Pointers

```
int   *countPtr, count;   // * does not distribute
                          // to all variables in a
                          // declaration.
count = 0;
countPtr = &count;
```

```
int *otherPtr = NULL;      // can initialize to NULL (or 0)
int *yetanotherPtr = &count; // or an address
```

```
int *badDecPtr = 10801141. // Error! Cannot
                           // assign any other literal value (only NULL)
```

& "ampersand"

& is a unary operator on variables that returns the address of the variable (or specific variable field or indexed element)

```
int arr[5] = {0,1,2,3,4};
int *elemPtr = &(arr[3]);
++(*elemPtr);
```

Operators (in order of precedence)	Associativity	Type
( ) , [ ]	L → R	Highest
+ - ++ -- ! * & (type)	R → L	unary
* / %	L → R	mult've
+ -	L → R	add've
< <= > >=	L → R	relational
== !=	L → R	equal/not
&&	L → R	Logic AND
||	L → R	Logic OR
? :	L → R	Conditional
= += -= /= %=	L → R	assignment
,	L → R	comma

```
#include <iostream>
using namespace std;
int cubeByValue(int n);
int main()
{
    int num = 5;
    cout << "Number is " << num << ".\n";
    num = cubeByValue(num);
    cout << "Number is " << num << ".\n";
}
int cubeByValue(int n)
{
  return n * n * n;
}
```

```
#include <iostream>
using namespace std;
int cubeByValue(int n);
int main()
{
    int num = 5;
    cout << "Number is " << num << ".\n";
    num = cubeByValue(num);
    cout << "Number is " << num << ".\n";
}
int cubeByValue(int n)
{
  return n * n * n;
}
```

```
#include <iostream>
using namespace std;
int cubeByReference(int &n);
int main()
{
    int num = 5;
    cout << "Number is " << num << ".\n";
    cubeByValue(num);
    cout << "Number is " << num << ".\n";
}
int cubeByReference(int &n)
{
   n = n * n * n;
}
```

```
#include <iostream>
using namespace std;
int cubeByValue(int n);
int main()
```

```
{
    int num = 5;
    cout << "Number is " << num << ".\n";
    num = cubeByValue(num);
    cout << "Number is " << num << ".\n";
}
int cubeByValue(int n)
{
    return n * n * n;
}
```

```
#include <iostream>
using namespace std;
void cubeByReference(int &n);

int main()
{
    int num = 5;
    cout << "Number is " << num << ".\n";
    cubeByReference(num);
    cout << "Number is " << num << ".\n";
}
void cubeByReference(int &n)
{
    n = n * n * n;
}
```

```
#include <iostream>
using namespace std;

void cubeByRefPtr(int *ptr);

int main()
{
    int num = 5;
    cout << "Number is " << num << ".\n";
    cubeByRefPtr(&num);
    cout << "Number is " << num << ".\n";
}
void cubeByRefPtr(int *ptr)
{
    *ptr = (*ptr) * (*ptr) * (*ptr);
}
```

unary *

binary *

In C++ functions:

I. - we can pass in a simple variable name

- if it is an array the address is always passed, but we treat it in the function just as an array (no need to dereference.)

```
int arr[5] = {0,1,2,3,4};
decThirdElt(arr, 5);
   :

void decThirdElt( int arr[], int sz)
{
    if (sz >= 3)    { arr[2]--;}
}
```

- if it is a simple int, float, string etc., the value is copied in and is NOT changed by the run of the program.

II. - we can pass in a pointer to a variable

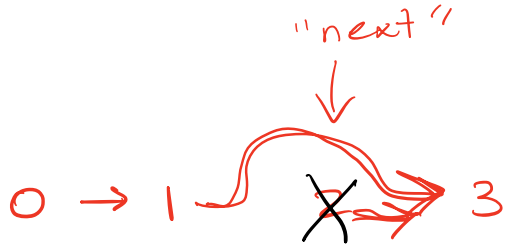
eg &count appears in the call

- then function must treat it as a pointer

III - we can declare in the prototype and function declaration that the parameter is passed-by-reference; then in the function, we just use it like its type, but the changed values persist.

Pointers

use case




```
struct Contact {
        int     phoneNum;
        Contact *next;
} calls[4];
```

← assign names and numbers

```
calls[0].next = &calls[1];
calls[1].next = &calls[2];
calls[2].next = &calls[3];
calls[3].next = NULL;
```

// 2 drops out

~~calls[1].next = calls[2].next~~

```
calls[1].next = calls[1].next -> next;
```