1. January 22, 2019 – Course Introduction – C++ Primer – GNU Compilation starters
   1. Today’s Concepts
      1. Differences between programming languages
         1. Compiled *vs* Interpreted programs
         2. Programming “style” restrictions
      2. C++ concepts (Learning C++ is only part of the deal)
      3. Classes
      4. Object-Oriented Programing
         1. ⮚ Encapsulation
         2. ⮚ Inheritance
         3. ⮚ Polymorphism
      5. Object-Oriented Design
         1. (But Remember: C++ is the “*Latin*” of programming languages)
   2. C++ Basic Info
      1. Created in 1979 by Bjarne Stroustrup – *the originator*. At Bell Labs (home of UNIX and C)
         1. The story: “Invent a computer programming language so arcane and complex that no one except him would be able to use it.”
         2. The background: “Languages like Simula and Ada.”
         3. The anecdote: “Colleagues laughed a his first cut at an inscrutably hard language. Then, he went back to his lair and didn’t emerge until he had added references and templates.”
      2. Added object-oriented features to C.
         1. Renamed to C++ in honor of auto-increment operator.
      3. Later standardized with several International Organization for Standards (ISO) specifications.
         1. ⮚ Constantly expanding/updating standards:
      4. Greatly influenced Java development (1991).
         1. ⮚ Popular modern OO language
         2. ⮚ Wide industry usage
         3. ⮚ Used in many types of applications
         4. ⮚ Object-Oriented
         5. ⮚ Portable (not as much as Java, but fairly so)
         6. ⮚ Efficient
         7. ⮚ Retains much of its C origins
   3. Procedural vs Object-Oriented
      1. Procedural
         1. Modular units: functions
         2. Program structure: hierarchical
         3. Data and operations are not bound to eeach other
         4. Examples
            1. C., Pascal, Basic, Python
      2. Object-Oriented (OO)
         1. Modular units: objects
         2. Program structure: a graph
         3. Data and operations are bound to each other
         4. Examples
            1. C++, Java, Python
      3. The ATM Machine paradigm
         1. Fundamental Software Development problems:
            1. Maintainability
            2. Adaptability (modularity)
            3. Expressiveness
      4. Procedural (P2)
         1. Focused on the question: “**What** should the program **do next**?” Structure program by:
            1. ⮚ Splitting into sets of tasks and subtasks.
            2. ⮚ Make **Functions** for tasks.
            3. ⮚ Perform **tasks in sequence** (computer). Large amount of data and/or tasks makes projects/programs unmaintainable.

A hierarchy of functions

* + 1. Object-Oriented (P2)
       1. Package-up **self-contained** & **modular** pieces of code. The world is made up of **interacting Objects**.  
          Pack away details into boxes (**Objects**), allowing us to consider them in their abstract form.
       2. Focus on their (numerous) interactions.
          1. Key concepts:

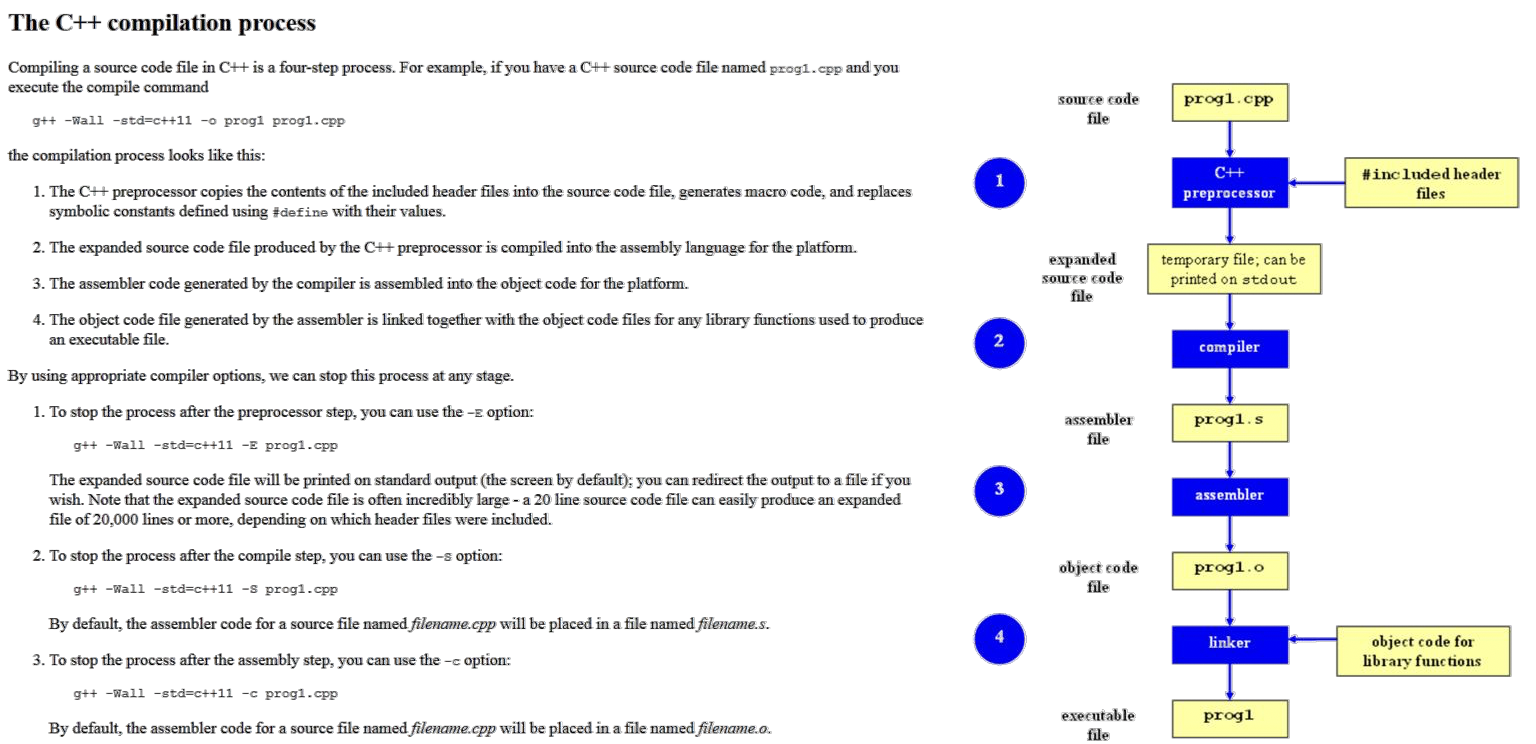
⮚ Encapsulation

⮚ Inheritance

⮚ Polymorphism

A collection of Objects

* 1. Classes
     1. *What is a ...* **Class** 
        1. C++ Classes are very similar to C Structs, in that they both include user-defined sets of data items, which collectively describe some entity such as a Student, a Book, an Airplane, or a data construct such as a String, a Complex Number, etc...
        2. A *non-primitive* & *user-defined* data type containing:
           1. ⮚ Attributes – make up the object’s state
           2. ⮚ Operations – define the object’s behaviors
     2. *What is an ...* ***Object***
        1. A particular instance of a class.
  2. Interpreters, Compilers, Hybrids
     1. Interpreted Languages (e.g. JavaScript, Perl, Ruby)
        1. Interpreter translates source into binary operations and executes it.
           1. Small, easy to write
        2. Interpreter is a *program* unique to each platform (i.e. operating system).
     2. Compiled Languages (e.g. C, C++)
        1. Compiler is platform dependent.  
           Code once compiled is expressed in the *instructions* of the target machine
           1. (e.g. target architecture).
     3. Many other models: (e.g. C, C++)
        1. Bytecode is platform independent  
           Java *Virtual Machine* (the target platform) is an interpreter that is platform dependent.
  3. C++ Compilation and Linkage



* 1. Hands-on C++ Compilation
     1. Connect to the **CSE** Ubuntu Virtual Machine Environment:
        1. https://unr.canvaslms.com/files/2324648
     2. 1)  Right click on the desktop screen, choose “Create Document” → “Empty File”, name it **proj\_1.cpp**
     3. 2)  Double-click to open it (with the default text editor – gedit ), write your code and save it.
     4. 3)  Back on the Desktop screen, click the Blue sign on the top-left, and then click on “Terminal Emulator”
     5. 4)  In the terminal give the command “**cd Desktop**” to go to the Desktop folder
     6. 5)  In the same terminal give the command “**g++ -o proj\_1 proj\_1.cpp**” to compile your code
     7. 6)  If it compiles correctly you will have a **proj\_1** file created which is your executable
     8. 7)  You can run to test the executable by giving the command “**./proj\_1**” in the same terminal screen.
        1. If it compiles and runs, you can take the source code (**proj\_1.cpp**) and put it in an archive file (zip, tar.gz) together with the documentation file. Then upload this compressed archive file on WebCampus.
     9. 8)  You will find the program that creates compressed files by clicking on the Blue sign on the top-left, and then go to “Accessories” → “Archive Manager”. You can add the files to compress via drag-and-drop.
        1. Install the **NoMachineTM** Client: <https://www.nomachine.com/>
  2. C++ Identifiers and Variables
     1. C++ Identifiers
        1. Can’t is keywords/reserved words.
        2. Case-sensitivity and validity of identifiers
        3. Meaningful names!
        4. Used for variables, class names, and more
     2. Variables
        1. Must declare all data before use in program
           1. A memory location to store data for a program.
           2. Reference by the variable name.
  3. Variable Declarations
     1. C++ Variables
        1. When we declare a variable, we tell the complier:   
           When we declare a variable, we tell the compiler:  
           When and where to set aside memory space for the variable.   
           How much memory to set aside.  
           How to interpret the contents of that memory;
           1. ⮚ I.e., the specified data type.

**Int a;**   
**double b;**

* + - 1. What name we will be referring to that location in memory;
         1. ⮚ I.e. by its identifier, or name.

**int ;**

**double ;**

* + - 1. Syntax: **<type> <legal identifier>** 
         1. Examples:

**int sum**

**float average**

**double grade = 98**

* + - * 1. Must be declared before being used
        2. Must be declared to be of a specific & known type (e.g. **int**, **float**, **char**, etc.)
      1. Naming conventions are rules for names of variables to improve readability
         1. Different standards exist, suggested:  
            Start with a lowercase letter  
            Indicate "word" boundaries with an uppercase letter  
            Restrict the remaining characters to digits and lowercase letters

**topSpeed bankRate1 timeOfArrival**

* + - * 1. Indicate "word" boundaries with an underscore

**top\_speed bank\_rate\_1 time\_of\_arrival**

* + - * 1. Note: variable names are still case sensitive!
  1. Data Types
     1. Same data types from C language (look up if forgot)
  2. Data Assignment
     1. You can (and often *should*) initialize data in declaration statement
        1. **int myValue; ????????  
           int myValue = 0; OK**
     2. Assigning data during execution
        1. Value Categories (C++ Heritage from C)
        2. *Lvalues* (left-side) & *Rvalues* (right-side)
     3. *Note:* Otherwise results *can* be “Undefined” if you don’t initialize!
        1. 4.1 Lvalue-to-rvalue conversion  
           1 - A glvalue of a non-function, non-array type T can be converted to a prvalue. ... If the object to which the glvalue refers is not an object of type T ... or if the object is uninitialized, a program that necessitates this conversion has undefined behavior.
     4. Note: “Reading” from an Uninitialized variable is
     5. **Undefined Behavior** 
        1. *Lvalues* must be variables
        2. *Rvalues* can be any expression
     6. Example: **distance = rate \* time;** 
        1. Lvalue: "**distance**“   
           Rvalue: "**rate \* time**" (Note: The *entire* expression)