Equality vs Identity

- **x equals y**: x and y, in some precise sense, have the same value.
  - In C++, this is equivalent to `x == y`.

- **x is identical to y**: x and y are actually the same object.
  - In C++, this is equivalent to `&x == &y`.

Side notes:
- If x and y are pointers, then x equals y if and only if x and y have the same target.
- In other words, two pointers are equal if and only if their targets are identical.

Pointers as Parameters

Passing a pointer to a function gives the function access to the target of the pointer. In effect, this implies another protocol for parameter passing: **pass-by-pointer**.

```c++
void capitalizeString( string* Str ) {
    if ( Str == NULL ) return;
    for (unsigned int Pos = 0; Pos < Str->length(); Pos++) {
        Str->at(Pos) = toupper(Str->at(Pos));
    }
}
```

Here, the pointer is passed by value (which is the default), but its actual target may be modified by the function.

Similar to pass-by-reference, but we must use explicit pointer syntax in the function.
**Pointers and Arrays**

The name of an array is essentially a pointer to the zeroth cell of the array. So, when you pass an array as a parameter you are effectively passing the array by pointer.

However, there is one fundamental difference. If you make the declaration

```cpp
double Weights[100];
```

then you cannot subsequently modify the value of the array name. So, an array name is more like a pointer that is constant.

If you allocate an array dynamically, using a pointer variable, you can then use the usual array indexing syntax with the pointer:

```cpp
double *Weights = new double[100];
Weights[17] = 42.73;
```

---

**Pointer Arithmetic**

Increment/decrement operations are allowed on C++ pointers. However, this is safe only if the target of the pointer is a cell of an array.

```cpp
void zeroArray( int* List, unsigned int Sz ) {
    if ( List == NULL ) return;
    int *Curr = List;
    for (unsigned int Pos = 0; Pos < Sz; Pos++, Curr++) {
        *Curr = 0;
    }
}
```

There's not much reason to increment the pointer here instead of simply using the loop counter as an array index.
From B. Stroustrup, “The C++ Programming Language”:

The result of applying the arithmetic operators +, -, ++, or -- to pointers depends on the type of the pointed to target object.

When an arithmetic operator is applied to a pointer p of type T*, p is assumed to point to an element of an array of objects of type T; p+1 points to the next element of that array, and p-1 points to the previous element.

This implies that the integer value of p+1 will be sizeof(T) larger than the integer value of p.

---

The combination of const with pointers raises some interesting capabilities:

```cpp
int* p = new int(5);        // nothing is constant
const int* p = new int(5);  // p is a pointer to an int which // is constant
int* const p = new int(5);  // p is a constant pointer to an // int (which may be modified)
const int* const p = new int(5);  // p is a constant pointer // to an int, which is // also constant
```

Used sensibly, const can prevent many problems. For example:

```cpp
void zeroArray( int* const List, unsigned int Sz ) {
    . . .
}
```
Besides pointer variables, C++ also has reference variables. Reference variables:
- store the address of an object (like a pointer)
- cannot be assigned a value after they are declared (like a const pointer)
- cannot ever be set to NULL (unlike a pointer)
- are implicitly dereferenced with no special syntax (unlike a pointer)
- cannot be targeted by delete (unlike a pointer)

```c++
string Quote("To be or not to be...");
string &refToQuote = Quote;
cout << refToQuote.length() << endl
    << refToQuote << endl;
```

References are somewhat less flexible, and hence somewhat safer than pointers. However, it is still possible to have a reference that has no target.