CMPE1250 – Pointers (General Theory, C)

Memory Organization

The memory in a computer is analogous to a giant array of bytes. The byte is the smallest unit that is given a unique address, and the address increases for each adjacent byte of memory. This means that every byte of memory in your computer has a discrete address, and may be individually accessed.

Every variable you create has an address in memory. Because the variable could be bigger than one byte, we always interpret the address of that variable to be the address of the ***first*** byte of memory that the variable consumes in memory.

For example, if we consider an **`int`** that is 32 bits in size, it would require four bytes of memory. If you were told that the address of that **`int`** was **`5`**, you would interpret that as: it starts at address **`5`**, and continues on to use address **`6`**, **`7`**, and **`8`** as well.

The specific addresses of variables in memory will rarely have any direct meaning to us. Instead, these address' will be provided to operations that need to know them. This will seem a little odd to us initially, as we tend to be very fact oriented.

To determine how many bytes something requires, we use the **`sizeof()`** operator. This operator will return the size of a type or instance in bytes. This will have utility in the future.

```
printf("The size of an int is %d bytes!", sizeof(int)); // The size of an int is 4 bytes!
```

Pointers

A pointer is a variable, like any other, but it holds a memory address. The address generally refers or "points to" other variables (or even functions). Pointers are critical to many operations in C, and are therefore necessary to learn before we are able to do much of anything.

Pointer size is dependent on the platform under which the program is compiled. Visual Studio may be set to build 32-bit (x86) or 64-bit (x64) code. Normally we will build for x86 which will result in 4-byte address sizes. Addresses that are 32 bits in size are address space limited to about 4GB.

On the micro, a pointer is 16-bits in size, so the address space is limited to 65536 addressable bytes. Discuss why this is.

Addresses and the AddressOf operator &

In order to retrieve the address of a variable or object, the "address of" operator is used. The ampersand (&) prefixes the variable that you want the address of:

```
int iValue = 46;
printf ("iValue's address: %p", &iValue); // iValue's address: 00B7F908

// Output is 32bit hexadecimal value (%p format specifier does this)
```

The address-of operator is often used to initialize pointer variables and arguments for pointer parameters.

Pointer Declaration and initialization

Pointers are declared using the following syntax:

```
datatype * ptrVariableName; // ptrVariableName is a pointer to type datatype
```

Where the type is any system or user-defined type and the asterisk falls between the type and pointer name.

```
// For pointer declarations :
// Read from Right to Left, pToInt is a pointer to int
int * pToInt;
double * pD = 0; // explicit initialization to NULL/nothing
char * pC = NULL; // again with NULL macro define of 0
```

Spacing is irrelevant, and you will see pointer declarations of all forms, our preference is shown above.

The pointer type is used to determine how the memory it points at is interpreted. Because of this mechanism, it is imperative that the pointer type matches the intended target. Remember that memory consists of values that have extremely specific alignment. Start location, and interpretation is therefore critical.

We will see a little bit farther along in the notes why this is so important.

Assigning Pointers

Pointers hold addresses. When a pointer holds the value **`0`** (has **`typedef`** name of **`NULL`**), it is considered empty, or not holding the address of anything. Normally, pointers are assigned an address using the addressof operator on a same type variable, meaning a pointer to an **`int`** is assigned the address of an **`int`** (not a **`double`**). There are many tricks pointers can play in C by mismatching the pointer and target types, but we will stick to the basics here.

```
int iVal = 46;
int * pPtToInt = &iVal; // Now pPtToInt has iVal's Address
// Or pPtToInt now "points-to" iVal.
```

The variable **`iVal`** holds the value **`46`**, and the pointer holds the address of **`iVal`** (some weird 32-bit value). The pointer may be indirectly used to read or write the contents of **`iVal`**, as it knows where it is in memory.

This is the purpose of a pointer. It holds the address of a variable, and has a type. The pointer has the capacity to interpret the memory with the perspective of its type, and read from or write to the memory in that context.

Dereferencing pointers

To access the value of the memory address that a pointer points to, the pointer is "dereferenced". This is done by reuse of the asterisk as the dereference operator. It is unfortunate that the * appears in so many different expressions in C, but it can easily be sorted out by the context.

- * used during declaration is always declaration of a pointer
  ```
  int * pNum = NULL; // pointer decl
  ```
- * used to the left of a pointer is always dereferencing that pointer
  ```
  int iVal = *pNum; // dereferencing
  ```
- * used between two numeric types is multiplication
  ```
  int q = 6 * 4; // multiplication
  ```

Again, when you deference a pointer, you are interpreting the memory found there under the context of the pointer type. You are allowed to read from or write to the target memory with a dereferencing operation. If this is the location of a variable (as it should be), the contents of the variable will change precisely as they would, had you used the variable itself.

```
int iVal = 10;
printf("\r\niVal is %d", iVal);

int * pVal = &iVal;  // take address of iVal, make contents of ptr
printf("\r\niVal lives at %p, but who cares?", pVal);

printf("\r\nLooking through the pointer, iVal is %d", *pVal); // dereference pointer
```

```
iVal is 10

iVal lives at 006FFE58, but who cares?
Looking through the pointer, iVal is 10
```

Once again, the actual address of a variable is not important to us, but we will use these values to feed other operations that do need to know them.

The type of pointer and type of target must match, or the interpretation of the memory will be incorrect. Imagine:

```
int iVal = -45;
float * pVal = &iVal;       // wrong pointer type
printf("\r\nLooking through the wrong pointer type, iVal is %d", *pVal);
```

```
Looking through the wrong pointer type, iVal is 1610612736
```

The compiler will NOT help you match types, and will happily assume that you know what you are doing. This freedom is nice when you are pulling off shenanigans, but is indifferent when you have made an honest mistake.

Modifying Variables out-of-Scope

It would not be particularly helpful to use a pointer to a type in the same scope – after all, you could just use the variable directly.

Pointers become quite handy when passing arguments to functions. Remember that having the address of a variable allows us to access that variable, indirectly. This continues to be true, even out of the scope of that variable. Consider the following code:

```
void foo(int * pVal)
{
	*pVal += 10;
}

int main()
{
	int iVal = 7;
	printf("\r\niVal before call to foo() : %d", iVal);
	foo(&iVal);
	printf("\r\niVal after call to foo() : %d", iVal);

	getchar();
}
```

```
iVal before call to foo() : 7
iVal after call to foo() : 17
```

By writing the function to take a pointer to a variable, we are giving the function the ability to modify that variable. This is handy, as the function may only return one value, but we could pass as many arguments as we want.

One philosophy in C is that functions should only return values indicating success or failure. If we follow this pattern, we would always modify arguments through pointers as shown above.

Arrays in C

C supports arrays. The syntax is a little different from C#:

```
int stuff[5];        // uninitialized array, 5 elements

for (int i = 0; i < 5; ++i)
       printf("\r\n%d", stuff[i]);
```

```
-858993460
-858993460
-858993460
-858993460
-858993460
```

If uninitialized, you get whatever junk was in memory at that position!

If initialized, initializing values correspond the array positions; the rest are assigned zero:

```
int stuff[5] = { 5, -42 };         // all elements initialized

for (int i = 0; i < 5; ++i)
       printf("\r\n%d", stuff[i]);
```

```
5
-42
0
0
0
```

The dimension of the array must be a constant value, or may be deduced from the initialization list:

```
int stuff[] = { 5, -42 };          // dimension deduced from initializer
```

As seen above, the array uses subscript notation `[]` to access elements of the array. This mechanism works as you learned it in C#. Unlike C#, however, *the name of an array is a pointer to the first element of the array*.

This single, mind-bending fact, and the rules that govern it, are one of the reasons C is so fast.

```
int stuff[] = { 5, -42 };   // dimension deduced from initializer
printf("\r\n%d", *stuff);   // name of an array is ptr to first element!
```

```
5
```

In C, the array is allocated as a contiguous block of elements. This means that the address of each element may be easily calculated, if you know the address of the first element, and the size of the elements. This is the case, and C uses a mechanism known as *pointer arithmetic*.

Pointer Arithmetic

The name of an array is a pointer to the first element of the array. *The reverse is also true.* This means that an array is just a pointer, and any pointer *could be treated as an array.*

When you dimension an array, the compiler sets aside a block of memory that is the element size times the array count. We will use this ideal definition, although in reality the compiler can use various alignments to organize the memory. This will not yet confront us, so we will use the ideal definition.

If you increase a pointer by one, it will be adjusted to point at the address of the next element. Consider the case of a pointer to a 32-bit **`int`**. Since type **`int`** has a size of four bytes, we would expect the address to increase by four:

```
int i = 0;
int * p = &i;
printf("\r\n%p, %p", p, p + 1);
```

```
00B5F75C, 00B5F760
```

And so it does.

Pro tip: Although the compiler will not stop you from manipulating memory you have not specifically allocated, consequences will result if you do so. Any attempt to access memory other than memory you own could cause the program to crash, could alter local variables, or the operating system may intervene to protect itself against a perceived breach in security.

Adding a value to a pointer increases the value by n x size, which perfectly matches the allocation form for an array. So really, the use of subscript is just a shorthand of pointer arithmetic:

```
int buff[] = { 1, 2, 3, 4, 5, 6 };

printf("\r\n5th value is %d", buff[4]);
printf("\r\n5th value is %d", *(buff + 4)); // exactly the same...
```

If **`buff + 4`** yields the address of the 5th element, then that pointer could be dereferenced to yield the target value - same as what subscript is doing. You get to choose what syntax you use, unless directed otherwise.

Pointers will allow **`++`**, **`--`**, **`+`**, **`-`**, **`+=`**, and **`-=`** as operations.

Subtraction of two pointers will yield the displacement in number of elements.

Passing Arrays to Functions

Passing an array is no different from passing a pointer. Since the name of an array is a pointer to the first element of the array, an array name is just a pointer:

```
void Show(double * pArray)
{
	printf("Array address is %p!", pArray);
}

int main()
{
	double dArray[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
	Show(dArray);
…
```

C supports a number of different variants of syntax for passing an array:

```
void Show(double * pArray)   // as above
void Show(double pArray [])  // empty [], implies an array argument
void Show(double pArray [5]) // some size supplied, but ignored, used for header docs
```

This looks great, except for a major omission - the **`Show`** function does not have access to the size of the array! Unlike C# arrays and **`List`**, which have properties that hold their current size, the C array is only a pointer to the first element, and has no properties to hold the size/length.

There is no C magic to get around this. As you have seen, array size values must be constant integral variables or literal integral values. Our standing rule that no meaningful literals are allowed unless they are held in a constant, will help manage this shortcoming.

When arrays are passed to a function, you may supply the array size as an additional argument, or use a global definition that describes the size. The array pointer and its length are all that is required safely access the array argument.

```
// parameters : pointer to start of array + num elements
void ShowArray(double * pArray, int iLength)
{
	// using standard subscript syntax :
	for (int i = 0; i < iLength; ++i)
		printf ("\r\n[%d] : %f", i, pArray[i]);

	// using pointer dereferencing and pointer arithmetic
	for (int i = 0; i < iLength; ++i)
		printf("\r\n[%d] : %f", i, *(pArray + i));
}

// calling code:
double dArr[5] = { 1.6, 2.0, 3.0, 4.0 }; // note size vs. initializer count
ShowArray(dArr, 5); // ptr + size
```

This form is typical for handling arrays in C, though you will occasionally see array sizes specified as global definitions, thereby eliminating the requirement to include them as function parameters:

```
#include <stdio.h>

// definition for the length in global space as a
//  find and replace definition. Names would need
//  to be super specific if this technique is used
#define GlobalArrayLengthDefinition 5

// parameters : pointer to start of array
void ShowArray(double * pArray)
{
	// using standard subscript syntax :
	for (int i = 0; i < GlobalArrayLengthDefinition; ++i)
		printf ("\r\n[%d] : %f", i, pArray[i]);

	// using pointer dereferencing and pointer arithmetic
	for (int i = 0; i < GlobalArrayLengthDefinition; ++i)
		printf("\r\n[%d] : %f", i, *(pArray + i));
}

int main()
{
	double dArr[GlobalArrayLengthDefinition] = { 1.6, 2.0, 3.0, 4.0 };
	ShowArray(dArr); // ptr only, length comes from global

	getchar();
}
```

We would want to limit the use of this technique for a few reasons:

- Global space is full of definitions already
- You do not want to accidentally use the wrong definition

NOTE: Other (more standardized and newer) versions of C will allow the use of constants to solve this problem. Sadly, this compiler does not support this mechanism in C.

NOTE: Some patterns include reserving space in the array to denote the element count, or using a special character in the array to indicate the end of the array. Strings use the character value 0 to indicate the end of the string. This way, the size of the string need not be passed to a function with the pointer. Care must be taken to ensure that the string is terminated before calling a function that operates on this assumption!