(PRACTICAL)
COMPUTATIONAL PHYSICS

Physics 551
Lecture 6

NOTATION

Extra Reading

Optional Exercise

Recommended

- 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.

ANNOUNCEMENTS

ANNOUNCEMENTS

- To clone this week's **C++ demonstration materials** please invoke

```
$git clone https://github.com/hughdickinson/CompPhysL6CPP.git
 /home/computationalphysics/Documents/cPlusPlus/lecture6
```

- To clone this week's **shell command demonstration materials** please invoke

```
$git clone https://github.com/hughdickinson/CompPhysL6Shell.git
 /home/computationalphysics/Documents/theShellGym/lecture6
```

☞ You can also find these commands on the Blackboard Learn website.

ANNOUNCEMENTS

- The following will be covered during the **shell demonstration** for this lecture.
- There is some new software to install **on VirtualBox**.
- For future reference, the **shell commands** that will be invoked are

```
$ sudo apt-get install libgsl0ldbl
$ sudo apt-get install libgsl0-dev
$ sudo apt-get install source-highlight
```

☞ **If** you had **difficulties** with the Git demonstration from **Lecture 5**, try invoking

```
$ git config --global push.default simple
```

ANNOUNCEMENTS

- The STL is a **very extensive resource** that this course cannot cover **in depth**.
- The **C++ Reference** material that is available at

  http://www.cplusplus.com/reference/

  is a **comprehensive** guide for all the ***classes***, ***functions*** and ***utilities*** that the STL provides.
- The reference is organized according to the **header files** that provide the **associated functionality**.
- **Class references** for the various **container types** e.g. `<vector>` and **data processing algorithms** are particularly useful.

ANNOUNCEMENTS

- **Homework and Project Advice** - How to maximize your marks.
  - **Read the question**. Make sure you implement the specified components e.g. if the question specifies **3** functions, write **3** functions!
  - **The web is a rich resource**. Use online resources for inspiration! If you use **code** from the web, **cite the website URL** using a comment and add your own **comments** to prove **your** understanding.
  - The same guidelines apply to the use of **literary resources** i.e. books.
  - **Your colleagues are a rich resource**. Discuss homework, midterms and final projects with your fellow students or supervisors. Do not **copy** each other's code **without citation**!

ANNOUNCEMENTS

For **general regulations** and **possible sanctions** upon violation see the ***Academic Dishonesty*** section of http://catalog.iastate.edu/academiclife/regulations/regulations.pdf

- **What is acceptable?**
  - Simple, ***isolated***, one-line statements copied from the web need not be cited, although you should still add a **comment** annotating the **intended functionality**.
  - **Entire *functions***, ***code blocks***, or ***classes*** that are **copied** into your source code should be **cited appropriately** and **re-commented**.
  - The use of **external libraries** and **header files** should be **clearly cited**. The exception is the STL, which may be used without citation.

CLARIFICATIONS

CLARIFICATIONS

- **Reminder:** How to **compile** standalone **binary executables** from C++ source code using `clang++`. Don't forget the `-std=c++11` flag

```
$ clang++ -std=c++11 -o pathToExecutable sourceCodeFiles…
```

- **If** function or method **arguments** are passed **by value** then the entity that is available **inside** the function or method body is a **copy** of the entity that is passed.

☞ Be aware that for classes that **you define**, this will be a **shallow copy** unless you have explicitly **overloaded** the **assignment operator**.

CLARIFICATIONS

- Use of the `this` identifier is **only valid** within class method definitions.
- Within a class method definition, `this` refers to a **pointer** to the **instance** of the class upon which the method being defined is invoked.
- Consider the following simple class definition

```
class ThisExample {
    public :
    ThisExample * thisThisExample(){
        return this;
    }
};
```

CLARIFICATIONS

- Now consider invoking the `thisThisExample()` method on two **distinct instances** of `ThisExample`

```
ThisExample firstExample;
ThisExample secondExample;

ThisExample * thisThis = nullptr;
bool isSameThis = false;

thisThis = firstExample.thisThisExample();
// Recall the “address-of” operator “&”
isSameThis = (&firstExample == thisThis); // true
isSameThis = (&secondExample == thisThis); // false

thisThis = secondExample.thisThisExample();

isSameThis = (&firstExample == thisThis); // false
isSameThis = (&secondExample == thisThis); // true
```

CLARIFICATIONS

`firstExample` and `secondExample`
are **separate instances** of the `ThisExample` class.

`firstExample`

```
class ThisExample {
    public :
    ThisExample * thisThisExample(){

        return this;

    }
};
```

INSTANCE

CLASS

`secondExample`

```
class ThisExample {
    public :
    ThisExample * thisThisExample(){

        return this;

    }
};
```

```
*firstExample.thisThisExample()
```

```
*secondExample.thisThisExample()
```

`thisThisExample` returns `this`. Dereferencing `this` provides a reference to the **instance** on which the method was called.

GITHUB

GITHUB INSTRUCTIONS

- **Thank you** for sending me your GitHub usernames.
- I have now created **private** Git repositories for each of you.
- You will be able to use these repositories to **back up**, **develop** and **submit** parts of your weekly homework from now on.

☞ The **URL** of **your** private repository is:

```
https://github.com/ISUComputationalPhysics/
<Firstname><Surname>Homework.git
```

☞ Replace <*Firstname*> and <*Surname*> with your own details.

DEMONSTRATION

Cloning from and **Pushing** to your **private** GitHub repository

Clone the Shell demonstration material from Github:

```
$git clone https://github.com/hughdickinson/CompPhysL6Shell.git
 /home/computationalphysics/Documents/theShellGym/lecture6
```

OO PROGRAMMING:

INHERITANCE

Important additional information is included in the **C++ demonstration material** for **Lecture 5**.

INITIALIZING THE BASE CLASS

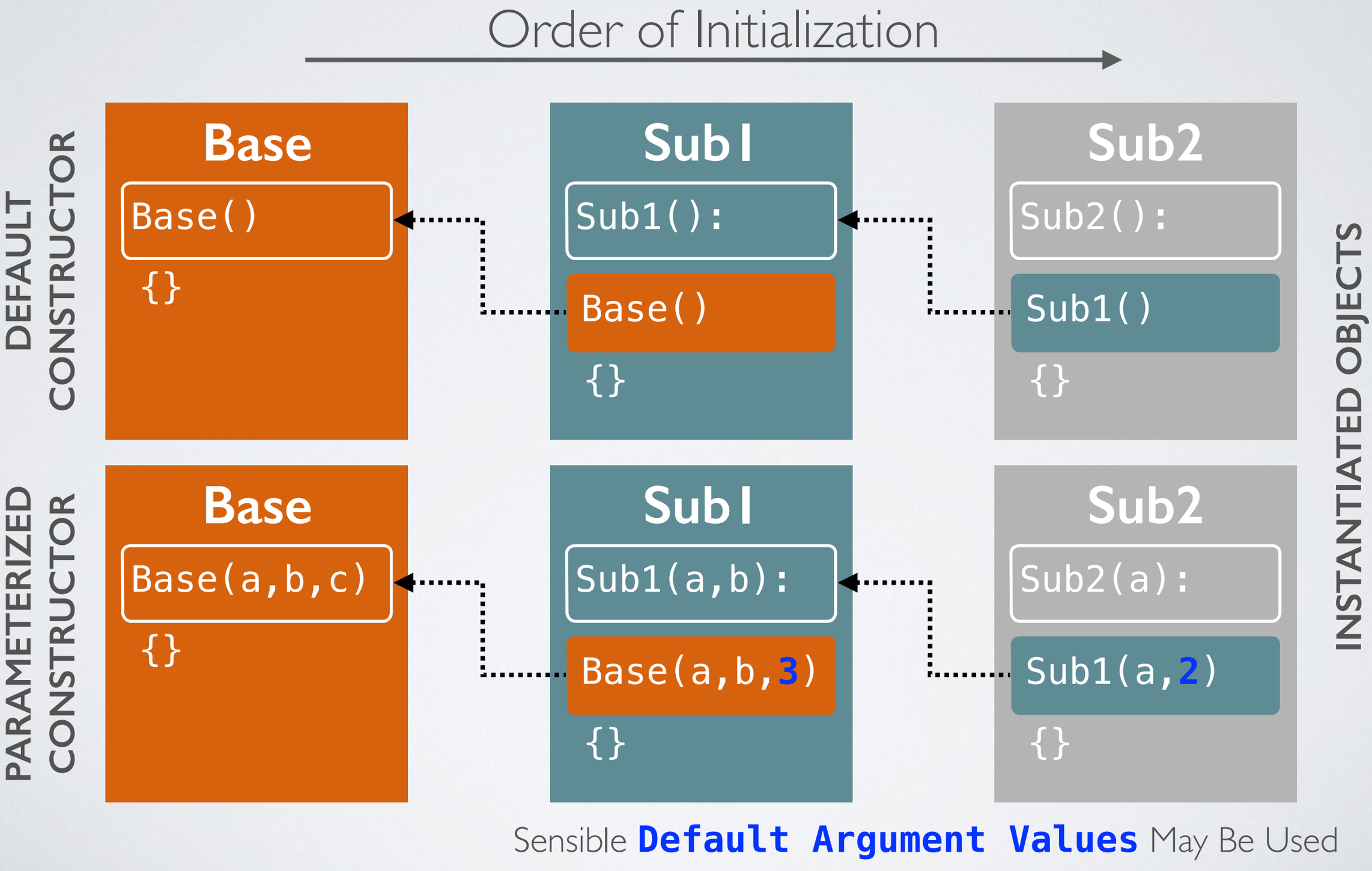
- Derived classes **inherit** the member data and methods of their parent classes.
- Depending upon their definitions, the member data of the **parent** class may require **initialization** in order for its methods to function as expected.
- If **any** of those methods are called by the **derived** class during its initialization, it is **essential** that the **base** class is **already** properly initialized.

INITIALIZING THE BASE CLASS

- When a derived class is instantiated, C++ initializes its parent class, by calling its constructor **before** it initializes the derived class.
- This behavior **propagates** upwards through the inheritance hierarchy
- This ensures that all the **ancestors** of particular derived class in the hierarchy are properly initialized **before** the initialization of that derived class proceeds.

☞ Derived classes may call **any** of the constructors that are provided by their parent classes, using **sensible default arguments** if necessary.

INITIALIZING THE BASE CLASS



Sensible **`Default Argument Values`** May Be Used

DEMONSTRATION

Object Oriented Programming: Initializing the Base Class

Clone the C++ demonstration material from Github:

```
$git clone https://github.com/hughdickinson/CompPhysL5CPP.git
 /home/computationalphysics/Documents/cPlusPlus/lecture5
```

THE C++ STANDARD TEMPLATE LIBRARY

C++ Standard Library → Input/Output with files

from the C++ Reference language tutorial:

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

WRITING TO AND READING FROM FILES

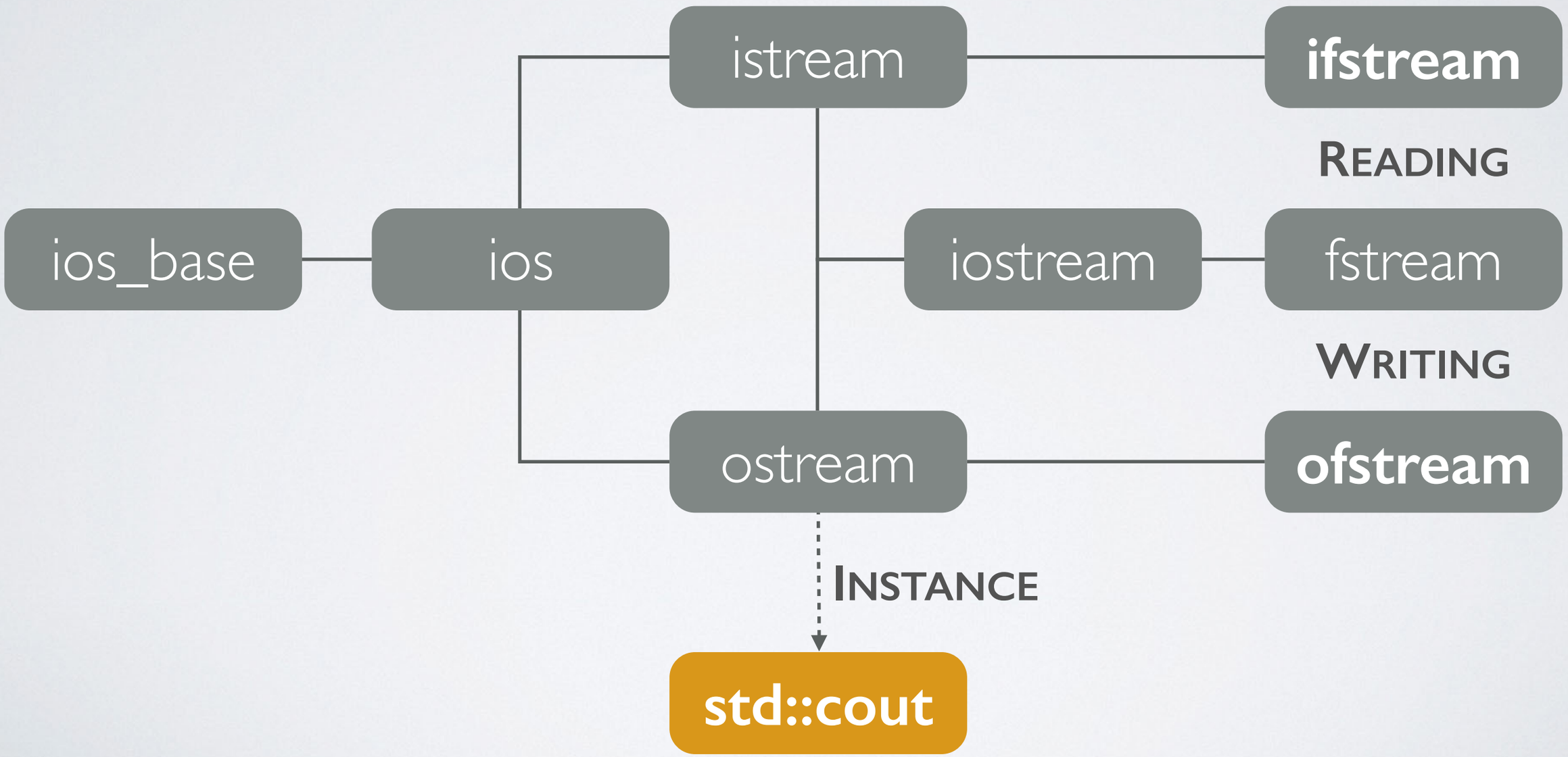
- In the context of **data analysis**, one of the most useful facilities that is provided by the STL is **data input** and **output** (I/O) using **files**.
- The STL supports I/O using arbitrary **textual** and **binary** file formats.

☞ File I/O functionality is provided by the `<fstream>`, `<ofstream>` and `<ifstream>` header files.

☞ For textual file I/O the required **stream operator syntax** closely resembles that which is used to output data to the terminal.

WRITING TO AND READING FROM FILES

STL I/O Inheritance Hierarchy



WRITING TO AND READING FROM FILES

- To open a file called "`infile`" for **reading** instantiate a `std::ifstream` object.

```
std::ifstream inputFile("infile");
```

- To open a file called "`outfile`" for **writing** instantiate a `std::ofstream` object.

```
std::ofstream outputFile("outfile");
```

☞ The single (required) constructor arguments are **C-strings** specifying the **paths** of the input or output file.

☞ **Recall** that a C-string (of type `const char *`) can be extracted from a `std::string` using its `c_str()` method.

WRITING TO AND READING FROM FILES

- Both the `ifstream` and `ofstream` classes provide methods to **check the status** of their instances.

☞To **check** if a file has been **successfully opened** use the `is_open()` method.

☞To **check** if a file is in a **readable or writeable** state use the `good()` method.

☞To check if the **end of a file** that is **being read** has been reached, use the `eof()` method.

☞To close a file that has been opened, use the `close()` method.

WRITING TO AND READING FROM FILES

- **Reading** from and **writing** to **formatted textual** files is accomplished using the **stream input** and **output operators**.

☞ *The **STREAM INPUT OPERATOR**, "**>>**" is used to **extract** a sequence of **non-whitespace characters** from a textual file and **may** also interpret the extracted characters another data type e.g.* `double` *or* `int`.

☞ *The **STREAM OUTPUT OPERATOR**, "**<<**" is used to convert data to a character representation and append the characters to an **output file**.*

DEMONSTRATION

STL Algorithms and File I/O

Clone the C++ demonstration material from Github:

```
$git clone https://github.com/hughdickinson/CompPhysL5CPP.git
 /home/computationalphysics/Documents/cPlusPlus/lecture5
```

C++ LEGACY CODE: HEADER FILES AND LIBRARIES

LEGACY CODE

- Although this is a course about computation, it is important to remember the **ultimate purpose** of the computations that are performed - supporting **physics research**!
- If **legacy codebases** and **software libraries** that provide the required functionality exist, reimplementation is not an efficient use of your time.
- Fortunately, modern software development tools and paradigms provide numerous facilities for **straightforward adoption** of and **interoperation** with legacy code and libraries.

C++ HEADER FILES

- An obvious example of a **legacy codebase** that has already been encountered during the course is the **STL**.
- Functionality provided by the STL is made available in source code using `#include` statements e.g.

  `#include <vector>` or `#include <iostream>`

☞ The `vector` token refers to a **file** called a **header file**.

☞ ***HEADER FILES*** *describe the functionality provided by legacy code and typically comprise* ***commented declarations*** *and partial* ***definitions*** *of classes, functions, constants and variables.*

C++ HEADER FILES

- **Including** a header file is equivalent to inserting the declarations and definitions it contains directly into your source code.
- This means that **your code** can **reference** the identifiers of the variables, **call** the functions or **instantiate** the classes that are declared or defined in the header file.

☞ You can place **your own code** into header files to make it easier to **reuse** and (ultimately) **share** with others.

☞ User defined header files normally use the "***.h***" suffix.

C++ HEADER FILES

- Recall that when `clang++` is invoked, header files are included during the **preprocessing** stage of the build.

☞ *Preprocessing is actually performed by a separate utility called the* ***C PREPROCESSOR****.*

- **By default**, the C preprocessor only searches particular **standard locations** to find the header files you specify should be included.
- You can specify **additional search locations** by providing a `-I` flag when invoking `clang++`.

C++ HEADER FILES

- To specify that the C preprocessor should include ***`headerDir`*** among the locations it searches for header files, the following invocation is required

```
$ clang++ -std=c++11 -IheaderDir -o output inputs…
```

- Recall that **identifiers** in C++ may be specified **at most once** in any C++ program.
- A header file may include variable declarations and may be **included several times** in a program.

C++ HEADER FILES

- A mechanism is required to prevent the compiler encountering **multiple declarations** of the same variable.
- The most commonly adopted approach involves enclosing the source code within header files between **include guards**.
- ***INCLUDE GUARDS*** *comprise several* ***special tokens*** *that the C preprocessor uses to determine whether a particular header file has already been included in the program.*

C++ HEADER FILES

- To implement the include guard mechanism for your header file, **prepend the source code** with statements that begin a C preprocessor **conditional block**

```
// If INCLUDE_GUARDS_H is NOT defined.
#ifndef INCLUDE_GUARDS_H
// Define it for the remainder of the build.
#define INCLUDE_GUARDS_H
```

☞ *The* `INCLUDE_GUARDS_H` *token is called a* ***PREPROCESSOR MACRO****. Its name is* ***arbitrary****, but should be* ***unique*** *to the header file being guarded and should be defined* ***exactly*** *once.*

C++ HEADER FILES

- Upon encountering the opening include guard, the C preprocessor will **only** consider subsequent tokens if the `INCLUDE_GUARDS_H` macro is **not** defined.
- The next expression defines `INCLUDE_GUARDS_H`, ensuring that the subsequent code will only be considered **once**.
- The **conditional block** that was opened by the `#ifndef` token should be terminated by a single `#endif` token at the end of the header file.

DEMONSTRATION

Creating and **Including** a **Header File**.

C++ SHARED LIBRARIES

- Although header files **may** provide complete **definitions** of simple classes, this is **not typical**.
- Instead, header files are used to **document** and **declare** programatic entities that are provided in separate **shared library** files.

☞ ***SHARED LIBRARY*** *files contain* ***compiled binary code*** *that can be* ***loaded*** *and* ***invoked at runtime*** *by a binary executable.*

☞ Shared libraries are **associated** with a binary executable during the **linking** stage of the build process.

C++ SHARED LIBRARIES: RATIONALE

- Upon compilation, inclusion of **the same source code** into **different programs** generally produces **very similar binary code**. If multiple standalone binaries incorporate essentially identical binary code, this is not efficient use of system **storage space**.
- Storage space can be saved by **precompiling logically distinct** and **reusable** subsets of source code and encapsulating the resulting **binary code** in a separate **shared library** file that can be accessed by multiple standalone binaries.

C++ SHARED LIBRARIES: RATIONALE

- If **improvements** are made to the source code that is used to build a shared library, and that library is used by **multiple standalone binaries**, then **only** the shared library needs to be recompiled for **all** the standalone binaries that use it to benefit from the improved source code.
- If shared libraries were **not** used, then **all** the standalone binaries would need to be recompiled.

☞ Shared libraries provide an excellent mechanism for **reuse of source code**.

C++ SHARED LIBRARIES: CREATION

- On **Linux** operating systems, the names of shared libraries typically incorporate a "*`lib`*" prefix and "*`.so`*" suffix.
- Shared libraries can be **created** from C++ source code by specifying the `-shared` and `-fPIC` flags when invoking `clang++`.

```
$ clang++ -std=c++11 -shared -fPIC -o
  libSharedLibrary.so sourceFiles…
```

☞ Source code used to create shared libraries should **not** contain a `main` function.

C++ SHARED LIBRARIES: USEAGE

- Recall that when `clang++` is invoked, libraries are associated with the executable during the **linking** stage of the build.

☞ *Linking is actually performed by a separate utility called the* ***LINKER***

- **By default**, the linker only searches particular **standard locations** to find the libraries you specify should be linked against.
- You can specify **additional search locations** by providing a `-L` flag when invoking `clang++`.

C++ SHARED LIBRARIES: USEAGE

- To specify that the linker should include *`libDir`* among the locations it searches for libraries when linking an executable, the following invocation is required

```
$ clang++ -std=c++11 -Llibdir -o output inputs…
```

- The *`inputs`* may now include one or more shared libraries.
- **If** the library name uses the conventional "*`lib`*" and "*`.so`*" prefix and suffix are used, then an **abbreviation** can be used.
- Specifically, the library *`libLibraryName.so`* can **also** be specified using the token *`-lLibraryName`*.

DEMONSTRATION

Creating and **Linking Against** a **Shared Library**.

LECTURE 6 SUMMARY

- After reviewing the material in this lecture **and completing the reading exercises** you should know:

  1. How to **push** updates to your own **private homework repository** on GitHub.
  2. How to write classes that properly **initialize the base classes** from which they derive.
  3. How to perform **textual file I/O** using instances of the C++ stream classes.

LECTURE 6 SUMMARY

4. That **header files** and **shared libraries** can be employed to **efficiently reuse** the code you write.
5. That C++ **header files** usually have a "***.h***" suffix.
6. That C++ header files typically contain **commented declarations** and partial **definitions** of classes, functions, constants and variables.
7. How to use **include guards** to ensure that the code in header files is parsed **exactly once** by the compiler.

LECTURE 6 SUMMARY

8. How to control where the C preprocessor **searches for header files** by supplying the `-I` flag to `clang++`.
9. That shared libraries contain **compiled binary code** that can be **loaded** and **invoked at runtime** by a binary executable.
10. That **shared library names** on **Linux** systems typically begin with "*`lib`*" and have a "*`.so`*" suffix.
11. That source code that is intended to be compiled into a shared library should **not** define a `main()` function.

LECTURE 6 SUMMARY

12. How to **create shared libraries** from C++ source code by invoking `clang++` and supplying the `-fPIC` and `-shared` flags.
13. How to **link** a binary executable with **shared libraries** by supplying them as input files to an invocation of `clang++`.
14. That shared library **names** supplied to clang++ can be **abbreviated** if they begin with "*`lib`*" that have a "*`.so`*" suffix e.g. *`libExample.so`* abbreviates to *`-lExample`*.

LECTURE 6 HOMEWORK

Be sure to thoroughly review the C++ demonstration material from Lectures **5** and **6**!

- Complete the **Lecture 6 Homework Quiz** that you will find on the course Blackboard Learn website.

As requested by several survey participants, **additional** reading that is **required** or **suggested** for **particular questions** will be specified in the question text.