Moving from C to C++

1. libraries change names

```
#include <stdio.h>  /* C library */

#include <cstdio>    /C++ library */
```

2. Compiler is g++ instead of gcc

   makefiles for C++ should be the same as for C (with the change of the compiler name)

3. C++ has a const keyword for constant variables, to replace the use of #define

```
/* C code */
#define size  10

// C++ code
const int size = 10;
```

4. C++ has true pass-by-reference parameter syntax

```
/* c code to swap two values */
void swap ( int *v1, int *v2)
{
  int temp = *v1;
  *v1 = *v2;
   *v2 = temp;
}

/* call */
int a, b;
 swap ( &a, &b);

 //  C++ code to swap to values
void swap2 ( int &v1, int &v2)
{
  int temp = v1;
  v1 = v2;
  v2 = temp;
}
```

```
// call
int c, d;
swap ( c, d);
```

5. Use of cin and cout with << and >> for input/output instead of scanf() and printf()

   However, using cin and cout with our OWN classes requires us to
   use and understand operator overloading (which is kind of hard)

6. Operator Overloading - quite advanced

7. Use of new and delete operators instead of the malloc( ) and free( ) functions

```
/* in C using malloc*/
/* replace TYPE with the correct type/struct name */

/* to dynamically allocate 1 instance of a TYPE on the heap */
TYPE *ptr;
ptr = (TYPE*) malloc ( sizeof(TYPE) );

/* to dynamically allocate an array of N instances of a TYPE on the heap */
TYPE *dynArr;
dynArr = (TYPE*) malloc (sizeof(TYPE) * N );

/* to de-allocate */
 free (ptr);
 free (dynArr);


// in C++ using new
// replace TYPE with the correct type/struct/class name

 // to dynamically allocate 1 instance of a TYPE on the heap
 TYPE *ptr;
  ptr = new TYPE;

  // to dynamically allocate an array of N instances of a TYPE on the heap
  TYPE *dynArr;
  dynArr = new TYPE[N];

   // to de-allocate
   delete ptr;
   delete []  dynArr;        // NOTE the use of the square brackets!!!
```

8. inline functions/methods for replace C #define macros

9. Classes instead of Structs

9a. Access Modifiers of public, private (and protected)

9b. Use of Constructors

9c. setters and getters

```
class MyDate
{
    private:
        int month;

    public:
        void setMonth (int m)
        {
            if ( m >=1 && m <= 12)   // validation check is important
                month = m;
        }

        int getMonth ()
        {
            return month;
        }

        // constructors
        MyDate ( )
        { … }

        MyDate ( int m, int d, int y )
        {  ….  }

};

// code in main

 // created on the stack
 MyDate d1;    // the default constructor gets called
```

```
MyDate d2 ( 10, 31, 1017 );    // calls the constructor taking 3 integers

// creates on the heap
MyDate *d4 = new MyDate();   // calls the default constructor
```

10. The C++ "Big Three"
- Copy Constructor
- Destructor
- Overloaded Assignment Operator

When copying an instance of a class, do you need a "Deep Copy" or is a "Shallow Copy" good enough?

This normally only impacts a class that has a pointer as a data member.

Without a pointer as a data member, there can be no difference between "deep" and "shallow" copying

To understand the difference between deep and shallow copying, check out:
https://www.cs.utexas.edu/~scottm/cs307/handouts/deepCopying.htm

The general idea is assume that we have a pointer referring to some data.

In a copy, we create another pointer.

If the two pointers both refer to the same memory location, we have a shallow copy

If the first pointer refers to the original data and the second pointer refers to a second memory location that is a duplicate of the first, we have a deep copy.

In C++, the "automatically generated" Copy Constructor, Destructor and Overloaded Assignment only do shallow copying. So if shallow copying is "good enough", we don't need to write the "Big Three".

If we need the class to do a deep copy, then we must write our own versions of the "Big Three" for the class.