Pointers 1a

Dominic Connor
Dominic@PaulDominic.com

What is a pointer ?

- It’s an address.
- Most, but not all items within your executing program have an address.
- It is possible, though quite rare to set the address of an item in your code, normally this is done by a combination of the run time libraries and the operating system itself.
- Not only do data items have an address, but so does code, and in particular, so do functions.

What a pointer isn’t

- An integer

In many environments an integer and a pointer are the same size. Some code assumes this to be true.

Even when integers and pointers are the same size they aren’t the same. Integers are by default signed, ie they can have negative values. You can’t have a negative value for a pointer.

Actually that’s not true (let’s see how), but if you do, confusion will follow.

Parameters

- In C++, you pass parameters to a function *by value.* This means a copy of the variable is made, used and then thrown away.
- Value parameters can’t be changed.
- To pass a variable that you want changed you can pass it’s address, ie a pointer.

```
void Fail(int victim)
{
    victim++;
}
void Succeed (int * Victim)
{
    *Victim=3;
}
```

&

- We call it by taking the address of the variable using the **&** operator, as in

Succeed (&x);

For a given instance of a given variable, the address is constant, however two instances of the same variable will have different addresses, because of course they are storing different data.

Dereferencing

De-referencing is where you get the thing pointed-to by the pointer. Thus in `Success`, `Victim` is an address that is dereferenced by * to give you a place to put the value.

* In this context is a *unary* operator, ie it operators upon one operand.

Thus Victim is still passed by “value”, since it does not change. What changes is the memory that it points to.

```
void Succeed (int * Victim)
{
   Victim++;
}
```

Does not change anything outside the function. It moves where the *copy* of `Victim` points to, but this copy of the variable disappears when the function returns. Thus this function has no lasting effect.

So how do we increment Victim ?

```
void Succeed (int * Victim)
{
   *Victim++;
}
```

No

- The increment gets executed first, so we move along Victim, leaving the memory location we want changed untouched.
- We need brackets…

```
(*Victim)++;
```

However there’s something interesting about the “increment” of the pointer. It doesn’t go up by one, it goes up by 4. (or 8 if you’re working in 64 bit). However if we do subtraction, it can look as if we’ve only gone up by one.

Pointers are strongly typed

- The compiler knows how big an integer is.

Thus ++ **does not** mean add one. It means “next”. Same applies to - -

If we do the same thing with a double, it will move along by 8.

double *p = Victim;

Will be flatly rejected by the compiler.

Although a pointer to a double is the same size as a pointer to int (or string or horribly complex sparse matrix class), it’s almost certainly an error to use pointers to two different types to reference the same memory.

How to do bad things

At the very base level a simple assignment gets executed by the processor like this

*x= 42.0;

Find out where x is.

Find out what size y is

Copy *size* bytes from y to x

```
 double Threat =42.0;
 double * Accomplice = reinterpret_cast<double *>(Victim);

*Accomplice =Threat;
```

If x is an integer then if you don’t have strong typing, 8 bytes will get copied into a 4 byte space. Where do the other 4 bytes go ?

Apparently they fit…

…In Debug Version

- Let’s see why this code works…

In debug compiles, VC++ puts empty space between values on the stack. This lets it catch errors like this.

In optimised release compiles, you’d expect them to be back to back, and when you overwrite a variable, you won’t always notice instantly what has happened.

To make this easy to digest, I’ve taken a simple and not very realistic case. However this does happen in real life programs. Aside from general crashes, there are a respectable number of programs that only work if they are compiled in debug mode. They are of course slower and larger than they need to be, and “work” means that the stray pointers are “lucky” by only trashing memory that isn’t being used.

Arrays and Pointers

So why does ++ make the pointer move by the size of the item pointed to ?

When incremented the pointer, I went to the “next” item. C-style arrays are implemented as a block of memory which has space for a fixed number of items of one type.

```
	double Array[ArraySize];
	int i;
	for (i=0; i!= ArraySize; i++)
	{
		Array[i] =i ;
	}
	cout << Mean(Array)<<"\n";
double Mean(double * a)
{
	int i;
	double sum=0;
	for (i=0; i != ArraySize; i++)
	{
		sum += a[i];
	}
	return (sum/ArraySize);
```

I could have written

```
sum += *a +i ;
```

Note that since * indirection has a higher precedence than +, the indirection will be evaluated first.

This is not the most common (or best) way of writing this , but you will find it’ and it is necessary to understand this style of coding because you will find it in C++ that is in service.

When i==I, you are not adding 1 to the pointer, but as you may work out from the ++ example earlier, by adding “1” we are adding 1*sizeof(double)

Getting Elements From An Array

Find Element zero

- Find Address of of A
- Return address

Find Element one

- Find Address of A
- + Sizeof (element)

Find Elment

- Find Address of A
- + 2* Sizeof (element)

Find Element N

Find Address of A

+ N * Sizeof (element)

Sizeof()

- sizeof() is often refered to as an operator, but it works in a different way to new(), *, etc.
- At compile time, *not* execution time it inserts the number of bytes occupied by an object. The size of items must be known when the program is compiled. It is constant during the execution of a program. Thus sizeof() cannot be overloaded, and in the life cycle of most pieces of software will always return the same number for the same type. However, for user defined types it will change, and as you move from 32 to 64 bit environments some sizes will increase.

In the call to Mean I could have just as well written

```
double Mean(double a[])
```

And both access styles will work. Arrays and pointers in many cases are simply syntactic sugar for the same thing, and you can use whichever suits what you are trying to achieve.

Array bound checking

- Doesn’t exist.

In my loop I used the same constant that defined the array size. That’s usually the best way, but if you go off the end, as in the classic newbie bug of assuming that

double Array [N];

Lets you write

Array[N]=42;

What’s scary of course is that this looks like it works (usually).

If you want array bound checking, then you either code it up yourself, or better still use the STL.