Getting started with the training room terminals

- Log on with your BU username
  - If you don’t have a BU username:
  - Username: Choose tutm1-tutm18, tutn1-tutn18
  - Password: on the board.

- On the desktop is a link to MobaXterm. Double click to open it.
Getting started with the training room terminals

- Click the + sign at the top of the window and enter:

  ```
  ssh username@scc2.bu.edu
  ```

- Use your SCC account if you have one. If not, see the board for a username and password.

- When prompted enter the password.
Getting started on the SCC

- Load the Eclipse module:
  
  $ module load eclipse/2019-06

- Enter this command to create a directory in your home folder and to copy in tutorial files:
  
  `/scratch/intro_to_cpp.sh`
Run the Eclipse software

- Enter this command to start up the Eclipse development environment.
  
  `eclipse &`

- When this window appears just click the Launch button:
Run the Eclipse software

- When this window appears just leave it be for now.
Tutorial Outline: All 4 Parts

- Part 1:
  - Intro to C++
  - Object oriented concepts
  - Write a first program

- Part 2:
  - Using C++ objects
  - Standard Template Library
  - Basic debugging

- Part 3:
  - Defining C++ classes
  - Look at the details of how they work

- Part 4:
  - Class inheritance
  - Virtual methods
  - Available C++ tools on the SCC
Tutorial Outline: Part 1

- Very brief history of C++
- Definition object-oriented programming
- When C++ is a good choice
- The Eclipse IDE
- Object-oriented concepts
- First program!
- Some C++ syntax
- Function calls
- Create a C++ class
Very brief history of C++

1962
Simula I was invented by Kristen Nygaard and Ole-Johan Dahl as a simulation language

1967
Simula 67 developed as the first object-oriented language

1969-1973
The C language was invented by Dennis Ritchie at Bell Labs

1972
D. Ritchie and Ken Thompson re-write the Unix OS in C

1979
Bjarne Stroustrup began developing "C with Classes"

1983
"C with Classes" renamed to C++

1985
1st commercial C++ compiler, Cfront, released by AT&T

2014
Minor update: C++14 released.

2011
Major update: C++11 standard released

1989
C++ 2.0 standard released.

For details more check out A History of C++: 1979–1991
Object-oriented programming

- Object-oriented programming (OOP) seeks to define a program in terms of the *things* in the problem:
  - files, molecules, buildings, cars, people, etc.
  - what they need to be created and used
  - what they can do

- Internal data:
  - Balance
  - Transaction record
  - Name of owner

- Functions defined for this class:
  - Add money
  - Withdraw money
  - Get balance

```pseudo-code```
```java
class Account

Accounts my_account;

dollars = my_account.withdraw(10.00);
print(my_account.get_balance());
```
```

Objects (instances of a class)

"pseudo-code"
Object-oriented programming

- OOP defines *classes* to represent these things.
- Classes can contain data and methods (internal functions).
- Classes control access to internal data and methods. A *public* interface is used by external code when using the class.
- This is a highly effective way of modeling real world problems inside of a computer program.
Characteristics of C++

- C++ is...
  - Compiled.
    - A separate program, the compiler, is used to turn C++ source code into a form directly executed by the CPU.
  - Strongly typed and unsafe
    - Conversions between variable types must be made by the programmer (strong typing) but can be circumvented when needed (unsafe)
  - C compatible
    - call C libraries directly and C code is nearly 100% valid C++ code.
  - Capable of very high performance
    - The programmer has a very large amount of control over the program execution, compilers are high quality.
  - Object oriented
    - With support for many programming styles (procedural, functional, etc.)
  - No automatic memory management (mostly)
    - The programmer is in control of memory usage

“Actually I made up the term ‘object-oriented’, and I can tell you I did not have C++ in mind.”
- Alan Kay (helped invent OO programming, the Smalltalk language, and the GUI)
When to choose C++

- Despite its many competitors C++ has remained popular for ~30 years and will continue to be so in the foreseeable future.

- Why?
  - Complex problems and programs can be effectively implemented
    - OOP works in the real world.
  - No other language quite matches C++'s combination of performance, libraries, expressiveness, and ability to handle complex programs.

Choose C++ when:

- Program performance matters
  - Dealing with large amounts of data, multiple CPUs, complex algorithms, etc.

- Programmer productivity is less important
  - You'll get more code written in less time in a language like Python, R, Matlab, etc.

- The programming language itself can help organize your code
  - In C++ your objects can closely model elements of your problem
  - Complex data structures can be implemented

- Access to a vast number of libraries

- Your group uses it already!

“If you’re not at all interested in performance, shouldn’t you be in the Python room down the hall?”
— Scott Meyers (author of Effective Modern C++)
Eclipse [https://www.eclipse.org](https://www.eclipse.org)

- In this tutorial we will use the Eclipse integrated development environment (IDE) for writing and compiling C++

- About Eclipse
  - Started in 2001 by IBM.
  - The Eclipse Foundation (2004) is an independent, non-profit corporation that maintains and promotes the Eclipse platform.
  - Cross-platform: supported on Mac OSX, Linux, and Windows
  - Supports numerous languages: C++, C, Fortran, Java, Python, and more.

- A complex tool that can be used by large software teams.
IDE Advantages

- Handles build process for you
- Syntax highlighting and live error detection
- Code completion (fills in as you type)
- Creation of files via templates
- Built-in debugging
- Code refactoring (ex. Change a variable name everywhere in your code)
- Much higher productivity compared with plain text editors!
  - …once you learn how to use it.

IDES available on the SCC

- Eclipse (used here)
- geany – a minimalist IDE, simple to use
- Netbeans – used in past C++ tutorials. Simpler than Eclipse but still capable.
- Spyder – Python only, part of Anaconda
- Emacs – The one and only.

Some Others

- Xcode for Mac OSX
- Visual Studio for Windows
- Visual Studio Core plus plugins
- Code::Blocks (cross platform)
A first program

- Click *Create a new C++ project* in the Eclipse window.
A first program

- For a project name use *hello_world*
- Choose a *Hello World C++ Project* and the *Linux GCC* toolchain.
- This version of Eclipse is their “IDE for Scientific Computing” package.
- Then click the Next button.
A first program

- Add your name
- Everything else can stay the same.
- Click Next.
A first program

- Last screen. Don’t change anything here, just click Finish.
A first program

- Now click the *Workbench* button in the welcome screen to go to the newly created project.
- `hello_world.cpp` has been auto-generated.

- Under the Project menu select the `Build Project` option.

- Then click the Run button:
Behind the Scenes: The Compilation Process

header files
- iostream.h
- my_header.h

main.cpp

C++ preprocessor
- * Expanded source code file
- * not normally visible
- * g++ -E to see output

C++ compiler
- *

C++ library files
- system library files

Object code file
- main.o

assembler
- *
- * Assembler code file
- * not normally visible
- * g++ -S to see output

linker

Executable
- main

```
g++ -o main main.cpp
```
Hello, World! explained

The main routine – the start of every C++ program! It returns an integer value to the operating system and (in this case) takes arguments to allow access to command line arguments.

The two characters // together indicate a comment that is ignored by the compiler.

The return statement returns an integer value to the operating system after completion. 0 means “no error”. C++ programs must return an integer value.
Hello, World! explained

- loads a header file containing function and class definitions
- Loads a namespace called std.
- Namespaces are used to separate sections of code for programmer convenience. To save typing we'll always use this line in this tutorial.

```cpp
#include <iostream>
using namespace std;

int main() {
    // prints !!!Hello World!!!
    cout << "!!!Hello World!!!" << endl;
    return 0;
}
```

- `cout` is the object that writes to the stdout device, i.e. the console window.
- It is part of the C++ standard library.
- Without the “using namespace std;” line this would have been called as `std::cout`. It is defined in the `iostream` header file.
- `<<` is the C++ insertion operator. It is used to pass characters from the right to the object on the left.
- `endl` is the C++ newline character.
Header Files

- C++ (along with C) uses header files as to hold definitions for the compiler to use while compiling.
- A source file (file.cpp) contains the code that is compiled into an object file (file.o).
- The header (file.h) is used to tell the compiler what to expect when it assembles the program in the linking stage from the object files.
- Source files and header files can refer to any number of other header files.
- When compiling the linker connects all of the object (.o) files together into the executable.
Make some changes

- Let’s put the message into some variables of type `string` and print some numbers.
- Things to note:
  - Strings can be concatenated with a `+` operator.
  - No messing with null terminators or `strcat()` as in C
- Some string notes:
  - Access a string character by brackets or function:
    - `msg[0]` → “H” or `msg.at(0)` → “H”
    - C++ strings are `mutable` – they can be changed in place.
- Re-run and check out the output.

```cpp
#include <iostream>
using namespace std;

int main() {
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;
    return 0;
}
```
A first C++ class: string

- string is not a basic type (more on those later), it is a class.
- string hello creates an instance of a string called hello.
- hello is an object. It is initialized to contain the string “Hello”.
- A class defines some data and a set of functions (methods) that operate on that data.

```cpp
#include <iostream>
using namespace std;

int main() {
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;
    return 0;
}
```
A first C++ class: string

- Update the code as you see here.
- After the last character is entered Eclipse will display a large number of methods defined for the msg object.
- If you click or type something else just delete and re-type the trailing period.

```cpp
#include <iostream>
using namespace std;

int main() {
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;
    msg.
    return 0;
}
```
A first C++ class: `string`

- Start typing the word `size` until it appears in the menu.
- Hit the Enter key to accept it.
- Now hover your mouse cursor over the `msg.size()` code and a help window will pop up.
A first C++ class: `string`

- Tweak the code to print the number of characters in the string, build, and run it.
- `size()` is a `public` method, usable by code that creates the object.
- The internal tracking of the size and the storage itself is `private`, visible only inside the string class source code.

```cpp
#include <iostream>

using namespace std;

int main()
{
    string hello = "Hello";
    string world = "world!";
    string msg = hello + " " + world;
    cout << msg << endl;
    msg[0] = 'h';
    cout << msg << endl;

    cout << msg.size() << endl;
    return 0;
}
```

- `cout` prints integers without any modification!
Break your code.

- Remove a semi-colon. Re-compile. What messages do you get from the compiler and Eclipse?

- Fix that and break something else. Capitalize *string* → *String*

- C++ can have elaborate error messages when compiling. Experience is the only way to learn to interpret them!

- Fix your code so it still compiles and then we’ll move on…
Basic Syntax

- C++ syntax is very similar to C, Java, or C#. Here’s a few things up front and we’ll cover more as we go along.
- Curly braces are used to denote a code block (like the main() function):

```
{ ... some code... }
```

- Statements end with a semicolon:

```
int a;
a = 1 + 3;
```

- Comments are marked for a single line with a // or for multiline with a pair of /* and */:

```
// this is a comment.
/* everything in here
   is a comment */
```

- Variables can be declared at any time in a code block.
Functions are sections of code that are called from other code. Functions always have a return argument type, a function name, and then a list of arguments separated by commas:

```c
int add(int x, int y) {
    int z = x + y;
    return z;
}
```

A `void` type means the function does not return a value.

Variables are declared with a type and a name:

```c
int x = 100;
float y;
vector<string> vec;
```

// Sometimes types can be inferred in C++11
```c
auto z = x;
```
A sampling of arithmetic operators:

- Arithmetic: $+$, $-$, $\ast$, $/$, $\%$, $\mathbf{++}$, $\mathbf{--}$
- Logical: $\&\&$ (AND), $\|\|$(OR), $\neg$ (NOT)
- Comparison: $==$, $>$, $<$, $\geq$, $\leq$, $\neq$

Sometimes these can have special meanings beyond arithmetic, for example the “+” is used to concatenate strings.

What happens when a syntax error is made?

- The compiler will complain and \textbf{refuse} to compile the file.
- The error message \textit{usually} directs you to the error but sometimes the error occurs before the compiler discovers syntax errors so you hunt a little bit.
Built-in (aka primitive or intrinsic) Types

- “primitive” or “intrinsic” means these types are not objects.
  - They have no methods or internal hidden data.
- Here are the most commonly used types.
- Note: The exact bit ranges here are **platform and compiler dependent**!
  - Typical usage with PCs, Macs, Linux, etc. use these values
  - Variations from this table are found in specialized applications like embedded system processors.

<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>unsigned char</td>
<td>8-bit integer</td>
</tr>
<tr>
<td>short</td>
<td>unsigned short</td>
<td>16-bit integer</td>
</tr>
<tr>
<td>int</td>
<td>unsigned int</td>
<td>32-bit integer</td>
</tr>
<tr>
<td>long</td>
<td>unsigned long</td>
<td>64-bit integer</td>
</tr>
<tr>
<td>bool</td>
<td></td>
<td>true or false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32-bit floating point</td>
</tr>
<tr>
<td>double</td>
<td>64-bit floating point</td>
</tr>
<tr>
<td>long long</td>
<td>128-bit integer</td>
</tr>
<tr>
<td>long double</td>
<td>128-bit floating point</td>
</tr>
</tbody>
</table>

http://www.cplusplus.com/doc/tutorial/variables/
Need to be sure of integer sizes?

- In the same spirit as using \texttt{integer(kind=8)} type notation in Fortran, there are type definitions that exactly specify exactly the bits used. These were added in C++11.
- These can be useful if you are planning to port code across CPU architectures (ex. Intel 64-bit CPUs to a 32-bit ARM on an embedded board) or when doing particular types of integer math.
- For a full list and description see: \url{http://www.cplusplus.com/reference/cstdint/}

```
#include <cstdint>
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>uint8_t</td>
<td>8-bit integer</td>
</tr>
<tr>
<td>int16_t</td>
<td>uint16_t</td>
<td>16-bit integer</td>
</tr>
<tr>
<td>int32_t</td>
<td>uint32_t</td>
<td>32-bit integer</td>
</tr>
<tr>
<td>int64_t</td>
<td>uint64_t</td>
<td>64-bit integer</td>
</tr>
</tbody>
</table>
Reference and Pointer Variables

- Variable and object values are stored in particular locations in the computer’s memory.
- Reference and pointer variables **store the memory location of other variables**.
- Pointers are found in C. References are a C++ variation that makes pointers easier and safer to use.
- More on this topic later in the tutorial.

```cpp
string hello = "Hello";
string *hello_ptr = &hello;
string &hello_ref = hello;
```

- The object `hello` occupies some computer memory.
- A **pointer** to the `hello` object string. `hello_ptr` is assigned the memory address of object `hello` which is accessed with the “&” syntax.
- `hello_ref` is a **reference** to a string. The `hello_ref` variable is assigned the memory address of object `hello` automatically.
Type Casting

- C++ is strongly typed. It will auto-convert a variable of one type to another where it can.

```cpp
short x = 1;
int y = x;  // OK
string z = y;  // NO
```

- Conversions that don’t change value work as expected:
  - increasing precision (float → double) or integer → floating point of at least the same precision.

- Loss of precision usually works fine:
  - 64-bit double precision → 32-bit single precision.
  - But…be careful with this, if the larger precision value is too large the result might not be what you expect!
Type Casting

- C++ allows for C-style type casting with the syntax: `(new type) expression`

```c++
double x = 1.0;
int y = (int) x;
float z = (float) (x / y);
```

- But when using C++ it’s best to stick with deliberate type casting using the 4 different ways that are offered…

Type Casting

- **static_cast<new type>( expression )**
  - This is exactly equivalent to the C style cast.
  - This identifies a cast at compile time.
  - This makes it clear to another programmer that you really intended a cast that reduces precision (ex. double → float) even if it would happen automatically.
  - ~99% of all your casts in C++ will be of this type.

- **dynamic_cast<new type>( expression)**
  - Special version where type casting is performed at runtime, only works on reference or pointer type variables.
  - Usually created automatically by the compiler where needed, rarely done by the programmer.

```cpp
double d = 1234.56;
float f = static_cast<float>(d);
// same as
float g = (float)d;
// same as
float h = d;
```
Type Casting – rarely used versions

- `const_cast<new type>( expression )`
  - Variables labeled as `const` can’t have their value changed.
  - `const_cast` lets the programmer remove or add `const` to reference or pointer type variables.
  - If you need to do this, you probably want to re-think your code!

- `reinterpret_cast<new type>( expression )`
  - Takes the bits in the expression and re-uses them `unconverted` as a new type. Also only works on reference or pointer type variables.
  - Sometimes useful when reading or writing binary files or when dealing with hardware devices like serial or USB ports.

**Danger!**

“unsafe”: the compiler will not protect you here!

The programmer must make sure everything is correct!
Functions

- Open the project “FunctionExample” in the Part 1 Eclipse project file.
  - Compile and run it!
- Open main.cpp
- 4 function calls are listed.
- The 1st and 2nd functions are identical in their behavior.
  - The values of L and W are sent to the function, multiplied, and the product is returned.
- RectangleArea2 uses const arguments
  - The compiler will not let you modify their values in the function.
  - Try it! Uncomment the line and see what happens when you recompile.
- The 3rd and 4th versions pass the arguments by reference with an added &

The return type is float.

```c
float RectangleArea1(float L, float W) {
    return L*W ;
}
```

Product is computed and returned

```c
float RectangleArea2(const float L, const float W) {
    // L=2.0 ;
    return L*W ;
}
```

```c
float RectangleArea3(const float& L, const float& W) {
    return L*W ;
}
```

```c
void RectangleArea4(const float& L, const float& W, float& area) {
    area= L*W ;
}
```
Using the Eclipse Debugger

- To show how these functions work we will use the Eclipse interactive debugger to step through the program line-by-line to follow the function calls.
- Click the Debug button:
Add Breakpoints

- Breakpoints tell the debugger to halt at a particular line so that the state of the program can be inspected.
- Right-click over the line numbers, go to Breakpoint Types and choose C/C++ Breakpoints.
- Double click next to the line numbers in the functions to add breakpoints.
- Click the green arrow in the toolbar to resume the program.
- The debugger will pause the program at the first breakpoint.

- In the right hand window you’ll see the argument values. Click one for details.

- Let’s step through this: