### + **Arrays**

|  |  |  |  |
| --- | --- | --- | --- |
| **arrays.cpp** | |  | line 1 -> Declaring a pointer.  line 4 -> Calling new to allocate memory on the heap for the pointer, however we are using brackets to indicate that it is an array. We also need to pass the size of the array (variable size in this case).  line 10 -> Calling delete to free the heap memory, however, we allocated an array using brackets so we need to add brackets to the delete to free memory.  (The brackets indicated that the constructor/destructor is called on each object in the array) |
| 1  2  3  4  5  6  7  8  9  10 | int \*x;  int size = 3;  x = new int[size];  for (int i = 0; i < size; i++) {  x[i] = i + 3;  }  delete [] x; |

### **Function parameters** - there are three ways to pass an argument to a function:

* + **Pass by value** - it means that the object passed in is a copy of the original object and by changing it, we do not change the original. This is safe, but less efficient because it needs extra memory.

|  |  |
| --- | --- |
| joinCubes-byValue.cpp | |
| 15  16  …  20  21  22  …  29  30  31  32  3334 | Cube joinCubes(**Cube c1, Cube c2**) {  double totalVolume = c1.getVolume() + c2.getVolume();  ...  Cube result(newLength);  return result;  }  int main() {  Cube \*c1 = new Cube(4);  Cube \*c2 = new Cube(5);  Cube c3 = joinCubes(**\*c1, \*c2**);  return 0;  } |

* + **Pass by pointer** - means that we are passing a pointer to the original data and by changing the parameter, we are changing the original. This is also more efficient, but more risky. However, here we can also get an invalid parameter (NULL) passed in.

|  |  |
| --- | --- |
| joinCubes-byPointer.cpp | |
| 15  16  …  29  30  31  32  3334 | Cube joinCubes(**Cube \* c1, Cube \* c2**) {  double totalVolume = c1**->**getVolume() + c2**->**getVolume();  ...  }  int main() {  Cube \*c1 = new Cube(4);  Cube \*c2 = new Cube(5);  Cube c3 = joinCubes(**c1, c2**);  return 0;  } |

* + **Pass by reference** - it means that we are passing an alias to the variable and by changing the parameter, we are changing the original. This is more efficient, but risky because we are changing the original value.

|  |  |
| --- | --- |
| joinCubes-byRef.cpp | |
| 15  16  …  29  30  31  32  3334 | Cube joinCubes(**Cube & c1, Cube & c2**) {  double totalVolume = c1.getVolume() + c2.getVolume();  ...  }  int main() {  Cube \*c1 = new Cube(4);  Cube \*c2 = new Cube(5);  Cube c3 = joinCubes(**\*c1, \*c2**);  return 0;  } |

**Then to summarize, we have the following:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **By Value** | **By Pointer** | **By Reference** |
| The copied content | The entire data | The memory address | Very little to none - just an alias |
| Does modification go through to the caller’s object? | No. | Yes | Yes |
| Always valid? | Yes. | No - could be null | Yes |
| Relative speed among 3 ways to pass function parameters | Slow - depending  ‘  on the data size | Fast - always 8 bytes | Fast |
| The relative programming safety | Safest | Not safe | Safe-ish |

#### 

### **The Const Function Parameter**

* + The keyword *const* is a way to prevent the parameters passed in to be changed. We are saving the memory because we are not passing by value and at the same time avoid the risk of changing the original.

|  |  |
| --- | --- |
| joinCubes-byRef-const.cpp | |
| 15  16  …  29  …  32  33 | Cube joinCubes(**const** Cube & c1 , **const** Cube & c2) {  double totalVolume = c1.getVolume() + c2.getVolume();  ...  }  int main() {  ...  Cube c3 = joinCubes(\*c1, \*c2);  return 0;  } |

* If function getVolume() can guarantee that it is not modifying the value, then we can have:

|  |  |  |  |
| --- | --- | --- | --- |
| Cube.h | | Cube.cpp | |
| 1  2  3 | public:  double getVolume() ; | double Cube::getVolume() **cohorizontal linenst** {  return length\_\*length\_\*length\_;  } | |