Pointers and References in C++

Dr Ian Cornelius

Hello

Hello (1)

Learning Outcomes

1. Understand the concept of pointers and references and their purpose in C++
2. Understand how to manage memory in C++

Pointers

Pointers (1)

- Pointers hold the address for a declared variable
- The asterisk (`*`) after `int` means **pointer to**
- Pointers are intended to map directly to addressing mechanisms of the machine
- There are three ways of declaring a pointer variable:
  1. `int * pIntExample1`
  2. `int *pIntExample1`
  3. `int* pIntExample1` - **preferred method**

```
int intExample1 = 32;
int* pIntExample1 = &intExample1;
```

```
intExample1 -> 32
&pIntExample1 -> 0x5556c43fe018
```

Pointers (3)

Variables and Memory i

- Variables are stored in the memory and visualised as a series of unique addressed boxes
- The operating system will pick an unused memory location, e.g. `0x1234`
  - there must be enough space available to store the variable
    - e.g. four bytes for an `int`
- `intExample1` is the name of the reserved memory location

```
int intExample1 = 32;
int* pIntExample1 = &intExample1;
```

```
intExample1 -> 32
&pIntExample1 -> 0x5577552b5018
```

Pointers (4)

Variables and Memory ii

- Array elements are stored sequentially in a contiguous block of memory
  - larger objects may span multiple blocks, i.e. arrays, classes, and floats

```
#include <iostream>
int arrayExample1[4] = {5, 0, 6, 2};
int main() {
  for(int &element : arrayExample1) {
    std::cout << "element -> " << element << " [Address: " << &el
  }
  return 0;
}
```

```
element -> 5 [Address: 0x55e34c89f010]
element -> 0 [Address: 0x55e34c89f014]
element -> 6 [Address: 0x55e34c89f018]
element -> 2 [Address: 0x55e34c89f01c]
```

Pointers (5)

Null Pointers

- Pointers can also point to a *null* object
  - can be achieved with the `nullptr` keyword
- `nullptr` can be assigned to any pointer type, but not to built-in data types
- There is only one `nullptr` and can be used for every pointer type

```
int* pIntExample1 = nullptr;
double* pDoubleExample1 = nullptr;
int intExample1 = nullptr; // Throws an error as intExmaple1 is not a
```

Pointers (6)

Pointers into Arrays i

- Pointers and arrays are closely related in C++
- The name of an array can be used as a pointer to its initial element
  - requesting an element address before the initial or beyond size of an array should be avoided

- Create an array, `arrayExample1` containing four values
  - `pointer1` is a pointer to the initial element
  - `pointer2` is a pointer to the initial element (in a different syntax)
  - `pointer3` is a pointer to a different element in the array
  - `pointer4` is a pointer to a memory address beyond the last element

```
int arrayExample1[4] = {5, 0, 6, 2};
int* pointer1 = arrayExample1;
int* pointer2 = &arrayExample1[0];
```

```
pointer1 -> 0x55fce58ad010
pointer2 -> 0x55fce58ad010
```

```
int arrayExample1[4] = {5, 0, 6, 2};
int* pointer3 = arrayExample1+2;
int* pointer4 = arrayExample1+6;
```

```
pointer3 -> 0x560882c44018
pointer4 -> 0x560882c44028
```

Pointers (7)

Pointers into Arrays ii

- Arrays can be iterated through using a pointer
- Often chosen by developers based on aesthetic or logical reasoning
- There is no performance gain over the *usual* method

Iterating using a Pointer

```cpp
#include <iostream>
char arrayExample1[] = {'5', '0', '6', '2'};
int main() {
  for(char* pointer = arrayExample1; *pointer != 0; pointer++) {
      std::cout << "*pointer -> " << *pointer << std::endl;
  }
  return 0;
}
```

```
*pointer -> 5
*pointer -> 0
*pointer -> 6
*pointer -> 2
```

Iterating Normally

```cpp
#include <iostream>
char arrayExample1[] = {'5', '0', '6', '2'};
int main() {
  for(char &element : arrayExample1) {
      std::cout << "element -> " << element << std::endl;
  }
  return 0;
}
```

```
element -> 5
element -> 0
element -> 6
element -> 2
```

References

References (1)

- Reference variables are an alias, another name for a variable that exists
- Once initialised, either the variable name or reference name can be used to refer to the variable
- References are often confused with pointers, but have three differences:
  - you cannot have a `null` reference
  - once initialised to an object, it cannot be changed to refer to another object
  - references must be initialised when created
- References are often used for:
  - function argument lists
  - function return values

References (2)

Creating a Reference

- References are initialised using the ampersand (`&`) character
- The first `int` declaration is a new object being created
- The second `int` declaration (with the `&`) is the reference object
  - this will refer to the memory address of `intExample1`

```cpp
#include <iostream>
int main() {
  int intExample1 = 32;
  int& rIntExample1 = intExample1;
  std::cout << "intExample1 -> " << intExample1 << "  [Address: "
  std::cout << "rIntExample1 -> " << rIntExample1 << " [Address:
  return 0;
}
```

```
intExample1 -> 32  [Address: 0x7ffc9877cf8c]
rIntExample1 -> 32 [Address: 0x7ffc9877cf8c]
```

References (3)

Pass by Reference

- The `swap()` function consists of two parameters
  - each refers to the address location

```
#include <iostream>
void swap(int &x, int &y) {
  int tmpX;
  tmpX = x;
  x = y;
  y = tmpX;
}
int main() {
  int intExample1 = 5;
  int intExample2 = 10;
  std::cout << "[Before] intExample1 -> " << intExample1 << "  [A
  std::cout << "[Before] intExample2 -> " << intExample2 << " [Ad
```

```
[Before] intExample1 -> 5  [Address: 0x7ffd36ff8250]
[Before] intExample2 -> 10 [Address: 0x7ffd36ff8254]

[After]  intExample1 -> 10 [Address: 0x7ffd36ff8250]
[After]  intExample2 -> 5  [Address: 0x7ffd36ff8254]
```

References (4)

Return as Reference from a Function

- C++ functions can return a reference, similar to how they can return pointers
- When returning a reference, it returns an **implicit** pointer to the return value
  - take care not to return a reference outside the scope of an array

```cpp
#include <iostream>
int arrayExample1[4] = {5, 0, 6, 2};
int arrayLength = sizeof(arrayExample1) / sizeof(int);
int& set_value(int i) {
  return arrayExample1[i];
}
int main() {
  for(int i = 0; i < arrayLength; i++) {
    std::cout << "[Before] arrayExample1[" << i << "] -> " << arr
  }
  set_value(1) = -9;
  for(int i = 0; i < arrayLength; i++) {
```

```
[Before] arrayExample1[0] -> 5
[Before] arrayExample1[1] -> 0
[Before] arrayExample1[2] -> 6
[Before] arrayExample1[3] -> 2

[After] arrayExample1[0] -> 5
[After] arrayExample1[1] -> -9
[After] arrayExample1[2] -> 6
[After] arrayExample1[3] -> 2
```

Memory Management

Memory Management (1)

- C++ has the feature of allocating the memory of variable at run time
  - this is known as *dynamic memory allocation*
- Python automatically manages the memories that are allocated to variables
  - whereas C++ does not
- Therefore, you will be required to deallocate the dynamically allocated memory manually
  - dynamically allocated memory is deallocated *manually* when the variable has no further use
- Allocation and deallocation of memory can be achieved using `new` and `delete` keywords, respectively
- Memory in C++ is divided into two parts:
  1. stack
  2. heap

Memory Management (2)

Allocation of Memory

- Memory allocation is achieved using the `new` keyword

```
int* pIntExample1 = new int;
*pIntExample1 = 32;
```

```
pIntExample1 -> 32 [Address: 0x55be05d432b0]
```

- Memory has been dynamically allocated for `int` using the `new` keyword
- Pointers have been used to aid in memory allocation
  - the `new` keyword returns the address of the memory location
  - in case of an array the `new` keyword returns the address of the *first* element

Memory Management (3)

Deallocation of Memory

- Deallocating the memory is achieved using the `delete` keyword

```
#include <iostream>
int main() {
  int* pIntExample1 = new int;
  *pIntExample1 = 32;
  std::cout << "pIntExample1 -> " << *pIntExample1 << " [Address: " << pIntExample1 << "]" << std::endl;
  delete pIntExample1; // Deletes the variable and reserved memory
  return 0;
}
```

```
pIntExample1 -> 32 [Address: 0x562ec2cc92b0]
```

Goodbye

Goodbye (1)

Questions and Support

- Questions? Post them on the **Community Page** on Aula
- Additional Support? Visit the Module Support Page
- Contact Details:
  - Dr Ian Cornelius, ab6459@coventry.ac.uk