Heap Memory – Allocating Arrays

```cpp
int *x;
int size = 3;
x = new int[size];
for (int i = 0; i < size; i++) {
    x[i] = i + 3;
}
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:

```cpp
/*
 * Creates a new Cube that contains the exact volume
 * of the volume of the two input Cubes.
 */
Cube joinCubes(Cube c1, Cube c2) {
    double totalVolume = c1.getVolume() + c2.getVolume();
    double newLength = std::pow( totalVolume, 1.0/3.0 );
    Cube result(newLength);
    return result;
}
```

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

### Alternative #1: Pass by Pointer

```cpp
Cube joinCubes(Cube * c1, Cube * c2) {
    double totalVolume = c1->getVolume() + c2->getVolume();
    double newLength = std::pow( totalVolume, 1.0/3.0 );
    Cube result(newLength);
    return result;
}
```

### Alternative #2: Pass by Reference

```cpp
Cube joinCubes(Cube & c1, Cube & c2) {
    double totalVolume = c1.getVolume() + c2.getVolume();
    double newLength = std::pow( totalVolume, 1.0/3.0 );
    Cube result(newLength);
    return result;
}
```

Contrasting the three methods:

<table>
<thead>
<tr>
<th>By Value</th>
<th>By Pointer</th>
<th>By Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exactly what is copied when the function is invoked?</td>
<td>Does modification of the passed in object modify the caller’s object?</td>
<td>Is there always a valid object passed in to the function?</td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safety</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using the `const` keyword

1. Using `const` in function parameters:

```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)
```

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

```cpp
#pragma once
namespace cs225 {
class Cube {
public:
  Cube();               // default ctor
  Cube(double length);  // 1-param ctor
  double getVolume();   // 1-param ctor
  double getSurfaceArea();
private:
  double length_;
};
}
```

Copy Constructor

When a non-primitive variable is passed/returned by value, a copy must be made.
All copy constructors will:

The automatic copy constructor:

1. 

To define a custom copy constructor:

```cpp
class Cube {
public:
  Cube();               // default ctor
  Cube(double length);  // 1-param ctor
  double getVolume();   // 1-param ctor
  double getSurfaceArea();
private:
  double length_;
};
```

Bringing Concepts Together:
How many times do our different `joinCubes` files call each constructor?

<table>
<thead>
<tr>
<th></th>
<th>By Value</th>
<th>By Pointer</th>
<th>By Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cube(double)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cube(const Cube &amp;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

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

Return by reference:

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

...remember: never return a reference to stack memory!

Return by pointer:

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

...remember: never return a reference to stack memory!

CS 225 – Things To Be Doing:

1. Go to lab and work on lab_into
2. Start on mp_stickers