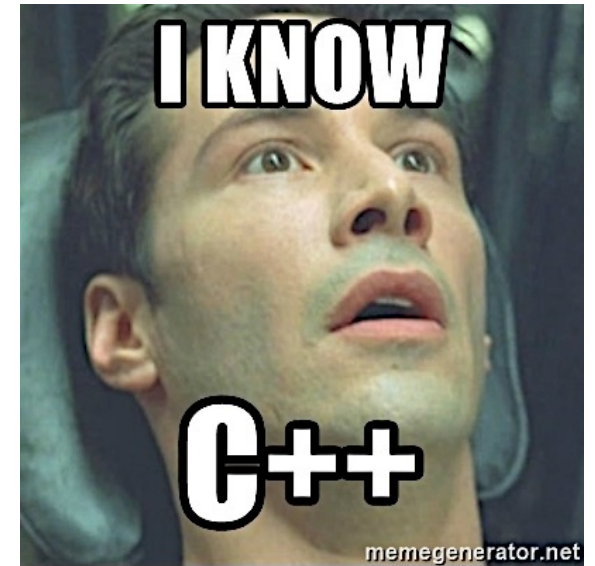
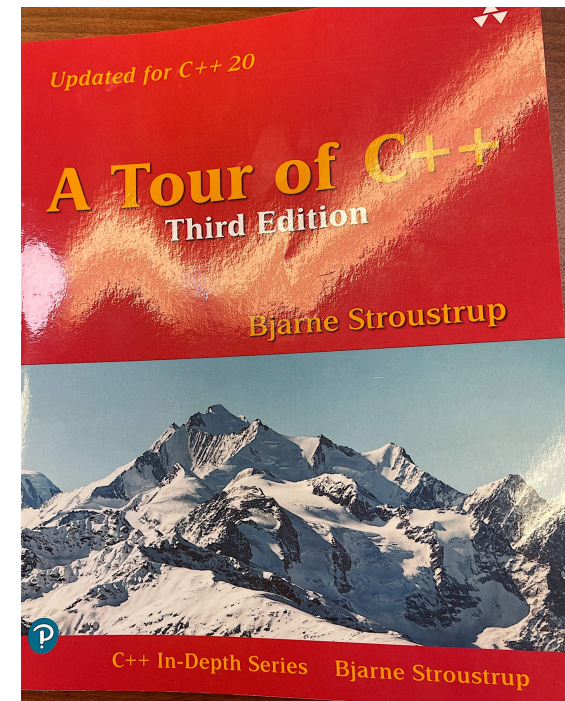


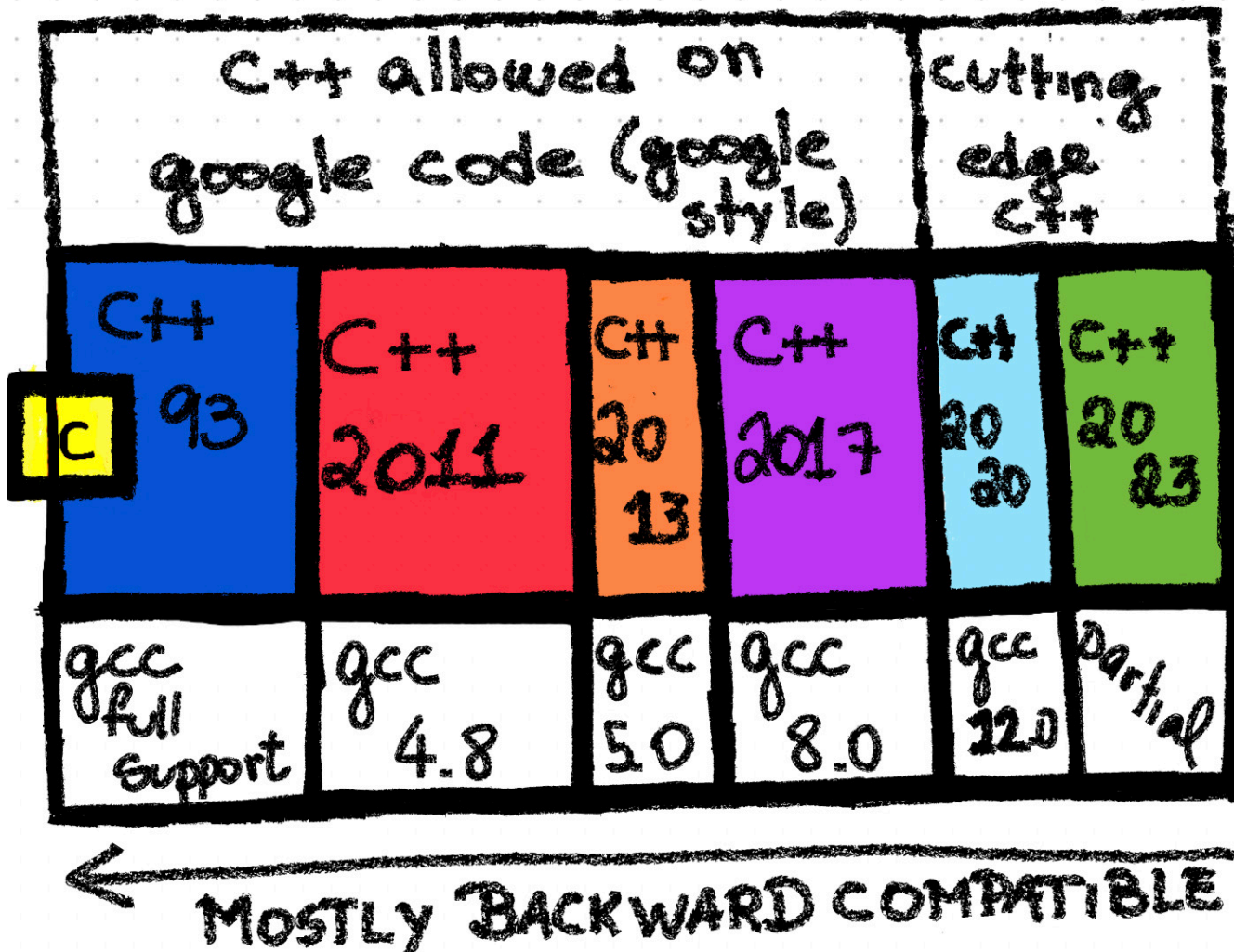
Lecture 16: C++ part I



Suggested Literature



Welcome to C++. Important Advice



C++ is a vast language that is constantly evolving (new standard every three years)

Not all features available by the latest standard are available yet

C++ file extensions: **cpp** and **hpp** (header). I also use **.c** and **.h** (old habits die hard)

Welcome to C++. Important Advice

C++ allowed on google code (google style)				cutting edge C++	
 C++ 93	C++ 2011	C++ 2013	C++ 2017	C++ 2020	C++ 2023
gcc full support	gcc 4.8	gcc 5.0	gcc 8.0	gcc 12.0	partial

← MOSTLY BACKWARD COMPATIBLE

You need to be careful about the compiler version.

We are going to use **gcc** version **5**

The new standard is **C++2026** (gcc implementation of C++2023 is still incomplete)

Use the compiler flag **-std=c++XX** to choose the standard

Welcome to C++. Important Advice

C++26 is done! — Trip report: March
2026 ISO C++ standards meeting
(London Croydon, UK)

News flash: C++26 is done! 🎉

And now we have
C++26! And C++29 is
coming

Completed
C++26
London Croydon,
UK (2026)

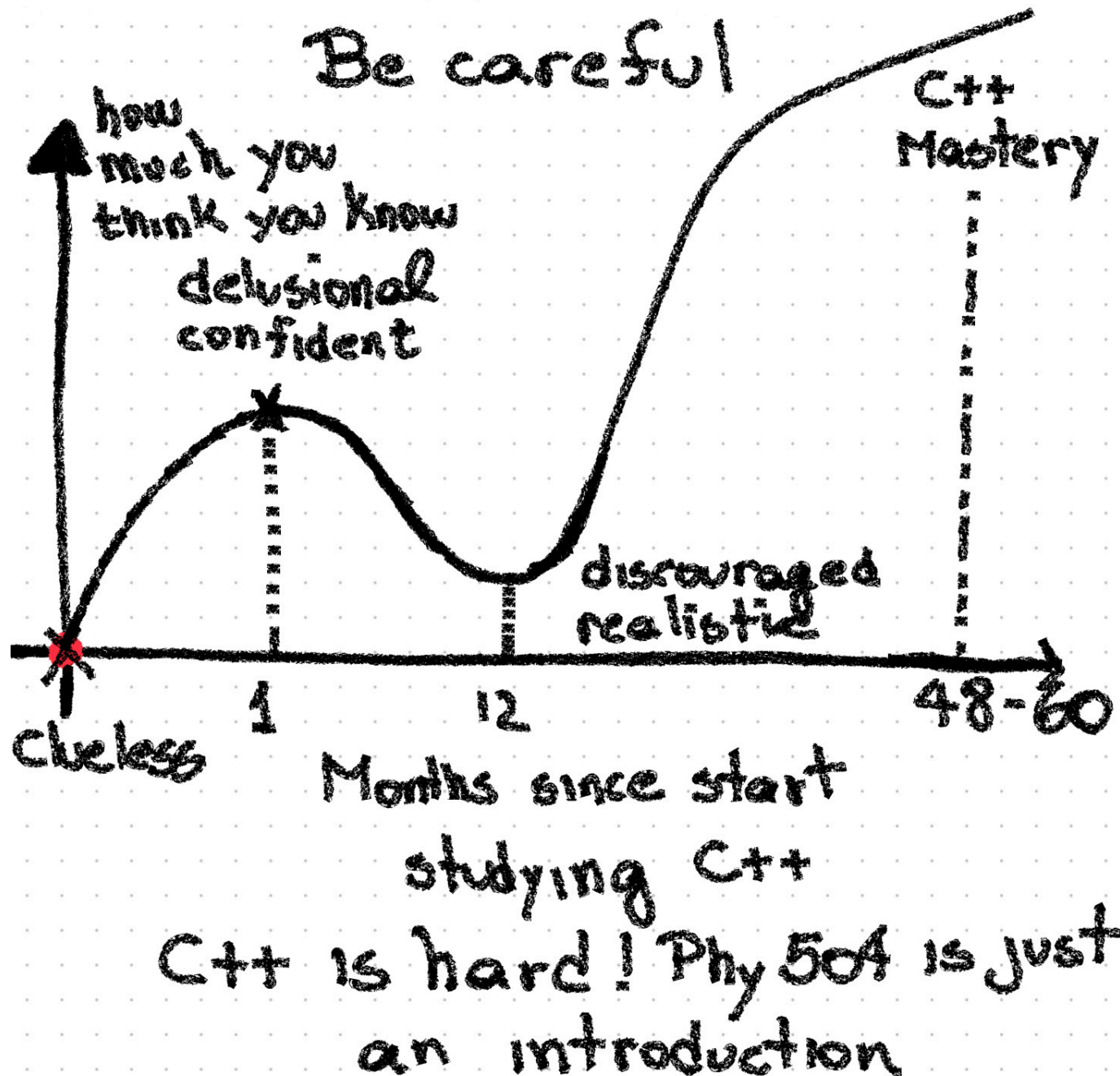


Work on C++29, especially on more memory safety and
profiles

At this meeting we also adopted the schedule for C++29, which will be another
three-year release cycle. To no one's surprise, a major focus of the discussion about
C++29-timeframe material was about **further increasing memory safety**.

<https://herbsutter.com>

Welcome to C++. Important Advice



There is no need to learn all C++ features at once to do research.

I love C++, but it takes a long time to master it.

[Google C++ style](#)

is a good conservative choice of features

Different teaching style



Learning
C++ after
learning
python



Learning
Python
after
learning
C++

So far, I have created a well-structured ladder. I avoided exposing you to advanced concepts prematurely.

C++ has so many concepts that we will make an overview. It will be fun but wild

In C++, I will throw you in the jungle and hope you will come out with a fur coat and a headband made out of some leaves.

That is how I learned at UChicago.

Don't Panic! We have AI to help us

Different teaching style

Most important: Don't Panic. Learn how to embrace seeing code with concepts you don't understand. Use AI to help you understand.

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";

    return 0;
}
```

I am having difficulties understanding "\n" vs std::endl in this code. Can you transcribe this code and explain me std::endl?

This is what ChatGPT is really good for!



Explanation

1. Using `printf("Hello World\n");` :

- `printf` comes from the C standard library (`<stdio.h>`).
- The `"\n"` character sequence represents a newline in the output, so this prints "Hello World" followed by a newline.

2. Using `std::cout << "Hello World" << std::endl;` :

- `std::cout` is an output stream from the C++ standard library (`<iostream>`).
- `std::endl` also inserts a newline character, **but** it additionally **flushes** the output buffer. Flushing means any data buffered in the output stream is forced to be written immediately to the console (or wherever the stream is directed).
- In contrast, a simple `"\n"` just inserts a newline without necessarily flushing the buffer.

3. Using `std::cout << "Hello World\n";` :

- This is very similar to using `printf("Hello World\n");` —it just prints "Hello World" plus a newline, **without** flushing the stream.
- Whether it flushes depends on the system (for example, some implementations may flush on a newline anyway), but it is **not** guaranteed the way `std::endl` is.

Why Use `std::endl` ?

- `std::endl` = `newline ( '\n' ) + flush`.
- Flushing the stream ensures that any buffered text is written out immediately. This can be useful if you need to guarantee that output is visible right away (e.g., before waiting for user input, or

Hello World (old C++)

Many things to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ hello.cpp -o hello1 --std=c++20
whovian@phy504:~/host/CPP$ ./hello1
Hello World
Hello World
Hello World
```

Hello World (old C++)

Much to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";

    return 0;
}
```



no .h at the end
of the header

```
whovian@phy504:~/host/CPP$ g++ hello.cpp -o hello1 --std=c++20
whovian@phy504:~/host/CPP$ ./hello1
Hello World
Hello World
Hello World
```

Hello World (old C++)

Much to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;
    std::cout << "Hello World\n";

    return 0;
}
```

With **printf**, the code offloads the string to the terminal after every command

When printing a lot of stuff that is slow and C++ is crazy about optimization

Hello World (old C++)

Many ways to write Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;
    std::cout << "Hello World\n";

    return 0;
}
```

C++ **std::cout**
has an
intermediate
buffer

 **std::endl = “\n”**
+ **flush buffer**
now

Hello World (old C++)

Much to discuss from a simple Hello World in C++

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";
    return 0;
}
```



C++ **std::cout** has
an intermediate
buffer

std::endl = “\n”
+ **flush buffer**
now (safer but
slower - speed
only relevant in
extreme cases)

Hello World (old C++)

What is **std::**? C++ namespaces (std = standard)

```
#include <stdio.h> // C Library
#include <iostream> // C++17 and below

int main ()
{
    printf("Hello World\n");

    std::cout << "Hello World" << std::endl;

    std::cout << "Hello World\n";

    return 0;
}
```

It is ok to use
using namespace std
in **cpp** files

Not good practice
to use it in **hpp**
(headers).

This would destroy
the encapsulation
purpose of
namespaces

Hello World (new C++)

Informal rule: C++ offers >1 ways to solve problems

```
include <fmt/core.h>
// FMT: C++20 implementation of std::format
// FMT: C++23 implementation of std::print
// (not implemented in gcc yet)
// TO RUN THIS CODE: g++ hello3.cpp -o hello3 -lfmt

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 (not implemented on **gcc-14** yet but ok on g++15)

Standard based on the

[FMT library](#)

Hello World (new C++)

Informal rule: C++ offers >1 ways to solve problems

```
include <print>

// C++23 - still in experimental stage
// I had to download gcc15 snapshot
// TO RUN THIS CODE: upgrade your docker container
// /opt/gcc-latest/bin/g++ hello4.cpp -o hello4 -std=c++23

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

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

Implementation
of C++ new
features lags
years behind

Suggestion: use
FMT lib (but
g++15 ok)

```

#include <iostream>
#include <fmt/core.h>
// FMT: C++20 implementation of std::format
// FMT: C++23 implementation of std::print
// (not implemented in gcc yet)

whovian@phy504:~/host$ g++ init.cpp -o init -std=c++20 -lfmt
whovian@phy504:~/host$ ./init
Ints are 2, 3, 3 and 3
Ints are 2, 3, 3 and 3
Ints are 2, 3, 3 and 3

// this is NOT an array. It is int initialization
int y(3);

// init an integer from std::initializer_list<int>
// the type of {3} is std::initializer_list<int>
int z = {3};

int w{3};

// not formatted output (basic way since C++<11)
std::cout << "Ints are " << x << ", " << y << ", " << z << " and "
<< w << std::endl;

// Second way of printing: formatted (C++20)
std::cout <<
    fmt::format("Ints are {:1d}, {:1d}, {:1d} and {:1d}", x, y, z, w)
<< std::endl;

// Third way of printing: formatted w/ std::print (C++23)
fmt::print("Ints are {:1d}, {:1d}, {:1d} and {:1d}\n", x, y, z, w);
}

```



Variable initialization

There are >1 way to initialize even basic types in C++

Advanced Topic: they have slightly different behaviors

```

#include <iostream>
#include <fmt/core.h>
// FMT: C++20 implementation of std::format
// FMT: C++23 implementation of std::print
// (not implemented in gcc yet)

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

int main () whovian@phy504:~/host$ ./init
{           Ints are 2, 3, 3 and 3
  int x = 2; Ints are 2, 3, 3 and 3
           Ints are 2, 3, 3 and 3

// this is NOT an array. It is int initialization
int y(3);

// init an integer from std::initializer_list<int>
// the type of {3} is std::initializer_list<int>
int z = {3};

int w{3};

// not formatted output (basic way since C++<11)
std::cout << "Ints are " << x << ", " << y << ", " << z << " and "
  << w << std::endl;

// Second way of printing: formatted (C++20)
std::cout <<
  fmt::format("Ints are {:1d}, {:1d}, {:1d} and {:1d}", x, y, z, w)
  << std::endl;

// Third way of printing: formatted w/ std::print (C++23)
fmt::print("Ints are {:1d}, {:1d}, {:1d} and {:1d}\n", x, y, z, w);
}

```

Variable initialization

Example of
standard and
bleeding-edge
(C++23)
ways to print
variables

Two
commands:
print and
format

Namespaces

Namespaces allow
duplication of
function names with
same type / num args

Writing
using namespace std
pollutes your code
(acceptable on **cpp**)

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

namespace vivian
{
    void Print(double x)
    {
        std::cout << "Namespace Vivian\n";
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

namespace phy504
{
    void Print(double x)
    {
        std::cout << "Namespace Phy504" << std::endl;
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

int main()
{
    vivian::Print(2.0);
    phy504::Print(3.0);
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ nano functions4.cpp
whovian@phy504:~/host/CPP$ g++ functions4.cpp -o func4 -std=c++20
whovian@phy504:~/host/CPP$ ./func4
Namespace Vivian
We will print a double. x = 2
Namespace Phy504
We will print a double. x = 3
```

Namespaces

You can also declare global variables inside **namespaces** (minimum level of encapsulation)

Writing **using namespace std** pollutes your code

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

namespace vivian
{
    int myglobal = 10;

    void Print(double x)
    {
        std::cout << "Namespace Vivian\n";
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

namespace phy504
{
    int myglobal = 20;

    void Print(double x)
    {
        std::cout << "Namespace Phy504" << std::endl;
        std::cout << "We will print a double. x = " <<
            x << std::endl;
    }
}

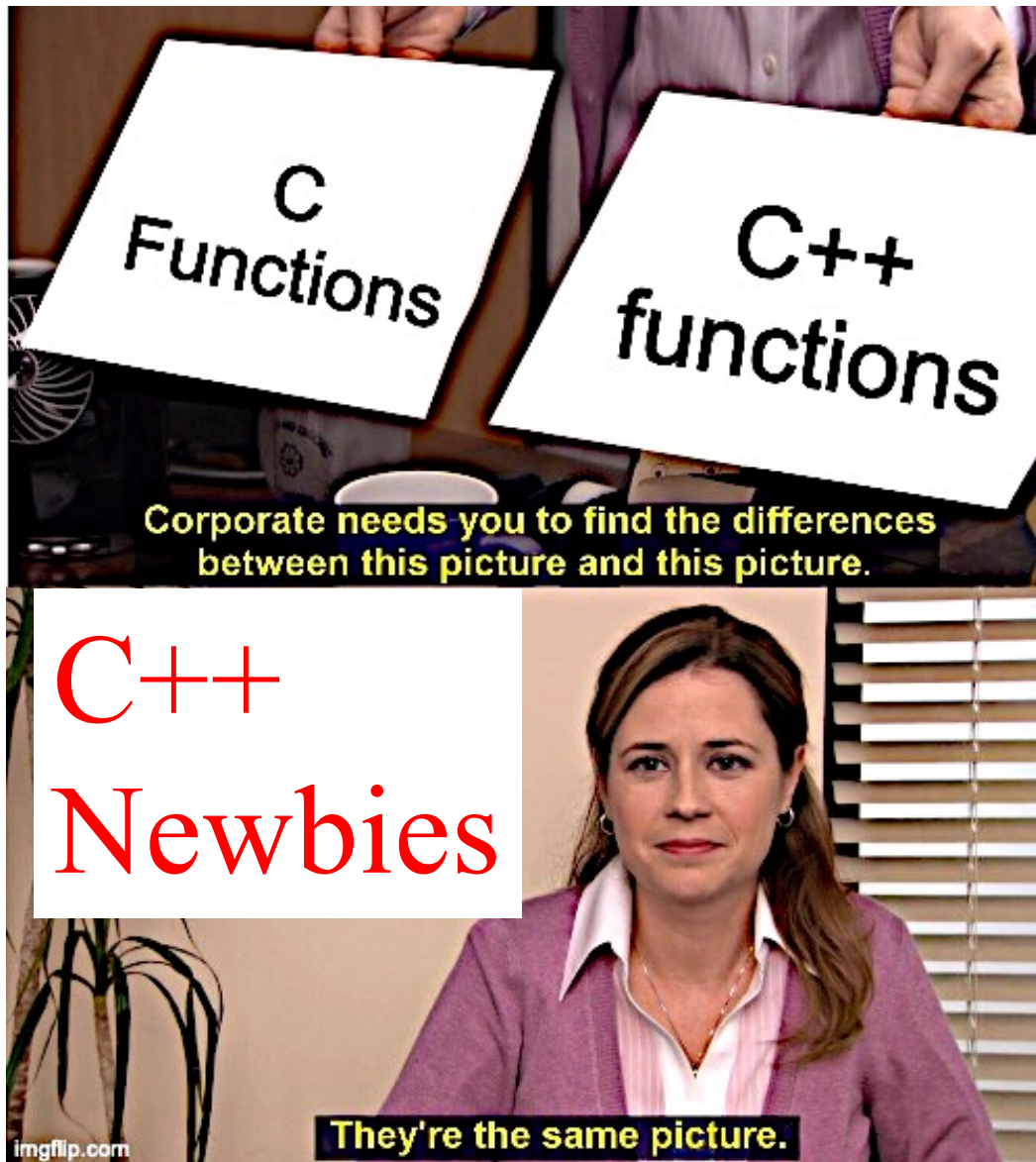
int main()
{
    std::cout << vivian::myglobal << " " <<
        phy504::myglobal << std::endl;
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ nano functions5.cpp
whovian@phy504:~/host/CPP$ g++ functions5.cpp -o func5 -std=c++20
whovian@phy504:~/host/CPP$ ./func5
10 20
```

Ok on **cpp** files, terrible on **hpp** (headers)

Functions C vs C++

They look the same. Trust me – they are **NOT**!



Functions C vs C++

Functions in C++ allow
naming overloading

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

Overloading works on
types. Order imp. **f(int,
double)** neq **f(double, int)**

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

void Print(double x)
{
    std::cout << "We will print a double. x = " <<
        x << std::endl;
}

void Print(int x)
{
    std::cout << "We will print an int. x = " <<
        x << std::endl;
}

void Print(std::string x)
{
    std::cout << "We will print a C++ string " <<
        x << std::endl;
}

int main()
{
    Print(1);
    Print(2.0);

    double x = 3.0;
    Print(3.0);

    std::string str = "HelloWorld";
    Print(str);
    Print("What?");

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ -std=c++20 functions.cpp -o func
whovian@phy504:~/host/CPP$ ./func
We will print an int. x = 1
We will print a double. x = 2
We will print a double. x = 3
We will print a C++ string HelloWorld
We will print a C++ string What?
```

```

#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 C vs C++

Functions in C++ allow
naming overloading

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

Overloading works on
types. Order imp. **f(int,
double)** neq **f(double, int)**

```

#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, 1); // error - compiler can't
                     // decide which function
                     // to use
    */
    // We need to do explicit casting
    // C++ static_cast
    Print(static_cast<double>(1), 1);
    Print(1, static_cast<double>(1));
    return 0;
}

```

```

whovian@phy504:~/host/CPP$ g++ -std=c++20 functions3.cpp -o func3
whovian@phy504:~/host/CPP$ ./func3
Print2: 0
Print1: 2

```

Functions C vs C++

Functions in C++ allow
naming overloading

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

Overloading works on
types. Order imp. **f(int,
double)** neq **f(double, int)**

Functions

C vs C++

Functions + STL
allows the return of
multiple objects.

How?

C++ **tuples** and
pairs (C++11)

tuples / pairs
allow elements
with different types
to be stored

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

int main()
{
    std::tuple<std::string, int> x =
        std::make_tuple("Vivian Miranda", 37);

    // Access tuple elements by position
    fmt::print("name = {0}. Age = {1}\n",
        std::get<0>(x), std::get<1>(x));

    // Access tuple elements by type
    // As long as the type of elements are diff
    fmt::print("name = {0}. Age = {1}\n",
        std::get<std::string>(x), std::get<int>(x));

    // If you only have 3 elements, use std::pair
    // std::pair has nice syntax to access elements
    std::pair<std::string, int> y =
        std::make_pair("Vivian Miranda", 37);

    fmt::print("name = {0}. Age = {1}\n",
        y.first, y.second);

    return 0;
}
```

```
whovian@phy504:~/host$ g++ tuples.cpp -o tuple -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple
name = Vivian Miranda. Age = 37
name = Vivian Miranda. Age = 37
name = Vivian Miranda. Age = 37
```

Functions C vs C++

In C++, types can have other types as arguments

`tuple<int,int>` is not the same as `tuple<string,int>`

```
std::tuple<std::string, int> x =  
    std::make_tuple("Vivian Miranda", 37);  
  
// Access tuple elements by position  
fmt::print("name = {0}. Age = {1}\n",  
    std::get<0>(x), std::get<1>(x));  
  
// Access tuple elements by type  
// As long as the type of elements are diff  
fmt::print("name = {0}. Age = {1}\n",  
    std::get<std::string>(x), std::get<int>(x));
```

Functions like `std::get<arg1>(arg2)` can have types
(or int!) and objects as arguments

`get<string>(x)` and `get<int>(x)`: 2 different outputs!

```

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

int main()
{
    // tuple types are too long
    //std::tuple<std::string, int> x =
    //    std::make_tuple("Vivian Miranda", 37);

    // Modern C++ has a powerful tool to
    // facilitate the use of complicated types

    // compiler can figure out the type based on RHS!
    auto x = std::make_tuple("Vivian Miranda", 37);

    // Access tuple elements by position
    fmt::print("name = {0}. Age = {1}\n",
        std::get<0>(x), std::get<1>(x));

    auto y = std::make_pair("Vivian Miranda", 37);

    fmt::print("name = {0}. Age = {1}\n",
        y.first, y.second);

    return 0;
}

```

```

whovian@phy504:~/host$ nano tuples2.cpp
whovian@phy504:~/host$ ./tuple2
name = Vivian Miranda. Age = 37
name = Vivian Miranda. Age = 37

```

Functions

C vs C++

Need a special function to make
tuple / pair
(**make_X**)

You cannot access tuple elements via **[] operator**. Need **std::get<index>** or **std::get<type>**

```

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

// example from en.cppreference.com/w/cpp/utility/tuple

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
  // 1: with tuples
  // Yes, we can apply const correctness to auto!
  const auto student0 = get_student(0);

  fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
    0,
    std::get<double>(student0),
    std::get<char>(student0),
    std::get<std::string>(student0));

  fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
    0,
    std::get<0>(student0),
    std::get<1>(student0),
    std::get<2>(student0));

  return 0;
}

```

Functions

C vs C++

We need `std::get<index>` or `std::get<type>` to access tuple elements. How?

`std::get<index>` is never ambiguous. On the other hand, `std::get<type>` can only be used when all types are different

```

whovian@phy504:~/host$ g++ tuples3.cpp -o tuple3 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple3
ID: 0, GPA: 3.8, grade: A, name: Lisa Simpson
ID: 0, GPA: 3.8, grade: A, name: Lisa Simpson

```

Functions

C vs C++

Can we adapt the example from the previous slide to return auto?

Yes! But in this case, we need to create the tuples w/
std::make_tuple

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

// example from en.cppreference.com/w/cpp/utility/tuple

auto get_student(const int id)
{
    switch (id)
    {
        case 0:
            return std::make_tuple(3.8, 'A', "Lisa Simpson");
        case 1:
            return std::make_tuple(2.9, 'C', "Milhouse Van Houten");
        case 2:
            return std::make_tuple(1.7, 'D', "Ralph Wiggum");
    }

    throw std::invalid_argument("id");
}

int main()
{
    const auto student0 = get_student(0);

    fmt::print("ID: {0}, GPA: {1}, grade: {2}, name: {3}\n",
        0,
        std::get<0>(student0),
        std::get<1>(student0),
        std::get<2>(student0));

    return 0;
}
```

```
whovian@phy504:~/host/Phy504Docker/CP$ g++ tuples3v2.cpp -o tuples3v2 -std=c++20 -lfmt
whovian@phy504:~/host/Phy504Docker/CP$ ./tuples3v2
ID: 0, GPA: 3.8, grade: A, name: Lisa Simpson
```

Functions

C vs C++

What if you don't want
to create a tuple to
receive multiple returns?

1st option: **std::tie**
(C++11). Great, except
that you can't use it with
the **auto** keyword (write
the types explicitly)

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

// example from en.cppreference.com/w/cpp/utility/tuple

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

    // 2: with tie
    // a bit clunky because you need to define vars separately
    double gpa;
    char grade;
    std::string name;

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

    return 0;
}
```

```
whovian@phy504:~/host$ g++ tuples4.cpp -o tuple4 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple4
ID: 2, GPA: 1.7, grade: D, name: Ralph Wiggum
```

Functions

C vs C++

What if you don't want to create a tuple to receive multiple returns?

Second option: with C++17 **structured binding** (allows the use of **auto**, types can be deducted)

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

// example from en.cppreference.com/w/cpp/utility/tuple

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);

    return 0;
}
```

```
whovian@phy504:~/host$ g++ tuples5.cpp -o tuple5 -std=c++20 -lfmt
whovian@phy504:~/host$ ./tuple5
ID: 1, GPA: 2.9, grade: C, name: Milhouse Van Houten
```

C++ Structured binding

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

struct Person {
    std::string name;
    int age;
    double score;
};

int main() {
    Person p{"Alice", 30, 9.5};

    auto [name, age, score] = p;

    std::cout << name << "\n";
    std::cout << age << "\n";
    std::cout << score << "\n";

    return 0;
}
```

C++17 Structured binding allows you to also unbind from a C Struct

```
whovian@whovianphy504:~/host/Lecture16$ g++ structure_binding.cpp -std=c++17 -o sb
whovian@whovianphy504:~/host/Lecture16$ ./sb
Alice
30
9.5
```

C++ Structured binding

```
#include <complex>
#include <iostream>

int main() {
    std::complex<double> z{3.0, 4.0};

    auto [re, im] = z;

    std::cout << "re = " << re
               << ", im = " << im << '\n';
}
```

C++26 new
feature – allow
C++ complex to
have tuple like
behavior
(including
structure binding)

```
whovian@whovianphy504:~/host/Lecture17$ nano complex.cpp
whovian@whovianphy504:~/host/Lecture17$ g++ complex.cpp -o c -std=c++26
whovian@whovianphy504:~/host/Lecture17$ ./c
re = 3, im = 4
```

C++ Structured binding

```
#include <complex>
#include <iostream>

int main() {
    std::complex<double> z{3.0, 4.0};

    std::cout << z.real()
               << ", "
               << z.imag() << '\n';
}
```

Before **C++26**

```
whovian@whovianphy504:~/host/Lecture17$ g++ complex2.cpp -o c2 -std=c++26
whovian@whovianphy504:~/host/Lecture17$ ./c2
3, 4
```

Curious question: How to print a Tuple?

```
#include <format>
#include <print>
#include <tuple>
#include <utility>
```

```
int main() {
    std::pair<int, int> p{10, 20};
    std::tuple<int, int, int> t{1, 2, 3};
    std::tuple<int, int> kv{7, 99};

    std::print("pair default : {}\n", p);
    std::print("pair as map   : {:m}\n", p);
    std::print("pair no paren: {:n}\n", p);

    std::print("tuple default: {}\n", t);
    std::print("tuple no paren: {:n}\n", t);

    std::print("2-tuple as map: {:m}\n", kv);
}
```

C++23 print is quite versatile on STL containers (like tuple)

{:m} and **{:n}** are tailored to format pairs and tuples

```
whovian@whovianphy504:~/host/Lecture16$ ./t9
pair default : (10, 20)
pair as map   : 10: 20
pair no paren: 10, 20
tuple default: (1, 2, 3)
tuple no paren: 1, 2, 3
2-tuple as map: 7: 99
```

Curious question: How to print a Tuple?

```
#include <format>
#include <iostream>
#include <tuple>
#include <utility>

int main() {
    std::pair<int, int> p{10, 20};
    std::tuple<int, int, int> t{1, 2, 3};
    std::tuple<int, int> kv{7, 99};

    std::cout << std::format("pair default : {}\n", p);
    std::cout << std::format("pair as map  : {:m}\n", p);
    std::cout << std::format("pair no paren: {:n}\n", p);

    std::cout << std::format("tuple default: {}\n", t);
    std::cout << std::format("tuple no paren: {:n}\n", t);

    std::cout << std::format("2-tuple as map: {:m}\n", kv);
}
```

Remember that
format can
introduce this
functionality to
old C++ streams

```
whovian@whovianphy504:~/host/Lecture16$ ./t9
pair default : (10, 20)
pair as map  : 10: 20
pair no paren: 10, 20
tuple default: (1, 2, 3)
tuple no paren: 1, 2, 3
2-tuple as map: 7: 99
```

Homework I

All C++ HW: Each HW should be written in a function. The function should be defined inside a namespace with your name

- Write a C++ function to generate and display a table of n and n^2 , for **integer** values of n ranging from **integer** a_1 to **integer** a_2 (a_1 and a_2 should be arguments of the function). Be certain to print appropriate column headings. Use stream and FMT library (two different functions).
- Overload previous HW solution (maintain same function name – use C++ overloading). Write a C++ function to generate and display a table of n and n^2 for two double values of n ranging from **double** a_1 to **double** a_2 (a_1 and a_2 should be arguments of the function). Use stream and FMT library (two different functions).

Homework II

All C++ HW: Each HW should be written in a function. The function should be defined inside a namespace with your name

- Write a program that subtracts the value **double a_1** from **double a_2** and displays the result with an appropriate message at the terminal. Use stream and FMT library.
- Overload previous HW solution (maintain same function name – use C++ overloading). Write a program that subtracts the value **integer a_1** from **integer a_2** and displays the result with an appropriate message, at the terminal. Use stream and FMT library (two different functions). Make sure to write a unique message for each overload.
- Write a program that converts **double a_1** degrees Fahrenheit (F) to degrees Celsius (C) with the following formula: **$C = 5 (F - 32) / 9$** . Use stream and FMT library. Pay attention to the variable types!

Homework III

All C++ HW: Each HW should be written in a function. The function should be defined inside a namespace with your name

- C++ has a built-in type for complex numbers. (See more information at: <https://www.geeksforgeeks.org/complex-numbers-c-set-1/>). Write a program that subtracts the value **complex<double> a_1** from **complex<double> a_2** and displays the result with an appropriate message at the terminal. Use stream and FMT library (two different functions). Use overloading C++ to maintain the same function name as the mandatory HW.

Homework IV

All functions should be defined inside a namespace with your name

- Generalize the following code to return the name, age, country (using 3-digit code) and wealth (US dollars) of any of your five favorite movie stars. The wealth can be an integer; no need to count the cents. The function should use the switch statement to provide concrete answers about any of your five favorite stars.
- Use C++17 Structure binding to exemplify in main() the return values of your function
- Use C++11 **std::tie** to exemplify in main() the return values of your function
- Use **std::get<position>** to exemplify in main the return values of your function

```
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"};
    }

    throw std::invalid_argument("id");
}
```