(PRACTICAL) COMPUTATIONAL PHYSICS

Physics 551
Lecture 4

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

- The **homework quizzes** from Lecture 2 have been marked.

☞ An **anonymous survey** has been posted on the course's Blackboard Learn website. The survey is designed to help make the course more relevant for all participants.

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

```
$git clone https://github.com/hughdickinson/CompPhysL4CPP.git /home/computationalphysics/Documents/cPlusPlus/lecture4
```

☞ You can also find this command on the Blackboard Learn website.

CLARIFICATIONS

- Distinguishing variable declaration and initialization
  - A variable **declaration** looks like

    `type-specifier identifier;`

  - It informs the compiler that the programmer intends to refer to `identifier` at a later point in the program and that `identifier` should be interpreted as referring to a specific `type`.
  - **Declaring** a variable **may** result in **default initialization** i.e. an `int` **might** be initialized to 0 by default.

☞ You should **not rely** on default initialization.

CLARIFICATIONS

- A **basic** variable **initialization** looks like:

```
identifier = value;
```

- This assigns a `value` to the memory that is referenced by `identifier`.
- Declaration and initialization can be combined in a single statement using several syntaxes e.g.

```
int intVar = 0; int intArray[2] = {0, 1};
double doubleVar(2.5); float * ptrToFloat{nullptr};
```

- **Immutable** (constant) variables **must** be initialized when they are declared.

CLARIFICATIONS

- When to include the `<iostream>` header.
  - Almost all of the examples in the reading assignments include the `<iostream>` header.
  - This seems to have given some students the impression that this header is mandatory in any C++ code.
  - In fact the `<iostream>` header is **only** required if the program is required to perform textual input/output.
  - Most of the examples in the reading assignments do perform I/O and this is why they include the `<iostream>` header.

CLARIFICATIONS

- How C++ passes **arrays** as function **arguments**:
  - ☞ When **array-types** are specified as **function parameters**, C++ will pass a **pointer** containing the address of the **first element** of the array.
  - This is why functions accepting array arguments almost always require an **integer argument** specifying the **number of elements** in the array.
  - This means that **mutations** to arrays that are passed as function arguments **are retained *after*** the function returns.

DEMONSTRATION

Even More C++
(Continued from Lecture 3)

Clone the C++ demonstration material from Github:

```
$ git clone https://github.com/hughdickinson/CompPhysL3CPP.git
  /home/computationalphysics/Documents/cPlusPlus/lecture3
```

"COMPILING" C++ CODE

- So far, we have used the **cling** interpreter to experiment with the components of the C++ language.
- Cling permits algorithmic **evaluation** and **prototyping** using code snippets that do not constitute a valid C++ program.
- However, all **complete** C++ programs should be **compiled** and executed as **standalone binary executables**.
- Several compiler utilities exist for C++, including **proprietary** compilers like Microsoft's `Visual C++` as well as **open-source** options like the GNU Compiler Collection's `g++` and `cling`'s backend compiler, `clang++`.

THE MAIN FUNCTION

- **All compiled** C++ programs **must** define a function called `main`.
- *The `main` function is called the **ENTRY POINT** of the compiled program.*
- When the operating system runs your program it will automatically call the `main` function.

☞ The `main` function **must** return an **integer** value.

- Several function **signatures** are permitted for the `main` function.

THE MAIN FUNCTION

- The **most commonly used** definition of the main function in C++ looks like:

```
int main(int argc, char * argv[]){
    /* …code goes here… */
    return 0;
}
```

- **This** function signature has **two parameters** that relate to the shell command your program was invoked with
  1. The integer parameter **`argc`** (**arg**ument **c**ount) counts the number of command line tokens **including the program name** comprising the invocation.

THE MAIN FUNCTION

2. The second parameter **`argv`** (**arg**ument **v**ector) is actually an array of **`argc`** pointers to **arrays** of characters (i.e. strings) that specify the **values** of the command line tokens.

☞ The main function should return 0 on success and non-zero values otherwise.

- Consider the second parameter of the **`main`** function:

```
char * argv[]
```

- How should this expression be interpreted? What are its components?
- Beginning from the **right**, the "`[]`" token makes it clear that this parameter should be an **array**.

THE MAIN FUNCTION

- The "`argv`" token is simply the identifier that can be used to reference the array within the `main` function.
- The token pair "`char *`" is the **type specifier** for the elements of the array i.e. **pointer-to-character**.
- Assembling these token interpretations, we should read:

  "`argv` is an **array** of **pointers-to-characters**"
- Finally, recall that a pointer-to-character could **actually** point to the **first element** of an **array of characters**. In this case **it does**!

THE MAIN FUNCTION

- Putting everything together, the statement

  "`argv` is an **array** of **pointers** to the **first elements** of **arrays of characters**"

  is the correct way to interpret

  `char * argv[]`

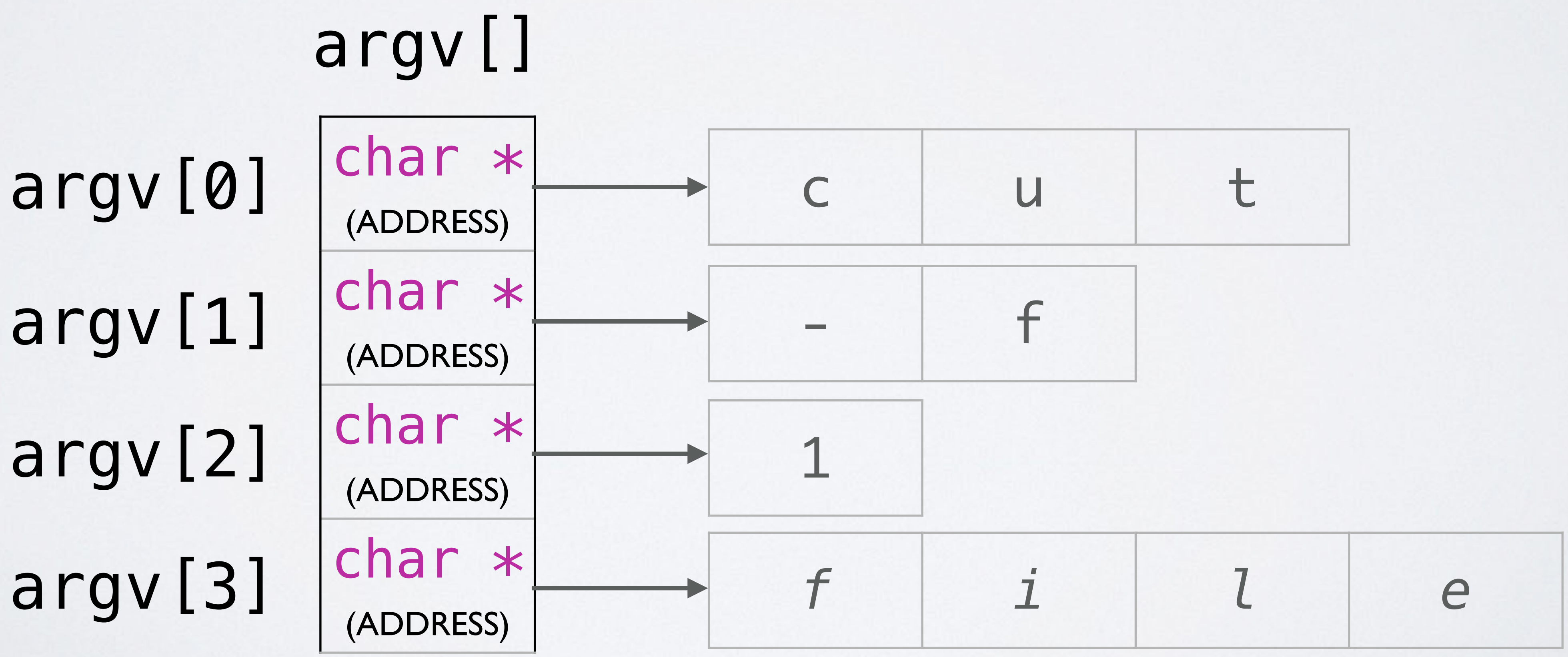
- The obvious question is then "How is the C++ program able to infer the **lengths** of the arrays of characters?"
- The first part of this week's reading assignment addresses this issue.

THE MAIN FUNCTION

- **Assume** that the `cut` shell command was implemented using C++.
- Invoking

```
$ cut -f 1 file
```

Results in `argc = 4` and the following layout for `argv`



BUILDING OUR PROGRAM

- We will use the `clang++` utility to convert our C++ code into a binary executable. man clang (**not** man clang++!)
- To simplest possible invocation of `clang++` looks like

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

- This command actually does several things - compilation being one of them!
  1. **Preprocesses** each of the *`sourceCodeFiles`*. The code in included **header files** is merged with your source code at this stage.
  2. **Compiles** each of the **preprocessed** *`sourceCodeFiles`* into intermediate **assembler** files.

INVOKING THE COMPILER

- ☞ Assembler code is expressed in a special language with instructions that are optimized for a **specific computer architecture**.

3. **Links** each of the assembled **binary object files** and any required **static libraries** into the main binary executable.
4. Adds references to any **dynamic** (or **shared**) **libraries** that will provide executable code **at runtime**.

- ☞ Even the simplest C++ programs will probably use elements of the **C++ standard library** at runtime.

5. Generates the specified binary executable with the correct filesystem **permissions** (check with `ls -l`) to enable its execution.

DEMONSTRATION

Building a simple C++ program

Clone the C++ demonstration material from Github:

```
$ git clone https://github.com/hughdickinson/CompPhysL3CPP.git
  /home/computationalphysics/Documents/cPlusPlus/lecture3
```

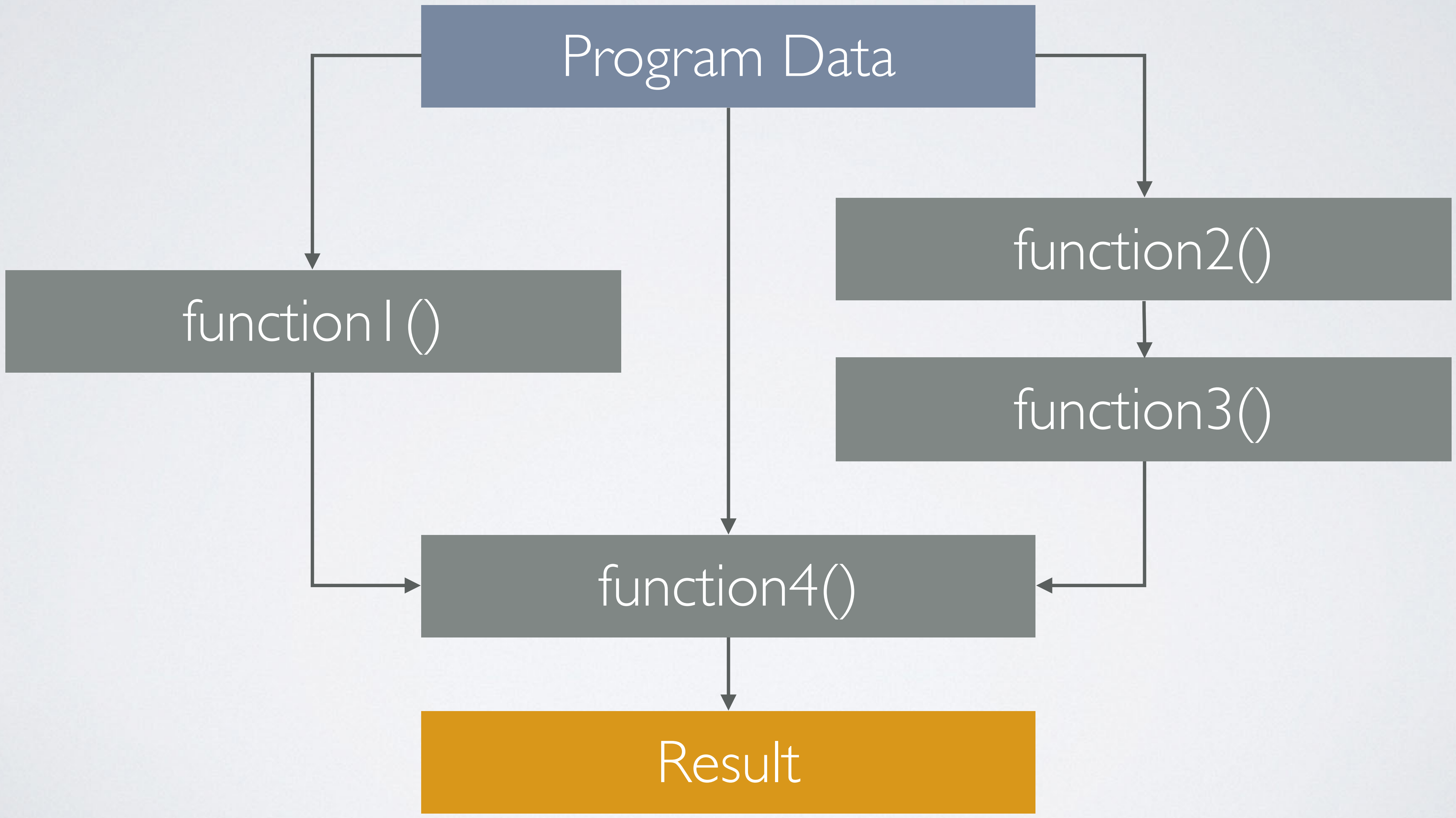
OBJECT ORIENTED PROGRAMMING

- The code we have written so far exemplifies the so-called **procedural programming** paradigm.

➾ ***PROCEDURAL PROGRAMMING*** *entails definition and invocation of **functions** (or **procedures**) that **operate** upon **data** that are passed to them.*

- In procedural programs, programatic data that are passed are typically available to all functions.
- Particular data are not explicitly associated with particular functions.

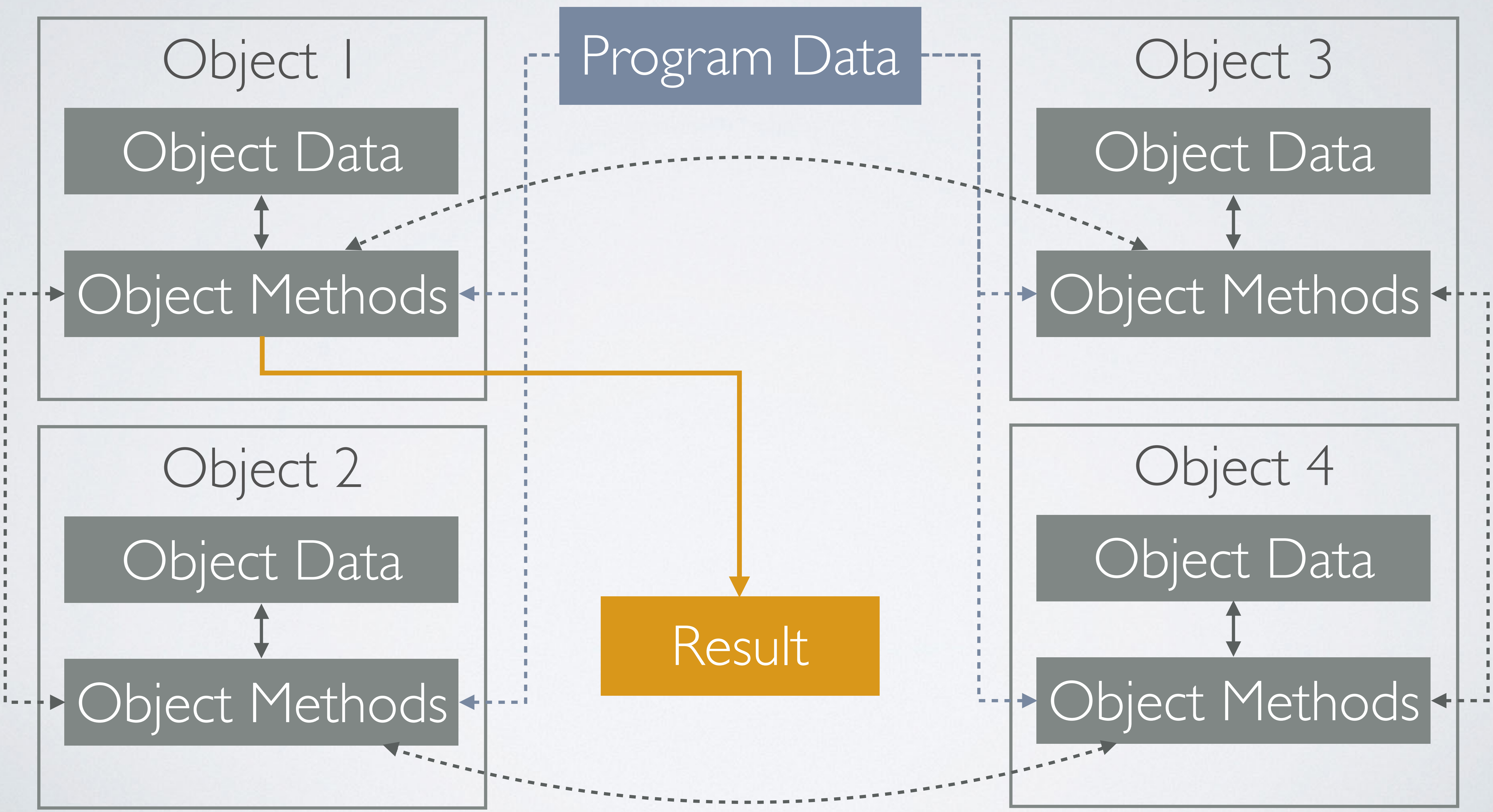
PROCEDURAL PROGRAMMING MODEL



OBJECT ORIENTED PROGRAMMING

- C++ is designed to implement another programming paradigm known as **object orientation**.
- ***OBJECT ORIENTED (OO) PROGRAMMING*** *entails* ***definition*** *and* ***management*** *of* ***objects*** *that have* ***associated data*** *that they may* ***exchange*** *or operate upon* ***internally****.*
- *In OO programs, programatic data are not globally accessible. Instead they are* ***ENCAPSULATED*** *within the* ***objects*** *that comprise the program.*
- ***External access*** *to encapsulated data is enabled by the* ***METHODS*** *of an object. Methods behave* ***like*** *functions* ***but*** *are explicitly associated with a* ***type*** *of object.*

OBJECT ORIENTED PROGRAMMING MODEL



DEMONSTRATION

Object Orientation in C++

Clone the C++ demonstration material from Github:

```
$ git clone https://github.com/hughdickinson/CompPhysL4CPP.git /home/computationalphysics/Documents/cPlusPlus/lecture4
```

LECTURE 4 SUMMARY

- After reviewing the material in this lecture **and completing the reading exercises** you should know:
  1. How to **dynamically allocate memory** for **pointer-to-array** types using the `new[]` operator.
  2. How to **free dynamically allocated memory** using the appropriate `delete` or `delete[]` operators.
  3. How to **index** the elements of **pointer-to-array types** i.e. in an identical fashion to normal array types.

LECTURE 4 SUMMARY

4. The implications of passing function arguments **by value** for the persistence of in-function modifications.
5. That arrays are **always** passed as **pointers to their first element**.
6. The implications of this for the persistence of in-function modifications to array elements.
7. How to declare and initialize **references** in C++,

LECTURE 4 SUMMARY

8. That **all** references **must** be **initialized when they are declared**.
9. That references define an **alias** for a **preexisting** identifier.
10. How to write functions that specify that their arguments are passed **by reference**.
11. The implications of this for the persistence of in-function modifications to such arguments.

LECTURE 4 SUMMARY

12. That **literal** values **cannot** be passed to functions as (mutable) references.
13. The meaning and purpose of **namespaces** in C++.
14. How to **define** namespaces in C++.
15. How to **explicitly specify** that an identifier belongs to a namespace using the **scope resolution operator** "**: :**".

LECTURE 4 SUMMARY

16. That entities provided by the C++ standard library are defined within the "`std`" namespace.
17. That mathematical functions and constants are provided by the "***`cmath`***" header file.
18. How to write complete, **compile-able** C++ programs that contain the **required** `main` function.

LECTURE 4 SUMMARY

19. That the `main` function **must** return an **integer** value upon completion - conventionally, **zero on success**.
20. The form of a commonly used **signature** for the `main` function that gives access to the **shell tokens** used to invoke the **compiled executable**.
21. How to **build** a simple C++ program using the `clang++` compiler.

LECTURE 4 SUMMARY

22. A **basic definition** of Object-Oriented programming.
23. How OO programming **differs** from **procedural programming**.
24. How to write **classes** that ***describe*** the **properties** and **behaviour** of objects that comprise an OO program. ***In particular***
    i) The syntax required to **declare** a class in C++.
    ii) The syntax required to **define** a class in C++.

LECTURE 4 SUMMARY

25. The meanings of the terms **member data** and **method** in the context of OO programming.
26. How to control member data and method **accessibility** using the ***public*** and ***private*** keywords.
27. How to **declare** member data and methods **within** a class definition.
28. How to provide **definitions** of methods **inside and outside** of the class definition.

LECTURE 4 SUMMARY

1. The purposes of the special ***constructor*** and ***destructor*** methods in a C++ class definition.
2. How to **initialize** member data in the constructor and safely **deallocate** acquired resources in the destructor.
3. How to **instantiate *classes*** in order to create ***objects*** in your program.
4. How to **access *public*** member data or **call *public*** methods that are exposed by objects using the "**.**" operator.

LECTURE 4 SUMMARY

29. How to create **pointers-to-objects** using the **new** keyword.
30. How to **access *public*** member data or **call *public*** methods of **pointers-to-objects** using the "**->**" operator.
31. How to **obtain the address** of an object or normal variable using the "**&**" operator.

LECTURE 4 HOMEWORK

Read sections:

- Compound data types → Character sequences
- Classes → Classes (I)
- Classes → Classes (II)
- Classes → Special members

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

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

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

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

LECTURE 4 HOMEWORK

In preparation for next week's lecture, you may want to read Chapter 2.5 from the **Git Pro Book**

- Complete the **Lecture 4 Homework Quiz** that you will find on the course Blackboard Learn website.

☞In response to **student requests**, all questions will be made available immediately and you will **not** be required to complete them in order.

☞Please complete the **anonymous survey** that you can find on the Blackboard Learn website.