Strings, Classes & Pointers

COSC1076
Semester 1 2019
Week 02

Admin

- Ask non-personal questions on the forum
  - If you email a non-personal question I will ask you to re-post it on the forum
- Assignment 1
  - Released after lecture
  - Discussed next week
    - You can read up on it, and start now if you wish
- Labs
  - Run 1 week behind lectures

C++ Style & Course Style Guide

RMIT UNIVERSITY

Declaration vs Definition vs Initialisation

- Declaration
  - Introduce a name (variable, class, function) into a scope
  - Fully specify all associate type information
- Definition
  - Fully specify (or describe) the name/entity
  - All definitions are declarations, but not vice versa
- Initialisation
  - Assign a value to a variable for the first time

Namespaces

- Define a new scope
  - Similar to packages in Java
  - Useful for organising large codebases

```
namespace myNamespace { … }
```

- Function, Class, Variables, etc labels can be enclosed within a namespace
  - The namespace must be referenced to access the entity, using ::

```
<namespace>::<label>
```

  - Namespaces can be nested

```
namespace namespace1 { namespace namespace2 { … } }
<namespace1>::<namespace2>::<label>
```

Namespaces

- Namespace entities can be exported

```
using std::cout
```

- Everything in a namespace can be exported

```
using namespace std
```

  - This is banned within this course
- The `std` namespace
  - Most STL entities we will use exist within the std namespace

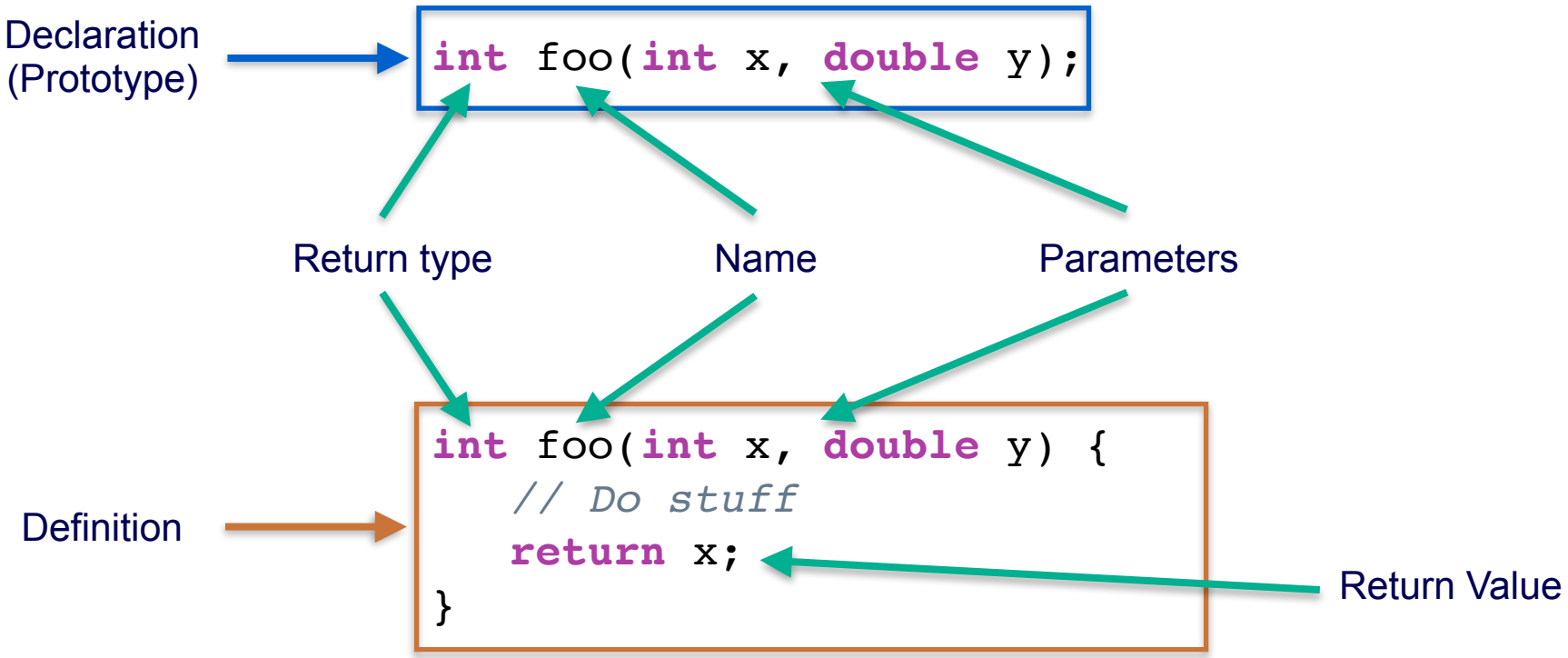
Global Variables

- So far, all variables have been *defined* within the *scope* of a function.
  - The variable only exists within that function
  - The variable cannot be referenced from elsewhere
- A variable defined *outside* of any function is global
  - Can be used within any function, so long as the definition appears before the variable is used
  - These are incredibly bad design and style

Functions

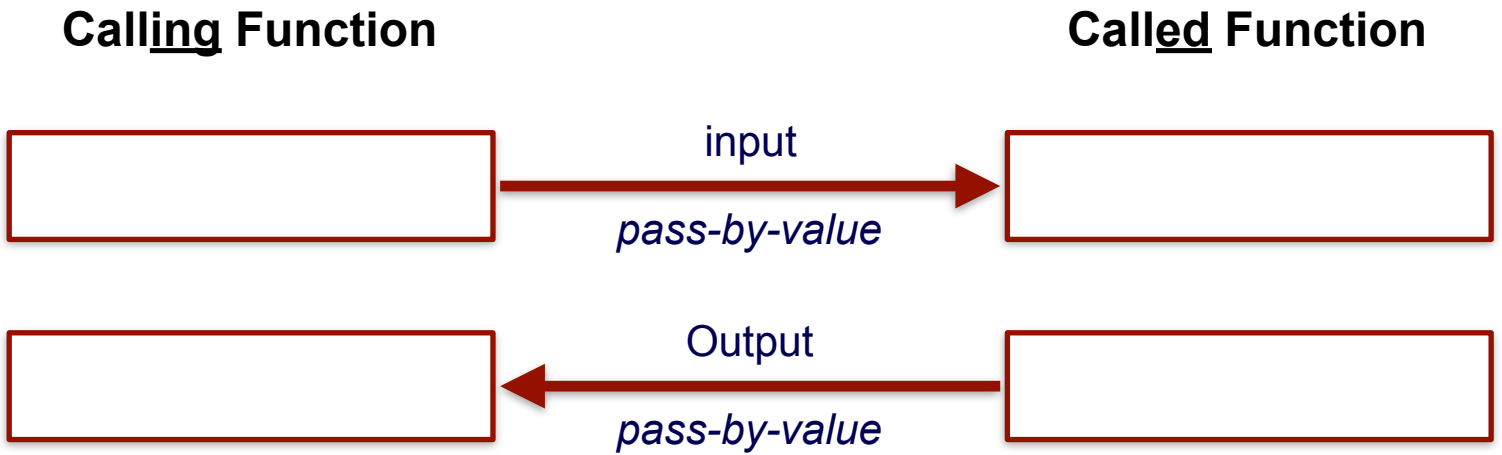
- Similar in concept to Java Methods
- Functions are not associated with a class, and sit in the “Global" scope
- Usage:
  - Functions must be *declared* before they can be used (called)
    - A function declaration is also called a ***function prototype***
  - Functions must only be *defined* once
    - This can be after it is called
    - It doesn’t not even have to be in the same cpp file! (more on this later)
- Pass-by-value
  - Pass-by-reference later (next week)
  - Array passing (next week, more detail)

Functions


Declaration
(Prototype)
int foo(int x, double y);
Return type
Name
Parameters
Definition
int foo(int x, double y) {
// Do stuff
return x;
}
Return Value

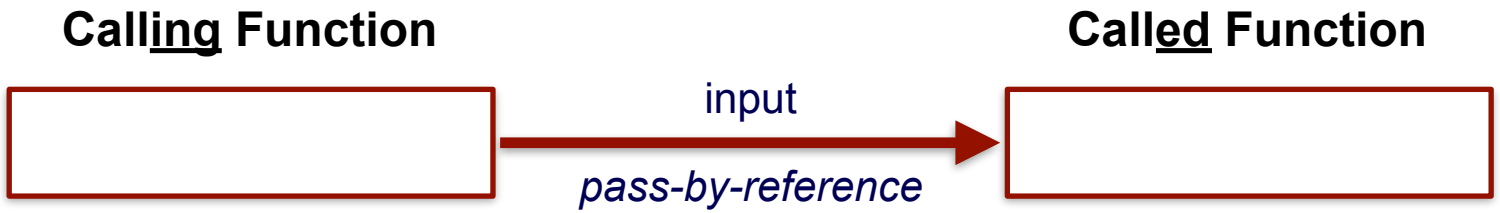
Functions

- Function calls operate through an approach called ***pass-by-value***
  - The value of the parameter is ***copied*** when it is given to the function
  - Changing the parameter within a function does not modify the value from the calling entity
  - This is similar to primitive types in Java



Functions

- Arrays are different** (sort-of)
  - Arrays (as parameters) operate through ***pass-by-reference***
  - The actual array is passed.
    - Changing a value in the array within the called function modifies the value from the calling function



** As we will see next week:

- Under-the-hood an array is implemented using a *pointer*
- The pointer is copied (pass-by-value)
- The high-level effect to the programmer is pass-by-reference

Arrays

- Similar to Java Arrays
  - Largely syntactic difference when declaring
  - No need to “new” the array

```
int a[LENGTH];
```

  - Can be initialised when declared

```
int a[LENGTH] = {1};
```

- BUT, not automatic bounds checking!

  - Cells “before” and “after” and start/end of the array can be accessed!
  - It is the programmer’s responsibility to ensure that a program does not access outside an array’s limits.

Multi-Dimensional Arrays

- Multi-dimensional arrays
  - Again, similar to Java

```
int a[DIM1][DIM2];
```

  - Inline initialisation is trickier

```
int a[DIM1][DIM2] = { {1,2,3}, {4,5,6}, …};
```

Arrays - Passing to Functions

- Use the “array type” (square brackets) as the parameter
  - That is:

```
void foo(int array[]);
```

  - This is technically “pass-by-reference”, we will see this later in the lecture
  - This *does not* work with multi-dimensional arrays

Strings

RMIT UNIVERSITY

Array of Characters

- What is a string?
  - Sequence (or array) of characters
  - In original C, “strings” did not exist - just an array of `char`’s!

```
char string[LENGTH];
```

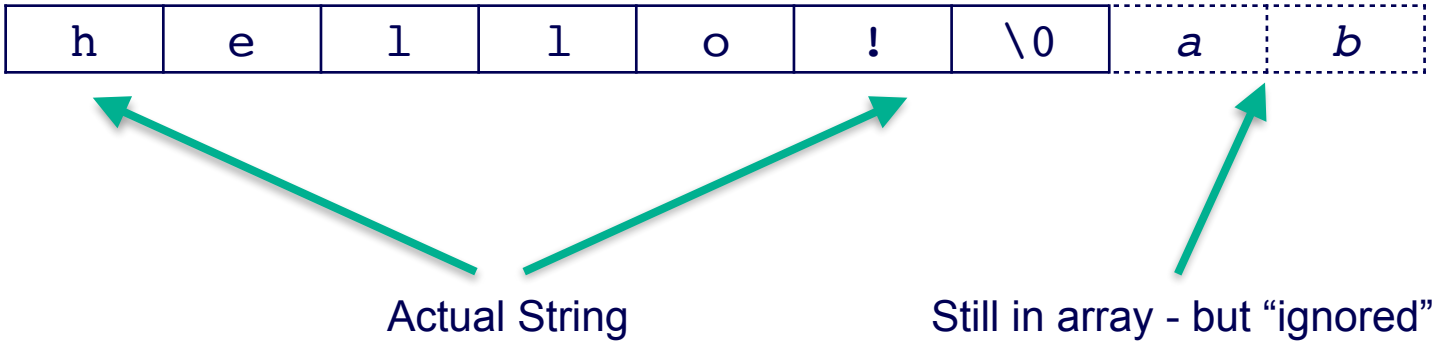
h	e	l	l	o	!

Except there is a problem.
How do you find the “end” of a string?

String Termination Character

- The null '\0' termination character denotes the "end" of a string

- If the '\0' appears in the middle of a "string" it actually terminates earlier
  - The rest of the array is ignored



Constant Strings

- Anything that is enclosed in double-quote “” are a ***constant string***
  - The compiler generates:
    - A character array
    - Of the exact length required
    - Guaranteed to end in a ‘\0’
  - Constant strings cannot be modified
  - Can be assigned or copied to ***mutable*** strings

std::string Class

- For this course, we will mostly use the std::string class
- As std::string is a class it has many useful methods

length()	Get the length of the string
at(int)	Get the character at the i-th position in the string
append(string)	Append another string to the end of ‘this’ string
substr(int,int)	Return the sub-string between two locations
c_str()	Get a c-style version of the string (if it is needed)

- Also works with typical operators
  - + to concatenate
  - == to compare
- See https://en.cppreference.com/w/cpp/string/basic_string

Pointers & References

The most important topic in the course!

RMIT UNIVERSITY

Computer Memory

- At the lowest level, Memory is a grid of ***bits***
  - Each bit only stores one value - 0 or 1.

0	1	1	0	0	1	0	0
0	0	1	0	0	0	0	1
...	...	...	...	...	...	...	...

Computer Memory

- For a “high-level” language this is not very useful.
  - Memory is grouped together into ***8-bit*** chucks, called a ***byte***
  - The entire byte is interpreted as a whole, for example a character

0	1	1	0	0	1	0	0	=	d
0	0	1	0	0	0	0	1	=	!
…	…	…	…	…	…	…	…		

Computer Memory

- Each ***byte*** has a unique ***address***
  - This is how a computer can find a piece of memory
  - Addresses are stored in hex, and adjacent memory locations are sequential
  - This is how variables in Java/C/C++, etc are stored

0x0004fca4	0	1	1	0	0	1	0	0
0x0004fca5	0	0	1	0	0	0	0	1
0x0004fca6	…	…	…	…	…	…	…	…
0x0004fca7								
0x0004fca8								
0x0004fca9								
0x0004fcaa								
0x0004fcab								

Variable

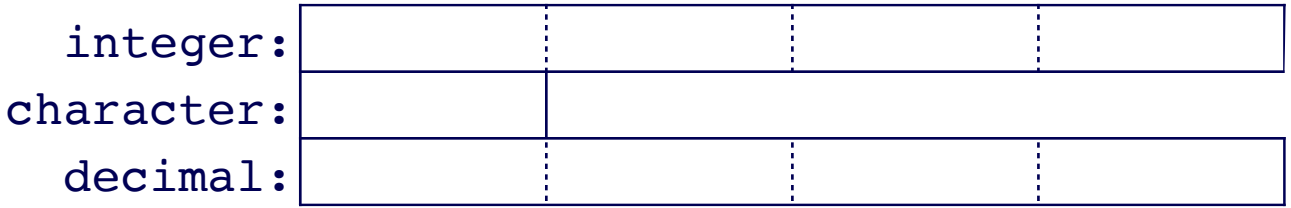
- Each variable in a program is stored in one or more bytes of memory

`char`	8 bits	1 byte
`int`	32 bits	4 bytes
`long`	64 bits	8 bytes
`short`	16 bits	2 bytes
`float`	32 bits	4 bytes
`double`	64 bits	8 bytes
`long double`	80 bits	10 bytes

Declaring a Variable

- When a variable is declared, the program (operating system):
  - Reserves the correct number of bytes in memory
- A variable name acts as a label for the location in memory

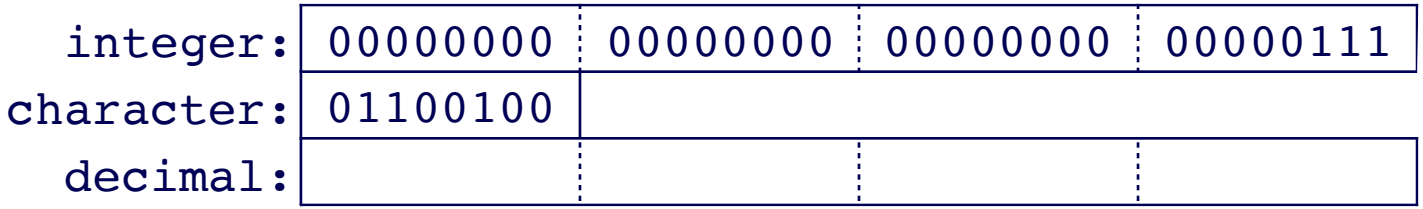
```
int    integer;
char   character;
float  decimal;
```



Assigning a Variable

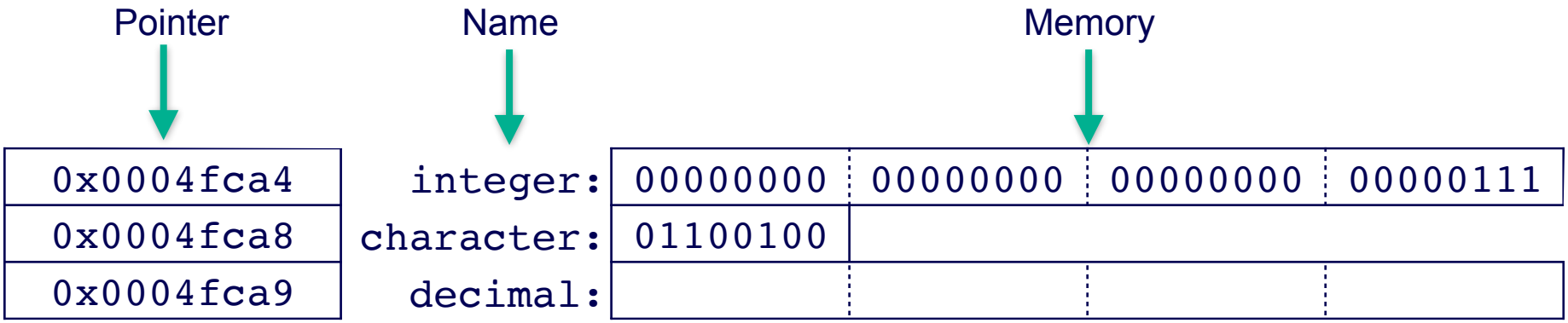
- When a value is assigned to a variable
  - The program locates that variable in memory, and
  - Updates the memory element

```
integer = 7;
Character = ‘d’
```



Pointers

- A ***pointer*** is the ***address*** of a variable
- A pointer is called this because it is said to ***"point to a variable"***

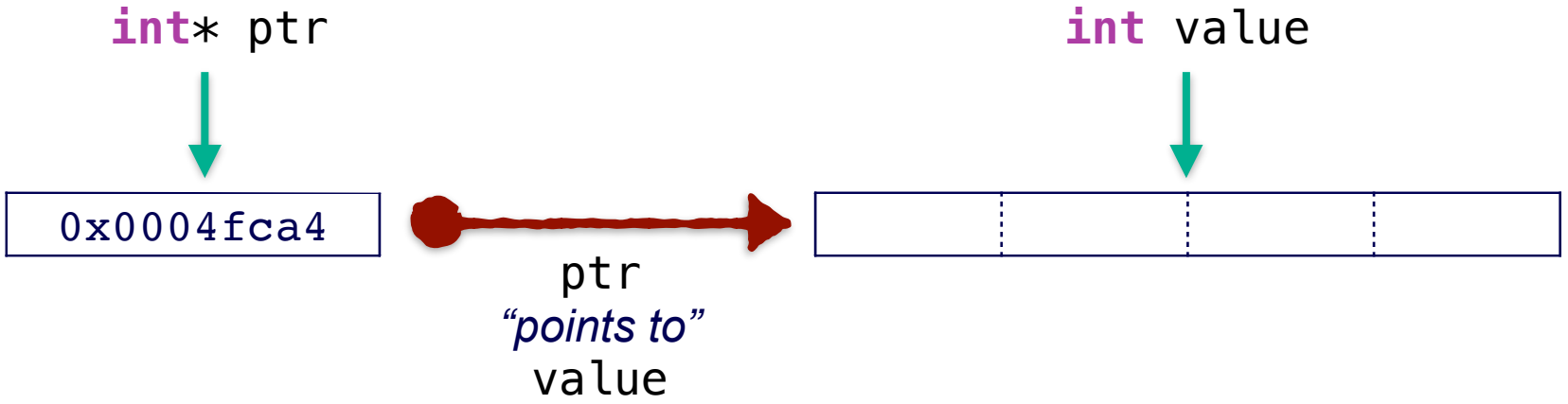


Pointers

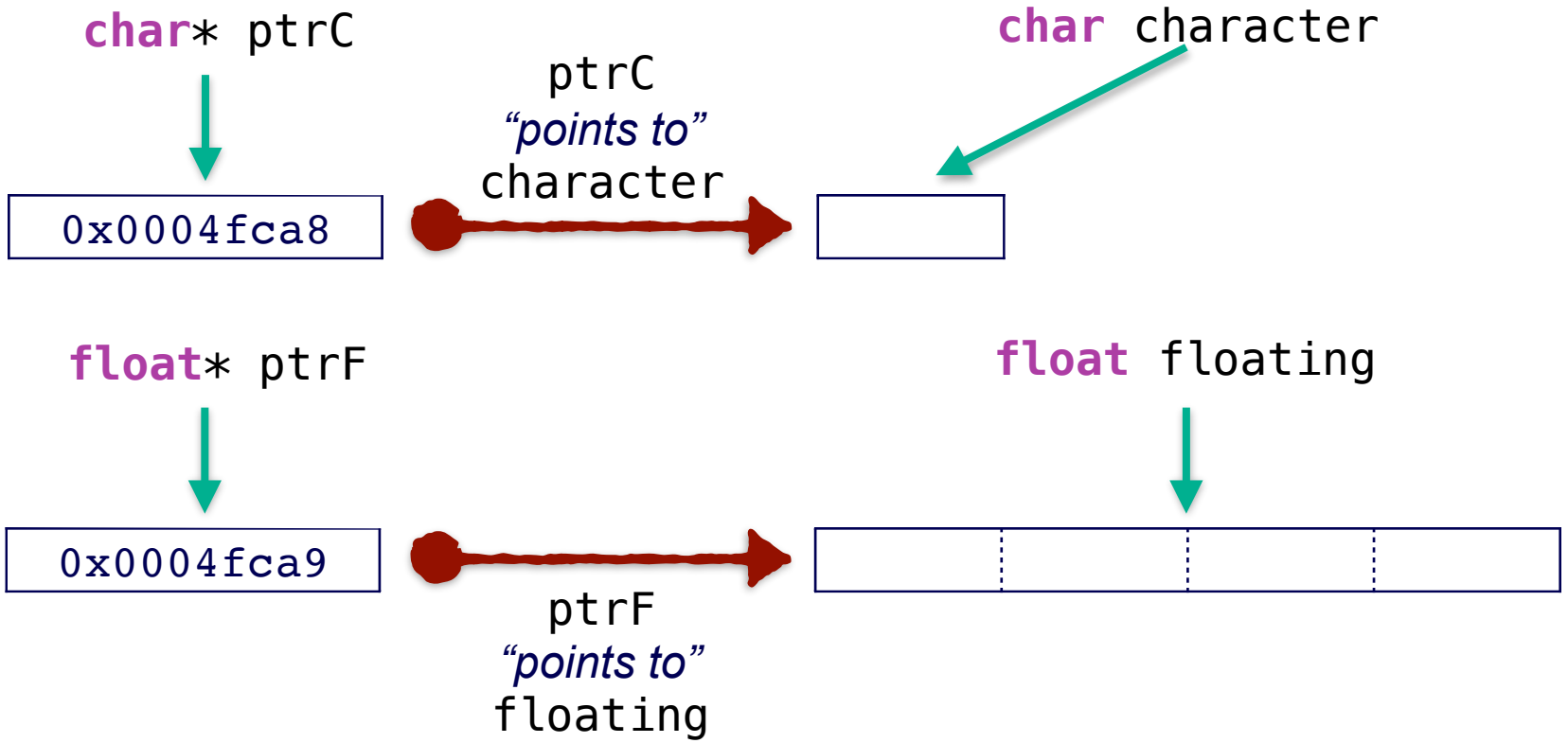
- A pointer type is denoted in syntax using a *
  - Syntax:

```
<type>* <name>
```

  - It is possible to have a pointer to *any* valid type
  - Pointers must be of the correct type for them to work!

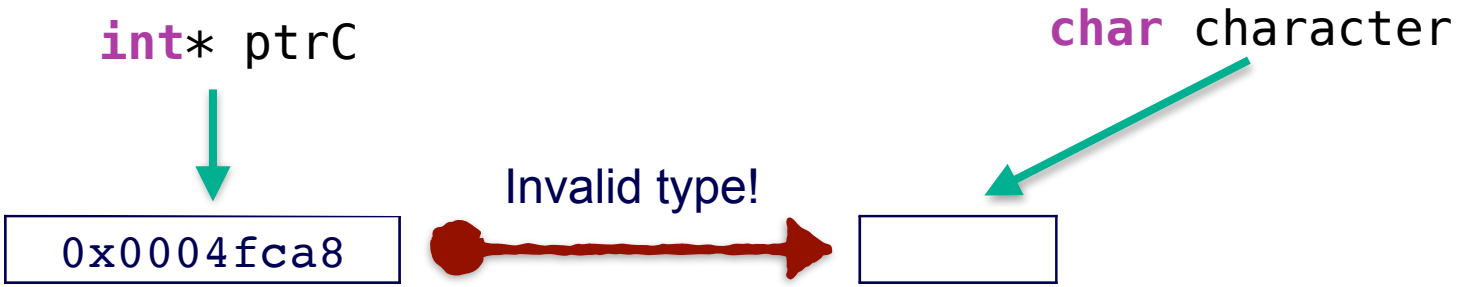


Pointers


char* ptrC
0x0004fca8
ptrC
"points to"
character
char character
float* ptrF
0x0004fca9
ptrF
"points to"
floating
float floating

Pointers

- Pointers must be of the correct type



Pointers - Addressing

- The place that a pointer “points to” must have been previously ***allocated***
  - Recall that defining a variable, allocates the memory for it
- The address of an existing variable is retrieved using the ‘&’ operator

```
int  value = 7;
int* ptr = &value;
```

Pointers - Addressing

- If a pointer does not have an appropriate variable to “point to”:
  - The value is set to 0x00000000,
  - That is, NULL

```
int* ptr = NULL;
```

  - (The capitals are necessary, lower case gives an error)

Pointers - Dereferencing

- To get the actual value of what a pointer “points to”:
  - The pointer needs to be ***dereferenced***
  - If a pointer addresses a location which has not been allocated, or NULL, dereferencing this results in a ***segmentation fault***
- The address of an existing variable is retrieved using the ‘*’ operator
  - Don’t get confused with using ‘*’ when *declaring* a pointer.

```
int  value = 7;
int* ptr = &value;
*ptr = 10;
```

Pointers

- Pointers can be copied between compatible types

```
int  value = 7;
int  other = 10;
int* ptr1 = &value;
int* ptr2 = &other;
*ptr = 10;
ptr1 = ptr2;
*ptr1 = 15;
```

Pointers

- Pointers can be “chained”

```
int  value = 7;
int* ptr = &value;
int* ptrptr = &ptr;
*ptr = 10;
**ptrptr = 15;
```

Pointer Uses: Functions

- Pointers allow a function to change the value of a variable that exists outside of the scope of the function
  - This process is known as ***pass-by-reference***
  - The value of the pointer is copied (that is the memory address)
  - Thus, when dereferencing the parameter, you get the same memory location

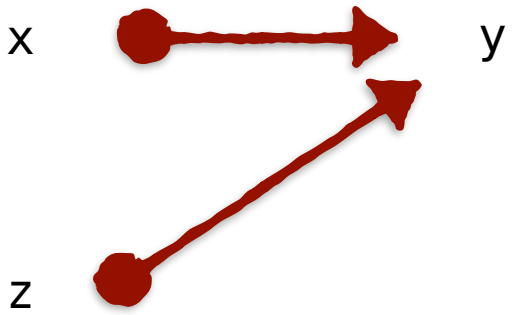
```
void foo(int* x) {
    *x = 10;
}

int y = 7;
foo(&y);
```

Pointer Uses: Functions

```
void foo(int* x) {
    *x = 10;
}

int y = 7;
int* z = &y;
foo(z);
```



Pointer Uses: scanf

- Pointers is how the `scanf` function works!

```
int value;
scanf("%d", &value);
```

- By giving `scanf` the address of a variable:
  - The function can then update the value variable itself

References

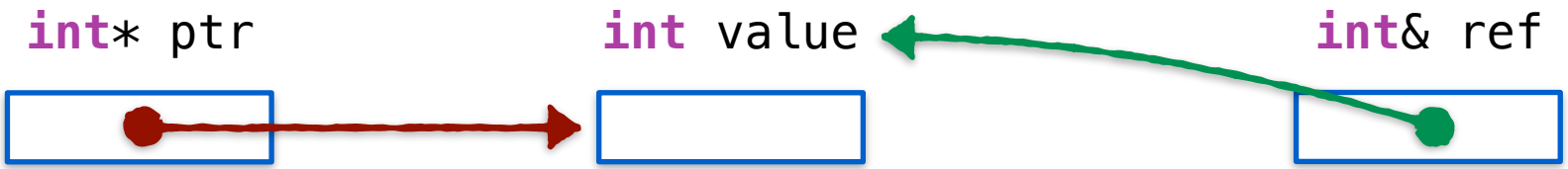
- A ***reference*** is an ***alias*** for a variable
  - That is, it serves as another name for the same variable
- You can think of them as hybrid between concrete types and pointers
  - In Java, every non-non-primitive variable is technically a reference.
- Technical details
  - You can only have a reference if the variable *already exists*
  - A reference must be initialised at the same time it is defined.
  - You can't "see" the underlying "value" of the reference that is stored in memory (In Java you can - it looks like a pointer!)

References

- A reference type is denoted in syntax using a '&'
  - (yes this can be confusing - yay for operator overloading!)
  - Syntax:

```
<type>& <name>
```

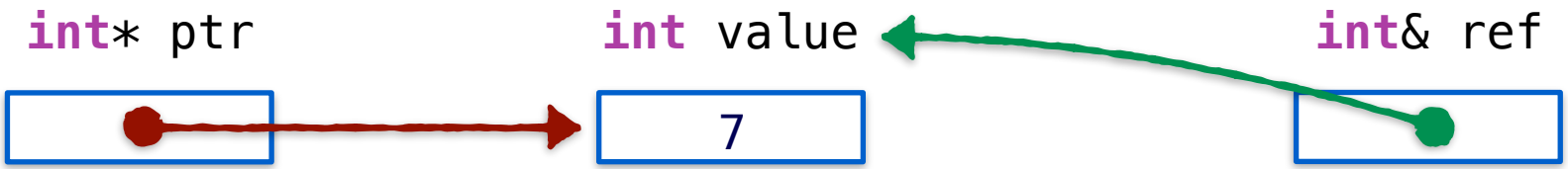
  - It is possible to have a reference to *any* valid type
  - References must be of the correct type for them to work!
- You can think of references as "pointing" to the "name" of a variable



References

- A reference type is denoted in syntax using a ‘&’

```
int  value = 7;
int& ref = value;
ref = 10;
```



References Use: Function

- References literally use ***pass-by-reference***
  - The parameter is a direct reference to the passed variable
  - The reference can be use as a normal variable

```
void foo(int& x) {
    x = 10;
}

int y = 7;
foo(y);
```

References - Declaration Error

- A reference must be initialised when it is declared.

```
int  value = 7;
int& ref; // ERROR!
```

Classes

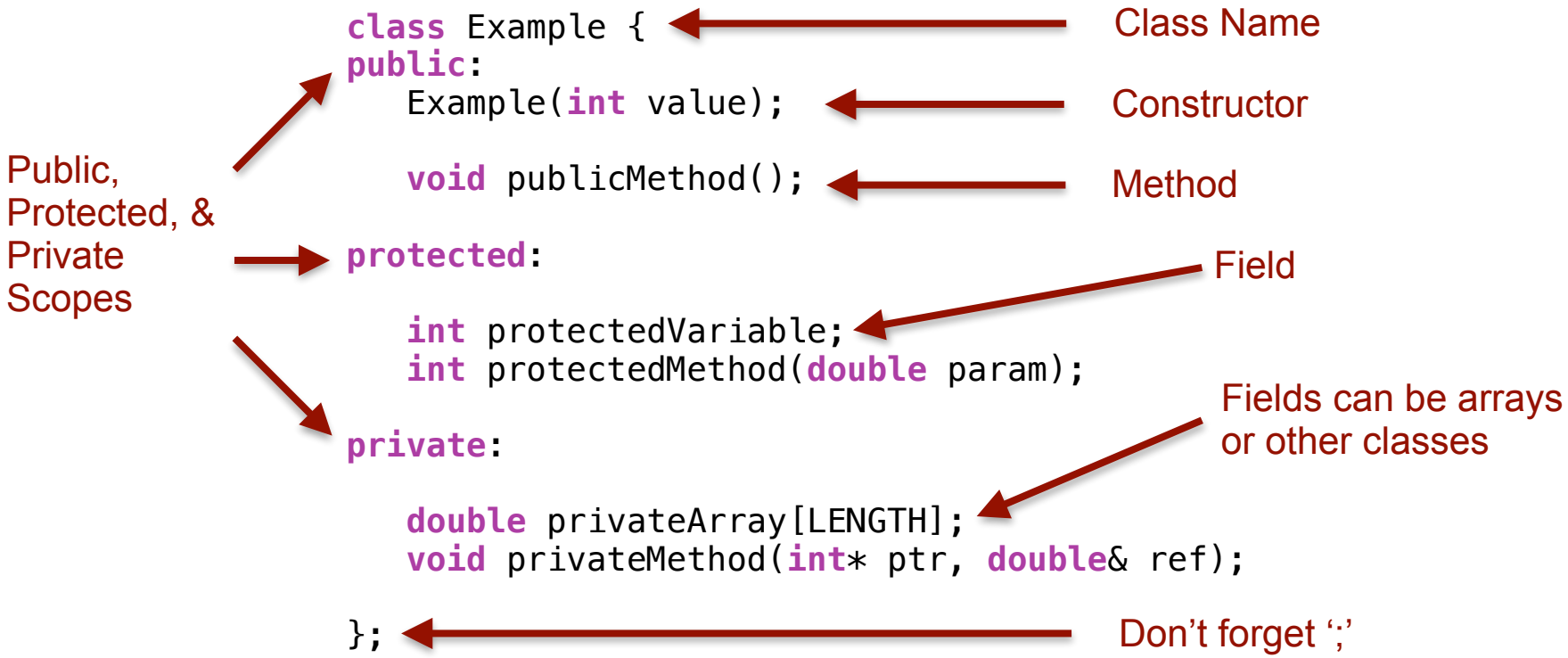
RMIT UNIVERSITY

Classes

- C++ Classes are similar to Java Classes
- Divide creating a class into ***declaration*** and ***definition***
  - A declaration is like a Java interface
    - Describes the components of the class
  - The definition is like a Java class file,
    - Provides the implementation of the class methods.

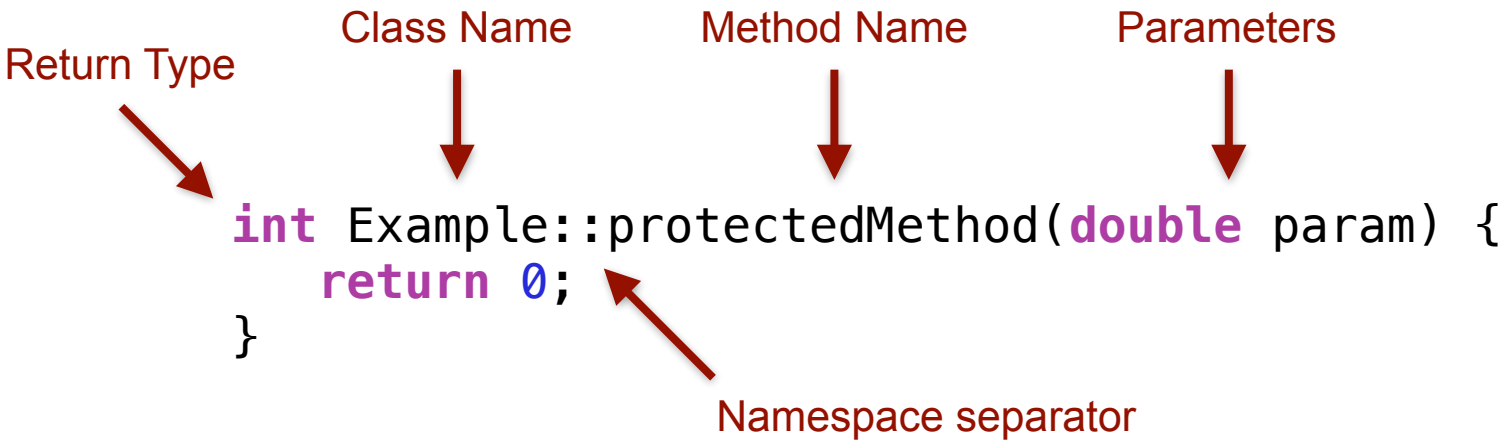
Classes Declaration

- C++ Class ***Declaration***



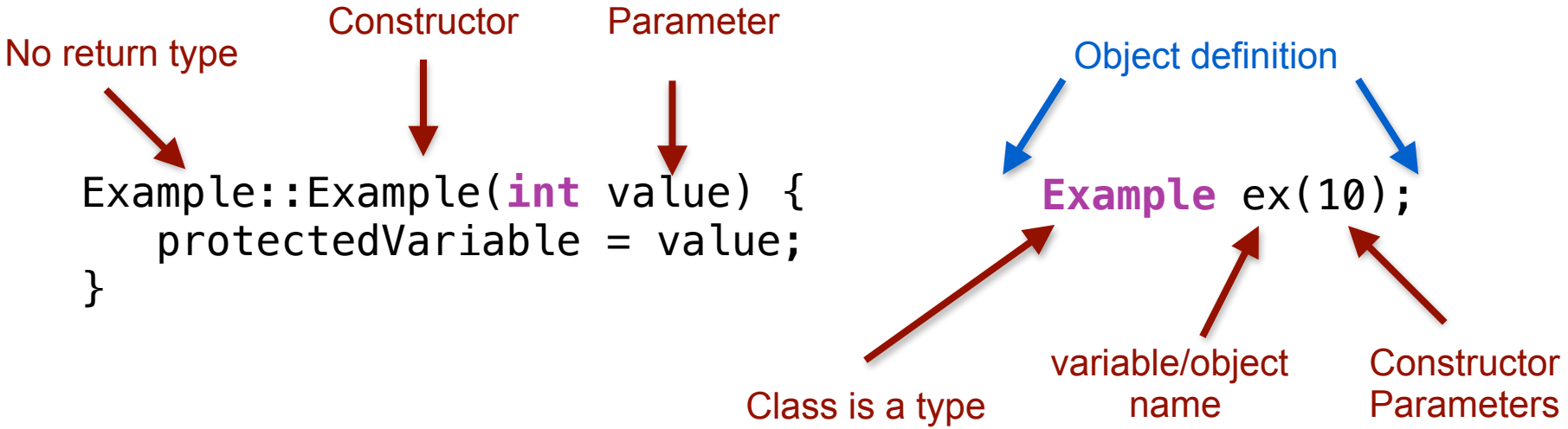
Class Method Definitions

- C++ Class method definitions provide the implementation of each method
  - Definitions provided individually
  - Scope is not relevant to the definition
  - The Class name creates a namespace!



Class Initialisation

- Objects (variables of a given class) can be created like any other variable
  - Does not need to be “new’ed”
- The constructor is called when defining the variable
  - Use bracket notation to provide the parameters to a class object



Access Class Members

- Class members (variables and methods) are accessed using dot '.' Syntax

```
Example ex(10);
ex.publicMethod();
```

- For pointers to object, arrow syntax '->' is a shortcut for dereferencing

```
Example* ptrEx = &ex;
(*ex).publicMethod();
ex->publicMethod();
```

- Class members can only be accessed from the correct scope
  - Public members are always accessible
  - Private members are only accessible only from within the class
  - Protected members can be accessed from this class and all children

Class & Functions

- Pass classes to functions either by
  - Pointer
  - Reference
- Passing the class directly:
  - Is possible
  - BUT!
    - Requires a special constructor (called a copy constructor)
    - We will cover this in future week(s)

RMIT
UNIVERSITY