(PRACTICAL) COMPUTATIONAL PHYSICS

Physics 551
Lecture 2

NOTATION

Extra Reading

Optional Exercise

- This lecture slides for this course will attempt to use a uniform notation throughout. A normal paragraph looks like this.

✒ *Italicized paragraphs with pen bullets will indicate definitions, with the defined word or phrase shown in* ***SMALL-CAPS****.*

✏ Pencil bullets will indicate the introduction of **new notation**.

☞ Pointing hand bullets indicate important points that might otherwise be overlooked.

GIT BASICS RECAP

- In Lecture 1, we learned how to modify the behavior of Git using an **existing** repository.
- We saw how to check the **status** of files in a working directory
- We learned how to **add** changed or newly created files to the Git staging area
- Finally, we learned how to **commit** any changes or updates to the Git repository's database as required.
- In this lecture, we will see how to create a new Git working directory by **init**ializing a new repository.

GIT BASICS CREATING A NEW REPOSITORY

Git Pro Book Chapter 2.1

- Begin by navigating to the directory you wish to convert into a Git working directory.
- Initialize a new Git repository by invoking

```
$ git init
```

- This will create a hidden *`.git`* directory. It will **not** stage or commit any files that already exist in the directory.
- Verify this by invoking.

```
$ git status
```

- Files can be staged and committed as required.

GIT BASICS
CLONING REPOSITORIES

Git Pro Book
Chapter 2.1

- *It is also possible to use an existing Git repository to **generate** a working directory that is a **CLONE** of the last committed state.*
- Clone the **last committed state** of an existing Git repository by invoking

```
$ git clone url_of_repository local_path
```

- This will set up the **directory** specified by `local_path` as a Git working directory that is identical to the cloned repository's last committed state.
- Git supports **several protocols** that can be used to specify the `url_of_repository`, including downloading over HTTP and SSH.

DEMONSTRATION

Cloning existing Git repositories and creating new ones.

SHELL COMMANDS TO CREATE NEW DIRECTORIES

- To create a new directory **in the current directory**, use the mkdir command with a **single-token relative** path e.g.

  `$ mkdir` *`newDirectoryName`* `man mkdir`

- To create a new directory in another **existing** directory, use the mkdir command with a **multi-token relative or absolute path** e.g.

  `$ mkdir` *`relative/path/to/parent/newDirectoryName`*

  `$ mkdir` *`/absolute/path/to/parent/newDirectoryName`*

- The `mkdir` command can also create a hierarchy of directories.

SHELL COMMANDS TO CREATE NEW DIRECTORIES

- To create any missing directories that enclose the new directory you wish to create, use the mkdir command with the `-p` flag e.g.

  ```
  $ mkdir -p relative/path/to/parent/newDirectoryName
  $ mkdir -p /absolute/path/to/parent/newDirectoryName
  ```

- Now any non-existent directories in the supplied path will also be created.

☞ Directories **and** their contents can be deleted by using the `rm` command with the `-r` flag e.g.

  ```
  $ rm -r path/to/directory
  ```

SHELL COMMANDS TO HANDLE COLUMNAR DATA

- Use the `paste` command to combine columnar files into a single file

  e.g.

  `$ paste` *`file1 file2 file3… > outputFile`*

  man paste

- If *`file1`*, *`file2`* and *`file3`* contain:

file1	*file2*	*file3*
The Spain in	rain falls the	in mainly plain

- Then *`outputFile`* will contain:

outputFile		
The	rain	in
Spain	falls	mainly
in	the	plain

SHELL COMMANDS TO HANDLE COLUMNAR DATA

- The `paste` command accepts two flags, `-d` and `-s`.
- The `-d` flag allows an alternative column delimiter to be specified e.g.

```
$ paste -d" " file1 file2 file3 > outputFile
```

- Replaces the **default tab character** using a single space, yielding:

outputFile
The rain in
Spain falls mainly
in the plain

- Note that the specified delimiter may comprise **multiple characters** specified **between the quotes**.

SHELL COMMANDS TO HANDLE COLUMNAR DATA

- The `-s` flag instructs paste to concatenate **all** of the lines of **each** input file **then** output the result of each concatenation on a **separate** line.
- This can be used to affect a **transpose** operation. Consider e.g.

wrongOrder1	*wrongOrder2*
Top left Top right	Bottom left Bottom right

- Then, using the `-s` flag e.g.

```
$ paste –s wrongOrder1 wrongOrder2 > outputFile
```

outputFile	
Top left	Top right
Bottom left	Bottom right

Try an experiment using the `-s` and `-d` flags together.

SHELL COMMANDS TO PARSE COLUMNAR DATA

- Select specific columns from a file using the `cut` command e.g.

```
$ cut -f 1-2,4-6 file > outputFile
```

`man cut`

- Selects columns 1, 2, 4, 5, and 6. If *`file`* contains

file					
1	+	-	1	=	2
2	*	/	3	=	6

- Then *`outputFile`* will contain

outputFile				
1	+	1	=	2
2	*	3	=	6

SHELL COMMANDS TO PARSE COLUMNAR DATA

- The value of the `-f` flag should be a **comma-separated** list of column **range specifiers**.

➩ ***CLOSED*** *range specifiers comprise a pair of column numbers separated by a hyphen e.g.* `4-12`

- Closed ranges extend from the first to the second numbered columns **inclusive**. Column numbering starts from **1**.

➩ ***OPEN*** *range specifiers comprise a single column number followed by a hyphen e.g.* `3-`

- Open ranges extend from the numbered column to the last column in the input file **inclusive**.

DEMONSTRATION

Parsing Columnar Data

COMPUTER PROGRAMS

- Computers are simply machines that transform a string of input bits into a string of output bits in a deterministic fashion.
- ✑ ***Programmable** computers enable the applied transformation to be arbitrarily specified using a **COMPUTER PROGRAM**.*
- In principle, the operation of a computer program **could** be defined as a sequence of bitwise mutations of the input data.
- Practically, such a definition would be difficult to write and interpret and would typically depend upon the hardware platform for which the program was defined.

PROGRAMMING LANGUAGES

- ***PROGRAMMING LANGUAGES*** *provide an* ***abstract*** *description or* ***encoding*** *of the operations performed by a computer program.*
- ***One way*** *that different programming languages can be distinguished is by the way they describe a program - their* ***GRAMMAR*** *or* ***SYNTAX****.*
- This course will introduce the grammars of **three different** programming languages.
- We will explore the ways each can be used to **express common programatic operations** and solve computational problems that arise in physics research.

WHICH LANGUAGES?

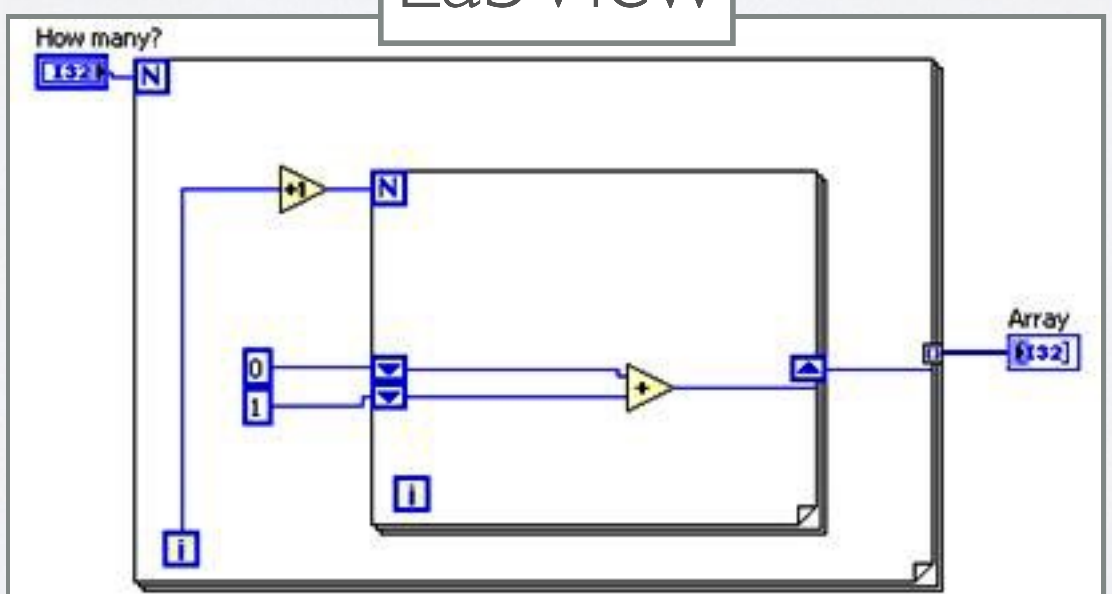
C++

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

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

    int numIterations(10), iteration(0);
    std::vector<int> output;

    while (iteration < numIterations) {
        if (iteration <= 1) {
            output.push_back(iteration);
        } else {
            output.push_back(*(output.end() - 1)
                             + *(output.end() - 2));
        }
        ++iteration;
    }
    return 0;
}
```

Python

```python
import numpy as np

numIterations = 10
iteration = 0

output = np.empty(numIterations)

while iteration < numIterations :

    if iteration <= 1 :
        output[iteration] = iteration

    else :
        output[iteration] = output[iteration - 2]
        output[iteration] += output[iteration - 1]

    iteration += 1
```

LabView



COMMON COMPONENTS OF PROGRAMMING LANGUAGES

TOKENS

- Programming languages represent computational operations and data using a series of **tokens** arranged in conformance with that language's grammar.

☞ ***TOKENS*** *are analogous to words and punctuation marks in written English. They are simply sequences of characters that have meaning as a group.*

☞ For some languages, the meaning of a token is **context dependent**.

- Programming language grammars typically specify several distinct **categories** of token including ***keywords***, ***identifiers***, ***literals***, ***variables***, ***type specifiers***, ***operators***, ***mutability specifiers*** and ***comments*.**

TOKEN CATEGORY DEFINITIONS

- *A **KEYWORD** is a token which has a meaning that is defined by the **programming language syntax**.*
- *Keywords are often **RESERVED WORDS**, which are tokens that cannot be assigned any other meaning.*
- ***IDENTIFIERS** are tokens that define human-readable names for programatic entities that is defined by the **computer program**.*
- Most languages define a subset of possible tokens that cannot be used as identifiers e.g. C++ identifiers cannot replicate reserved words.

TOKEN CATEGORY DEFINITIONS

- ✑ *A **Literal** is a token that can be directly interpreted as a value e.g.* 2 *or* `'b'`.
- The value of a literal is not modifiable at program runtime.
- ✑ *A **Variable** is an **identifier** that refers to data at a particular location (or address) in computer memory.*
- Programmers use variables to straightforwardly ***reference***, ***assign*** and ***modify*** their associated data at program runtime.
- ☞ **Mutable** variables can usually be assigned the value of a literal or another variable.

TOKEN CATEGORY DEFINITIONS

- A program's variables are represented in computer memory as sequences of bits.

➾ *A **TYPE SPECIFICATION** is a token that defines the **interpretation** to be applied to a variable e.g. as an integer value.*

➾ *Redefining the interpretation of an variable is known as **TYPE CASTING**.*

➾ ***WEAKLY TYPED** computer languages will attempt to infer the type of a variable based upon its assigned value.*

➾ ***STRONGLY TYPED** computer languages require the type of a variable to be **explicitly** specified and may **only** be assigned conforming values.*

TOKEN CATEGORY DEFINITIONS

- *An **OPERATOR** is a token that specifies an operation to be applied to one or more programatic entities e.g. addition of two numeric values.*
- *A **MUTABILITY SPECIFIER** is a token that is associated with a particular variable and specifies whether that variable may be modified **after** its initial assignment.*
- ***SOURCE CODE** refers to the textual representation of computer program using a particular programming language.*
- *A **COMMENT** is a non-functional portion of the source code that serves to annotate the remaining functional code.*
- It is **very important** to comment your code. In this course, you will be awarded marks for doing so!

EXPRESSIONS, STATEMENTS, AND INITIALIZATIONS

- *An **EXPRESSION** is a sequence of one or more tokens that conforms to the programming language grammar.*
- *A **STATEMENT** is an expression that specifies an **action** to be performed by a computer program.*
- *A **DECLARATION** is an expression that associates an identifier and possibly a type with a variable.*
- *An **INITIALIZATION** is a statement that assigns a value to a declared variable.*

- Many programming languages allow declaration and definition to be **combined** within a single statement.

DEMONSTRATION

Introducing Cling and C++

LECTURE 2 SUMMARY

- After reviewing the material in this lecture **and completing the reading exercises** you should know:
  1. How to **clone** existing Git repositories and initialize new Git working directories.
  2. How to create new directories in the Linux filesystem.
  3. How to use `cut` and `paste` to work with columnar text files.

LECTURE 2 SUMMARY

4. A working definition of a programmable computer and a computer program.
5. What is meant by the terms: ***token***, ***keyword***, ***reserved word***, ***identifier***, ***literal***, ***variable***, ***type specifier***, ***operator***, ***mutability specifier*** and ***comment*** in the context of computer programs and C++ in particular.
6. How to start and use the **Cling** C++ interpreter to **test and prototype** C++ code.

LECTURE 2 SUMMARY

7. The **basic** components of a simple C++ program including: ***declarations***, ***initializations***, ***assignments***, ***arithmetic expressions***, ***boolean expressions*** and ***flow control statements***.
8. How to **include header files** in order to enable extra functionality in Cling and C++.
9. How to **output text and numeric values** to the terminal in C++.

LECTURE 2 SUMMARY

10. How to determine the memory that is allocated to a particular type in C++ using the `sizeof()` operator.
11. How to explicitly specify the memory that is allocated to ***literal numeric, character*** and ***string*** values that you declare in your C++ code.
12. That it is **very important** to add numerous detailed comments to your code. I remind you that you will get **credit** for doing so in this course!

LECTURE 2 HOMEWORK

Read sections:

- Basics of C++ → Variables and types
- Basics of C++ → Constants
- Basics of C++ → Operators

from the **C++ Reference** language tutorial:

http://www.cplusplus.com/doc/tutorial

Read Chapter 2.1 from the **Git Pro Book**.

- Complete the **Lecture 2 Homework Quiz** that you will find on the course Blackboard Learn website.