Introduction to C++

Mark Anderson
anderson.mark@queensu.ca

May 7, 2020

Review of Unix commands

pwd	current directory
ls	list files in directory
cd ⟨directory_path⟩	change directory
cp ⟨file_path⟩⟨destination_path⟩	copy a file
mkdir ⟨directory_path⟩	make a directory
rmdir ⟨directory_path⟩	remove a directory
mv ⟨file_path⟩⟨destination_path⟩	move a file
mv ⟨file_path⟩⟨file_path_new⟩	rename a file
rm ⟨file_path⟩	remove a file

Review of Unix commands

- Options can be specified for most commands
- Options can be grouped together
- Use the --help option for most commands to obtain the descriptions of all available options

ls **-l**	detailed list
ls **-a**	all files (including hidden)
ls **-la**	detailed list *and* all files
rm **-i** ⟨file_path⟩	confirmation prompt
rm **-r** ⟨file_path⟩	recursive (delete directories)

Review of Unix commands

- To get all of the available options, most programs implement a `--help` option
  - `ls --help`
  - `mv --help`
  - `rm --help`

- Pretty much all system programs will also have more thorough documentation via the `man` command (for manual) and an even more thorough set of documentation via the `info` command
  - `man` and `info` are documentation systems
  - Try `ssh --help` vs. `man ssh`
  - Try `man ls` vs. `info ls`

- Short options can generally grouped together, while long options can not

Introduction

- A program is a sequence of commands or instructions used to accomplish a task
  - That task might be to make a plot, or solve a system of differential equations
- Three types of errors you are likely to encounter
  - Syntax error
    - While humans can handle syntax errors without spewing out error messages, computers can not
    - The program must be syntactically correct, or it cannot run
    - Even a small mistake (capitalization, missing a semicolon, ...) will return an error message
  - Runtime error
    - Named this way because they occur during the runtime of the program, not the compilation
    - Examples are accessing an element of a list or accessing a file that doesn't exist
  - Semantic error
    - This is the worst type of error!
    - A semantic error is one in which the computer does not actually generate an error message
    - Rather, it is when the computer does something different than you expect
    - Remember, however, that the computer is doing *exactly* what you told it to do!

Introduction to C++

- C++ is an extension of C
  - Provides numerous additions, tools, and features
- Many libraries are written in C++
  - ROOT – which you will likely use a lot this summer
- C++ is an intermediate level language
  - Has components of both high and low level languages
- C++ is object-oriented (although supports multiple programming paradigms)
- C++ is statically-typed
- C++ is compiled (as opposed to interpreted)

Introduction to C++

- More complicated than Python
  - Python is a high-level language
  - Python handles a lot of declarations, memory management, etc.
    - This sacrifices performance
    - May not be an issue depending on your application
  - Python provides many useful functions and libraries to facilitate quick data analysis and prototyping of ideas
    - Very useful for testing ideas
- C++ is more strict than Python
  - Unlike Python, you *must* declare data types when defining variables
  - Unlike Python, you *must* end each statement with a semicolon
  - As a result, it is safer and has better performance

Hello, world!

- Start with a hello world problem, as is usual for these tutorials

```
#include <iostream>

using namespace std;

int main(int argc, char ** argv)
{
  // Print a message.
  cout << "Hello, world!" << endl;
  return 0;
}
```

- Compare that with Python

```
print("Hello, world!")
```

Hello, world!

- This example contains many of the concepts we will discuss today
  - Variables
  - Data types
  - Functions
  - Blocks and scope
  - Namespaces
  - Libraries

```
#include <iostream>

using namespace std;

int main(int argc, char ** argv)
{
  // Print a message.
  cout << "Hello, world!" << endl;
  return 0;
}
```

- First thing to note is that a comment starts with //
  - Anything after the // is ignored by the compiler
- Comments are useful to explain to someone reading the code what is going on, why something is done a certain way, etc.
  - That "someone" could be you several months later!

How do I run that?

- C++ is a compiled language
  - A compiler translates the source code into lower-level code that the machine can read
  - g++ or clang
- To compile the "Hello, world!" example, put the code into a file with the extension ".cc" (or ".cpp") with your favourite editor
- Assuming that we have named the file `hello.cc`, we can compile the program

```
g++ -o run_hello hello.cc
```

- Once the program compiles, we can run it

```
./run_hello
```

The ROOT Interpreter

- There is also a C++ interpreter in ROOT
  - CINT or Cling depending on ROOT version
  - Useful for testing out individual commands
  - Not recommended for anything more than a few lines of simple tests
- The interpreter loads several useful C++ libraries
  - We will see this a bit later in the tutorial
  - For example, can type `cos(3.14159)` and it will work out of the box

```
mark@m:~$ root -l
root [0]
root [0] float pi_5 = 3.14159
(float) 3.14159f
root [1] float 5_pi = 3.14159
ROOT_prompt_1:1:7: error: expected unqualified-id
float 5_pi = 3.14159
      ^
root [2] double pi_5 = 3.14159
(double) 3.1415900
root [3] char c = 'l'
(char) 'l'
root [4] 5*6+7-8
(int) 29
root [5] 5%2
(int) 1
root [6]
root [6]
```

The ROOT Interpreter

- Keep in mind that the interpreter will not catch various errors, or may give different errors than the compiler
- For this tutorial, I would recommend having the interpreter open on a second screen and trying some of the examples yourself
  - Log on to Neutrino with the nuguest account as you did yesterday
  - In the terminal, run `root -l` (the `-l` option is to run without the initial message/banner)
- To quit, use the command `.q` (or `.qqq`, or `.qqqqq`...)
- For help, use the command `.h` or `.?`

Variables and data types

- Address in memory which stores a value
  - First create a variable with a given name (identifier)
  - Assign that variable a value via the assignment (=) operator
  - Accessed via its name (identifier)
- Variables must start with a letter and only contain alphanumeric characters and underscores
- In C++, you must declare the data type when initializing the variable (cannot be changed)
- Common basic types include integers, floats, and booleans
- You *should* (but technically don't have to) initialize the variable when you allocate it

Variables and data types

- Some of the most important primitive types that you will be using
  - `int` 5, 8, 100, 10438
  - `double` 3.14159, 1.0, 564.6673
  - `char` ‘c’, ‘y’, ‘q’, ‘\n’, ‘\t’
  - `bool` `true`, `false`
- We will look into integers and doubles in more detail
- Try some of the commands below in the ROOT interpreter

```
int i = 5;
float pi_5_decimal = 3.14159;
float 5_decimal_pi = 3.14159; //Will fail!
char c = 'c';
bool b = true;
```

Variables and data types (numbers)

- Humans use the decimal (base 10) system to represent numbers
  - Probably because we have ten fingers and ten toes
- Computers use the binary (base 2) system to represent numbers
- Many computing applications use the hexidecimal (base 16) system
- A binary digit is called a bit, and 8 bits are called a byte
  - Fun fact: 4 bits is called a nibble (rarely used term)

Decimal: 10 symbols (0-9), called digits

$$\begin{aligned} 8142 &= 8000 + 100 + 40 + 2 \\ &= 8 \cdot 10^3 + 1 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0 \end{aligned}$$

Binary: 2 symbols (0,1), called bits

$$\begin{aligned} 11001b &= 10000b + 1000b + 000b + 00b + 1b \\ &= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 25 \text{ (in decimal)} \end{aligned}$$

Variables and data types (integers)

(Signed) integers

- `short` $[-2^{15}, 2^{15}-1]$ (16 bits/2 bytes)
- `int` $[-2^{31}, 2^{31}-1]$ (32 bits/4 bytes)
- `long` $[-2^{63}, 2^{63}-1]$ (64 bits/8 bytes)

Unsigned integers

- `unsigned short` $[0, 2^{16}-1]$ (16 bits/2 bytes)
- `unsigned int` $[0, 2^{32}-1]$ (32 bits/4 bytes)
- `unsigned long` $[0, 2^{64}-1]$ (64 bits/8 bytes)

- Be careful with underflow and overflow
- If the result is bigger than the range, it will "loop around" and give unexpected results

```
short i = 32760;
i += 100; //This will equal -32676 (overflow)

short j = -32760;
j -= 100; //This will equal 32676 (underflow)
```

Variables and data types (floating points)

- Floating points are represented as $s \cdot m \cdot 2^e$
  - ***s***: the **sign**, $\pm$
    - Either 0 for positive or 1 for negative
  - ***m***: the **mantissa**
    - The "decimal" portion – only one digit to the left of the dot (as with scientific notation)
    - Normalized – for example, $1234.567 \longrightarrow 1.234567 \cdot 10^3$ in base ten
  - ***e***: the **exponent**

Single-precision (`float`)

- 1 bit for s
- 23 bits for m
- 8 bits for e

Total: 32 bits (4 bytes)

$2^{2^8/2} \longrightarrow \sim[-3 \cdot 10^{38}, 3 \cdot 10^{38}]$

Double-precision (`double`)

- 1 bit for s
- 52 bits for m
- 11 bits for e

Total: 64 bits (8 bytes)

$2^{2^{11}/2} \longrightarrow \sim[-2 \cdot 10^{308}, 2 \cdot 10^{308}]$

Variables and data types (floating points)

- Be careful with precision of floating points
  - Floating point numbers have limited precision and will be rounded
  - Operations may produce results you wouldn't expect! (well, now you will expect it)
- Machine precision / machine ε
  - `float` $\varepsilon = 1/2^{23} \approx 1.19 \cdot 10^{-7}$
  - `double` $\varepsilon = 1/2^{52} \approx 2.22 \cdot 10^{-16}$
- Always ensure that floating point precision and machine epsilon will work for you

Rounding/cancellation

```
float  pi1 = 3.14159265358979;
double pi2 = 3.14159265358979;

pi1 - 3.1415926; // 1.410e-7
pi2 - 3.1415926; // 5.359e-8
```

Absorption

```
float  y1;
double y2;

y1=1+1e-8; // 1.0000000000000000
y2=1+1e-8; // 1.0000000099999999
```

Variables and data types (floating points)

- For reasons of finite precision, it is dangerous to compare floating points with equality (==)

Equality comparison: bad

```
float a = 1.0/3.0;

if(a*3.0 == 1)
{
  // Expect this to run.
}
else
{
  // Uh oh! This might
  // run instead.
}
```

Comparing absolute error: still bad

```
float a = 1.0/3.0;
abs(a*3.0 - 1) < 1e-7 // True.

float b = 1e8/3.0;
abs(b*3.0 - 1e8) < 1e-7 // False.
```

Comparing relative error: better

```
float a = 1.0/3.0;
abs((a*3.0 - 1.0)/1.0) < 1e-7) // True.

float b = 1e8/3.0;
abs((b*3.0 - 1e8)/1e8) < 1e-7) // True.
```

Variables and data types

- Other useful primitive types include
  - Booleans (either true or false)
  - Characters ('s', '4', 'n')
- C++ has various more advanced data types in the standard library
  - Strings, vectors, complex numbers, ...
  - More on this a bit later
- Type modifiers can change the behaviour of a variable
  - `const` ensures that the variable value cannot change (will give compiler error)
    - Very good practice to use `const` when you don't want/expect the value to change
    - Can prevent annoying bugs
  - `static` ensures that the variable is declared once in the program
    - Allocates memory for entirety of the program
    - Useful for keeping track of the state of a function or number of functions calls
    - Useful if looking for condition that only needs to be met once

Operators

Arithmetic Operators		Comparison Operators		Logical Operators	
+	addition	>	greater than	&&	and
-	subtraction	<	less than	||	or
*	multiplication	>=	greater than or equal	!	not
/	division	<=	less than or equal		
~~**~~	~~exponential~~	==	equals		
%	modulo	!=	does not equal		

- Arithmetic operators have the same rules for precedence as is standard in math
- Comparison and logical operators require numbers or booleans on both sides and return a boolean

Operators

Compound Assignment Operators

+=	addition
-=	subtraction
*=	multiplication
/=	division
%=	modulo

Increment/Decrement Operators

++	increment by 1
--	decrement by 1

```
int x=0;

// All of the below do the same.
x=x+1; // x is now 1.
x+=1; // x is now 2.
x++; // x is now 3.
```

- The increment/decrement operator can go on either side of the variable
- This matters if returning the value

```
int x=0;
int y;

y = x++; // y==0, postcrement
y = ++x; // y==2, precrement
```

Blocks, scope, and conditions

- Blocks in C++ are surrounded by curly braces, { }
- Blocks can be nested within other blocks
- Any variables defined in a block are only valid until the end of the block
  - The memory allocated at their declaration is released at the }

```
{
  int i = 0;
  {
    short s = 35;
  } // Memory released for s

  i += s; // Trying to access s here will give an error

} // Memory released for i
```

Blocks, scope, and conditions

- Can declare variables with the same name in separate blocks
  - This includes nested blocks
    - Uses first instance found when traversing upwards through the nested blocks
  - Not advised to use same names in nested blocks as that is extremely prone to error

```
{
  int m;
  int block1 = 1;
  {
    int block1 = 2;
    m = block1 + 1; // Evaluates to 3
  }
  m = block1 + 1; // Evaluates to 2
}
```

Blocks, scope, and conditions

- A condition can be defined to control the flow of the program
- The block following the condition is executed if the condition is met

```
int i = 0;
if (i > 0)
{
  i += -1;
}
else if (i < 0)
{
  i += 1;
}
else
{
  i += 80;
}
```

- Can also use the switch syntax for simple comparisons

```
int i = 0;
switch(i)
{
  case 0:
    // Do something...
    break;
  case 1:
    // Do something else...
    break;
  default:
    //If none of the cases apply...
}
```

Iteration

- Can iterate in several ways
  - `for(initialization; condition; increment){ }`
  - `while(condition){ }`
    - Enters the block if `condition` is met
  - `do{ } while(condition)`
    - Executes the block, then checks `condition` and will repeat until `condition` not met

```
// Factorial
// with for loop
int n = 1;
for (int i=1; i<10; i++)
{
  n *= i;
}
```

```
// Factorial
// with while loop
int n = 1;
int i = 1;
while (i<10)
{
  n *= i;
  i++;
}
```

Iteration

- To finish current *iteration*, use the keyword `continue`
- To finish current *loop*, use the keyword `break`
- Both keywords only apply to the current loop (so if nested, will go back to parent)

```
// Example of break.
int n = 1;
for (int i=1; i<20; i++)
{
  if ( n > 21474836 )
  {
    break;
  }

  n *= i;
} // Goes here when break called
```

```
// Example of continue.
int n = 0;
for (int i=1; i<200; i++)
{
  // Add even numbers only.
  if ( i%2 != 0 )
  {
    continue;
  } // Below is ignored if continue
  n += i;
}
```

Functions

- Defined operation that executes a sequence of statements
- Very useful to avoid constantly copying and pasting code
- Should execute a single specific task
- Can take arguments that are passed to the function and used internally
- Must specify the return type
  - Can return `void`, which is essentially returning nothing

```
// Cube of a number.
double cube(double x)
{
  return x*x*x; // In older versions of ROOT, can do x**3 (but don't)
}

cube(9.0) // Will return 729.0 (since it is a double).
```

The main function

- Each program should have one (and only one) main function
- The main function is the first to be called
  - Functions can be called from within main to define a program
- Can be defined with or without arguments
- Convention to return 0 if no failures in the program
  - Anything else indicates error (and perhaps the specific type of error)

```
int main(){
  // Code to be called goes here.
  // Can call other functions,
  // define variables, ...
  return 0;
}
```

The main function

- Can provide the main function with command line arguments
  - argc is the number of arguments
  - argv is the array of the arguments (first element is the program name)

```
int main(int argc, char **argv){
  // Code to be called goes here.
  // Can call other functions,
  // define variables, ...
  return 0;
}
```

```
int main(int argc, char *argv[]){
  // Code to be called goes here.
  // Can call other functions,
  // define variables, ...
  return 0;
}
```

Header files

- Non-trivial programs will typically consist of many files
- Functions and variables must be declared before being used
- Declaring each function before use in each file quickly becomes tedious and error prone
- C++ uses header files to contain all declarations of functions defined in the source file
  - Functions can be reused by including the header file (typical extensions are ".hh" or ".hpp")

Header file: f.hh

```
// Declaration.
// Note the semicolon!
void func1(int x, int y, int z);
```

Source file: f.cc

```
#include "cube.hh"

// Definition.
void func1(int x, int y, int z)
{
  // Program goes here.
}
```

Header files

- Note the use of `#include` – this is a pre-compiler directive
  - This step is done before compilation (as the name suggests)
  - `include` copies and pastes the contents of the file into the program

- Two options for including: `#include "f.hh"` or `#include <f.hh>`
  - `#include <f.hh>` searches for "f.hh" in the entire include path of the compiler
  - `#include "f.hh"` searches first in the current directory, then acts as `#include <f.hh>`

- Useful to note that the include path can be specified as an option to the compiler

```
g++ -o run_hello -I/path/to/header hello.cc
```

Namespaces

- Namespaces are used to group together functions, classes, variables
  - This defines the scope of these things
  - Similar to blocks
  - Useful to prevent accidentally sharing function or variable names from external programs
- By default, functions and variables in header files are in the global namespace
  - Available everywhere where the header files are included

```
namespace my_functions
{
  void func1(int x, int y, int z)
  {
    // Program goes here.
  }
}

my_functions::func1(3,65,999); // Note the namespace prefix here.
```

Namespaces

- Can use the keyword `using` to avoid the namespace prefix
  - Compiler knows that functions used are in that namespace, so the prefix in unnecessary
  - Use with care
    - Definitely do not use `using` in header files, as any file that includes the header will also have the `using` instruction – this can cause issues
    - Even in source files, limit usage if you can – namespaces make code clearer

```
using namespace my_functions;

func1(3,65,999); // Note the lack of the namespace prefix here.
```

Back to "Hello, world"

- Let's revisit the "Hello, world!" example with this new information
- How does the compiler know what `cout` and `endl` are when they haven't been defined anywhere?
  - From the `include` statement at the top
- Second line tells the compiler that variables and functions are in the `std` namespace

```
#include <iostream>

using namespace std;

int main(){
  // Print a message.
  cout << "Hello, world!" << endl;
  // This would be required without the using instruction.
  std::cout << "Hello, world!" << std::endl;
  return 0;
}
```

The C++ standard library

- The C++ standard library is a collection of useful classes and functions
  - Containers (set, vector, list, map)
  - Strings and complex numbers
  - Common algorithms
  - Input/output
  - Streams for files and strings

- Don't reinvent the wheel!
  - If you need an algorithm, a mathematical function, a container, or anything like that, check the standard library

Streams

- Streams are for input (`istream`) and output (`ostream`)
- Streams can be written to (« operator) and/or read from (» operator) (depends on type)
- `cout` is a predefined variable which access the standard output (prints to the screen)
- Similarly, `cin` is a predefined variable which accesses standard input (from the keyboard)

```
#include <iostream>

// Output stream.
cout << "Hello, world!" << endl;
cout << 55 << "hi" << 'r' << 3.14159 << endl;

//Input stream.
int x;
cout << "Please enter a number." << endl;
cin >> x;
```

Streams

- Streams can also be used to read and write to files

```
#include <fstream>

// Output file stream.
ofstream ofile("output.txt");
ofile << 5 << endl; // Write integer to file.
ofile.close();

//Input stream.
ifstream ifile("output.txt");
int x;
ifile >> x; // Read the contents into x.
ifile.close();
```

Containers

- Several container types are implemented in the standard library
  - `vector` we will discuss this in more detail on the next slide
  - `map` container that is indexed by a unique key to obtain value (associative array)
  - `set` container for unique elements
  - `queue` container for inserting elements in one end and extracting from another (FIFO)
  - `stack` container for inserting and extracting elements from one end only (LIFO)
- The containers are template types
  - Containers can contain any data type
- Containers will all have the following function methods

`insert(x)`	insert element
`clear()`	remove all elements
`size()`	return number of elements
`empty()`	return boolean indicating whether or not the container is empty

Vectors

- A vector is a dynamic array (can change size)
  - If you want a fixed-length vector (e.g., for performance or safety), use `valarray`
- Some of its functions are shown below (in addition to those on the last slide)

```
#include <vector>

vector<int>    vector1; //Vector containing elements of type int
vector<double> vector2; //Vector containing elements of type double

vector1.push_back(1); // Insert 1 at the end. {1}
vector1.push_back(2); // Insert 2 at the end. {1,2}

vector1.resize(5); // Set length to 5 (pad with zeros). {1,2,0,0,0}

vector1.pop_back(); // Remove the last element. {1,2,0,0}
vector1[1]; // Return element at index 1, which is 2 in this case.
```

Strings

- C strings are `char` arrays terminated by null
  - C strings are ugly and annoying to deal with
- Luckily, the C++ standard library provides the `string` class
  - Much easier to deal with
- If you need C strings for whatever reason, use the `c_str()` method
  - For example, ROOT typically requires C strings as input

```
#include <string>

string str1("Hello");
string str2("world");
string str3; // Empty string.

// Instantiate ROOT object which requires C string.
TH1D h1(str1.c_str(), str2.c_str());
```

Strings

- Comparison (==, !=) and concatenation (+, +=) operator also implemented for strings

```
#include <string>

string str1("Hello");
string str2("world");
string str3;

if(str2 == "world")
{
  str3 = str1 + ", " + str2;
}
else
{
  str3 = str1 + "!";
}
```

Strings

- Some important methods include

`find(substring)`	returns the position (zero-indexed) of first instance of `substring`
`rfind(substring)`	returns the position (zero-indexed) of last instance of `substring`
`substr(pos, n)`	returns the substring from index `pos` to `pos+n`

```
#include <string>

// Can also initialize as: string str1 = "hello_hello_hello"
string str1("hello_hello_hello");

str1.find("hello"); // Will return 0.
str1.rfind("hello"); // Will return 12 (i.e., start of 3rd "hello").

str1.substr(12,5); // Will return "hello" (specifically, the 3rd one).
```

Other useful aspects

- `cmath` or `math.h`
  - Includes trigonometric functions, exponential and logarithmic functions, absolute value, constants, ...
- `limits`
  - Information about types (e.g., minimum and maximum value of type `int`) for your platform
- `complex`
  - Complex numbers and functions to operate on them
- `random`
  - For random number generation

Objects and classes

- Class
  - Can define a custom data type
  - Blueprint/details to define properties and behaviour that an object should have
  - Contains data (properties) and functions (behaviour)
    - Data: data members, attributes
    - Functions: methods, function methods

- Object
  - Specific instance created from the class (blueprint)
  - General structure of every object is the same
  - Properties may be different

Objects and classes: cars

Common properties (data)

- Make
- Model number
- Colour
- Year
- Mass
- ...

Common functions (behaviour)

- Start engine
- Stop engine
- Increase speed
- Apply brakes
- Turn steering wheel
- ...

- The above properties might be used to construct a class called "Car"
- An object of type "Car" might be a 2007 Blue Honda Civic

Classes in C++

- Classes can have data members and/or methods (functions)
  - If a data member or function is *private*, it can only be accessed/called from within the class
  - If a data member or function is *public*, it can be accessed from outside of the class
    - It is bad practice for data members to be public (use getters and setters to be explicit)
    - Not the case for methods – this will depend though
  - If a data member or function is *protected*, it can be accessed by classes that inherit from it
- In C++, structs are classes where all data members and methods are *public*
  - More typical to use structs if only storing data members
  - Note that C and C++ structs are not the same
- Classes have two methods that must exist
  - The constructor: runs when the object is instantiated
    - Used to initialize data members
    - Has the same name as the class itself
  - The destructor: runs when the object is destroyed
    - Frees allocated memory (if using pointers, must use `delete` here – we will get to pointers)
    - Has the same name as the class with a tilde ($\sim$) prefix

Creating a class in C++: a rectangle

Header: Rectangle.hh

```
class Rectangle
{
  public:
    // Constructor.
    Rectangle(double length=1.0,
              double width=1.0);

    // Destructor.
    ~Rectangle();

    double area();

  private:
    double fLength;
    double fWidth;
};
```

Source: Rectangle.cc

```
#include "Rectangle.hh"

Rectangle::Rectangle
(double length,
 double width)
{
  fLength = length;
  fWidth = width;
}

double Rectangle::area()
{
  return fLength*fWidth;
}
```

Some remarks about C++ classes

- It is convention to prefix all private data members with an 'f'
  - Allows someone reading the code to immediately infer that the variable is a data member
  - This is not enforced by the compiler or anything, but your experiment's code will likely do this and I strongly recommend adopting a similar convention for your own code
- Declarations of the class, methods, and data members are done in the header file
  - Allows for the class to easily be used in other files
- Definitions/implementations of the function methods is done in the source file
  - This is what gets compiled
- Default arguments of function methods, if used, should be specified in the declaration (i.e., the header file)

Object oriented programming

1. Abstraction
   - Show relevant details and hide the inner workings (e.g., you can drive a car without understanding its inner workings)
2. Encapsulation
   - Protect data from being unintentionally modified (e.g., you cannot access the internal parts of a computer easily except via the I/O ports, as intended)
3. Inheritance
   - Extend blueprint to avoid rewriting code (e.g., a class which inherits from general cars might be trucks, with additional properties like 4WD...)
4. Polymorphism
   - Do same operation with a different input (e.g., a function could either add to integers of concatenate two strings depending on the input type)

Pointers

- Just another data type to store a value
  - int stores an integer
  - double stores a floating point number of double precision (2*32=64)
  - string stores a string of characters
  - A pointer stores an address in memory
- Like other data types
  - A pointer variable can be declared
  - Pointers have operations
- Pointers tell you *where* something is
- Variables tell you *what* something is

Pointers

- Address of a variable can be accessed via the *reference* operator (&)
- Value at address can be accessed via the *dereference* operator (*)
  - A little confusing since * is also used to declare a pointer
  - Keep that in mind
- For variables, the dot is used for function methods and attributes of classes
- For pointers, the arrow is used for function methods and attributes of classes
  - The arrow can be though of as a dereference operator followed by the dot

```
//Variables
histogram.Draw()

//Pointers
histogram->Draw()
(*histogram).Draw() //equivalent to simply using ->
```

Pointers

- The following example illustrates the concept of addresses and the reference operator
- Full example in the tutorial folder

```
int i = 3;
cout << "The value of i is: " << i << endl;
//The value of i is: 3

cout<< "The address of i is: " << &i << endl;
//The address of i is: 0x1001054a0
//Note that this address will change
//depending on the computer
//It will still be in this format
```

Pointers

- The following example illustrates the difference in initializing a variable and a pointer

```
//Initialize a variable of type TH1D (a ROOT histogram object).
//Call the constructor to create the object.
TH1D hist_var = TH1D();

//Initialize a pointer variable of type TH1D.
//Allocate memory (new).
//Call the constructor to create the object.
//Fill the allocated memory with that object.
TH1D * hist_ptr = new TH1D();
```

What's the point of pointers?

- When passed to a function, variables are copied
  - A new variable is instantiated internally (in scope of function)
  - Any modifications to the variable in the function are local
  - Copying can be expensive if the object is large
- Instead, can pass a pointer to a function
  - The function is passed the address and can thus modify the value stored at the address
  - Operations can change the value
  - Generally, it is cheaper to copy a pointer (4 or 8 bytes) and dereference it than to copy the entire object around (could be many times larger)
    - No need for pointers of primitive data types like int, float, double
    - With modern C++compilers, copying isn't terribly expensive any more
  - Moving around pointers rather than the object itself is often favourable

Some remarks about pointers

- "If you don't understand pointers, don't use them."
  - someone???
- Pointers can be misused fairly easily
  - For most applications, using regular variables is totally fine
  - If in doubt, either ask someone or just use regular variables
- Passing by reference is often a good alternative
  - If the variable is not to be modified, use the `const` type modifier!
- For every instance of `new`, there should be an instance of `delete`
  - If you use the `new` keyword, *you* are responsible for memory management – ultimately for freeing the memory

Bonus: using precompiler directives to obfuscate code

- Can use precompiler directive `#define` to create macros
  - Compiler will substitute all instances of the identifier (first string) with the token string (second string)
  - Can use for more than just strings
    - Do mathematical operations, generate random numbers, ...
- This code is perfectly valid in C++ ...
  - ... but I would not recommend it

```
#include <iostream>
#define e using
#define ee namespace
#define eee std
#define eeee ;
#define eeeee int
#define eeeeee main
#define eeeeeee (
#define eeeeeeee )
#define eeeeeeeee while
#define eeeeeeeeee true
#define eeeeeeeeee {
#define eeeeeeeeeee }
#define eeeeeeeeeeee cout
#define eeeeeeeeeeeee cerr
#define eeeeeeeeeeeeee <<
#define eeeeeeeeeeeeeee 'e'
#define eeeeeeeeeeeeeeee return

e ee eee eeee
eeeee eeeeee eeeeeee eeeeeeee
eeeeeeeeee
eeeeeeeee eeeeeee eeeeeeeee eeeeeeee
eeeeeeeeee
eeeeeeeeeeee eeeeeeeeeeeeee eeeeeeeeeeeeee eeee
eeeeeeeeeeeee eeeeeeeeeeeeee eeeeeeeeeeeeee eeee
eeeeeeeeeee
eeeeeeeeeeeeeeee eeeeeeeeeeeeee eeee
eeeeeeeeeee
```

For tomorrow...

- I will be giving an introduction to Git and GitHub
- Make a GitHub account (if you do not already have one)
  - `https://github.com/`
- Choose an appropriate user name
  - You will likely use this account in the future (perhaps throughout your career)
  - I suggest you make it related to your name
- Take a few moments to familiarize yourself with the website

References and further resources

- There is lots of material not covered here
- Much of the material in these slides is from
  - `http://www.hlnum.org/publications/cppscicomp.pdf`
  - `https://kriemann.name/Ronald/publications/cppscicomp.pdf`
- Google and Stack Overflow are your friends
  - Often people have similar questions
- Documentation can be confusing at first, but is informative
  - `https://en.cppreference.com/w/`
- Books from the creator of C++ himself, Bjorn Stroustrup
  - Programming: Principles and Practice Using C++ for complete beginners
  - A Tour of C++ for people with previous programming experience