CS 225

#5: Parameters

January 25, 2018 · *Fagen-Ulmschneider, Zilles*

Heap Memory – Allocating Arrays

heap-puzzle3.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:

joinCubes-byValue.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:

-
-

Alterative #1: Pass by Pointer

joinCubes-byPointer.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

joinCubes-byReference.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:

	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
```
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:

Cube.h
```
#pragma once

namespace cs225 {
  class Cube {
    public:
      Cube();
      Cube(double length);
      double getVolume()          ;
      double getSurfaceArea()          ;

    private:
      double length_;
  };
}
```

Cube.cpp
```
…
  double Cube::getVolume()                        {
    return length_ * length_ * length_;
  }

  double Cube::getSurfaceArea()                   {
    return 6 * length_ * length_;
  }
…
```

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

Return by reference:
```
Cube &joinCubes(const Cube &s1, const Cube &s2)
```

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

Return by pointer:
```
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
```
class Cube {
  public:
    // …
    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