{0}\"\n", str3);

    // via copy (not affecting the original string)
    string str4 = to_upper_copy(str1);
    print("str4: \"{0}\"\n", str4);

    // below, we need to type the template parameter explicitly
    // because it cannot be inferred from the string literal
    // string literal != std::string
    string str5 = to_upper_copy<string>("Vivian Miranda");
    print("str5: \"{0}\"\n", str4);

    return 0;
}
```

```
whovian@phy504:~/host/Lecture22$ g++ strings5.cpp -o s5 -lfmt
whovian@phy504:~/host/Lecture22$ ./s5
str1: "Vivian Miranda"
str2: "VIVIAN MIRANDA"
str3: "VIVIAN MIRANDA"
str4: "VIVIAN MIRANDA"
str5: "VIVIAN MIRANDA"
```

Boost Library is a bit more complete.

E.g., uppercase or lowercase the entire string instead of doing **char** by **char** (**STL**)

Issue w/ boost. Install is not trivial/portable

C++ Strings

Boost Library offers

good string
functionality

[boost/algorithm/string](#)

```
#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
// code adapted from
// geeksforgeeks.org/boosttrim-in-cpp-library/
// geeksforgeeks.org/c-boost-string-algorithms-library/
int main () {
    string s1 = "    VivianMirandaPhy504    ";
    string s2 = "    VivianMirandaPhy504    ";
    string s3 = "    VivianMirandaPhy504    ";
    trim_left(s1);
    trim_right(s2);
    trim(s3);

    print("trimmed left s1: |{0}|\n", s1);
    print("trimmed right s2: |{0}|\n", s2);
    print("trimmed both sides s2: |{0}|\n\n", s3);

    string s4 = "viv1viv2viv3viv4viv5viv6";
    print("first_copy: {0}\n", erase_first_copy(s4, "viv"));
    print("nth_copy: {0}\n", erase_nth_copy(s4, "viv", 3));
    print("last_copy: {0}\n", erase_last_copy(s4, "viv"));
    print("all_copy: {0}\n\n", erase_all_copy(s4, "viv"));

    string s5 = "Vivian Miranda";
    print("erase_head_copy: {0}\n", erase_head_copy(s5, 7));
    print("erase_tail_copy: {0}\n\n", erase_tail_copy(s5, 7));

    string s6 = "Vivian-Miranda-1986-37";
    print("first_copy: {0}\n", replace_first_copy(s6, "-", "_"));
    print("last_copy: {0}\n", replace_last_copy(s6, "-", "_"));
    print("all_copy: {0}\n", replace_all_copy(s6, "-", "_"));
}
```

```
trimmed left s1: |VivianMirandaPhy504    |
trimmed right s2: |    VivianMirandaPhy504|
trimmed both sides s2: |VivianMirandaPhy504|

first_copy: 1viv2viv3viv4viv5viv6
nth_copy: viv1viv2viv34viv5viv6
last_copy: viv1viv2viv3viv4viv56
all_copy: 123456

erase_head_copy: Miranda
erase_tail_copy: Vivian

first_copy: Vivian_Miranda-1986-37
last_copy: Vivian-Miranda-1986_37
all_copy: Vivian_Miranda_1986_37
```

Casting Strings into bool / Numbers

C++ Strings

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

```
using std::string; using fmt::print;
using boost::lexical_cast; using std::to_string;
```

```
int main ()
```

```
{
```

```
    string str1("1.23456"); // init by string literal
    double x1 = lexical_cast<double>(str1); // if fails, then the
                                           // code doesn't compile!
```

```
    double x2 = stod(str1);
    print("x1 = {0}; x2 = {1} \n", x1, x2);
```

```
    // We should avoid strings with multiple numbers (we will learn soon)
```

```
    string str2 = "1.23456 2.34567";
    std::string::size_type sz; // size_type is basically unsigned int
    double x3 = stod(str2, &sz); // second argument useful when the
                                // string contains multiple numbers
    double x4 = stod(str2.substr(sz)); // .substr(sz) = create substring
    print("x2 = {0}; x3 = {1} \n", x3, x4);
```

```
    string str3("3"); // init by string literal
    int x5 = lexical_cast<int>(str3);
    int x6 = stoi(str3);
    print("x5 = {0}; x6 = {1} \n", x5, x6);
```

```
    double num = 3.1415;
    string str4 = to_string(num); to_string is part of the STL
    string str5 = lexical_cast<string>(num); // if fails, then the
                                           // code doesn't compile!
    print("str4 = {0}; str5 = {1} \n", str4, str5);
```

```
    return 0;
```

```
}
```

```
whovian@phy504:~/host$ g++ strings6.cpp -o string6 -std=c++20 -lfmt
whovian@phy504:~/host$ ./string6
x1 = 1.23456; x2 = 1.23456
x2 = 1.23456; x3 = 2.34567
x5 = 3; x6 = 3
str4 = 3.141500; str5 = 3.14150000000000002
```

STL offers functions
to cast strings to
numbers and vice
versa.

[boost::lexical_cast](#)
because it provides a
[unified API](#)

The problem is that it
throws exceptions

C++ Strings

General conversion to **bool**
via string comparison.

Difference between
boost::iequals and STL
string.compare (or **==**)?

boost::iequals is case
insensitive!

```
#include <string>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
using boost::lexical_cast; using boost::iequals;
C++ has a proper bool type
bool convert_string (const std::string s) {
    if (iequals(s, string("T")) ||
        iequals(s, string("True")) ||
        iequals(s, string("Yes")))
    {
        return true;
    }
    else if (iequals(s, std::string("F")) ||
             iequals(s, std::string("False")) ||
             iequals(s, std::string("No")))
    {
        return false;
    }
    else
    {
        const double tmp = lexical_cast<double>(s);
        return ((tmp < 0) == false) ? true : false;
    }
}

int main()
{
    bool x1 = convert_string("True");
    bool x2 = convert_string("truE");
    bool y1 = convert_string("False");
    bool y2 = convert_string("FaLSe");
    print("{0}, {1}, {2}, {3}\n", x1, x2, y1, y2);
}
```

```
whovian@phy504:~/host/CPP$ g++ strings9.cpp -o string9 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string9
true, true, false, false
```

C++ Strings

Modern C++ standard
provides new ways to cast
strings to numbers (and also
includes **std::expected** to
avoid Exceptions !

```
#include <expected>      // C++23 - std::expected, std::unexpected
#include <print>          // C++23 - std::print, std::println
#include <string>          // C++98 - std::string
#include <string_view>    // C++17 - std::string_view
#include <charconv>       // C++17 - std::from_chars, std::to_chars
#include <system_error>   // C++11 - std::errc (error codes)
#include <utility>        // C++11 - std::forward
#include <boost/lexical_cast.hpp> // Boost - header-only

std::expected<int, std::string>
parse(std::string_view s) {
    if (s.empty())
        return std::unexpected("empty string");

    int value;
    auto [ptr, ec] = std::from_chars(s.data(),
                                     s.data() + s.size(),
                                     value);

    if (ec == std::errc::invalid_argument)
        return std::unexpected("not a valid integer");
    if (ec == std::errc::result_out_of_range)
        return std::unexpected("number out of range for int");
    if (ptr != s.data() + s.size())
        return std::unexpected("trailing characters after number");
    return value;
}

std::expected<int, std::string>
parse_boost(std::string_view s) {
    try {
        return boost::lexical_cast<int>(s);
    } catch (const boost::bad_lexical_cast& e) {
        return std::unexpected(e.what());
    }
}

int main() {
    std::string_view input = "123";

    if (auto r = parse(input); r)
        std::println("parsed value: {}", *r);
    else
        std::println(stderr, "error: {}", r.error());

    if (auto r = parse_boost(input); r)
        std::println("parsed value: {}", *r);
    else
        std::println(stderr, "error: {}", r.error());
    return 0;
}
```

```
whovian@phy504:~/host/Lecture22$ nano string15.cpp
whovian@phy504:~/host/Lecture22$ g++ string15.cpp -o s15 -std=c++26
whovian@phy504:~/host/Lecture22$ ./s15
parsed value: 123
parsed value: 123
```

C++ Strings

Modern C++ standard
provides new ways to cast
strings to numbers (and also
includes **std::expected** to
avoid Exceptions !

```
#include <expected>      // C++23 - std::expected, std::unexpected
#include <print>          // C++23 - std::print, std::println
#include <string>         // C++98 - std::string
#include <string_view>    // C++17 - std::string_view
#include <charconv>       // C++17 - std::from_chars, std::to_chars
#include <system_error>   // C++11 - std::errc (error codes)
#include <utility>        // C++11 - std::forward
#include <boost/lexical_cast.hpp> // Boost - header-only

std::expected<int, std::string>
parse(std::string_view s) {
    if (s.empty())
        return std::unexpected("empty string");

    int value;
    auto [ptr, ec] = std::from_chars(s.data(),
                                     s.data() + s.size(),
                                     value);

    if (ec == std::errc::invalid_argument)
        return std::unexpected("not a valid integer");
    if (ec == std::errc::result_out_of_range)
        return std::unexpected("number out of range for int");
    if (ptr != s.data() + s.size())
        return std::unexpected("trailing characters after number");
    return value;
}

std::expected<int, std::string>
parse_boost(std::string_view s) {
    try {
        return boost::lexical_cast<int>(s);
    } catch (const boost::bad_lexical_cast& e) {
        return std::unexpected(e.what());
    }
}

int main() {
    std::string_view input = "Vivian";

    if (auto r = parse(input); r)
        std::println("parsed value: {}", *r);
    else
        std::println(stderr, "error: {}", r.error());

    if (auto r = parse_boost(input); r)
        std::println("parsed value: {}", *r);
    else
        std::println(stderr, "error: {}", r.error());
    return 0;
}
```

```
whovian@phy504:~/host/Lecture22$ nano string16.cpp
whovian@phy504:~/host/Lecture22$ g++ string16.cpp -o s16 -std=c++26
whovian@phy504:~/host/Lecture22$ ./s16
error: not a valid integer
error: bad lexical cast: source type value could not be interpreted as target
```

Split lines into words
(Still Strings)

C++ Strings

Boost Library offers an easy interface to split text into lines and lines into words

[boost/algorithm/string](#)

```
#include <string>
#include <vector>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
using std::vector; using boost::split;

int main() {
    std::string content =
        "First line Content\n"
        "Second line Content\n"
        "Third Line Content";

    vector<string> lines;
    boost::split( lines, content,
                  boost::is_any_of("\n"),
                  boost::token_compress_on
                );
    for (auto x : lines) {
        print("{0}\n", x);
    }
    print("\n");

    for(auto line : lines) {
        vector<string> words;

        boost::split( words, line,
                      boost::is_any_of(" \t"),
                      boost::token_compress_on
                    );
        for (auto x : words) {
            print("{0}\n", x);
        }
        print("\n");
    }
}
```

```
whovian@phy504:~/host/CPP$ g++ strings10.cpp -o string10 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string10
First line Content
Second line Content
Third Line Content

First
line
Content

Second
line
Content

Third
Line
Content
```

C++ Strings

Boost Library offers good string functionality

[boost/algorithm/string](#)

```
#include <string>
#include <vector>
#include <fmt/core.h>
#include <boost/algorithm/string.hpp>
#include <boost/lexical_cast.hpp>
using std::string; using fmt::print;
using namespace boost::algorithm;
using std::vector; using boost::split;
using boost::starts_with; using boost::is_any_of;

int main()
{
    string content = "#First line Content";
    vector<string> words;
    split( words,
           content,
           is_any_of(" \t"),
           boost::token_compress_on
           );

    if(starts_with(words[0], "#"))
    {
        print("This is a comment line\n");
    }

    // DEAL WITH key=value (no space) line
    vector<string> words2 = {"key=value"};

    auto pos = words2.at(0).find("=");
    if (pos != std::string::npos)
    {
        string key = words2.at(0);
        words2.resize(0); // zero size (reset vector)
        split( words2, key,
               boost::is_any_of("="),
               boost::token_compress_on
               );
        for (auto x : words2) {
            print("{0}\n", x);
        }
    }
}
```

```
whovian@phy504:~/host/CPP$ g++ strings11.cpp -o string11 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string11
This is a comment line
key
value
```

We will be able to check if the line is a comment (starts with **#**), and to separate **key=value** lines (no spaces)

REGULAR EXPRESSIONS (Still Strings)

C++ Strings

Boost Library and **STL**
offer support for regular
expressions.

```
#include <string>
#include <fmt/core.h>
#include <boost/regex.hpp>
#include <boost/algorithm/string/regex.hpp>
using std::string; using fmt::print;
using boost::regex; using boost::regex_match;

int main()
{
    regex re_has_equal("^([A-Za-z]+)?[=]+([A-Za-z]+)?$");

    string s1 = "viv=";
    if (regex_match(s1, re_has_equal)) {
        print("s1 matches re_has_equal\n");
    }
    string s2 = "viv=viv";
    if (regex_match(s2, re_has_equal)) {
        print("s2 matches re_has_equal\n");
    }
    string s3 = "=viv";
    if (regex_match(s3, re_has_equal)) {
        print("s3 matches re_has_equal\n");
    }
    string s4 = "==viv";
    if (regex_match(s4, re_has_equal)) {
        print("s4 matches re_has_equal\n");
    }
    string s5 = "ViV=viv";
    if (regex_match(s5, re_has_equal)) {
        print("s5 matches re_has_equal\n");
    }
    string s6 = "ViVviv";
    if (!regex_match(s6, re_has_equal)) {
        print("s6 doesn't match re_has_equal\n");
    }
}
```

```
whovian@phy504:~/host/CPP$ g++ strings14.cpp -o string14 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string14
s1 matches re_has_equal
s2 matches re_has_equal
s3 matches re_has_equal
s4 matches re_has_equal
s5 matches re_has_equal
s6 doesn't match re_has_equal
```

std::regex_match
(warning: a few regex
commands may require
double **** - example **\\d**
for digits)

C++ Strings

Boost Library and **STL** offer support for regular expressions

need double ****

```
#include <string>
#include <fmt/core.h>
#include <boost/regex.hpp>
#include <boost/algorithm/string/regex.hpp>
using std::string; using fmt::print;
using boost::regex; using boost::regex_match;

int main()
{
    // doesn't match scientific notation ex: 1e+20
    regex re_is_float("[+]?\d+.*\d*");

    string s1 = "7.";
    if (regex_match(s1, re_is_float)) {
        print("s1 matches re_is_float\n");
    }
    string s2 = "17.32";
    if (regex_match(s2, re_is_float)) {
        print("s2 matches re_is_float\n");
    }

    regex re_is_number("[+]?\d+.*\d*[eE]?[+-]?\d*");
    string s3 = "17.32";
    if (regex_match(s3, re_is_number)) {
        print("s3 matches re_is_number\n");
    }
    string s4 = "1.32e+20";
    if (regex_match(s4, re_is_number)) {
        print("s4 matches re_is_number\n");
    }
}
```

```
whovian@phy504:~/host/CPP$ nano strings13.cpp
whovian@phy504:~/host/CPP$ g++ strings13.cpp -o string13 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./string13
s1 matches re_is_float
s2 matches re_is_float
s3 matches re_is_number
s4 matches re_is_number
```

need double ****
because **\\d \\t \\n** has a special meaning

IO

C++ IO

Old C++ adopts
streams for both
input and output

C++23 add the
command **print()** that
has Python-style API
for formatting

```
#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip> // setw, setfill, setprecision
#include <fmt/printf.h>
using std::cout; using std::endl;
using fmt::print; using std::setfill;
using std::setprecision; using std::setw;

int main ()
{ // Before C++23, IO was dealt with streams
  const double x = 7.1234567890;
  const int y = 730;

  cout << "x = " << setprecision(6) << x << endl;
  cout << "y = " << setw(5) << y << endl;
  cout << "y = " << setfill('0') << setw(5) << y << endl;
  cout << "y = " << setfill('*') << setw(5) << y << endl;
  cout << endl;

  // fmt offers a preview of C++23
  print("x = {:.5f}\n", x); // = setprecision(6)
  print("y = {:5d}\n", y);
  print("y = {:05d}\n", y);
  // > is an alignment specifier
  // fmt.dev/latest/syntax.html#format-specification-mini-language
  print("y = {:*>5d}\n\n", y);

  // fmt offers a type safer version of printf
  // If format fails, it throws fmt::v9::format_error
  fmt::printf("x = %.5f\n", x);
  fmt::printf("y = %5d\n", y);
}
```

```
whovian@phy504:~/host/CPP$ g++ io1.c -o io1 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io1
x = 7.12346
y = 730
y = 00730
y = **730

x = 7.12346
y = 730
y = 00730
y = **730

x = 7.12346
y = 730
```

C++ IO

Printing **bool** with **streams** requires an extra flag; otherwise **streams** print an integer

Alignment specifier is a cool feature of **fmt::print()**

```
#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>

using std::cout; using std::endl;
using fmt::print; using std::setfill;
using std::boolalpha;

int main ()
{
    const bool a = false;
    const bool b = true;
    cout << "a = " << a << " b = " << b << endl;
    cout << boolalpha;
    cout << "a = " << a << " b = " << b << endl;

    print("a = {}, b = {}\n", a, b);

    // print alignment
    const int y = 730;
    print("{:*>20}\n", y); // left
    print("{:*^20}\n", y); // center
    print("{:*<20}\n", y); // right
}
```

```
whovian@phy504:~/host/CPP$ g++ io2.c -o io2 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io2
a = 0 b = 1
a = false b = true
a = false, b = true
*****730
*****730*****
730*****
```

```

#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>
using fmt::print;
using namespace std;

int main ()
{
    const int y = 125;
    cout << left << "|" << setfill('*') <<
        setw(10) << y << "|" << endl;
    cout << right << "|" << setfill('*') <<
        setw(10) << y << "|" << endl;
    // no center position!
    cout << endl;

    const double x = 300545900;
    cout << "x = " << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        defaultfloat << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        scientific << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << x << endl;
    cout << endl;

    // print offers more compact notation
    print("x = {:*>21.10g}\n", x);
    print("x = {:*>21.10e}\n", x);
    print("x = {:*>21.10f}\n", x);
    return 0;
}

```

C++ IO

streams allow the control of alignment (but not centering)

Scientific notation in **streams**
vs. **fmt::print()**

Print has more compact notation than streams.

```

whovian@phy504:~/host/CPP$ g++ io3.c -o io3 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io3
|125*****|
|*****125|

x = *****300545900
x = *****3.0054590000e+08
x = *300545900.0000000000

x = *****300545900
x = *****3.0054590000e+08
x = *300545900.0000000000

```

C++ IO

```
#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>
using fmt::print;
using namespace std;

int main ()
{
    const double x = 3125;
    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        noshowpoint << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        showpoint << x << endl;
    cout << endl;

    const double y = 1.2;
    cout << "y = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        noshowpoint << y << endl;

    cout << "y = " << right << setfill('*') <<
        setw(15) << showpoint << y << endl;
    cout << endl;

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io4.c -o io4 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io4
x = *****3125
x = *****3125.000
y = *****1.2
y = *****1.200000
```

streams additional
manipulators:

std::showpoint /
std::noshowpoint

std::noshowpoint hides
the dot on integers (3125
vs 3125.000)

C++ IO

```
#include <string>
#include <iostream>
#include <fmt/core.h>
#include <iomanip>
using fmt::print;
using namespace std;

int main ()
{
    const double x = 3125;
    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        scientific << uppercase << x << endl;

    cout << "x = " << right << setfill('*') <<
        setw(15) << setprecision(7) <<
        scientific << nouppercase << x << endl;
    cout << endl;

    const int y = 37;
    cout << "y = " << right << setfill('*') <<
        setw(15) << fixed << showbase <<
        dec << y << endl;

    cout << "y = " << right << setfill('*') <<
        setw(15) << fixed << showbase <<
        hex << y << endl;
    cout << endl;

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io5.c -o io5 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io5
x = **3.1250000E+03
x = **3.1250000e+03

y = *****37
y = *****0x25
```

streams additional
manipulators:

std::uppercase /

std::nouppercase;

std::showbase /

std::noshowbase and

std::hex / std::dec

C++ IO

```
#include <fmt/core.h>
#include <vector>
#include <fmt/ranges.h> // fmt::join
#include <fmt/color.h>
using fmt::print; using fmt::join;
using std::vector; using fmt::styled;

int main () {
    vector<double> v = {1.0, 2.0, 3.0, 4.0, 5.0};

    // FMT offers a more concise format in color
    print("Elapsed time: {0:.2f} seconds",
          fmt::styled(1.23, fmt::fg(fmt::color::red) |
                      fmt::bg(fmt::color::blue)));
    print("\n");

    print("{} ",
          fmt::styled(join(v, ", "),
                      fmt::fg(fmt::color::red) |
                      fmt::bg(fmt::color::blue)));
    print("\n");
}
```

How to print the
elements of a
container (vector,
list, tuple)?

FMT library
allows you to print
vectors in colors

```
whovian@whovianphy504:~/host/Lecture21$ g++ io15.c -o io15 -std=c++23 -lfmt
whovian@whovianphy504:~/host/Lecture21$ ./io15
Elapsed time: 1.23 seconds
1, 2, 3, 4, 5
```

C++ IO

How to print
elements of a
container?

Big difference if
(1) memory is
(non) contiguous
or (2) the container
is iterable

```
#include <iostream>
#include <fmt/core.h>
#include <fmt/ranges.h>
#include <tuple>
using fmt::print; using std::string;
using fmt::join; using namespace std;

int main ()
{
    tuple<double,string,int> v1 = {2.0, "Vivian", 1000};

    // tuple is noniterable - complicated to print all
    // elements using C++ streams (requires an advanced
    // feature named variadic generic lambdas/templates)

    // We won't cover variadics in this lecture

    // FMT offers a more concise/easier format (C++23)
    print("{} ", fmt::join(v1, ", "));
    print("\n");

    // C++17 - use initialization to guess template parameters
    tuple v2 = {10.0, string("VivianMiranda"), 2000};

    print("{} ", fmt::join(v2, ", "));
    print("\n");
}
```

```
whovian@phy504:~/host/CPP$ g++ io8.c -o io8 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io8
2, Vivian, 1000
10, VivianMiranda, 2000
```

Homework I

- Given `std::string s = "Hello, World!"`, write three lambdas: `to_upper`, `to_lower`, and `to_title` (capitalize the first letter of each word, lowercase the rest). Print the result using both `std::cout` and `fmt::println` to practice both styles.

Hints:

- The C standard library provides `std::toupper(int)` and `std::tolower(int)` in `<cctype>`. These work on single characters and return an int.
- `std::ranges::transform` applies a function to every element of a range and writes the results to a destination.
- For `to_title`, you'll need a bool flag to track whether the next character starts a new word. Capture it with l-value reference. A space means the next letter should be uppercase.
- Each lambda should take a `std::string` by value (not reference).

Homework II

- (1) Using `std::ranges::copy`, `std::ranges::views::transform` and `std::back_inserter`, write a C++ program that converts `std::string original = "Vivian Miranda"` to **VIVIAN MIRANDA**.
- (2) Now answer (1) using STL algorithm transform but not STL ranges!
- (3) Using C++ 23 `std::views::chunk_by`, write a C++23 program that groups adjacent chars in the following vector.

```
std::vector<char> chars = {'a', 'a', 'b', 'b', 'b', 'a', 'a', 'c'};
```

```
whovian@whovianphy504:~/host/Lecture20/HW$ ./string2
Grouped characters:
aa
bbb
aa
c
```

- (4) Using `std::views::reverse` and `std::ranges::copy` and `std::back_inserter`, reverse the string (`std::string original = "Vivian Miranda"`). Now use `std::reverse` but not ranges!

Homework III

Given `std::string s = "PHY504 C++ course is awesome"`;, write two lambdas that count the number of words in a string:

- Write a lambda that counts the number of space characters using `std::ranges::count(str, ' ')` and adds 1. Handle the edge case of an empty string (which has 0 words, not 1).
- Write a lambda that uses `std::istringstream` and the `>>` operator in a loop to extract and count words. The `>>` operator automatically skips whitespace. You'll **need** `#include <sstream>`.

Hints:

- Both lambdas should take a `const` l-value reference to avoid a copy.
- Approach A is simpler but fragile. Think about what happens with double spaces, leading spaces, or trailing spaces.

Homework IV

Given `vector<string> v1{"PHY504", "", "C++", "", "course", "is", "awesome"}`, use `std::views::filter` with a lambda to remove the empty strings, then `std::views::transform` with a lambda to append "!" to each remaining word. Materialize the result into a new `vector<string>` using `std::ranges::to`.

Hints:

- `std::views::filter` takes a lambda that returns true for elements you want to keep. An empty string has `.empty() == true`.
- `std::views::transform` takes a lambda that modifies each element. To append, remember the `+` operator on `std::string` concatenates.
- Both views are lazy, so nothing happens until you materialize with `std::ranges::to` or `iterate`.
- Pipe them together
- Use `using namespace std::string_literals`; if you want the `"!"s` suffix

Homework V

Given `std::string s = "PHY504 C++ course has 6 students"`, write four lambdas that count: (1) non-space characters, (2) vowels, (3) consonants, and (4) `digit_count`. Each lambda should take a string by l-value reference and return the count. Print a report using `fmt::println` with aligned columns. Look up the `<` and `>` alignment specifiers in `fmt`: for example `{:<15}` left-aligns in a 15-character field.

Hints:

- Use `std::ranges::count_if` for each counter. It walks through the range and counts how many elements make the predicate return true.
- The `<cctype>` header provides useful character classification functions like `std::isalpha`, `std::isdigit`, `std::tolower`. Think about which ones you need for each counter.
- Be careful with vowels vs consonants: a consonant is not simply "everything that isn't a vowel." What about digits, spaces, and punctuation?

Homework VI

Given `std::vector<std::string> names{"Vivian", "Vania", "Gilberto", "Isabella"}`, sort the vector three different ways using lambdas as comparators:

1. Alphabetically, case-insensitive (so "Gilberto" comes before "Isabella").
2. By length, shortest first (so "Vania" comes before "Gilberto").
3. By number of vowels, most vowels first (so "Isabella" comes before "Vania").

For each sort, print the result with `fmt::join`.

Hints:

- Use `std::ranges::sort`.
- For case-insensitive sorting, you need to compare lowercase versions of both strings
- For sorting by vowel count, write a separate lambda that counts vowels (from previous HW).
- Important: `std::ranges::sort` modifies the vector in place. If you want to show all three sorts, make a copy of the original vector before each sort: `auto v = names;`
- Think about tie-breaking: when two strings have the same length or same vowel count, `std::ranges::sort` is free to put them in any order.

Homework VII

- (1) C++ offers **string_view**, allowing users to get a reference (without copy) of a substring of long strings. Complete this code (replace XXX) and then run it and check the difference in runtime

```
#include <string>
#include <string_view>
#include <chrono>
#include <fmt/core.h>

int main() {
    const std::string text = "This is a large and expensive string used to"
                             " demonstrate string_view performance."
                             "and yes you can concatenate string literals";
    constexpr int iterations = 10'000'000; volatile size_t total = 0; // prevent optimization

    auto start_string = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < iterations; ++i) {
        // Q1: use std::string .substr() func to make a copy of a substring with 10 chars
        std::string part = XXX; total += part.size();
    }
    auto end_string = std::chrono::high_resolution_clock::now();
    // Q2 WE WANT TIME IN MILLISECONDS
    auto duration_string = std::chrono::duration_cast<XXX>(end_string - start_string).count();

    auto start_view = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < iterations; ++i) {
        std::string_view part = std::string_view(text).substr(10, 20); total += part.size();
    }
    auto end_view = std::chrono::high_resolution_clock::now();
    auto duration_view = std::chrono::duration_cast<XXX>(end_view - start_view).count();

    fmt::print("copy vs std::string_view time: {0} vs {1}ms \n", duration_string, duration_view);
    fmt::print("Total: {0} \n", total);
}
```

HW VIII

An input file is a file where users change the options of the code

```
key1 = 1  
key2 = -2.0  
key3 = 3  
key4 = 4.0
```

(1) With **boost::iequals** and **boost::lexical_cast** write the template func.

template<typename T> T convert_string (std::string s)

that converts all these options from string to double or int. Research a bit (using AI, a book, or Google) about template functions. The variable should be defined as **T result**

bool convert_string (std::string s)

(2) If **boost::lexical_cast** fails, it throws **boost::bad_lexical_cast** exception. Catch this exception and exit the program with the message “Game Over” when this exception is thrown.

HW IX

An input file is a file where users change the options of the code

```
key1 = True  
key2 = False // Caps should not matter  
key3 = T  
key4 = F  
key6 = 10 // True (>= 0)  
key7 = -2 // False (<0)
```

Using **boost::iequals** and **boost::lexical_cast** write a function that converts all these options about how to write a bool into a C++ bool type

```
template<> bool convert_string (std::string s)
```

This is known as template specialization

Homework X (*not going to be on exam*)

We learned how to read a multiline string and split it into words

- (1) Read the file Hamlet we used in Bash using C++ `std::ifstream`. This can be a formatted read.
- (2) The `std::ifstream` has a few nice member functions `.seekg()`, `.tellg()`, and `.read()` that, together with the flags `std::ios::beg` and `std::ios::end`, allow you to read the entire file into a single string. Do that!
- (3) Using the Boost string library, break the string into lines and words. All words must be saved in the vector `std::vector<std::string> allWords`
- (4) Create the following vector to store unique words and their counts.

`std::vector<std::pair<std::string,int>> uniqueWords`

In the pair `<string, int>`, the integer will save the frequency of each word. Write a loop that populates the STL vector `uniqueWords`.

- (5) Using an STL algorithm, sort the unique words in ascending order of frequency. That means more frequent words will appear last.
- (6) Count how many unique words were used in Hamlet. Print them all

Homework XI (*not going to be on exam*)

We learned how to read a multiline string and split it into words. In input files, lines that start with “#” delimitate comments lines

- (1) Using `boost::starts_with`, adapt the code you made to read a file by line/words (the $\frac{1}{2}$ half of the code you created in HWIII)
- (2) We want to adapt HWIII to read input lines. In these files, we want to read `key = value`, or `key == value`, or `key==value`. The regular expression that detects the presence of the equality sign is `(=+)`. So, adapt the code you use to answer (1), with `boost::regex` and `boost::regex_match`, only to read lines containing at least one `=` sign.

Homework XI (continuation of HW X)

1. One common mistake users make on input files is to forget to add the value, i.e., they write “key= ” and then nothing. Using `boost::regex`, and `boost::regex_match` adapt the code you wrote to read input files to correct such mistakes
2. One common mistake users make on input lines is adding , or ; at the end of the line. The regex `(;+)` and `(,+)` could detect them. Using `boost::regex`, and `boost::erase_regex` adapt the code you wrote to read input files to correct such mistakes
3. Now, we must erase the equal sign before saving the key and value. Using `boost::regex`, and `boost::erase_regex` adapt the code you wrote to read input files to erase the equal sign.