Pointers

In this lecture we will be looking at a powerful part of C++, pointers.

Introduction to Pointers

Pointers are a powerful part of C++. They can often be used in place of reference and they offer functionality that no other part of the language can.
In this lecture you will, hopefully, learn the basic mechanics of pointers and get an idea of what they are good for.
The main areas covered are therefore:

- Declaration and initialisation of pointers
- Dereferencing of pointers
- Using constants and pointers
- Passing and returning pointers
- Working with pointers and arrays

Pointer Basics

Pointers have a reputation for being difficult to understand, especially when explained by experts. Although I don't claim to be an expert hopefully after reading this you will be able to understand them…assuming you understand everything else covered so far.

In computer science a pointer is defined as:
A programming language datatype whose value is used to refer to ("points to") another value stored elsewhere in the computer memory. Obtaining the value that a pointer refers to is called dereferencing the pointer. A pointer is a simple implementation of the general reference datatype, although it is quite different from the type of reference referred to as references in Java or C++.
So what is the basic reality of pointers then?

I thought you said that pointers were simple, they are really.
Quite simply a pointer is:

Definition: A pointer is a variable that can contain a memory address.

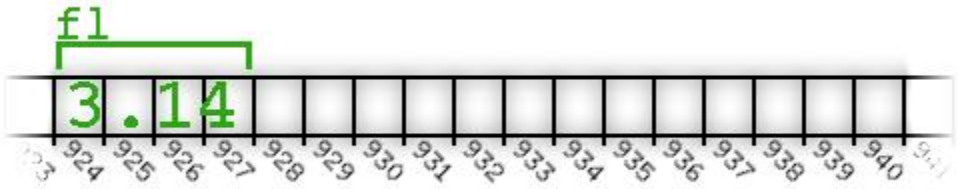
Memory Addresses

A computer's memory is organised into discrete addressable locations, each of which stores a fixed number of bits (**b**inary dig**its**). Each memory location contains 8 bits and is called a byte. Different computers number the memory using different complex schemes. As a programmer you don't really need to know the exact memory address of any variables because the compiler handles the details.

The memory addresses are arranged in a continuous ascending numerical sequence, usually starting at zero and ending at one less than the total number of memory locations in the computer's memory.

When we define a variable, such as a char or bool, whose value can be stored in a single byte, one memory location is allocated to store the variable and the address of that location becomes the address of the variable. If, on the other hand, we define a variable such as an int, whose value requires, say, four bytes to store it, then four consecutive memory locations are allocated and the address of the first of these locations becomes the address of the variable. Finally, if we define an array of, say, N elements of some type that requires, say, M bytes to store its value, then N*M consecutive elements are allocated to store the array values. The address of the first of these memory locations becomes the address of the whole array and also the address of the zeroth element of the array. More generally, the address of the element with index k is the address of the whole array plus k*M, although programmers rarely need to know this.

Here is an example of how variables a stored. This is an example of floating point variable f1.



The illustration that shows 3.14 in the computer's memory can be misleading. Looking at the diagram, it appears that "3" is stored in memory location 924, "." is stored in memory location 925, "1" in 926, and "4" in 927. Keep in mind that the computer actually uses an algorithm to convert the floating point number 3.14 into a set of ones and zeros. Each byte holds 8 ones or zeros. So, our 4 byte float is stored as 32 ones and zeros (8 per byte times 4 bytes). Regardless of whether the number is 3.14, or -273.15, the number is always stored in 4 bytes as a series of 32 ones and zeros.

***Note: We have talked about computer memory being a series of cubby holes with different addresses, this has been expanded upon and computer memory has been compared in one book to an estate. Instead of houses in which people store stuff you have memory locations where data is stored. In an estate houses sit side-by-side labelled with addresses, so do chunks of computer memory. Just as you could use the address, written down, to navigate to the house and therefore what is stored inside, you can use a pointer with a memory address to get a particular memory location and therefore what is stored inside it.*

Declaring Pointers

```
int* pIntPointer;   // Declare a pointer
```

This is how you declare a pointer.
Because pointers work in a unique way it is considered good practice to prefix the pointer name with the letter “p“ to remind you as the programmer, and anyone reading your code, that it is a pointer.

Pointers just like variables have to have a specific data type. The pointer then points to a specific data type. In this example the pointer `pIntPointer` is pointing to integer (`int`). This means that it can only point to an `int` value, it therefore could not point to a `float` or `char` for example.
To put this another, and perhaps more correct, way the pointer can only store the address of an `int`.

To declare a pointer of your own, you begin with the type of object to which the pointer will point, followed by an asterisk, followed by a pointer name. When you declare a pointer you can put whitespace on either side of the asterisk.
Therefore:

```
int* pIntPointer;   // Declare a pointer
int *pIntPointer;   // Declare a pointer
int * pIntPointer;  // Declare a pointer
```

All declare a pointer called `pIntPointer`.

If you remember when we covered variables, we could declare multiple variables of the same type like this:

```
int score, ammo;
```

If however we try to do that using pointers we get problems, the following code declares `pScore` as a pointer to an `int` and `ammo` as an `int.`

```
int* pScore, ammo;
```

Ammo is not a pointer it is variable of type `int`.
To make this clearer the statement could be re-written:

```
int *pScore, ammo;
```

The best way, however is to be more explicit and declare your pointers separately:

```
int* pScore;
int ammo;
```

Initialising Pointers

As with other variables you can initialise a pointer in the same statement you declare it:

```
int* pScore = 0;    // Declare and initialise a pointer
```

Assigning 0 to a pointer has special meaning, it is loosely translated as “point to nothing”. A pointer whose value is 0 is considered to be a NULL pointer. When declaring pointers it is important to remember to initialise them to something, even if that value is 0 (or NULL). This is because un-initialised pointers can cause many errors. They are known as wild pointers.

You may see the following:

```
// Declare and initialise a NULL pointer
int* pScore = NULL;
```

This is because the programmers who wrote the code wish to be explicit about the nature of the pointer.
NULL is a constant equal to 0 that is defined in multiple library files including `iostream`.

Assigning Addresses to Pointers

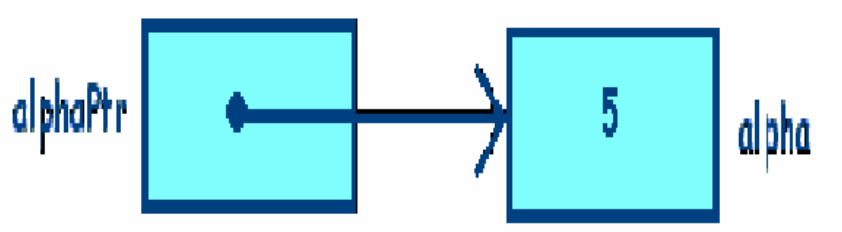
Because pointers store addresses of objects you need a way to get addresses into the pointers. One way to do that is to get the memory address of an existing variable and assign it to a pointer.

```
int* pAlphaPtr;
int alpha = 5;

/* assign pointer address of alpha to pAlphaPtr */
pAlphaPtr = &alpha;
```

The above statement gets the address of the variable `alpha` and assigns it to `pAlphaPtr`.

We get the address of `alpha` by preceding the variable name with the &, address of operator.
You should recognise this as it was used in the functions for pass-by-reference. In this context however it returns the address of an object.



The first part `int*` tells the compiler to declare a pointer for integers. `pAlphaPtr` will be the name of that pointer. In the last part of that statement, `&alpha`; specifies that the ***address*** of the variable alpha is what should be assigned to the pointer variable.
This is effectively what it does in terms of memory. `pAlphaPtr` contains the address of `alpha` whose value is equal to five.

As `pAlphaPtr` contains the address of `alpha` it's as if `pAlphaPtr` knows exactly where `alpha` is located in the computers memory.
Because of this you can use `pAlphaPtr` to get to `alpha` and manipulate the value stored by `alpha`.

We can test this, and we will in tutorial, by using the following code:

```
// address of alpha variable
cout << "&alpha is: " << &alpha << "\n";
// address stored in pointer
cout << "pAlphaPtr is: " << pAlphaPtr << "\n";
```

The values printed should be the same for both and should look something like this:

```
&alpha is: 0x4e772430
pAlphaPtr is: 0x4e772430
```

Dereferencing Pointers

The dereference operator is the asterisk - *

By using the dereference operator (*), by placing it in front of the pointer we display value to which the pointer refers not the pointer itself.
By placing the dereference operator (*) in front of a pointer you are saying "Treat this as the thing that the pointer references, not the pointer its self".

Therefore if we do the following:

```
// value pointed to by pointer
cout << "*pAlphaPtr is: " << *pAlphaPtr << "\n";
```

We should display the value pointed to by `pAlphaPtr`, which is currently 5 (see the code samples above). This is because `*pAlphaPtr` accesses the value stored in `alpha`.
`*pAlphaPtr` means the object to which `pAlphaPtr` points.

**Note: Don't de-reference a blind pointer or a null pointer as it could crash your program and cause disastrous results.*

We can also perform standard operations on the data pointed to by the pointer by using the dereference operator (*).
For instance:

```
*pAlphaPtr += 5;     // alpha will be equal to 10
```

Will add 5 to the value of `alpha`, this is because `pAlphaPtr` points to `alpha`.

You must be careful when performing manipulations using pointers. You must make sure that you use the dereference operator or strange and disastrous things may occur.
The following code:

```
// Add 5 to the memory address stored in pAlphaPtr
pAlphaPtr += 5;
```

Adds 5 to the address stored in `pAlphaPtr`, not to the value to which `pAlphaPtr` points.
As a result `pAlphaPtr` now points to a memory address that may contain anything.

**Note: Please note the use of the ampersand & reference operator and asterisk * de-reference operator.*
*You **cannot** use these operators inter-changeability:*
&P - means the address of a particular object P.
**P - means the contents of memory pointed to by P.*

Reassigning Pointers

Unlike references (those using the & operator) pointers can point to different objects at different times during the lifetime of a program.
Reassigning a pointer works like reassigning any other variable, we use the assignment operator (=) followed by a relevant value or object:

```
int newAlpha = 20;
/* assign pointer address of newAlpha to pAlphaPtr */
pAlphaPtr = &newAlpha;
```

In the code above we reassign `pAlphaPtr` to `newAlpha`. Therefore `pAlphaPtr` points to `newAlpha`.

Don't change the value to which a pointer points when you want to change the pointer its self. For example if I want to change `pAlphaPtr` to point to `newAlpha`. Then this would be wrong:

```
*pAlphaPtr = &newAlpha;
```

This would change the value to which `pAlphaPtr` points to the value stored in `newAlpha`.
If `alpha` was 5 and `newAlpha` was 20 then `alpha` would now equal 20.

Pointers and Objects

We have only covered pointers and integers so far, although we have mentioned that pointers can be used for other built in types and also with objects.
It works much the same as the code below demonstrates:

```
// string object
string sScore = "Score";
// pointer to a string object
string* pStr = &sScore;
```

As we have seen above `pStr` can point to any string object, we just need to reassign. `pStr` therefore points to the address of a string object.

```
// string pointed to by pointer
cout << "*pStr is: " << *pStr << "\n";
```

We can perform all the operations we have seen, this code for example prints out `Score` as that is the text stored in the string object.

Pointers and Constants

You can use the keyword const to restrict the way a pointer works. These restrictions can act as safe guards and can make your programming intensions clearer.
Since pointers are quite versatile using constant pointers is inline with the programming mantra of asking only for what you need.

Using constant pointers

By using the keyword const when we declare and initialise a pointer, we declare a *constant pointer*. This restricts the pointer to only point to the object it was initialised to point to.
With constants the value of the data stored cannot change, with constant pointers the memory address of the pointer cannot change.

```
int alpha;
// a constant pointer
int* const pAlphaPtr = &alpha;
```

The preceding code creates a constant pointer `pAlphaPtr` that points to `alpha`.
You declare a constant pointer just as you would a normal pointer, you just put the keyword const before the pointer name.

As with all constants you must initialise it when you first declare it or you will generate errors.

```
// illegal – error – must be initialised
int* const pAlphaPtr;
```

Because a constant pointer can only ever point to one memory location the following is illegal:

```
// a constant pointer
int* const pAlphaPtr = &alpha;
// illegal – can't point to different object
pAlphaPtr = &newAlpha;
```

Changing the pointees value
We said that you can't alter the memory address of the constant pointer, but what about altering the value of the object to which the pointer points (the pointee)?
You alter the value of the pointee with no problems, even using the pointer to accomplish this:

```
*pAlphaPtr = 6000;
```

This is because the constant restriction applies only to the value that the object stores, a pointer essentially stores a memory address. This is the value that cannot change.
You can look at it using the house example. A pointer can only hold one address at once, this address can change. A constant pointer can only one address and this cannot change. What is stored in the house is free to change though.

Although constant pointers have a similar syntax to references, references have a cleaner syntax than pointers and can make your code easier to read.

Using pointers to constants
As we have seen you can use pointers to change the values to which they point. Using the keyword const before all the rest of the declaration we can restrict a pointer so that it cannot be used to change the value to which it points.

```
// a pointer to a constant
const int* pNumber;
```

This declares a pointer to a constant. We can then assign the address to the pointer as we did before with the standard pointers. We do not have to use an actual constant, although we may, we can also use a standard variable:

```
int lives = 10;
pNumber; = &alpha;
```

You cannot however alter the value to which it points; the value of the pointee will remain the same.
This means that the following is illegal:

```
// illegal – can't use a pointer to a constant to change
// the value to which the pointer points
```

```
*pNumber += 1;
```

Although cannot change the value to which the pointer points, we can change the pointer so that it points to different objects in the program at different times.
The following code, where we change the pointee, is therefore perfectly legal:

```
const int MAX_LIVES = 10;
// a pointer to a constant
pNumber = &MAX_LIVES;
```

Using constant pointers to constants
So far we have seen constant pointers and pointers to constants; each of these had different restrictions. A *constant pointer to a constant* combines these restrictions.
This means that the pointer can only point to the object that it was initialised to and that neither the address of the pointer nor the value of the pointee can change.

```
// constant pointer to a constant
const int* const pARMOURBONUS = &ARMOUR;
```

The preceding code creates a constant pointer to a constant named `pARMOURBONUS` that points to a constant `ARMOUR`.

Like a pointer to a constant a constant pointer to a constant can point to either a constant or non-constant value.

You cannot reassign, change where it points, of a constant pointer to a constant. The following is therefore illegal:

```
// illegal - can't change to point
// at another object
pARMOURBONUS = &MAX_LIVES;
```

Neither can you use a constant pointer to a constant to change the value of the pointee:

```
// illegal - can't change value
// through this pointer
*pARMOURBONUS += 5;
```

In many ways a constant pointer to a constant acts like a constant reference, which can only refer to the value it was initialised to refer to and which can't be used to change that value.

Although you can use a constant pointer to a constant instead of a constant reference, you should stick with constant references where ever possible. References have a cleaner syntax than pointers and can make your code easier to read.

Summarising Constants and Pointers

Pointer constants and constant pointers are also something that many people simply don't use. If you have a value in your program and it should not change, or if you have a pointer and you don't want it to be pointed to a different value, you should make it a constant with the const keyword.

This aims to provide a brief summary to help make sure these concepts are understood.

Now consider the following three declarations assuming that char_A has been defined as a type char variable:

```
const char * myPtr = &char_A;
char * const myPtr = &char_A;
const char * const myPtr = &char_A;
```

What is the difference between each of the valid ones? Do you know?

They are all three valid and correct declarations. Each assigns the address of char_A to a character pointer. The difference is in what is constant.

The first declaration:

```
const char * myPtr
```

Declares a pointer to a constant character, you cannot use this pointer to change the value being pointed to:

```
char char_A = 'A';
const char * myPtr = &char_A;
// error - can't change value of *myPtr
*myPtr = 'J';
```

The second declaration,

```
char * const myPtr;
```

Declares a constant pointer to a character. The location stored in the pointer cannot change. You cannot change where this pointer points:

```
char char_A = 'A';
char char_B = 'B';

char * const myPtr = &char_A;
// error - can't change address of myPtr
myPtr = &char_B;
```

The third declares a pointer to a character where both the pointer value and the value being pointed at will not change.

Pretty simple, but as with many things related to pointers, a number of people seem to have trouble

Pointers and Functions

In the original C language, function parameters are always passed as value parameters, i.e. a copy of the data is made and the function deals with the copy. This means that in C the function cannot alter the data it was passed. However, if we pass a pointer as a value parameter, the function can de-reference the pointer and modify the data it points to.
This gives the same effect as reference parameters in C++, but the notation is a bit less convenient. It is therefore preferable to use pass-by-reference and to abstain from passing pointers.

Despite this you may sometimes need to pass a pointer, for instance when using a graphics engine that returns a pointer to a 3D object. If you want another function to use this object then it becomes necessary to pass the pointer to the object for efficiency.

Pointers pass any other object or data type instance. You must obviously declare that your function takes pointers as actual parameters though. Therefore in your function declaration the header needs to declare that its parameters are pointers.
For instance:

```
void swap(int* const px, int* const py)
```

Shows us that the function takes two pointers as arguments, the function will therefore accept two memory addresses as arguments.

When we call the function in our program, we therefore need to pass pointers. Remember though that pointers are memory addresses so we can pass data in the following way:

```
int myScore = 1;
int yourScore = 100;
// pass the memory addresses
// of the variables
swap(&myScore, &yourScore);
```

Swap pointers function
The following function swaps the values of two integer variables by passing copies of their addresses as pointer value parameters.

```
void swap(int* const px, int* const py)
{
      // store value pointed to
      // by px is temp
    int temp = *px;
      // stored value pointed to
      // by py in address pointed
      // to by px
```

```
    *px = *py;
      // store original value pointed
      // to by px in address pointed
      // to by py
    *py = temp;
}
```

This mirrors the swap function that we saw when we covered pass-by-reference in functions, but rather than using references it uses pointers.

Passing a constant pointer
We have seen functions obtain access to variables through pass-by-reference. We can accomplish this through pointers. When we pass a pointer we pass only the address of an object. This, like pass-by-reference can be very efficient. A good example is to view it like e-mailing a friend the URL to a website rather than e-mailing him the whole site.

The above function uses constant pointers as we don't wish to alter the pointer themselves; we want them to continue storing the memory addresses they hold. We do however want to alter their pointees values.

You can also pass a constant pointer to a constant. This works almost identically to passing by constant reference.

Returning Pointers
Before references the only option for games programmers to return objects efficiently was by using pointers.

Returning a Pointer
Before you can return a pointer from a function you must specify that you are returning one.

```
string* inventoryItem(string sInventory[], int index);
```

This function header declares that the function takes the parameters – an array of strings and an index
It also declares that the function returns a pointer to a string object and not a string object itself.

As you can see to specify that your function returns a pointer just put an asterisk after the type name of the return type.

```
string* inventoryItem(string sInventory[], int index)
{
     // return memory address of the
     // array and index
     return &(sInventory[index]);
}
```

This function then returns a pointer, a memory address; we do this using the address of (&) operator.

Essentially the return statement says return the memory address of the array at the index position.

We could then call the function and use the returned value in the following way:

```
// inventory array
string sInventory[4] = {"Sword", "Armour", "Shield",
"Potion"};
// pointer to string type
string* pString;
// call inventory item, storing return
// address in to pString
pString = inventoryItem(sInventory, 2);
// print out memory address stored in pString
cout << "pString: " << pString << "\n";
// print out the string that is stored in the pointee
cout << "*pString: " << *pString << "\n\n";
```

After performing some operations we should be familiar with – declaring instances of strings, arrays and pointers.
We then call the `inventoryItem` function passing some our array and our integer. The function the returns a memory address that is assigned to our pointer.

*Note: You must be careful, being aware of the scope of objects in your functions when you are returning pointers.

```
string* scopeProblem()
{
     string localScope = "This string will cease to exist
once the function call ends";
     string* pLocal = &local;
     return pLocal;
}
```

This function returns a pointer, if used the return value will most likely cause the program to crash.
This is because a pointer is returned to a string that no longer exists. The memory is freed for reuse. Attempting to dereference a pointer of this type, wild or dangling pointer, can lead to disastrous results for your program. To avoid this never return a pointer to a local variable.

Arrays and Pointers

Arrays and pointers have a special relationship. This is because an array name is really a constant pointer to the first element of the array.
Because the array elements are stored in a contiguous block of memory, you can use the array name as a pointer for random access to the elements. This relationship also has implications for the passing and return of arrays in functions.

Due to this relationship between array names and pointers, the following is legal.

```
// Array of 100 ints
int  Array_Scores [100];
// pointer to an int
int  *ptr;
// Assigns address of first element of My_Array to ptr
ptr = Array_Scores;
// Does the same as above
ptr = &Array_Scores[0];
```

When you declare an array, the name is a *constant pointer*, which cannot be altered, effectively making it a constant.
In the previous example, you could never make this assignment:

```
// ILLEGAL because My_Array is a constant, although the
// correct type
My_Array = ptr;
```

Because the array name is a pointer to the first element of the array you can dereference an array to access the first element:

```
// Array of 100 ints
int  Array_Scores[100];
cout << *Array_Scores << endl;
```

This prints out the value stored in the first element of the array.

This information can be used to randomly access array elements using an array name as a pointer. This just involves simple addition.
When you add an integer to a pointer, the integer is *multiplied by the type size* of the type that the pointer points to. If you have an array of characters this is one, if it is an array of integers this is 4. The reason this occurs is to move the address on by one, so we point at the next address holding information.

This enables you to access different elements of the array using just the array name.

```
// Array_Scores is an array of 100 ints
```

```
int Array_Scores [100];
// ptr is a pointer to an int
int *ptr;
// Assigns address of first element to an int
ptr = my_array;
// Adds 4 to ptr (4 equiv. to 1*sizeof(int))
ptr = ptr + 1;
```

This next set of code prints out the value stored in the array at the position indicated by the number added. Arrays are index from zero so by dereferencing the array name prints out the value stored in index zero.

```
// print out the value stored in the array at index 1
cout << *(Array_Score + 1) << endl;
// print out the value stored in the array at index 7
cout << *(Array_Score + 7) << endl;
```

Symbol Summary

*	De-reference operator, indirection operator	This is used ***to declare*** a variable as a pointer. It is also used when you want ***to access*** the value pointed to by the pointer variable.
&	Reference operator, address-of operator	Use before a variable to indicate that you mean the ***address of*** that variable. You'll often see this in a function header where the parameter list is given.
->	Member selection operator	This is used to refer to ***members of structures***

A Warning when using the ampersand *&* operator:

It is used in several different roles, for example it is used in:
– Pass by reference operator. i.e. Reference Parameter,
– Bitwise AND,
– Address of memory variable. i.e. an address of an object.

You must ensure that you are aware of which particular role the ampersand *&* operator is playing in the different parts of a program.

An & (ampersand) in front of an object name means, “give me the memory address of the object”:

Pointer Summary

Computer memory is organised in an ordered way with each chunk of memory having its own unique identifier

A pointer is a variable that contains a memory address

Programmers often pre-fix pointer variable names with p to remind themselves that the variable is a pointer

A pointer is declared to refer to the value of a specific type

It's considered good practice to initialise a pointer when you declare it

If you assign 0 to a pointer it's called a null pointer

To get the address of a variable put address-of operator (&) before the variable name

When a pointer contains the address of an object it's said to point to the object

Unlike references you can re-assign pointers. That is a pointer can point to different objects during the life of the program

You deference the pointer a pointer to access the object it points to with the * deference operator

You can use the -> operator with pointers for a more readable way to access object data members and member functions

A constant pointer can only point to the object it was initialised to point to. You declare a constant pointer by putting the keyword `const` right before the pointer name, as in `int* const p = &i`;

You can't use a pointer to a constant to change the value to which it points. You declare a pointer to a constant by putting the keyword `const` right before the pointer name, as in `const int*;`

A constant pointer to a constant can only point to the value it was initialised to point to, and it can't be used to change that value. You declare a constant pointer to a constant by putting the keyword `const` before the type name and right before the pointer name, as in `const int* const p = &i`;

You can pass pointers for efficiency or to provide direct access to an object

If you want to pass a pointer for efficiency, you should pass a pointer to constant or a constant pointer to a constant so that the object you're passing access to can't be changed through the pointer.

A dangling pointer is a pointer to an invalid memory address. Dangling pointers are often caused by deleting an object to which a pointer pointed. Dereferencing such a pointer can lead to disastrous results.

You can return a pointer from a function, but be careful not to return a dangling pointer

Pointers with Binky

See also: cslibrary.stanford.edu/104/ for a fun description of pointers.
This video uses the new operator something covered in a few weeks time, for now it is enough to know that it creates a new instance of a variable.