Pointers: Pass By Address, Pointers and Arrays

Pass By Address

- We've seen that regular function parameters are pass-by-value
  - A *formal parameter* of a function is a local variable that will contain a *copy* of the argument value passed in
  - Changes made to the local parameter variable do not affect the original argument passed in
- If a pointer type is used as a function parameter type, then an actual *address* is being sent into the function instead
  - In this case, you are not sending the function a *data value* -- instead, you are telling the function where to find a specific piece of data
  - Such a parameter would contain a *copy* of the address sent in by the caller, but not a copy of the target data
  - When addresses (pointers) are passed into functions, the function could affect actual variables existing in the scope of the caller
- Example:

```
void SquareByAddress(int * ptr)
// Note that this function doesn't return anything.  void function.
{
   *ptr = (*ptr) * (*ptr);             // modifying the target, *ptr
}

int main()
{
  int num = 4;
  cout << "num = " << num << '\n';     // num = 4
  SquareByAddress(&num);               // address of num passed in
  cout << "num = " << num << '\n';     // num = 16
}
```

Pointers and Arrays:

With a regular array declaration, you get a pointer for free. The **name** of the array acts as a pointer to the first element of the array.

```
int list[10];   // the variable list is a pointer
                // to the first integer in the array
int * p;        // p is a pointer.  It has the same type as list.
p = list;       // legal assignment.  Both pointers to ints.
```

In the above code, the address stored in list has been assigned to p. Now both pointers point to the first element of the array. Now, we could actually use p as the name of the array!

```
list[3] = 10;
```

```
p[4] = 5;
cout << list[6];
cout << p[6];
```

Pointer Arithmetic

Another useful feature of pointers is pointer arithmetic. In the above array example, we referred to an array item with p[6]. We could also say *(p+6). When you add to a pointer, you do not add the literal number. You add that number of units, where a unit is the type being pointed to. For instance, p + 6 in the above example means to move the pointer forward 6 integer addresses. Then we can dereference it to get the data * (p + 6).

What pointer arithmetic operations are allowed?

- A pointer can be incremented `(++)` or decremented `(--)`
- An integer may be added to a pointer `(+ or +=)`
- An integer may be subtracted from a pointer `(- or -=)`
- One pointer may be subtracted from another

Most often, pointer arithmetic is used in conjunction with arrays.

Example: Suppose ptr is a pointer to an integer, and ptr stores the address 1000. Then the expression (ptr + 5) does not give 1005 (1000+5). Instead, the pointer is moved 5 integers (ptr + (5 * size-of-an-int)). So, if we have 4-byte integers, (ptr+5) is 1020 (1000 + 5*4).

This code example illustrates pointers and arrays: parray.cpp

---

Pass By Address with arrays:

The fact that an array's name **is** a pointer allows easy passing of arrays in and out of functions. When we pass the array in by its name, we are passing the **address** of the first array element. So, the expected parameter is a pointer. Example:

```
// This function receives two integer pointers, which can be names of integer arrays.
   int Example1(int * p, int * q);
```

When an array is passed into a function (by its name), any changes made to the array elements do affect the original array, since only the array **address** is copied (not the array elements themselves).

```
void Swap(int * list, int a, int b)
{
  int temp = list[a];
  list[a] = list[b];
  list[b] = temp;
}
```

This Swap function allows an array to be passed in by its name only. The pointer is copied but not the entire

array. So, when we swap the array elements, the changes are done on the original array. Here is an example of the call from outside the function:

```
int numList[5] = {2, 4, 6, 8, 10};
Swap(numList, 1, 4);        // swaps items 1 and 4
```

Note that the Swap function prototype could also be written like this:

```
void Swap(int list[], int a, int b);
```

The array notation in the prototype does not change anything. An array passed into a function is always passed by address, since the array's name IS a variable that stores its address (i.e. a pointer).

Pass-by-address can be done in returns as well -- we can return the address of an array.

```
int * ChooseList(int * list1, int * list2)
{
  if (list1[0] < list2[0])
    return list1;
  else
    return list2;    // returns a copy of the address of the array
}
```

And an example usage of this function:

```
int numbers[5] = {1,2,3,4,5};
int numList[3] = {3,5,7};
int * p;
p = ChooseList(numbers, numList);
```

Using const with pass-by-address

- The keyword **const** can be used on pointer parameters, like we do with references. It is used for a similar situation -- it allows parameter passing without copying anything but an address, but protects against changing the data (for functions that should not change the original)
- The format:

```
const typeName * v
```

- This establishes v as a pointer to an object that cannot be changed through the pointer v.
- Note: This does **not** make v a constant! The pointer v **can** be changed. But, the **target** of v cannot be changed (through the pointer v).
- Example:

```
int Function1(const int * list);  // the target of list can't
                                  // be changed in the function
```

Note: The pointer can be made constant, too. Here are the different combinations:

1. Non-constant pointer to non-constant data

```
int * ptr;
```

2. Non-constant pointer to constant data

```
const int * ptr;
```

3. Constant pointer to non-constant data

```
int x = 5;
int * const ptr = &x;   // must be initialized here
```

An array name is this type of pointer - a constant pointer (to non-constant data).

4. Constant pointer to constant data

```
int x = 5;
const int * const ptr = & x;
```

---

const pointers and C-style strings

- We've seen how to declare a character array and initialize with a string:

```
char name[20] = "Marvin Dipwart";
```

Note that this declaration creates an array called `name` (of size 20), which can be modified.

- Another way to create a varible name for a string is to use just a pointer:

```
char* greeting = "Hello";
```

However, this does **NOT** create an array in memory that can be modified. Instead, this attaches a pointer to a fixed string, which is typically stored in a "read only" segment of memory (cannot be changed). So it's best to use `const` on this form of declaration:

```
const char* greeting = "Hello";               // better
```

- Note: It would be legal to modify the contents of `name` above, but it would NOT be legal to modify the contents of `greeting`:

```
name[1] = 'e';        // name is now "Mervin Dipwart"

greeting[1] = 'u';    // ILLEGAL!
```

- These two examples illustrate the above declarations (`name` and `greeting`). Note that in both of them, output of the string is done the same way, regardless of the type of declaration. But they differ in the attempts to change the string contents:
  - `str1.cpp` -- This one uses the first declaration of `greeting` (non-const pointer). Note that the attempt to change the contents *will* compile, but execution results in a run-time error
  - `str2.cpp` -- This one uses the second declaration of `greeting` (const used with the pointer). The attempt to change the target, therefore, will not compile.