#5: Parameters

September 7, 2018 · *Wade Fagen-Ulmschneider*

Heap Memory – Allocating Arrays

```
                  heap-puzzle3.cpp
 5 | int *x;
 6 | int size = 3;
 7 |
 8 | x = new int[size];
 9 |
10 | for (int i = 0; i < size; i++) {
11 |   x[i] = i + 3;
12 | }
13 |
14 | delete[] x;
```

*: new[] and **delete[]** are identical to **new** and **delete**, except the constructor/destructor are called on each object in the array.

Memory and Function Calls

Suppose we want to join two Cubes together:

```
                joinCubes-byValue.cpp
11 | /*
12 |  * Creates a new Cube that contains the exact volume
13 |  * of the volume of the two input Cubes.
14 |  */
15 | Cube joinCubes(Cube c1, Cube c2) {
16 |   double totalVolume = c1.getVolume() + c2.getVolume();
17 |
18 |   double newLength = std::pow( totalVolume, 1.0/3.0 );
19 |
20 |   Cube result(newLength);
21 |   return result;
22 | }
```

By default, arguments are "passed by value" to a function. This means that:

-
-

Alterative #1: Pass by Pointer

```
               joinCubes-byReference.cpp
15 | Cube joinCubes(Cube * c1, Cube * c2) {
16 |   double totalVolume = c1->getVolume() + c2->getVolume();
17 |
18 |   double newLength = std::pow( totalVolume, 1.0/3.0 );
19 |
20 |   Cube result(newLength);
21 |   return result;
22 | }
```

Alternative #2: Pass by Reference

```
                joinCubes-byPointer.cpp
15 | Cube joinCubes(Cube & c1, Cube & c2) {
16 |   double totalVolume = c1.getVolume() + c2.getVolume();
17 |
18 |   double newLength = std::pow( totalVolume, 1.0/3.0 );
19 |
20 |   Cube result(newLength);
21 |   return result;
22 | }
```

Contrasting the three methods:

	By Value	**By Pointer**	**By Reference**
Exactly what is copied when the function is invoked?			
Does modification of the passed in object modify the caller's object?			
Is there always a valid object passed in to the function?			
Speed			
Safety			

Using the `const` keyword

1. Using `const` in function parameters:

joinCubes-by*-const.cpp
15   Cube joinCubes(`const` Cube s1, `const` Cube s2)
15   Cube joinCubes(`const` Cube *s1, `const` Cube *s2)
15   Cube joinCubes(`const` Cube &s1, `const` Cube &s2)

> **Best Practice:** *"All parameters passed by reference must be labeled const."*   *– Google C++ Style Guide*

2. Using `const` as part of a member functions' declaration:

Cube.h
```
1    #pragma once
2
3    namespace cs225 {
4      class Cube {
5        public:
6          Cube();
7          Cube(double length);
8          double getVolume()          ;
9          double getSurfaceArea()          ;
10
11       private:
12         double length_;
13     };
14   }
```

Cube.cpp
```
…
11     double Cube::getVolume()                {
12       return length_ * length_ * length_;
13     }
14
15     double Cube::getSurfaceArea()           {
16       return 6 * length_ * length_;
17     }
…
```

Returning from a function

Identical to passing into a function, we also have three choices on how memory is used when returning from a function:

Return by value:

15	`Cube` joinCubes(const Cube &s1, const Cube &s2)

Return by reference:

15	`Cube &`joinCubes(const Cube &s1, const Cube &s2)

…remember: never return a reference to stack memory!

Return by pointer:

15	`Cube *`joinCubes(const Cube &s1, const Cube &s2)

…remember: never return a reference to stack memory!

Copy Constructor

When a non-primitive variable is passed/returned **by value,** a copy must be made. As with a constructor, an automatic copy constructor is provided for you if you choose not to define one:

All **copy constructors** will:

The **automatic copy constructor**:

1.

2.

To define a **custom copy constructor**:

Cube.h
```
4    class Cube {
5      public:
       // …
9        Cube(const Cube & other);    // custom copy ctor
```

CS 225 – Things To Be Doing:
1. Exam 0 is ongoing
2. lab_debug due Sunday (11:59pm)
3. MP1 due Monday (11:59pm)
4. Daily POTDs every weekday