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.

Reference Variable

A reference variable is an alias to an existing variable. Modifying the reference variable modifies the variable being aliased. Internally, a reference variable maps to the same memory as the variable being aliased. Three key ideas:

1. 
2. 
3. 

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></td>
<td></td>
</tr>
<tr>
<td>Does modification of the passed in object modify the caller's object?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is there always a valid object passed in to the function?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Speed

### Safety

---

Using the `const` keyword

1. Using `const` in function parameters:

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

   ```cpp
   #pragma once
   namespace cs225 {
   class Cube {
      public:
         Cube();
         Cube(double length);
         double getVolume();
         double getSurfaceArea();
      private:
         double length_;
   };
   }
   ```

   ```cpp
   #pragma once
   namespace cs225 {
   class Cube {
      public:
         Cube(const Cube &other); // custom copy ctor
   };
   ```

---

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

Return by reference:

```cpp
Cube & joinCubes(const Cube &s1, const Cube &s2)
```  
...remember: never return a reference to stack memory!

Return by pointer:

```cpp
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:

```cpp
class Cube {
   public:
      Cube(const Cube &other); // custom copy ctor
};
```