An Introduction to C++

A survey of what you will (or might) encounter this summer

Remington Hill

Contact Info: remington.hill@queensu.ca

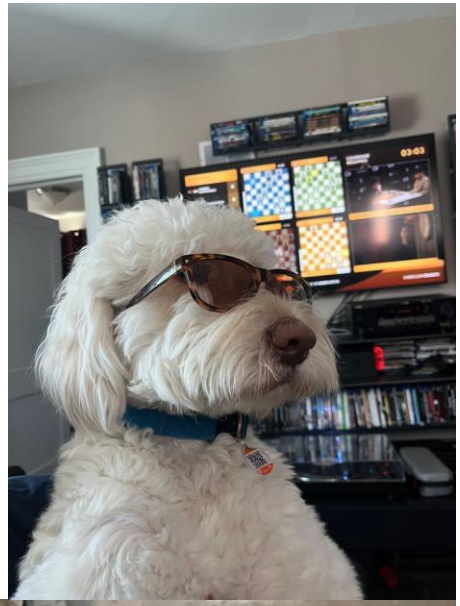
Supervisor: Dr. Stephen Sekula

Queen's University

May 12th, 2025

Introduction

- My name is Remington W. Hill.
- 3rd year PhD Student at Queen's.
  - PICO – Acoustic reconstruction.
- You can find me at SNOLAB (2nd floor cubical space, C15).
- Hobbies:
  - Baking and cooking .
  - Reading (sci-fi/fantasy, history, etc.).
  - I enjoy playing and watching chess (I am RemiTech on chess.com).
  - Also a big Warhammer 40k fan, enjoy painting and playing.



Outline

- Formatting.
- Compiler(s).
- Libraries.
- Namespaces.
- Control statements.
- if, else, while, for, etc.
- Functions.
- Vectors and Arrays.
- Pointers.
- Classes/structs.
- Templates.

What is and what isn't this talk?

- This is not a formal lecture on C++.
  - I am not a computer scientist.
- I assume that each of you have no experience coding in C++.
  - Will build up from the basics (no one gets left behind).
- My hope is that after today, you have enough of the basics to:
  - Understand your collaborations analysis tools and write code in their existing framework.
  - Run and understand Geant4 simulations (or other equivalent C++ simulations).
- A couple of questions before we get started:
  - How many of you have taken a computer science course before?
  - How many of you have written software in C++ before?
    - Be honest! No judgement. I am using this to gauge pace.
  - Who has written in Python before?

Some general remarks

- Don’t be afraid to ask for help!
  - Can always speak to your coworkers, supervisor, etc.
  - If you need help, I can be reached at remington.hill@queensu.ca or at C-15 on the 2nd floor at SNOLAB (for those who are on site this summer).
- You are going to make mistakes; learn from them.
- Make sure you keep a good reference handy.
  - I use “C++: How to program” by Deitel and Deitel (in fact, most examples in this talk are largely inspired by this textbook).
- Understanding the limits of googling and using someone else's code.
  - Stack overflow is an **asset**. However, do not just copy the code without understanding why it is written the way that it is.
  - This will make you a better programmer. Will also help with readability; whomever takes over your code will thank you.
- Where does generative AI models fit into this?
  - I would strongly advise you **not** to use ChatGPT, Copilot, Deepseek to write code.

* Here, when we say every line, we exclude control operators such as if, else, while, etc. Classes, structs (see later slides), will require semi-colons.

C++ structure

- Every line must be ended with a semi-colon* (;).
- Variable types need to be specified upon declaration.
  - Is it an int, double, float, TFile, TTree, G4int? You need to tell the compiler.
- Indentation is optional; however, please respect your fellow programmers.
  - When entering a conditional statement, class, struct, etc. it is good practice to increase the indentation by 1 tab or 4 spaces.
- Make sure you leave useful comments.
  - Each of you will work on code that someone else will then use (potentially).
  - Do not make them pull there hair out.
  - For today only, I am breaking this rule to avoid clutter.
- Choose appropriate variable names.
  - Do not store an energy value in a float called “temp”.
    - This means nothing.
  - Suppose you were looking to store both the electrical charge of an electron, and the permittivity of free space, what would be a good name for each?

A classic example

- The first thing you, as a programmer, will typically do in a new language is print the classic, “Hello World”.
  - What does this look like in C++?

hello_world.cpp > ...

```cpp
#include <iostream>

using namespace std;

int main(){

    cout << "Hello World" << endl;

    return 0;
}
```

A classic example

- The first thing you, as a programmer, will typically do in a new language is print the classic, “Hello World”.
  - What does this look like in C++?

hello_world.cpp

```cpp
#include <iostream>

using namespace std;

int main(){

    cout << "Hello World" << endl;

    return 0;
}
```

The method to include standard libraries or your files.

If enclosed by < >, compiler will look in your path for the library.

If enclosed by double quotes (i.e. #include “experiments.h”), local directories are checked.

A classic example

- The first thing you, as a programmer, will typically do in a new language is print the classic, “Hello World”.
  - What does this look like in C++?

hello_world.cpp > ...

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main(){
6
7       cout << "Hello World" << endl;
8
9       return 0;
10  }
```

Tells the compiler which namespace is being used throughout the program.

For example, cout and endl belong to the std namespace.

This line allows us to use these two functions without specifying their namespace.

Without this line, all calls to functions of this namespace need to include std::

A classic example

- The first thing you, as a programmer, will typically do in a new language is print the classic, “Hello World”.
  - What does this look like in C++?

hello_world.cpp > ...

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main(){
6
7       cout << "Hello World" << endl;
8
9       return 0;
10  }
```

This is the main function; it is the entry point for your C++ program.

Every function will have its arguments defined in (), and be closed by {}.

Function definitions before this will not be called or used unless called within the main() function.

int tells the compiler that this function returns an int value when it exits.

Can return any type (double, int, vector<int>, etc.).

A classic example

- The first thing you, as a programmer, will typically do in a new language is print the classic, “Hello World”.
  - What does this look like in C++?

hello_world.cpp > ...

```cpp
#include <iostream>

using namespace std;

int main(){

    cout << "Hello World" << endl;

    return 0;
}
```

Print to the terminal the string “Hello World”.

cout and endl are objects from std; more on this later (see Namespaces).

cout stands for ‘character output’. endl ends the current line.

A classic example

- The first thing you, as a programmer, will typically do in a new language is print the classic, “Hello World”.
  - What does this look like in C++?

hello_world.cpp > ...

```cpp
#include <iostream>

using namespace std;

int main(){

    cout << "Hello World" << endl;

    return 0;
}
```

Return value of the function.

Returning zero tells the compiler that main exited without issue.

Any other value is some error status you need to identify.

User defined functions return whatever the user provides.

Data types

- C++ requires you to be exact when defining variables.
  - You must tell the machine what it is handling.
- So what are they?
- **Declaration:** this is telling the compiler what the name **and** data type of your variable is.
- **Initialization:** This is supplying an initial value to the variable.

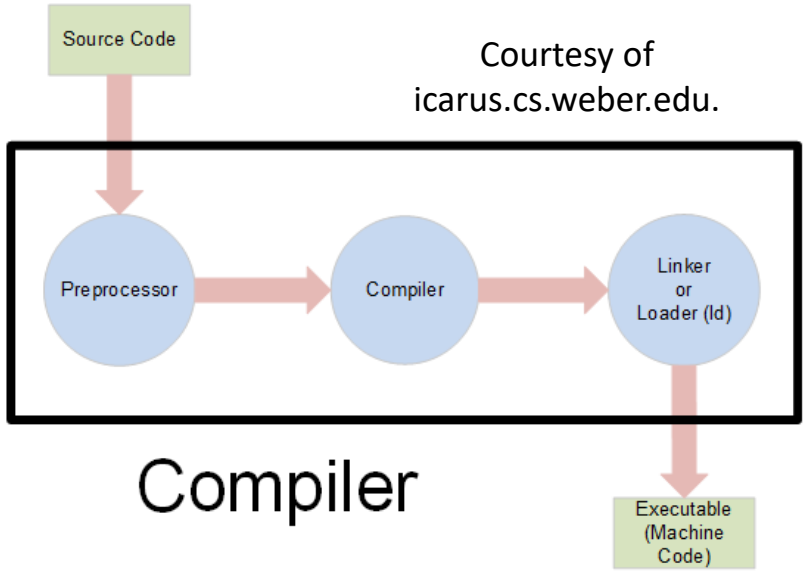
Key word	Size in bytes	Interpretation	Possible values
bool	1	boolean	true and false
unsigned char	1	Unsigned character	0 to 255
char (or signed char)	1	Signed character	-128 to 127
wchar_t	2	Wide character (in windows, same as unsigned short)	0 to 2^{16}-1
short (or signed short)	2	Signed integer	-2^{15} to 2^{15} - 1
unsigned short	2	Unsigned short integer	0 to 2^{16}-1
int (or signed int)	4	Signed integer	-2^{31} to 2^{31} -1
unsigned int	4	Unsigned integer	0 to 2^{32} - 1
Long (or long int or signed long)	4	signed long integer	-2^{31} to 2^{31} -1
unsigned long	4	unsigned long integer	0 to 2^{32} -1
float	4	Signed single precision floating point (23 bits of significand, 8 bits of exponent, and 1 sign bit.)	$3.4*10^{-38}$ to $3.4*10^{38}$ (both positive and negative)
long long	8	Signed long long integer	-2^{63} to 2^{63} -1
unsigned long long	8	Unsigned long long integer	0 to 2^{64} -1
double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)
long double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)

Courtesy of go4expert.

Compiling

- How do we run this code?
  - Will need to convert this from human readable to machine readable.
- Right now, we have what is often referred to as the **source code.**
  - For most daily applications, you don't have access to this (i.e. whatever video games you are playing right now).
- To get the machine-readable file, also called the **executable**, we must compile our program..

- **Preprocessor**: process all statements with #; copies C++ statements to temp. files.
- **Compiler**: translates source code to machine code.
- **Linker**: will link together object code files (headers, more on this later).



Compiling pt. 2

- Here, we are going to use the command line and g++ (on **LINUX).**
- To compile, run: `g++ Hello_World.cpp -o Hello_World`
- This will yield an executable of the name “Hello_World”.
  - It is standard to use the same source code name but drop the file extension.
  - If no name is supplied, the default is a.out.
- There are other arguments that can be included:
  - You will most certainly use –I/path/to/headers/, which tells the compiler to include the supplied directory.
- To run this file, simply put: `./Hello_World`
- If you do not have access to the command line and g++ today (or a windows compiler), can use this: https://www.programiz.com/cpp-programming/online-compiler/
- A temporary solution; you should seek access to a permanent solution. For large files or projects, I encourage you to use ‘make’ (or GNUmake).
  - Will automatically check which parts of the code need updating.
    - i.e. unchanged files are not recompiled.
- You can find documentation here: https://www.gnu.org/software/make/manual/make.html.

The Preprocessor

- The preprocessor can be used for the following:
  - Inclusion of other files (we've already seen this).
  - Defining constants (or macros).
  - Conditional compilation of code.
  - Conditional execution of preprocessor directives.
- Suppose you wanted to use some constant in your file, it *could* be defined as such:

```
#define REMINGTONS_AGE 28
```

- Within your cpp file, any occurrence of REMINGTONS_AGE is **replaced** with the integer, 28.
  - Removes function overhead.
- Another example:

```
#define f(x) (2*x*x + 3*x - 5)
```

Preprocessor (cont.)

- You might ask yourself, why use the preprocessor to
  - define a constant?
  - do a simple operation? (i.e. calculate area).
- The short answer is you usually won't (unless you are working in C).
- The only (non-conditional include) example I have ever used is:

```
// Here, we use the # operator to output the name of a variable.
#define UPDATE_USER(x) std::cout << #x << " is set as -----> " << x << std::endl;
```

- UPDATE_USER can be called as a normal function would, and yield:

```
string experiment = "PICO-40L";
int currentYear = 2025;

UPDATE_USER(experiment);
UPDATE_USER(currentYear);
```

```
experiment is set as -----> PICO-40L
currentYear is set as -----> 2025
```

Namespaces

hello_world_no_namespace.cpp > main()

```cpp
#include <iostream>



int main(){

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

    return 0;
}
```

hello_world.cpp > ...

```cpp
#include <iostream>

using namespace std;

int main(){

    cout << "Hello World" << endl;

    return 0;
}
```

- What are namespaces?
  - Suppose you include two header files, Alice.h and Bob.h.
  - Each are a different namespace.
  - They both have a function within them called person().
  - How do we specify which one we are using?
- Namespaces: define the scope where global identifiers and global variables are placed.
  - I.e. Within the namespace Alice, access to all identifiers in that space is possible.
- To use functions within a namespace, two approach's.
  - Specify namespace followed by function or variable name (Alice::person()).
  - Or 'use' the namespace (see left).

Control statements

- Control statements allow you to control where a program will go and what actions it will take.
- The most common are:

if (*condition){..code..}*

else if (*condition){..code..}*

else*{..code..}*

for*(start; stop; increment){..code..}*

while *(start/condition){..code..}*

do while*(condition){..code..}*

- Examples of each can be found on the right.
  - There are critical differences between all of them (lets discuss).

```cpp
#include <iostream>

using namespace std;

int main(){

    int a = 1;
    int b = 10;

    // Example of if statements and conditions.
    if (a < b) cout << "a < b" << endl;
    else if (a > b) cout << "a > b" << endl;
    else{
        cout << "a = b" << endl;
    }

    // While loop. Stop before a = b
    while (a < b){
        cout << "a = " << a << endl;
        a++
    }

    // Do while loop. Stop before a = b
    a = 1;
    do{
        cout << "a = " << a << endl;
        a++
    }while(a < b);

    // For loop. Equivalent to the while loop above.
    for (a = 1; a < b; a++){
        cout << "a = " << a << endl;
    }

    return 0;
}
```

Find the errors!

- We should have enough information to now begin to identify errors in the code.
  - There are 5/6 errors in the code below.
  - Take 1-2 minutes to identify what they are.

etiquette_example_1.cpp > main()

```cpp
#include <iostream>

int main(){

    a = 50;

    while (a < 100){
        cout << "a = " << a << endl
    }

}
```

Corrected code

etiquette_example_1_corrected.cpp > main()

```
1   #include <iostream>
2
3   int main(){
4
5       int a = 50;
6
7       while (a < 100){
8           std::cout << "a = " << a << std::endl;
9           a++;
10      }
11
12
13      return 0;
14  }
```

1) Missing variable type in declaration.

2/3) Missing namespace identifier(s).

4) Missing semi-colon at the end of the line.

5) No increment resulting in infinite loop.

6) Missing return statement.
(This technically won't give an error message, but good practice is to always return a value for non-void functions)

Listen to your compiler!

- Listen to your compiler.
  - If the compiler encounters an error, it will be very explicit on where it is.
- Let us look at the log from our previous example.
  - Some things it won't catch (i.e. infinite loop; that is your responsibility!).

```
(base) [rhill@nearline-login EIEIOO_2024]$ g++ etiquette_example_1.cpp -o etiquette_example_1
etiquette_example_1.cpp: In function ‘int main()’:
etiquette_example_1.cpp:5:5: error: ‘a’ was not declared in this scope
    5 |     a = 50;
      |     ^
etiquette_example_1.cpp:8:9: error: ‘cout’ was not declared in this scope; did you mean ‘std::cout’?
    8 |         cout << "a = " << a << endl
      |         ^~~~
      |         std::cout
In file included from etiquette_example_1.cpp:1:
/cvmfs/soft.computecanada.ca/easybuild/software/2020/Core/gcccore/9.3.0/include/c++/9.3.0/iostream:61:18: note: ‘std::cout’ declared here
   61 |   extern ostream cout;  /// Linked to standard output
      |                  ^~~~
etiquette_example_1.cpp:8:32: error: ‘endl’ was not declared in this scope; did you mean ‘std::endl’?
    8 |         cout << "a = " << a << endl
      |                                ^~~~
      |                                std::endl
In file included from /cvmfs/soft.computecanada.ca/easybuild/software/2020/Core/gcccore/9.3.0/include/c++/9.3.0/iostream:39,
                 from etiquette_example_1.cpp:1:
/cvmfs/soft.computecanada.ca/easybuild/software/2020/Core/gcccore/9.3.0/include/c++/9.3.0/ostream:599:5: note: ‘std::endl’ declared here
  599 |     endl(basic_ostream<_CharT, _Traits>& __os)
      |     ^~~~
```

Command Line / User Inputs

- Last week, you would have learned how to navigate the command line at Derek's talk.
  - I will assume you are familiar with the basics.
- Suppose I wanted to check which two numbers, a and b, was smaller.
  - Can I avoid having to recompile each time? Yes.
  - Two solutions we will cover:
    - Command line arguments
    - User input during execution (cin).
- For command line arguments, modify the main function to include two arguments (which we will talk about later):

```
int main(){}
int main(int argc, char* argv[]){}
int main(int argc, char** argv){}
```

Command line (cont.)

- The code on the right will take two integers as inputs.
- It then uses three conditional statements to check if:
  - a < b
  - a > b
  - a == b
- The pointer to the array of argv[] needs to be converted to the appropriate type.
  - atoi() for ints.
  - atof() for doubles.
- argv[0] is the filename of the executable.

```cpp
#include <iostream>
#include <stdlib.h>

using namespace std;

int main(int argc, char* argv[]){

    if (argc != 3){
        cout << "ERROR: Incorrect number of arguments given." << endl;
        return 1;
    }

    int a = atoi(argv[1]);
    int b = atoi(argv[2]);

    if (a < b) cout << "a < b" << endl;
    else if (a > b) cout << "a > b" << endl;
    else{
        cout << "a = b" << endl;
    }

    return 0;

}
```

Character inputs

- Another variation of this program is character input.
  - Called cin; opposite of cout.
  - Like cout, it is apart of the iostream header.
- This asks the user for **input** during execution.
- The general structure is to declare some variable of a given type, then use the overloaded cin operator to store it

```
int val = 0;
cin >> val;
```

- Let us see this in action.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6
7      int val = 0;
8
9      while (val > 10 || val < 1){
10         cout << "Please enter a number between 1 and 10." << endl;
11         cin >> val;
12     }
13
14     cout << "Thank you for entering " << val;
15     cout << ", which is in the correct bounds!" << endl;
16
17     return 0;
18 }
```

```
Please enter a number between 1 and 10.
0
Please enter a number between 1 and 10.
11
Please enter a number between 1 and 10.
12
Please enter a number between 1 and 10.
-1
Please enter a number between 1 and 10.
1
Thank you for entering 1, which is in the correct bounds
```

Functions

- Thus far, we have made use of functions from the standard library.
  - You are free to create your own.
- As is the case with our main, our custom functions will have the following format:

  type function_name(*args*){}
- *type*: Denotes the type returned by the function.
- function_name: The name you give to the function.
- *args*: the arguments supplied to the function (types must be specified and separated by commas).
- Lets talk about some of the examples on the right.

```cpp
#include <iostream>

using namespace std;

double add(double a, double b){
    return a + b;
}

double subtract(double a, double b){
    return a - b;
}

double multiple(double a, double b){
    return a*b;
}

double f(double x){
    return x*x - 13*x + 42;
}

int main(){

    cout << add(1,2) << endl;
    cout << subtract(1,2) << endl;
    cout << multiple(1,2) << endl;
    cout << f(2) << endl;

    return 0;
}

```

Question time!

- Will the following code give me an error during compilation?

```cpp
#include <iostream>

using namespace std;

double multiply(double x, double y){
    return x*y;
}

int multiply(int x, int y){
    return x*y;
}

int main(){

    int i_product = multiply(2, 4);
    double d_product = multiply(2.1, 4.2);

    cout << "Prodct using integar multiply function: " << i_product << endl;
    cout << "Prodct using double multiply function: " << d_product << endl;


    return 0;

}
```

NO!

(let's talk about overloading)

Function overloading

- Function overloading refers to creating two functions with the same name, but different parameters.
- This is allowed, if and only if, the arguments are different.
  - Compiler will find appropriate function call.
- An unrealistic example is on the right.
  - In practice, a sum function for array or vector of length N would be used.
- See this very often in ROOT, Geant4, etc.
  - TH1, Tfile, TTree, TBranch.
- We will talk about this more later in the context of constructors for classes.

```
1   #include <iostream>
2
3   using namespace std;
4
5   double add(double a, double b){
6       return a + b;
7   }
8
9   double add(double a, double b, double c){
10      return a + b + c;
11  }
12
13  double add(double a, double b, double c, double d){
14      return a + b + c + d;
15  }
16
17  int main(){
18
19      cout << add(1,2) << endl;
20      cout << add(1,2,3) << endl;
21      cout << add(1,2,3,4) << endl;
22
23      return 0;
24  }
25
```

Function overloading (cont.)

- How does one overload a function incorrectly?
  - Suppose we have two definitions for a function with the same name:

```
double f(double x){
    return x*x + 2*x - 3;
}

double f(double x){
    return 2*x + 3;
}
```

  - Yields a redefinition error.

```
playground.cpp:12:8: error: redefinition of 'double f(double)'
   12 | double f(double x){
      |        ^
playground.cpp:8:8: note: 'double f(double)' previously defined here
    8 | double f(double x){
      |        ^
```

Function prototyping

- When working with classes you'll see **function prototyping.**
  - A method of defining a function and its arguments prior to specifying its operation.
- Useful for when functions depend on one another.
  - Or for the sake of tidiness, a class is defined in a header file outside the current file.
- Make sure you give a function a corresponding definition.
  - Undefined reference error if lines 17-20 were removed.

```
1   #include <iostream>
2
3   using namespace std;
4
5   double f(double x);
6
7   int main(){
8
9       for (int i = -50; i < 50; i++){
10          cout << "x = " << i;
11          cout << ", f(x) = " << f(i) << endl;
12      }
13
14      return 0;
15  }
16
17  double f(double x){
18      return x*x + 3*x + 5;
19  }
20
21
22
```

Pointers

- What we have worked with thus far has been variables.
  - When we define an int, string, or double, it **directly** stores the value we assign it.
- Pointers hold the location in memory where a value is located.
- To declare a pointer, you must use the * operator.
  - It is equivalent to say either `int* p;` `int *p;`
- You can try and cout this pointer, all you will get is the **memory address**:

```
cout << p << endl;   ---> "0x8"
```

- To get the value at this address, we must **dereference** p.

```
int* p = new int(5);

cout << *p << endl; ---> 5
```

Pointers (cont.)

- You can also assign an already existing variable to a new pointer.
- To do so, you can use the 'address of' operator (&)

```
int* p;
int x = 5;
p = &x;
```

```
// Or more simply.
int x = 5;
int* p = &x;
```

- You need to be mindful with pointers. They just **point** to a place in memory.
  - There can be instances where that point in memory is updated.
  - Always verify you are using the value and type you intend to.
- Watch out for memory leaks.
  - Especially in Geant4 simulations (see anecdote about muon veto sims).
- Why use pointers?
  - Dynamic memory allocation, removes the copying of variables, etc.

Careful with pointers!

```cpp
#include <iostream>
#include "TFile.h"
#include "TTree.h"

using namespace std;

int main(){

    // First thing we shall do is open a TFile.
    TFile* f = new TFile("/project/pico/rhill/EIEIO_2025/sample.root", "READ");

    // Now we shall get the TTree contained in the file.
    TTree* t = (TTree*)f->Get("sample_tree");

    // We will do a few basic operations then close the file.
    int length = t->GetEntries();

    for (int i = 0; i < length; i++){
        t->GetEntry(i);
    }

    return 0;

}
```

File doesn't exist

- When working with pointers, there is an increased risk of computational error.
- This will compile just fine! But it will cause a **segmentation fault**.

Namespaces (cont.)

- You can create your own namespace.
- Suppose you have a set of custom functions that you use in your analysis.
  - Can place these in a header file and include this in future analysis to have access to the functions.
- To define a namespace:

namespace nsp_name*{..code..}*

- An example is on the right.

```cpp
#include <iostream>

namespace analysis{

    const double energy = 2.614;
    const double ratio = 0.99;

    double f(double x){
        return x*x;
    }

    namespace sub_analysis{
        const double momentum = 3.14;
        const double e2 = energy*energy;

        double g(double x){
            return x*x*x;
        }
    }

}

int main(){

    double a = 2.0;
    double a2 = analysis::f(a);
    double a3 = analysis::sub_analysis::g(a);
    std::cout << "A = " << a << ", A^2 = " << a2 << ", A^3 = " << a3 << std::endl;

    double e = analysis::energy;
    double e2 = analysis::sub_analysis::e2;
    std::cout << "Energy = " << e << ", Energy squared = " << e2 << std::endl;


    return 0;
}
```

Main namespace (analysis).

Nested namespace (sub_analysis).

sub_analysis has access to members of analysis.

Namespaces (cont.)

- With namespaces, you cannot inherit (more on this later) from other namespaces.
- If instead, you want to use members from a different namespace in a new namespace, utilize the **using** command.
- For example, here is a created namespace that has access to all std members.
- Important to remember here that the declaration of 'using …" is local to the namespace bubbleChamber.
- **using** can be applied to entire namespaces or just members.

```
#include <iostream>

namespace bubbleChamber{

    using namespace std;

    double target_mass = 500; // Units of tonnes.
    string target_fluid = "C3F8";

}

int main(){

    std::cout << bubbleChamber::target_fluid << std::endl;

}
```

A brief aside on scope

- Let's talk about scope.
- You might often hear programmers talk about scope. What does this mean?
- Scope tells us where a function, variable, struct, class, **is or is not** defined.
- We saw earlier namespaces, functions:
  - The rule of thumb is that when a variable is defined within one of these, it is **local** to this scope.
  - You cannot access it outside of this namespace or function.
- It is good practice to avoid global variables.

Scope error

- Let us take a couple minutes to find the scope error.
- Where is the error?

FOUND IT!
If we try to declare that d is equal to a, the compiler will recognize that a is "not declared in this scope".

```cpp
#include <iostream>

namespace top_namespace{

    int b = 5;
    double c = b*2.0;

    namespace inner_namespace{
        int a = b;
        double f(double x){
            return x*x + 2*x -3;
        }
    }

    int d = a;
    double g(double x){
        return 2*x;
    }


}

int main(){

    std::cout << "Just hanging out!" << std::endl;

    return 0;
}
```

Arrays and vectors

- Vectors and arrays serve a similar function. But are vastly different in execution.
  - Both can store multiple instances of the same type (i.e. int, double, etc.).
- Arrays are **static** entities (like structs).
  - This means that they **cannot** change size after declaration.

    type array_name*[size];*
- Vectors are **dynamically** sized entities.
  - They **can** change size after declaration (although, be careful of memory constraints).

    std::vector<*type*> vector_name;
- Each have their own respective functions.
  - std::vector reference (https://cplusplus.com/reference/vector/vector/).
  - std::array reference (https://en.cppreference.com/w/cpp/container/array).

```cpp
#include <iostream>
#include <vector>

int main(){

    int size = 50;
    int arr[size];

    std::cout << arr[0] << std::endl;

    arr[0] = 1;
    arr[1] = 2;

    for (int i = 0; i < size; i++){
        arr[i] = i;
    }

    std::cout << arr[0] << std::endl;

    std::vector<double> vec;

    vec.push_back(1); // Appends 1 to the back.
    vec.push_back(2);
    vec.clear(); // Clears all elements.


    for (int j = 0; j < size; j++){
        vec.push_back(j);
    }

    std::cout << vec.at(0) << std::endl;
    std::cout << vec[0] << std::endl;

    return 0;
}
```

Structs

- "Structures are aggregate data types".
  - Another way of saying collection.
- What are some examples of 'structures' in real life?
  - A person.
    - Height, name, hair colour, etc.
  - A SNOLAB experiment.
    - tivity, etc.
  - Any
- General ruct is:

  st {..*code*..};
- Do not f lon!
- Let us g mple.

```
Remington
Brown
Blue
185
99
Samwise the Brave
Dirty blonde
Blue
100
99
```

```cpp
#include <iostream>

using namespace std;

struct person{

    string name = "Remington";
    string hair_colour = "Brown";
    string eye_colour = "Blue";
    double height = "185"; // Units of cm
    double weight = "99"; // Units of kg


};

int main(){

    person Remington;
    cout << Remington.name << endl;
    cout << Remington.hair_colour << endl;
    cout << Remington.eye_colour << endl;
    cout << Remington.height << endl;
    cout << Remington.weight << endl;

    // I dye my hair dirty blonde, lose 80 cm in height
    // Also change my name to Samwise the Brave.
    Remington.name = "Samwise the Brave";
    Remington.height = 100;
    Remington.hair_colour = "Dirty blonde";

    cout << Remington.name << endl;
    cout << Remington.hair_colour << endl;
    cout << Remington.eye_colour << endl;
    cout << Remington.height << endl;
    cout << Remington.weight << endl;

    return 0;
}
```

Classes

- Classes allows for objects to have *attributes* **and** *operations*.
  - Extremely similar to structs, so what is the difference?
  - Structs – all members are **public** by default.
  - Classes – all members are **private** by default.
- In classes, you the user decide what is **private** or **public** (these are called member access specifiers).
  - **Private** members are only available to member functions of the class.
  - **Public** members/variables are available to functions external to the class.
- Like structs, classes have an optional requirement(s):
  - **Constructor**: allows the user to specify what functions are called, variables are set, etc. when the class object is first created.
  - **Destructor**: allows the user to specify what happens to the class variables at the end of the program.
- I won't touch on **protected** members here.

Private vs Public?

- When do you, the programmer, make something **public** or **private**?
  - There is no absolute standard.
  - Common for all members to be **private**, except the constructor, destructor, and user functions.
- As an example, SIN is private.
  - Hair_colour and eye_colour are public.
- Since there is no **public** function that can access SIN, the user will not be able to access and use SIN.

```
class Person{

    private:

        // Have the SIN be a private member.
        int SIN = 123456789;


    public:
        // Hair colour and eye colour are public.
        string hair_colour = "brown";
        string eye_colour = "blue";

}



;
```

Getters and Setters

- Since classes operate with private members it common to see what are called getters and setters.
- Within a class, you are free to update private members with the “.”
- Outside a class however, this will lead to a ‘private within this context error’.
- To remediate this, one can create functions within the class that set, or get, a private member variable.
- Suppose from our previous example, we wanted the user to be able to get and set the SIN number.
  - Within our class, we can simply add the two functions:

```
// Now we want the user to have access to the SIN.
int GetSIN(){ return SIN; }
void SetSIN(int x){ SIN = x; }
```

```cpp
class ComplexNumber
{
    private:
        double x;
        double y;

    public:
        ComplexNumber operator+(ComplexNumber const& num){
            return ComplexNumber(x + num.x, y + num.y);
        }

        ComplexNumber(double x1, double y1){
            x = x1;
            y = y1;
        }

        double GetX(){ return x; }
        double GetY(){ return y; }

        void SetX(double x1){ x = x1; }
        void SetY(double y1){ y = y1; }

};
```

Classes (cont.)

- It is possible to allow classes to access private members of another class.
- To do so, we must give it the **friend** identifier.

```
class Cube{

    private:

        double length = 2.0;
        double width = 1.0;
        double height = 4.0;

    // Forward declare.
    friend class CompanionCube;

};
```

```
class CompanionCube{


    public:
        double ComputeVolume(Cube& obj){
            return obj.width*obj.height*obj.length;
        }

};
```

- Here, we can use the CompanionCube class to compute the volume of our Cube class (make sure you forward declare).

Inheritance

- How could you reuse a set of functions from one class, in another?
  - You could copy and paste code.
  - Or you could inherit these functions.
- To do so, requires making a **derived** class.
- The **derived** class will have access to members of the parent class in accordance to the inheritance access chosen:
  - **Public inheritance:** public stay public, protected remain protected.
  - **Private inheritance:** makes public and protected members private.
  - **Protected inheritance:** makes public and protected members protected.
- In all inheritance schemes, **private** members of the parent class remain private.
  - If you want to access these you need to use getters and setters.

Inheritance (cont.)

- Suppose you have some class for a probability density function (PDF).
- We can use this PDF class as a basis for two new PDF classes.
  - Gaussian and Landau.
- How do we inherit PDF into our new classes?
- First thing is to decide what kind of inheritance we wish to use.
  - The vast majority of times you inherit it will be public.

```
class PDF
{
    private:
        double min;
        double max;

    public:
        double GetMean();
        void SetMin(double x){ min = x; }
        void SetMax(double x){ max = x; }
        double GetMin(){ return min; }
        double GetMax(){ return max; }

};

double PDF::GetMean(){
    return 2.0;
}
```

Inheritance (cont.)

```cpp
class Gaussian : public PDF{

    public:

        Gaussian(){

        }
        Gaussian(double x, double y){
            mu = x;
            sigma = y;
        }


    private:
        // Two parameters for the Gaussian c
        double sigma = 1.0;
        double mu = 4.3;

};
```

```cpp
class Landau : public PDF{

    public:

        Landau(){

        }
        Landau(double x, double y){
            mu = x;
            c = y;
        }

    private:
        // Two parameters for the Landau distribution.
        double c = 2.0;
        double mu = 3.14;

};
```

Constructors/Destructors

- You will often encounter classes that have multiple constructors:

```
TGraph (const TVectorD &vx, const TVectorD &vy);
TGraph (const TVectorF &vx, const TVectorF &vy);
TGraph (Int_t n, const Double_t* x, Double_t* y);
TGraph (Int_t n, const Float_t* x, Float_t* y);
```

- Using our example of **Gaussian** and **Landau** classes defined earlier:

```
class Gaussian : public PDF{

    public:

        Gaussian();
        Gaussian(double, double);


    private:
        // Two parameters for the Gaussian distribution.
        double sigma = 1.0;
        double mu = 4.3;

};
```

```
class Landau : public PDF{

    public:

        Landau();
        Landau(double, double);

    private:
        // Two parameters for the Landau distributi
        double c = 2.0;
        double mu = 3.14;
```

Here, in these constructors we could assign the parameters for each PDF.

Templates

- Often different object types share similar methods or functions.
  - It is also true that certain operations are permissible on many types.
- Suppose you have two classes: **Person** and **Vehicle**.
- These two classes have little in common.
  - The attributes of a vehicle might be EV status, fuel capacity, etc.
  - For class Person, it might be height, weight, age, etc.
- For the parts they do have in common, we can use templates.
- The below methods would update the position or mass of our **Person** and **Vehicle** classes (provided they have these attributes!).

```
template<class T>
void UpdatePosition(T obj, std::vector<double> r){
    obj.UpdateX(r[0]);
    obj.UpdateY(r[1]);
    obj.UpdateZ(r[2]);
}
```

```
template<class T>
void UpdateMass(T obj, double x){
    obj.Mass += x;
}
```

Project

- I have left about ~20 minutes for you to work on a small project to begin exploring how to use C++.
- Write a program that does the following:
  - Create a class that models SNOLAB experiments.
  - Have it model a detector technology (get creative).
  - Store a list of events in a vector, as well as another vector with their energies.
  - Create a vector operator (+) that allows you to sum these two 1xN vectors into a 2xN vector.
  - Create an arbitrary threshold for energy and remove these events from the vector.
- I will be here to help if you get stuck (just raise your hand or unmute).

Conclusions

- Throughout this talk we have established the **basics** of how to write in C++.
  - We looked at the different data types in C++ and how to perform basic numerical operations on them.
  - We established some of the basics for defining your own functions, while also talking about prototyping and overloading.
  - Arrays and vectors were introduced to give you an introduction on where each might be more appropriate or better to use.
  - We discussed classes and structs that will allow you to interpret more complex C++ code and organize your own analysis work.
  - We also covered pointers, templates, inheritance, and the preprocessor.
- I strongly encourage you to all attend the talk next week on ROOT.
  - Between now and then play around with C++ and see if you can begin to write more complex scripts.
  - ROOT is a toolset used by many experiments; it will help to have experience with it.
- If you ever have any questions or comments, feel free to stop by my cubical or reach out via email: remington.hill@queensu.ca.

Thank you for listening! Questions?

Queen's UNIVERSITY