

What are Pointers?

Pointers are basically the same as any other variable. However, what is different about them is that instead of containing actual data, they contain a pointer to the memory location where information can be found. This is a very important concept, and many programs and ideas rely on pointers as the basis of their design, linked lists for example.

Getting Started

How do I define a pointer? Well, the same as any other variable, except you add an asterisk before its name. So, for example, the following code creates two pointers, both of which point to an integer:

```
int* pNumberOne;  
int* pNumberTwo;
```

Notice the 'p' prefix in front of the two variable names? This is a convention used to indicate that the variable is a pointer.

Now, lets make these pointers actually point to something:

```
pNumberOne = &some_number;  
pNumberTwo = &some_other_number;
```

The & (ampersand) sign should be read as 'the address of', and causes the address in memory of a variable to be returned, instead of the variable itself. So, in this example, `pNumberOne` is set to equal the address of `some_number`, so `pNumberOne` now points to `some_number`;

Now, if we want to refer to the address of `some_number`, we can use `pNumberOne`. If we want to refer to the value of `some_number` from `pNumberOne`, we would have to say `*pNumberOne`. The `*` dereferences the pointer, and should be read as 'the memory location pointed to by', unless in a declaration, as in the line `int *pNumber`.

What we've learnt so far: an example:

Phew! That's a lot to take in, I'd recommend that if you don't understand any of those concepts to give it another read through; pointers are a complex subject, and it can take a while to master them.

Here is an example which demonstrates the ideas discussed above. It is written in C, without the C++ extensions.

```
#include <stdio.h>  
  
void main()  
{  
    // declare the variables:  
    int nNumber;
```

```

int *pPointer;

// now, give a value to them:
nNumber = 15;
pPointer = &nNumber;

// print out the value of nNumber:
printf("nNumber is equal to : %d\n", nNumber);

// now, alter nNumber through pPointer:
*pPointer = 25;

// prove that nNumber has changed as a result of the above by
// printing its value again:
printf("nNumber is equal to : %d\n", nNumber);
}

```

Read through, and compile the above code sample, and make sure you understand why it works. Then, when you are ready, read on!

A trap!

See if you can spot the fault in the program below:

```

#include <stdio.h>

int *pPointer;

void SomeFunction();
{
    int nNumber;
    nNumber = 25;

    // make pPointer point to nNumber:
    pPointer = &nNumber;
}

void main()
{
    SomeFunction(); // make pPointer point to something

    // why does this fail?
    printf("Value of *pPointer: %d\n", *pPointer);
}

```

This program firstly calls the `SomeFunction` function, which creates a variable called `nNumber`, and then makes the `pPointer` point to it. Then, however, is where the problem is. When the function leaves, `nNumber` is deleted, because it is a local variable. Local variables are always deleted when execution leaves the block they were defined in. This means when `SomeFunction` returns to `main()`, the variable is deleted, so `pPointer` is pointing at where the variable used to be, which no longer belongs to this program. If you do not understand this, it may be wise to read back over on local and global variables, and on scope. This concept is also important.

So how can the problem be solved? The answer is by using a technique known as *dynamic allocation*. Please be aware that this is different between C and C++. Since most developers are now using C++, this is the dialect that the code below is using.

Dynamic Allocation

Dynamic allocation is perhaps the key to pointers. It is used to allocate memory without having to define variables and then make pointers point to them. Although the concept may appear confusing, it is really simple. The following code demonstrates how to allocate memory for an integer:

```
int *pNumber;
pNumber = new int;
```

The first line declares the pointer, `pNumber`. The second line then allocates memory for an integer, and then makes `pNumber` point to this new memory. Here is another example, this time using a `double`:

```
double *pDouble;
pDouble = new double;
```

The formula is the same every time, so you can't really fail with this bit.

What is different about dynamic allocation, however, is that the memory you allocate is not deleted when the function returns, or when execution leaves the current block. So, if we rewrite the above example using dynamic allocation, we can see that it works fine now:

```
#include <stdio.h>

int *pPointer;

void SomeFunction()
{
    // make pPointer point to a new integer
    pPointer = new int;
    *pPointer = 25;
}

void main()
{
    SomeFunction(); // make pPointer point to something
    printf("Value of *pPointer: %d\n", *pPointer);
}
```

Read through, and compile the above code sample, and make sure you understand why it works. When `SomeFunction` is called, it allocates some memory, and makes `pPointer` point to it. This time, when the function returns, the new memory is left intact, so `pPointer` still points to something useful. That's it for dynamic allocation! Make sure you understand this, and then read on to learn about the fly in the ointment, and why there is still a serious error in the code above.

Memory comes, memory goes

There's always a complication, and this one could become quite serious, although it's very easy to remedy. The problem is that although the memory that you allocate using dynamic allocation is conveniently left intact, it actually never gets deleted

automatically. That is, the memory will stay allocated until you tell the computer that you've finished with it. The upshot of this is that if you don't tell the computer that you've finished with the memory, it will be wasting space that other applications, or other parts of your application could be using. This eventually will lead to a system crash through all the memory being used up, so it's pretty important. Freeing the memory when you've finished with it is very simple:

```
delete pPointer;
```

That's all there is to it. You have to be careful however, that you pass a valid pointer, i.e. that is a pointer that actually points to some memory you've allocated, and not just any old rubbish. Trying to `delete` memory that's already been freed is dangerous and can lead to your program crashing.

So here is the example again, updated this time so it doesn't waste any memory:

```
#include <stdio.h>

int *pPointer;

void SomeFunction()
{
    // make pPointer point to a new integer
    pPointer = new int;
    *pPointer = 25;
}

void main()
{
    SomeFunction(); // make pPointer point to something
    printf("Value of *pPointer: %d\n", *pPointer);

    delete pPointer;
}
```

One line difference is all it took, but this line is essential. If you don't delete the memory, you get what is known as a 'memory leak', where memory is gradually leaking away, and can't be reused unless the application is closed.

Passing pointers to functions

The ability to pass pointers to function is very useful, but very easy to master. If we were to make a program that takes a number and adds five to it, we might write something like the following:

```
#include <stdio.h>

void AddFive(int Number)
{
    Number = Number + 5;
}

void main()
{
    int nMyNumber = 18;
```

```

printf("My original number is %d\n", nMyNumber);
AddFive(nMyNumber);
printf("My new number is %d\n", nMyNumber);
}

```

However, the problem with this is that the `Number` referred to in `AddFive` is a copy of the variable `nMyNumber` passed to the function, not the variable itself. Therefore, the line `Number = Number + 5` adds five to the *copy* of the variable, leaving the original variable, in `main()`, unaffected. Try running the program to prove this.

To get around this problem, we can pass a pointer to where the number is kept in memory to the function, but we'll have to alter the function so that it expects a pointer to a number, not a number. To do this, we change `'void AddFive(int Number)'` to `'void AddFive(int* Number)'`, adding the asterisk. Here is the program again, with the changes made. Notice that we have to make sure we pass the address of `nMyNumber` instead of the number itself? This is done by adding the `&` sign, which (as you will recall) is read as 'the address of'.

```

#include <stdio.h>
void AddFive(int* Number)
{
    *Number = *Number + 5;
}

void main()
{
    int nMyNumber = 18;

    printf("My original number is %d\n", nMyNumber);
    AddFive(&nMyNumber);
    printf("My new number is %d\n", nMyNumber);
}

```

Try coming up with an example of your own to demonstrate this. Notice the importance of the `*` before `Number` in the `AddFive` function? This is needed to tell the compiler that we want to add five to the number pointed to by the variable `Number`, rather than add five to the pointer itself.

The final thing to note about functions is that you can return pointers from them as well, like this:

```
int * MyFunction();
```

In this example, `MyFunction` returns a pointer to an integer.

Pointers to Classes

There are a couple of other caveats with pointers, one of which is structures or classes. You can define a class as follows:

```

class MyClass
{
public:
    int m_Number;
}

```

```
char m_Character;
};
```

Then, you can define a variable of type `MyClass` as follows:

```
MyClass thing;
```

You should already know this, if not try reading up on this area. To define a pointer to `MyClass` you would use:

```
MyClass *thing;
```

as you would expect. Then, you would allocate some memory and make this pointer point to the memory:

```
thing = new MyClass;
```

This is where the problem comes in, how then would you use this pointer. Well, normally you would write `'thing.m_Number'`, but you can't with a pointer because `thing` is not a `MyClass`, but a pointer to it, so `thing` itself does not contain a variable called `m_Number`; it is the structure that it points to that contains `m_Number`. Therefore, we must use a different convention. This is to replace the `'.'` (dot) with a `->` (dash followed by a greater than sign). An example showing this is below:

```
class MyClass
{
public:
    int m_Number;
    char m_Character;
};

void main()
{
    MyClass *pPointer;
    pPointer = new MyClass;

    pPointer->m_Number = 10;
    pPointer->m_Character = 's';

    delete pPointer;
}
```

Pointers to Arrays

You can also make pointers that point to arrays. This is done as follows:

```
int *pArray;
pArray = new int[6];
```

This will create a pointer, `pArray`, and make it point to an array of six elements. The other method, not using dynamic allocation, is as follows:

```
int *pArray;
int MyArray[6];
pArray = &MyArray[0];
```

Note that, instead of writing `&MyArray[0]`, you can simply write `MyArray`. This, of course, only applies to arrays, and is a result of how they implemented in the C/C++ language. A common pitfall is to write `pArray = &MyArray;`, but this is incorrect. If you write this, you will end up with a pointer to a pointer to an array (no typo), which is certainly not what you want.

Using pointers to arrays

Once you have a pointer to an array, how do you use it? Well, let's say you have a pointer to an array of `ints`. The pointer will initially point to the first value in the array, as the following example shows:

```
#include <stdio.h>

void main()
{
    int Array[3];
    Array[0] = 10;
    Array[1] = 20;
    Array[2] = 30;

    int *pArray;
    pArray = &Array[0];

    printf("pArray points to the value %d\n", *pArray);
}
```

To make the pointer move to the next value in the array, we can say `pArray++`. We can also, as some of you probably guessed by now, say `pArray + 2`, which would move the array pointer on by two elements. The thing to be careful of is that you know the what the upper bound of the array is (3 in this example), because the compiler cannot check that you have not gone past the end of array when you are using pointers, so you could easily end up crashing the system. Here is the example again, showing this time the three values that we set:

```
#include <stdio.h>

void main()
{
    int Array[3];
    Array[0] = 10;
    Array[1] = 20;
    Array[2] = 30;

    int *pArray;
    pArray = &Array[0];

    printf("pArray points to the value %d\n", *pArray);
    pArray++;
    printf("pArray points to the value %d\n", *pArray);
    pArray++;
    printf("pArray points to the value %d\n", *pArray);
}
```

You can also subtract values as well, so `pArray - 2` is 2 elements from where `pArray` is currently pointing. Make sure, however, that you add or subtract to the

pointer and not to its value. This kind of manipulation using pointers and arrays is most useful when used in loops, such as the for or while loops.

Note that, also, if you have a pointer to a value, e.g. `int* pNumberSet`, you can treat it as an array, for example `pNumberSet[0]` is equivalent to `*pNumberSet`, and `pNumberSet[1]` is equivalent to `*(pNumberSet + 1)`.

One final word of warning for arrays, is that if you allocate memory for an array using `new`, as in the following example:

```
int *pArray;  
pArray = new int[6];
```

it must be deleted using the following:

```
delete[] pArray;
```

Notice the `[]` after `delete`. This tells the compiler that it is deleting a whole array, and not just a single item. You must use this method whenever arrays are involved, otherwise you will end up with a memory leak.

Last Words

One final note: you must not delete memory that you did not allocate using `new`, as in the following example:

```
void main()  
{  
    int number;  
    int *pNumber = number;  
  
    delete pNumber; // wrong - *pNumber wasn't allocated using new.  
}
```

Common Questions and FAQ

Q: Why do I get 'symbol undefined' errors on `new` and `delete`?

A: This is most likely caused by your source file being interpreted by the compiler as being a plain C file, and the `new` and `delete` operators are a new feature of C++. This is usually remedied by ensuring that you are using a `.cpp` extension on your source code files.

Q: What's the difference between `new` and `malloc`?

A: `new` is a keyword only present in C++, and is now the standard way (other than using Windows' memory allocation routines) to allocate memory. You should never use `malloc` within a C++ application unless absolutely necessary. Because `malloc` is not designed for object-oriented features of C++, using it to allocate memory for classes will prevent the constructor of the class being called, as just one example of the problems that can arise. As a result of the problems that arise from the use of

`malloc` and `free`, and because they are now for all intents and purposes obsolete, they are not discussed in any detail in this article, and I would discourage their use wherever possible.

Q: Can I use `free` and `delete` together?

A: You should free memory with the equivalent routine to that used to allocated it. For instance use `free` only on memory allocated with `malloc`, `delete` only on memory allocated with `new` and so on.

References

References are, to a certain degree, out of the scope of this article. However, since I've been asked many times by people reading this about them, I will discuss them briefly. They are very much related to pointers in that, in many cases, they can be used as a simpler alternative. If you recall above, I mentioned that the ampersand (&) is read as 'the address of' unless in a declaration. In the case of its presence in a declaration such as that shown below, it should be read as 'a reference to'.

```
int& Number = myOtherNumber;  
Number = 25;
```

The reference is like a pointer to `myOtherNumber`, except that it is automatically dereferenced, so it behaves just as it were the actual value type rather than a pointer type. The equivalent code to this, using pointers, is shown below:

```
int* pNumber = &myOtherNumber;  
*pNumber = 25;
```

The other difference between pointers and references is that you cannot 'reset' a reference, that is to say that you cannot change what it is pointing to after its declaration. For instance, the following code would output '20':

```
int myFirstNumber = 25;  
int mySecondNumber = 20;  
int &myReference = myFirstNumber;  
  
myReference = mySecondNumber;  
  
printf("%d", myFristNumber);
```

When in a class, the value of the reference must be set by the constructor in the following way:

```
CMyClass::CMyClass(int &variable) : m_MyReferenceInCMyClass(variable)  
{  
    // constructor code here  
}
```

Summary

This topic is very hard to master initially, so it is worth looking over at least twice: most people do not understand it immediately. Here are the main points again:

1. Pointers are variables that point to an area in memory You define a pointer by adding an asterisk (*) in front of the variable name (i.e. `int *number`).
2. You can get the address of any variable by adding an ampersand (&) in front of it (i.e. `pNumber = &my_number`).
3. The asterisk, unless in a declaration (such as `int *number`), should be read as 'the memory location pointed to by'.
4. The ampersand, unless in a declaration (such as `int &number`), should be read 'the address of'.
5. You can allocate memory using the 'new' keyword.
6. Pointers MUST be of the same type as the variables you want them to point to, so `int *number` will not point to a `MyClass`.
7. You can pass pointers to functions.
8. You must delete memory that you have allocated by using the 'delete' keyword.
9. You can get a pointer to array that already exists by using `&array[0]`.
10. You must delete an array that is dynamically allocated using `delete[]`, not just `delete`.

This is not an absolutely complete guide to pointers, there are a few other things that I could have covered in more detail, such as pointers to pointers, and also things that I chose not to cover at all, such as function pointers, which I believe are too complex for a beginner's article, and things that are used rarely enough so that a beginner would be better off not flummoxed by these unwieldy details.

That's it! Try running some of the programs presented here, and come up with some examples of your own.

Updates

- 10 July 2002 - FAQ added, some minor changes throughout article to improve clarity, updating of an incorrectly placed source code snippet, addition of section on references.