(PRACTICAL) COMPUTATIONAL PHYSICS

Physics 551

Lecture 5

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

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

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

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

```
$git clone https://github.com/hughdickinson/CompPhysL5Shell.git /home/computationalphysics/Documents/theShellGym/lecture5
```

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

ANNOUNCEMENTS

- **Appeal!** If possible, please avoid submitting homework as Rich Text Format (RTF) and use **plain text files** instead.
- RTF modifies characters like "**"**" which makes it more difficult for me to try compiling or running your code.
- **Expectations:** See Lecture 1 Slides, e.g. Slide 7: WHAT **NOT** TO EXPECT. "A course on **High Performance Computing** using parallel architectures."
- **Good news!** The course focus will soon shift from **programming fundamentals** to utilities for **data analysis**.

ANNOUNCEMENTS

- **Reminder!**

If you don't already have one, please sign up for a **GitHub account** at: https://github.com.

- Once you have your account, please **send me an email** with your **GitHub username**.
- This will allow me to invite you to join the GitHub **organization**:

ISUComputationalPhysics

- The facilities provided by membership of this organization will be **essential** when you carry out and submit your midterm and final projects.

CLARIFICATIONS

- The `clang++` invocation specified in the slides for Lecture 4

  ```
  $ clang++ -o pathToExecutable sourceCodeFiles…
  ```

  is sufficient for **most** of the **basic** C++ syntax we have covered so far.

☞ In fact, using the `nullptr` identifier and some of the more advanced features we will cover in this lecture require another flag `-std=c++11`. Accordingly, the **required invocation** is:

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

GIT BASICS: RECAP

- In **Lecture 2**, we learned how to create a new Git working directory by initializing a new repository using the `git init` command.
- We also saw how to make a **clone** of an existing Git repository using the `git clone` command.
- Next, we will see how Git can be used to **update our local working directory** to reflect the **most recent state** of the remote repository.
- We will also learn how to **update the remote repository** to reflect **locally committed changes**.

GIT BASICS: UPDATING A WORKING DIRECTORY

- To **update** a local working directory, navigate to that directory and invoke

  `$ git pull`

  Git Pro Book Chapter 2.5

- This will determine whether any differences exist between the current state of the working directory and that of the remote repository.
- If necessary, `git pull` will retrieve any updated files from the remote repository, **replacing** any that have **not** been locally modified.
- If the corresponding local files have **also** been modified Git will attempt to **merge** the two versions in a consistent way.

GIT BASICS: UPDATING THE REMOTE REPOSITORY

- To **update** a remote repository, navigate to the working directory that was **cloned** from the repository and invoke.

  ```
  $ git push
  ```

  Git Pro Book Chapter 2.5

- If you have committed changes to your local working directory since it was cloned, `git push` will **create a new snapshot** in the remote repository reflecting the current state of your local working directory.
- If the state of the remote repository has changed in a way that **may** conflict with your local changes Git push will fail.
- Invoke `git pull`, **resolve** any conflicts that **have** occurred, **commit** your files again and invoke `git push`.

DEMONSTRATION

Pulling updates to cloned working directories and **Push**ing committed changes to remote repositories.

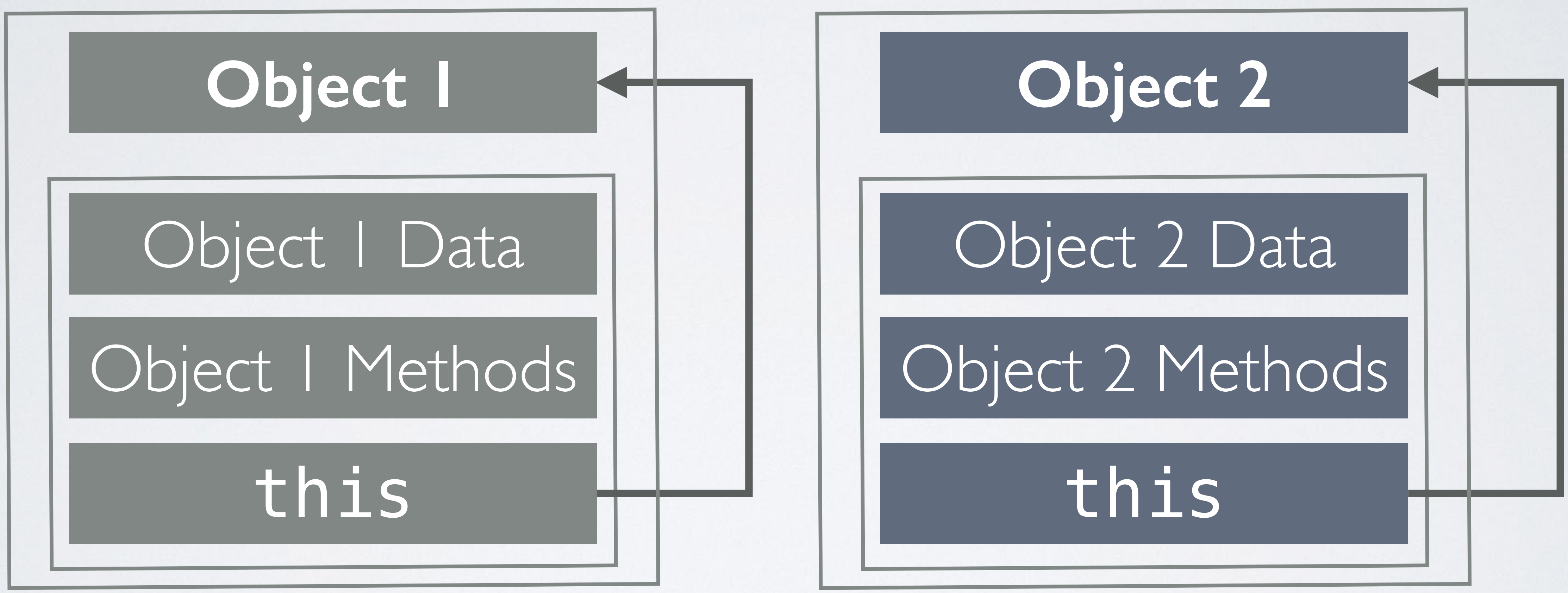
OO PROGRAMMING IN C++: THE `this` POINTER

OO PROGRAMMING IN C++: THE `this` POINTER

- When defining a class method, programmers may reference the **reserved keyword** `this`.
- `this` is an **identifier** that corresponds to **a pointer** to the **instance** of the class being on which the currently executing method was called.
- **Recall** that classes provide a **description** that is used to instantiate **distinct objects** that maintain their own **independent** member data.

☞ The `this` pointer provides a mechanism by which class **instances** can **explicitly** reference **themselves** and their own **members**.

OO PROGRAMMING IN C++: THE `this` POINTER



DEMONSTRATION

The `this` pointer

Clone the C++ demonstration material from Github:

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

DEEP AND SHALLOW COPYING OF POINTERS

DEEP AND SHALLOW COPYING OF POINTERS

- It is often desirable to **duplicate** the state of a **specific class instance**.
- For example, after invoking the assignment operator e.g.

```
classInstance1 = classInstance2
```

  one would probably expect that `classInstance1` behaves like a **perfect clone** of `classInstance2`.
- However, the definition of what constitutes a perfect clone-like behaviour is somewhat **unclear** for C++ class definitions that specify **pointer-type** member data.

DEEP AND SHALLOW COPYING OF POINTERS

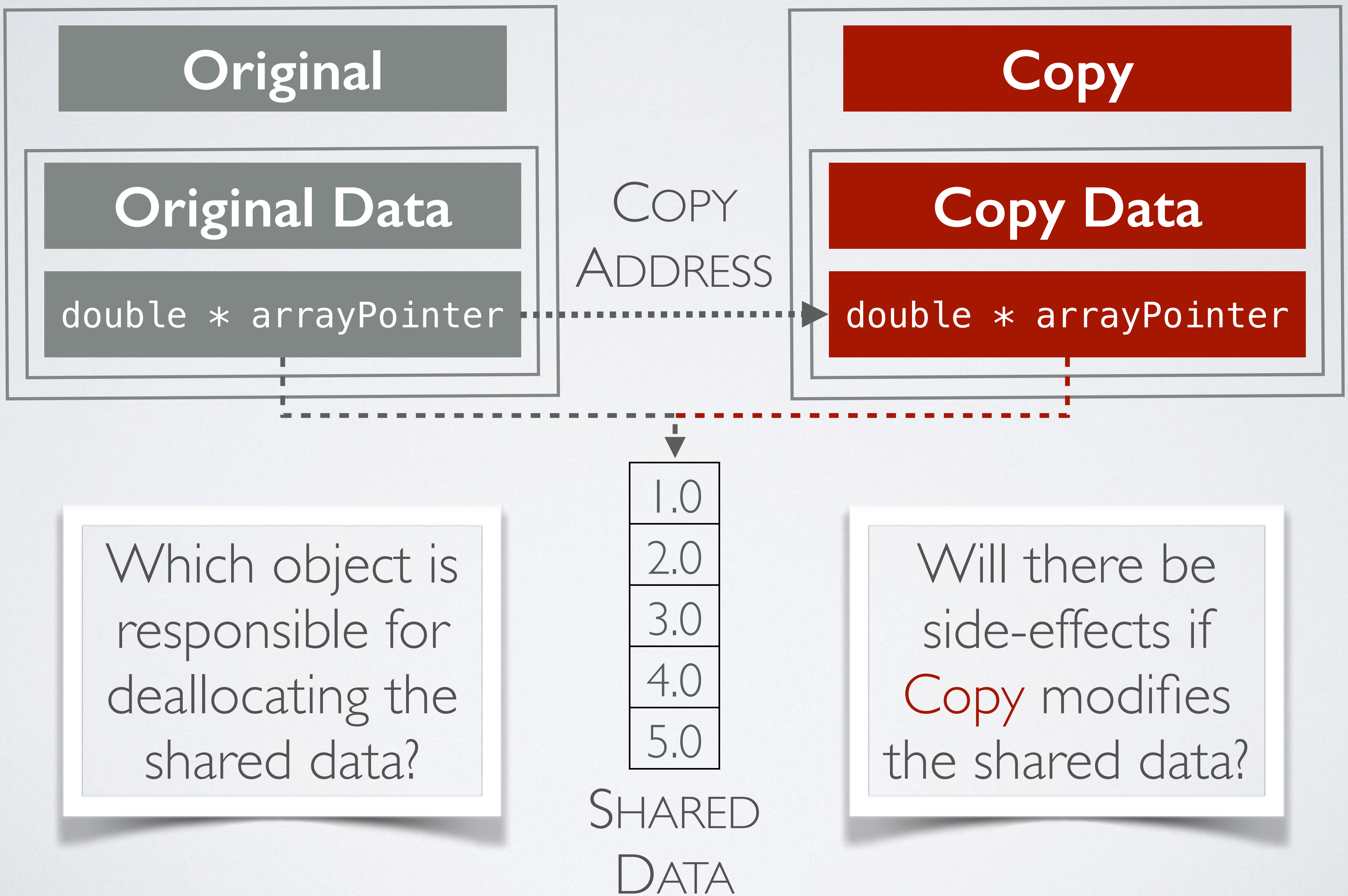
- The fundamental issue is whether to duplicate the **value** of the pointer itself or whether to duplicate the **data** at the address to which the pointer refers.

✑ *The former option is known as performing a **SHALLOW COPY**.*

- Shallow copying **can** be problematic when the pointed-to memory must be **deallocated**, and **may** produce unexpected behavior if multiple objects **modify** the associated data.

✑ *The alternative option is **usually preferred** and is known as performing a **DEEP COPY**.*

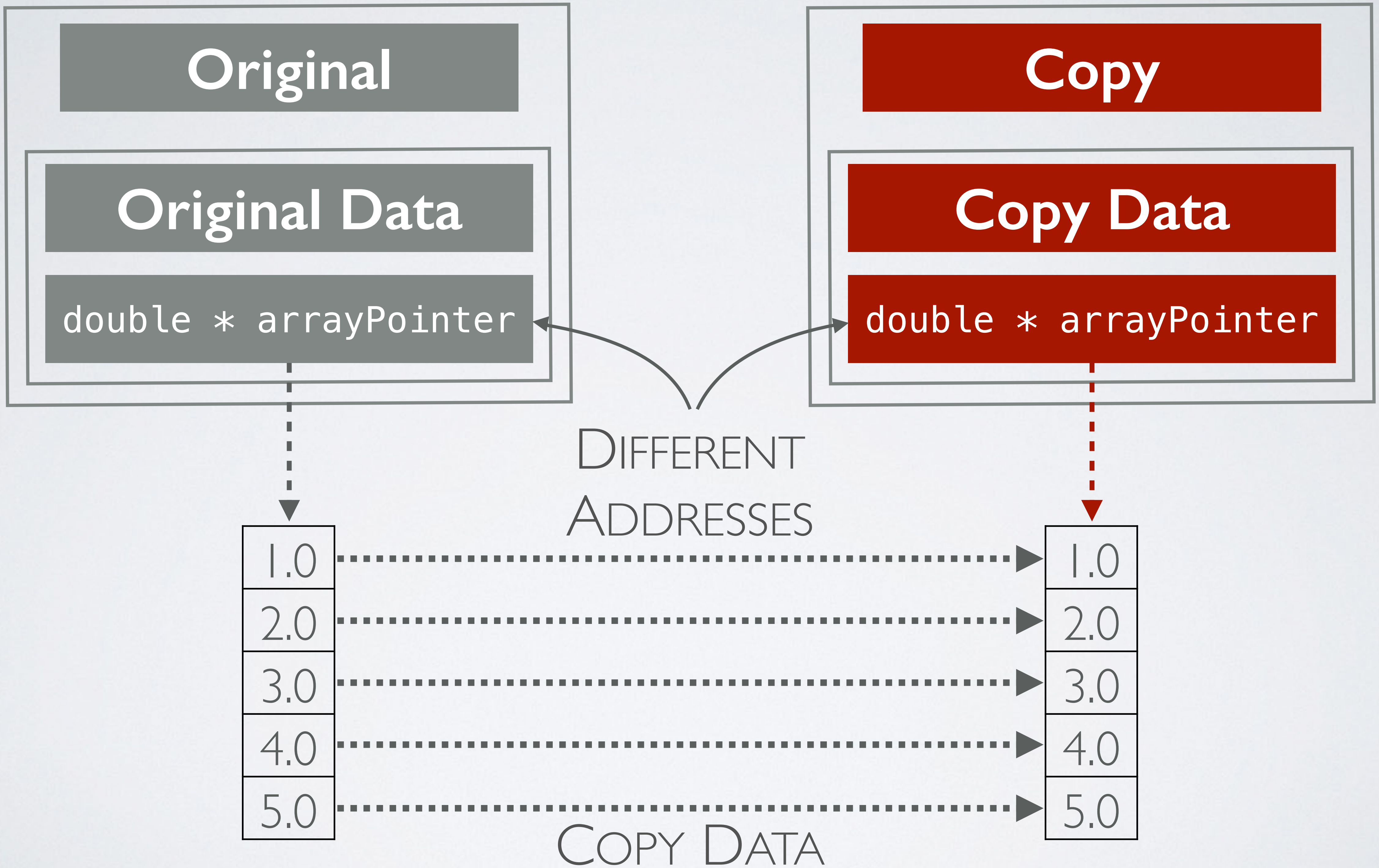
SHALLOW COPY



Which object is responsible for deallocating the shared data?

Will there be side-effects if Copy modifies the shared data?

DEEP COPY



Original
Original Data
double * arrayPointer
Copy
Copy Data
double * arrayPointer
Different Addresses
1.0
2.0
3.0
4.0
5.0
1.0
2.0
3.0
4.0
5.0
Copy Data

OO PROGRAMMING IN C++: OPERATOR OVERLOADING

OO PROGRAMMING IN C++: OPERATOR OVERLOADING

- C++ allows programmers to define the way that instances of their classes interact with the familiar C++ **operators** (**+**, **−**, **=**, etc).
- ✑ *The mechanism for implementing these definitions is known as* ***OPERATOR OVERLOADING****.*
- Operator overloading **reduces syntactic clutter** by replacing method calls with concise and expressions.
- For example the "**+**" operator is overloaded to perform concatenation when applied to **instances** of `std::string`, so e.g. `string1.append(string2)` becomes `string1 + string2`

OVERLOADING THE ASSIGNMENT OPERATOR

- In the absence of an explicit definition, C++ compilers provide a default implementation of the assignment operator for every **fully defined** class.
- The default assignment operator implements **shallow copy semantics** for **pointer-type member data**, which is normally sub-optimal.
- Accordingly, it is very common for C++ programmers to overload the assignment operator for the classes they define.
- Implementation of **deep-copy semantics** for pointer-type member data is a common motivation for doing so.

OVERLOADING THE ASSIGNMENT OPERATOR

- The assignment operator can be overridden for a class called `DemoClass` by defining a method with the signature

```
DemoClass & operator=(const DemoClass & otherInstance)
```

- The method returns a **reference** to an instance of `DemoClass`.
- In fact, this is a reference to the **class being assigned to** and is typically obtained by dereferencing the `this` pointer.
- The **single** method parameter is an **immutable** reference to **another** instance of `DemoClass`, the state of which will be copied to the class instance that is being assigned to.

DEMONSTRATION

Overloading the assignment operator.

Clone the C++ demonstration material from Github:

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

OO PROGRAMMING: **INHERITANCE**

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

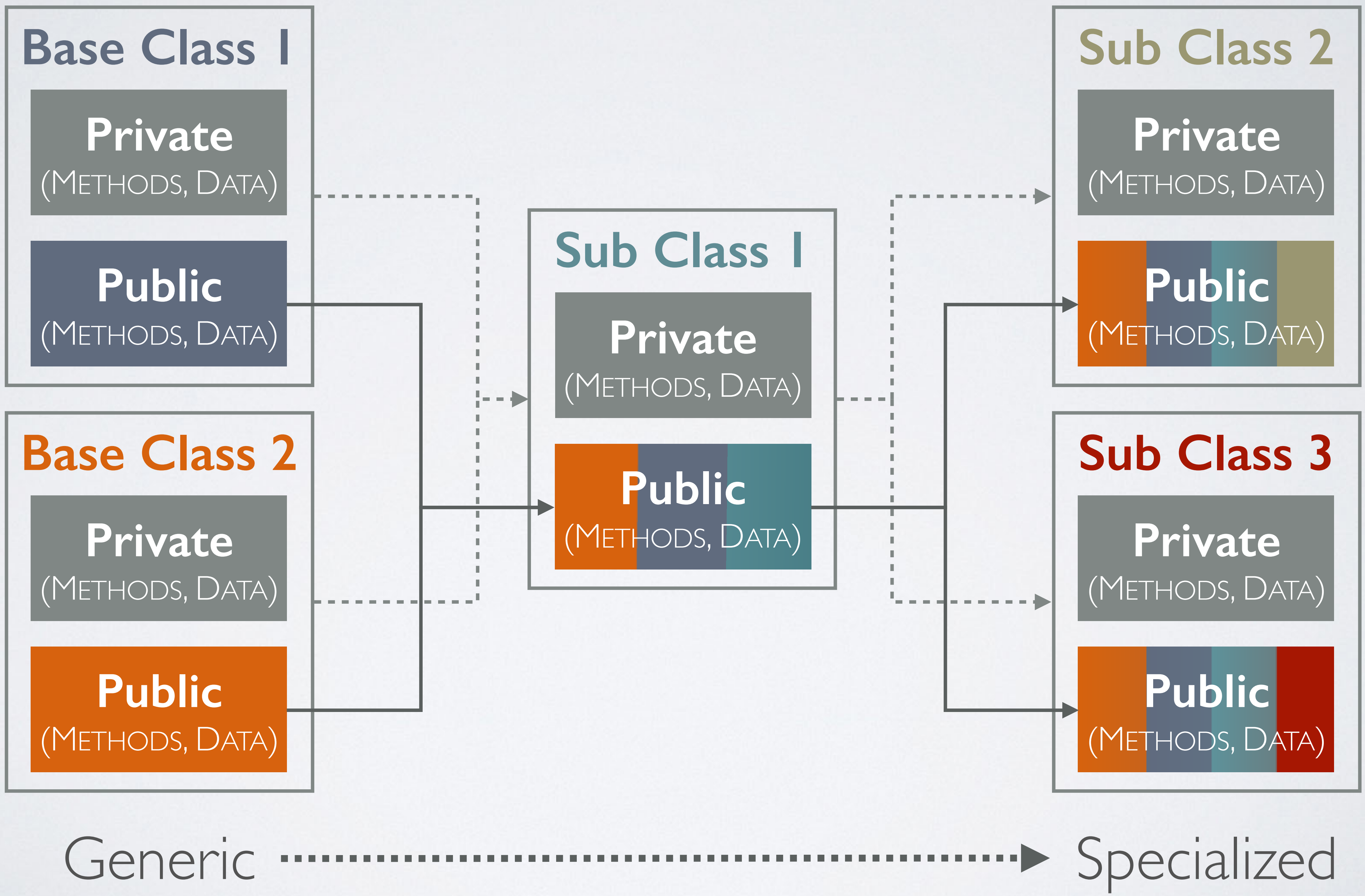
OO PROGRAMMING: INHERITANCE

- ***INHERITANCE*** *in OO programming is a* ***very*** *powerful mechanism for* ***reusing*** *and* ***extending*** *the functionality of preexisting classes.*
- Inheritance allows programmers to define a class that incorporates (or **inherits**) the method and member data definitions of another.
- By utilizing this capability, C++ programmers **do not need to redefine** methods and member data that **perform identical functions**.

OO PROGRAMMING: INHERITANCE

- *The class that **defines** the **inherited** functionality is called the **PARENT** (or **BASE**) class and the class that **inherits** the functionality is called the **CHILD** (or **DERIVED**) class.*
- Derived classes **may** inherit functionality from **several** base classes and **multiple** derived classes **may** inherit from **the same** base class.
- Derived classes **may** also act as the base class for further derivation, and **propagate** their inherited functionality to their children.

OO PROGRAMMING: INHERITANCE HIERARCHY



OO PROGRAMMING: INHERITANCE

- Inheritance also enables derived classes to **refine the functionality** of the methods they inherit by **providing customized definitions** of those methods.
- *This mechanism is called **METHOD OVERRIDING**.*
- To override an inherited method when defining a derived class, simply define a **new method** with an **identical signature** to the base-class method to be overridden

☞ Note the distinction between **overriding** methods that are inherited and **overloading** methods within a single class.

DEMONSTRATION

Inheritance and Overriding Methods

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

A **comprehensive reference** that describes **all the classes** provided by the **C++ Standard Template Library** can be found at: http://www.cplusplus.com/reference

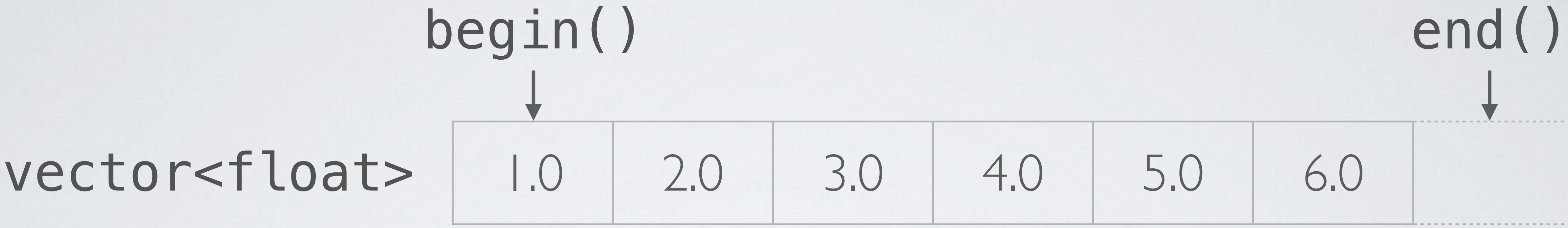
THE C++ STANDARD TEMPLATE LIBRARY

- The C++ Standard Template Library (STL) comprises a large number of **professionally developed** classes, that provide a rich array of functionality, including
  - Data **containers** that overcome many of the shortcomings of basic C++ arrays.
  - Highly **optimized** data processing **algorithms**.
  - Facilities for **reading** from and **writing** to **textual** and **binary** files.
  - Sophisticated facilities for **random number generation**.

CONTAINERS

- *The STL provides several **CONTAINER** classes that can be used to store collections of objects that have the **same type**.*
- Each STL container models a different **data structure**. Examples include `std::vector`, `std::deque`, and `std::map`.
- The **type** of a container explicitly includes the type of the objects it can contain. A `std::vector` that stores objects of type `X` is denoted `std::vector<X>`.
- **All** STL containers provide a ***uniform*** **iterator**-based interface that integrates seamlessly with the STL **algorithm** classes.

STL ITERATORS - BASICS



- ***ITERATORS** are abstract representations of an element **and** its position within an STL container.*
- **All** STL containers implement `begin()` and `end()` methods.
- The `begin()` method returns an **iterator instance** corresponding to the **first element** of the container.
- The `end()` method returns an **iterator instance** corresponding to **one-past-the-last element** of the container.

STL ITERATORS - BASICS

- Iterators provide **access** to their corresponding **container element** using an overloaded **dereference operator**, “`*`”.
- Iterators can be advanced to the **next container element** using overloaded **prefix or postfix** increment operators, “**++**”.
- Iterators can be compared using overloads of the familiar C++ comparison operators i.e. “`==`”, “`!=`”, “`<`”, “`>`”, etc.
- Each type of **container** has an **associated iterator type**.
- If the container’s type is *`container`*<X>, then the type of its **associated iterator** is *`container`*<X>`::iterator`.

STL ITERATORS - BASICS

- Iterators can be used to **process the elements** of an STL container using familiar C++ **loop statements** e.g.

```
// Instantiate and intialize a std::vector containing doubles
std::vector<double> doubleVec = { 1.2, 2.3, 3.4, 4.5 };
// Declare an appropriate iterator instance.
std::vector<double>::iterator doubleVecIt;

for(doubleVecIt = ++doubleVec.begin();// start at the 2nd element
    doubleVecIt != doubleVec.end(); // stop at the last element
    ++doubleVecIt) // move to the next position
  {
    // Add 2 to the vector element
    *doubleVecIt += 2.0;
  }
```

STL ITERATORS - BASICS

- C++ also provides a **compact loop syntax** that can be used to process **all** the elements in a container.

```
// Instantiate and intialize a std::vector containing doubles.
std::vector<double> doubleVec = { 1.2, 2.3, 3.4, 4.5 };

// loop over all elements of doubleVec.
for(double & element : doubleVec)
  {
    // Add 2 to the vector element.
    element += 2.0;
  }
```

- Iterators are also used to provide **input element ranges** to the STL **algorithm** classes e.g. the `std::sort` algorithm

```
// Sort all vector elements into ascending order.
std::sort(doubleVec.begin(), doubleVec.end());
```

DEMONSTRATION

The Standard Template Library (STL)

Clone the C++ demonstration material from Github:

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

LECTURE 5 SUMMARY

- After reviewing the material in this lecture **and completing the reading exercises** you should know:
  1. The **meaning** and **utility** of the `this` pointer in the context of OO programming in C++.
  2. The distinction between **deep** and **shallow copy semantics** for pointer-type member data.
  3. That **operators** can be **overloaded** for user-defined types in C++.

LECTURE 5 SUMMARY

4. That the default assignment operator that is provided **automatically** by the C++ compiler implements **shallow copy semantics**.
5. How to define an overload of the **assignment operator** that implements **deep copy semantics** for classes that you define.
6. The fundamental principles of **inheritance** in the context of object-oriented C++.

LECTURE 5 SUMMARY

7. The meanings of the terms **base** (or **parent**) **class** and **sub-** (or **child**) **class**.
8. How to specify that the classes you define **derive** (or **inherit**) from other pre-existing classes.
9. How to provide **specialized functionality** in a subclass by **overriding** methods that are inherited from a base class.

LECTURE 5 SUMMARY

10. A **summary** of the functionality that is provided by the the **C++ Standard Template Library**.
11. How to **instantiate** STL **containers** that can hold a **particular type**.
12. The basics of the STL **iterator interface** that is provided by STL container classes.
13. How to **instantiate an iterator** that can process the elements of a **particular container type**.

LECTURE 5 SUMMARY

14. That STL iterators provide **access to an element** of an STL container using an overload of the **dereference operator**, "*".
15. That STL iterators can be **advanced** to the next element of a container using the "**++**" operator.
16. That STL iterators can be **compared** using overloads of the familiar C++ comparison operators like "**==**".

LECTURE 5 SUMMARY

17. That **all** STL containers provide a `begin()` and `end()` method
18. That the `begin()` method returns an iterator that corresponds to the **first element** of the container.
19. That the `end()` method returns an iterator that corresponds to **one-past-the-last** element of the container.

LECTURE 5 SUMMARY

20. How to use a **normal** C++ loop statement to process elements of an STL container.
21. How to use the special **compact for loop syntax** to process **all** of the elements of an STL container.
22. How to invoke an STL **algorithm** to **sort the elements** of an STL container.
23. The basic principles of **textual file I/O** in C++.

LECTURE 5 SUMMARY

24. How to update **cloned** Git working directories to reflect the most recent state of a remote repository.
25. How to update **remote** Git repositories to reflect **committed** changes to a **cloned** working directory.

LECTURE 5 HOMEWORK

Read sections:

- Classes → Friendship and inheritance
- C++ Standard Library → Input/Output with files

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

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

Be sure to thoroughly review the C++ demonstration material!

Reminder! Please sign up for a **GitHub account** at: https://github.com then send me an email with your GitHub username.

- Complete the **Lecture 5 Homework Quiz** that you will find on the course Blackboard Learn website.