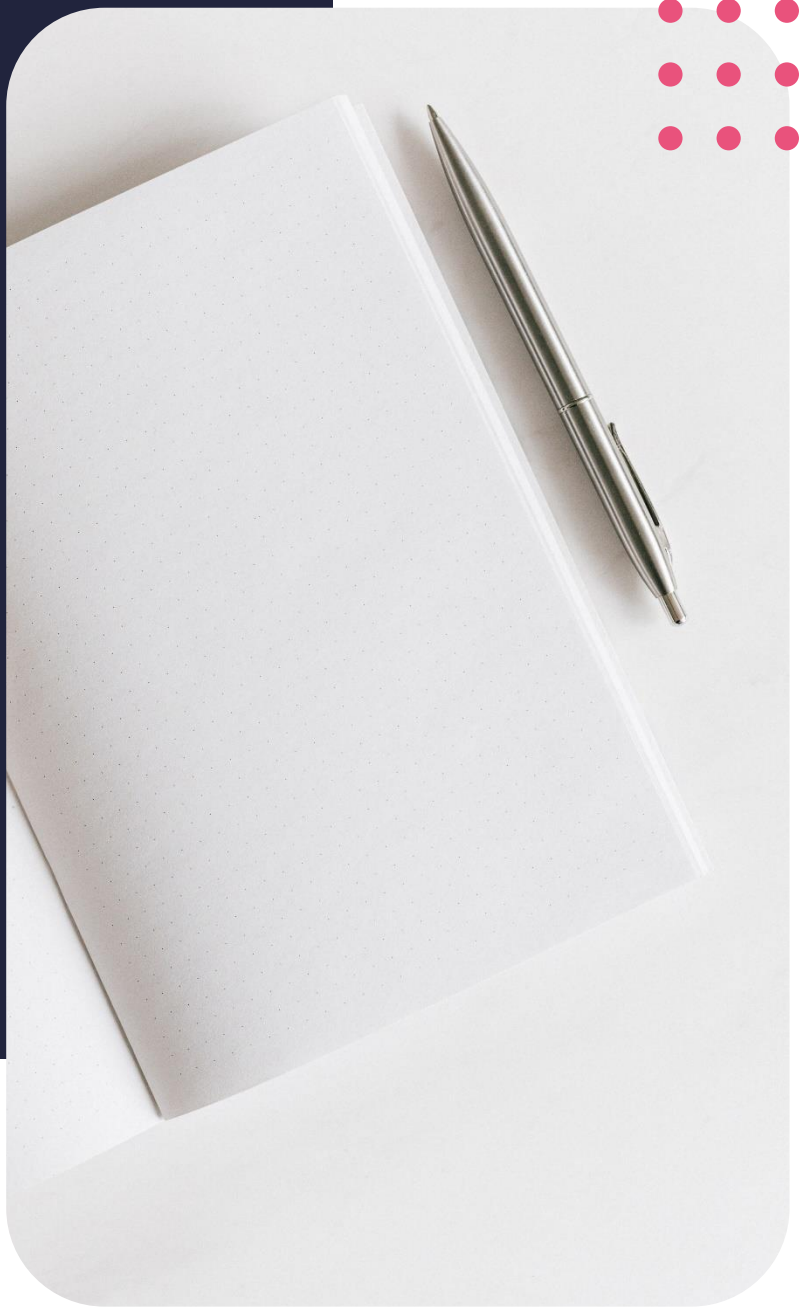
Diploma in Computer Science

Pointers

Contents

The science of pointers	3
A bit of theory…	3
Pointer operations	5
Syntax	5
Types of pointers	6
Pointers as arguments	6
Function pointers	7
Null pointers	7
Void pointer	8
Pointer arithmetic	8
Increment	9
Pointers vs arrays	10
Pointer to pointer	13
Linked lists	13
Things to note when using pointers	15
Conclusion	15



Lesson outcomes

By the end of this lesson, you should be able to:

Understand the concept of pointers as used in computer science

Apply the concept of arithmetic operators to pointers

Recognise the role of pointers in making software more efficient

Appreciate the idea of linked lists as a data structure that uses pointers

The science of pointers

We've already talked about variables and pointers themselves are variables, so why are we looking at this again?

Because a pointer is not just an ordinary variable. Instead of a pointer holding a variable from a source such as the keyboard or the result of a calculation, it holds the address of a memory location of another variable. You may be wondering why on earth would we want to do that. Stick with us and you'll see!

The reason why we are so interested in pointers is because they are a low-level idea and are closely related to how microprocessors work, both practically and theoretically. This is not just good for computer science, but it is useful in the practical world, especially if your program is just one layer away from assembly language, like C. If you're going to venture into operating system kernels, embedded systems, and device drivers then you better pay extra, lean-on-your-toes attention! This is pretty much how that low level software remains efficient. Still confused? We are going to look closely at how this works on a conceptual level in our next section.

In the meantime, an easy way of envisioning a pointer is to think of a road sign. When you get to a road sign that says you are 10 kilometres from the beach, it simply points to the direction you have to follow to get to the beach.

The sign itself is not at the beach, it is simply a representation.

A bit of theory...

Not everyone likes theory, but in this case it's a really good place to start.

Much like our example of a road sign, in computer science, a pointer is recorded as a kind of reference to a data primitive.

So, let's break that down to understand it better…

A data primitive is a piece of data that can be written or read from computer memory using an address.

If you group data primitives in a logically contiguous manner in memory and view them collectively as one piece of data, then they are referred to as a data aggregate. If you remember from module 1, we learned that a byte is the smallest addressable unit of memory.

If we group together several data primitives of the same type, then we get an array. Each of the data primitives within the array is stored in a separate distinct memory location, even though the array as a whole is contiguous.

Now the address of the first element in the array is considered to be the base address of the entire array. Generally, a pointer would use this address.

Let's revisit variables a bit. A variable can be associated with: a type - which dictates the size of the variable and the kind of data that you can store in it.

a name –to call the variable and identify it by an address - which is the physical or logical location of a variable in memory. For a computer to know where to find anything in memory, it uses an address. If you remember from our lesson on variables, there are some variables that sit on more than one byte, such as long int.

As just mentioned above, the base address come up which is the first address in that set of bytes is used is the address of that variable. This is because if we know the size of the variable, then we do not need to reference all the memory cells occupied by that variable. We just use the base address to calculate the span of the variable. Genius!

To get a clearer picture of what we're talking about, let's look at a code snippet. If we declare a variable of type long long int and give it a value of say 9, it will look like this:

```
Long long int age=9;
```

We established that the variable uses up 8 bytes.

Now, if you modify the code a bit and use an ampersand operator, it returns the base address of the variable in this modified code snippet.

Now if you compile and run this code, it returns an address. Despite the variable using up 8 bytes, we only get one address.

It looks like pointers can be a bit confusing , so why do we even need them? Well, let's take a look at some of the real world uses of pointers.

Why do we even need pointers?

Pointers might just look like a silly way of making things overcomplicated instead of just using the variable name, but they're pretty useful when you want to work with more complex data structures such as linked lists, graph trees and so on. Structures such as linked lists would be near impossible to create if it weren't for pointers.

Pointers are particularly useful to store and manage the addresses of time and amicably allocated blocks of memory such as those used to store arrays and data objects.

Pointers are also critical to reducing the system footprint of a program. If you look at the explanation that we gave earlier, it is clear that it's much easier to reference the base address of a chunk of data then to reference all the individual addresses in the block of data. Not only does it need loads and loads of memory, but it also quickly becomes a whole lot of work for the processor as it needs to resolve each address individually.

Pointers also allow modifications to a variable by the function that did not create the memory object, that is, the function can allocate memory and another function can modify the values without the use of global variables.

Passing on things like arrays saves a whole lot of memory because we only pass the address of the array instead of all the elements of the array, which would mean that a copy of the whole array would be made so that the function has data to work on instead of just having a reference to the array (as we shall see later in the lesson) Pointers allow you to return more than one value from a function. this is especially useful when you are using objects such as structures and Aries in different parts of a program. Without pointers there is no real easy way of passing such structures to a function and returning values from them. More about this in our next module.

In our next lesson we will look closely at structures and their use in storing data of different kinds. We will also look at typecasting which allows us to change an expression's data type.

Pointer operations

Now that we have established exactly what pointers are, let's look at what you can do with them.

We have a dimension that pointers are variables, this means that most of what you can do with ordinary variables, you can do with pointers. This includes things like arithmetic logic and passing them to functions and blocks of code.

Syntax

As usual, we'll look at how we put this concept into actual code. So, let's look at the syntax for the general office pointer in C. It looks like this:

```
Data type *pointer_name=&variable
```

Wait a minute, this looks a lot like a variable declaration! There are two things that we introduced here that are used by the compiler to tell apart the pointer from regular variables:

firstly, a pointer is always declared using an asterisk before its name. Secondly, the value assigned to the pointer needs to be an address. That is why we slapped an ampersand next to the variable that we assigned to the pointer.

Now let's look at this code snippet:

```
#include<stdio.h>
Int main(){
Int age=9;
Int *pointertoage;
Pointertoage=&age;
Printf("%d",pointertoage);
Printf("%d",pointertoage);
}
```

Notice that in this program, when we first declare the pointer, we use an asterisk, but from then on, when we reference the pointer name without the asterisk. This is because if you reference the pointer with an asterisk, it will take you to the value that is contained in the address that the pointer holds.

Feel free to copy the code in the snippet and run it yourself. You will notice that each time you run the program; the address contained in the pointer is different. This is because memory is not fixed, it is allocated as and when needed, so the value will always be different.

This code snippet shows some valid ways in which you can declare pointers:

```
Int *ptr_name;/* this is a pointer to an integer */
Int *ptr1,name;/* ptr1 is a pointer to type integer and name is an integer variable */
Double *ptr2; /* pointer to a double */
Float *ptr3; /* pointer to a float */
Char *ch1; /* pointer to a character */
Float *ptr, variable;/*ptr is a pointer to type float and variable is an ordinary float variable
*/
```

C gives you with quite a lot of flexibility in the ways in which you can declare pointers. Just a word of caution: just like we saw with variables, remember that you need to initialise pointers to avoid weird unpredictable and potentially catastrophic results.

Types of pointers

You can imagine that there can't be just one type of pointer.

Pointers can be categorised in many ways based on what they can do. We're going to

look at some of the most common pointer types and briefly explore what they can

do. We'll be using some of these pointer types in later lessons.

Pointers as arguments

Pointers can be used as function parameters, passed to the function when it is called. This functionality is also referred to as 'pass by reference'. This means that changes that we effect on the variable inside the function will be reflected on the original variable even after we exit the function. This quickly becomes the method of choice when working with large data structures because, as mentioned before,

no copy of the actual value is made but a reference to the memory location of the value is made which results in far fewer operations for the CPU.

It is worth noting here that you should only reserve pass-by-reference for functions that need to modify the value in question. This ensures code security.

Functions can also return pointers to the calling function. You also need to be careful here as pointers to local variables become invalid when the computer exits the function.

Function pointers

All along we have been referring to pointers that point to data. That's not the whole story, though. We can also use pointers to point to functions. A function pointer stores the address of a function that can then be called through that function pointer. This is useful where you want different behaviours for the same piece of code.

The syntax of a function pointer looks like this:

```
Void (*funtion_pointer)(data_type) = &function_name;
```

The address of the function as well as the returned data type is referenced in the pointer. If you've called a function using the pointer you can pass a value to it at runtime.

Using a function pointer, you can do clever things like replacing a switch statement. Also, the function pointer can be passed as an argument to a function and returned from a function. This functionality can be used instead of the ordinary method to avoid code redundancy.

Null pointers

variable that you want to reference, it is good programming practice to initialise the pointer as a null pointer.

The pointer will carry a zero value until a value is assigned to it.

The syntax looks like this:

```
Int *pointertoage=NULL;
```

Just like with ordinary variables, when you do not yet have the address of the variable, it is good practice to initialise it to zero.

If we were to print the value of pointerToAge, we would get the value 0.

It is better to use the word NULL than the number 0 because 0 is an actual value and could confuse readers, even though it works on the computer.

Void pointer

The last type of pointer we are going to talk about is the void pointer. This is also known as a generic pointer which does not have any data type. This type of pointer can store the address of any variable. To create a void pointer, we use the keyword ‘void’. It does sound like they are more flexible but there is some sad news. Void pointers cannot be dereferenced, meaning that you cannot use the* operator to retrieve the contents of the address to which they point. Also, you cannot perform pointer arithmetic with void pointers with the exception of GNU C, which considers the void pointer size to be 1. You may want to avoid pointer arithmetic using the void pointer because it makes your program less portable.

Pointer arithmetic

Pointers, being variables, can also be used in arithmetic operations.

Let's now look at some of the ways in which we CAN perform arithmetic using pointers.

We've already mentioned that pointers are just the same as regular variables, the only difference being that pointers store addresses and regular variables store data. Because pointers store addresses, there are fewer arithmetic operations that you can perform on them. It seems rather silly that you can perform some operations and not others, but it is for a good reason.

To better understand this, think of pointers as the house numbers on each of the houses in your street. A house number itself is not very useful if the house is not there. Now, you can perform an operation like subtracting a larger house number from a smaller one. You probably won't run into any errors unless if you subtract a smaller house number from a larger one then you get a negative house number which doesn't exist. And the same applies if you add house numbers. You are more likely to get a house number that doesn't exist than to get a meaningful address. The problem gets even bigger if you try to multiply or divide house numbers, then you get values like house number 2.67, which doesn't even make any sense.

This is why you can only perform the single most sensible operation of them all which is subtraction.

When adding, multiplying, or dividing addresses, we have no idea what you end up pointing to, so it is a pretty useless exercise.

Also, memory is not necessarily contiguous, and by addition, you might end up encroaching into some other function’s memory space. To sum it up the difference between two pointers gives you the number of elements

of the type that can be stored between two pointers but spending them doesn't give you any meaningful functionality.

The set of permitted arithmetic operations permitted on pointers include:

Increment and decrement

Addition or subtraction of an integer

Subtracting one pointer from another

Comparison of two pointers, and

Assignment

Let's now look at each of the operations in detail.

Increment

The increment operator increases the value of a pointer by the size of the data object to which the pointer points. This operator is exactly the same as the operator

we use on normal variables, which is the double plus sign (++) The general syntax looks like this:

```
Pointer++;
```

Similarly, you can decrease the pointer using the decrement operator which, just like in normal variables, is denoted by a double minus sign (--)

The general syntax looks, almost like that of the increment operation like this: If, for example, our variable of type int is stored at address 12345 and referred to by a pointer p, executing the following line of code will change the address to 12349.

```
P++;
P--;
```

If that line of code is immediately followed by p--, then the address would revert to 12345.

You may be wondering why it did not become 12346. This is because the size of int is 4, so the computer first multiplies the value of the increment which is 1 by the size of the data type which is 4. This means that for us to get the next address space we need to move 4 units, hence our new address will be 12349. The same applies to the decrement operator. It is first multiplied by the size of the data type which is 4 then subtracted from the address.

The addition and subtraction operators work in pretty much the same way as what we saw with the increment and decrement operators.

Say we add four units to our pointer p as shown in this code snippet:

```
P+4;
```

```
P-2;
```

This time we'll do it a little bit differently and go back 2 spaces instead of the original 4. For the computer to calculate the new address it first looks at the size of the data type pointed to by the pointer. In this case our pointer is an int, so it needs 4 memory units. The number by which we want to increment the pointer is bit

multiplied by this size, which means that we are going to move 16 units from the original address. From our previous example, our initial address was 12345. If we add 16, your address will be 12361.

For our next operation we are subtracting two address spaces from our new address, 12361. Again, the computer will calculate the size of the data type that the pointer is pointing to, which is int. we have already established that int is 4 bytes. The computer that multiplies the number of moves that we are subtracting from the address, which is 2, by the size of the data type that it just established which is 4. This means that we are moving back 8 address spaces, hence our new address will be 12353. After doing this, you might be tempted to add two pointers together. This is an illegal operation in C and will result in an error.

If two pointers are pointing to the same array they can be compared. This can be achieved using all relational operators that we looked at in our lesson on operators. There is a rather complicated explanation of how comparisons are evaluated in the official C reference guide, but basically pointers can only be compared if they are contents of the same object. Anything other than that results in in error. Remember earlier in the lesson we referred to “the same object” as data that has been grouped and is part of the same data aggregate, for example an array or a structure. If for example you try to compare a pointer from an array a, and another pointer from an array b, it cannot be computed. In case you're wondering, this is one of those things that would, if allowed, produce results that are useless in the real world and will just introduce bugs.

We seem to have ignored assignment as one of the operations that can be performed on pointers. We have, in fact, covered it already. Assignment refers to transferring an address into a pointer. We have already done this in previous examples.

Pointers vs arrays

You may have noticed that we have been quietly avoiding arrays in this discussion, but it’s time to address the elephant in the room.

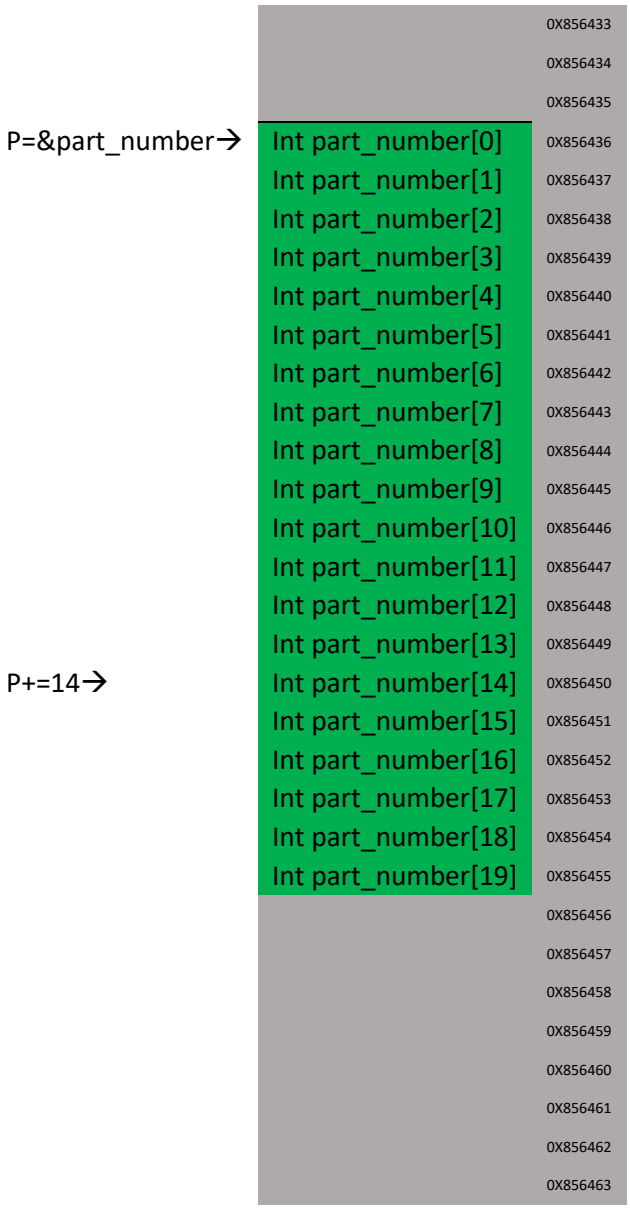
Arrays in particular work hand in hand with pointers to achieve faster execution speeds and a smaller system footprint. Let’s take for example an array named part_number . Here, the variable part_number refers to the base address of the array. If we declare a pointer p to the array, we can essentially access the array elements using the pointer. Let’s take a look at this scenario in the code snippet:

```
Int *p;
P = part_number;
// is the same as
P = &part_number[0]
```

This means that whenever you call an array without specifying an index, it will always refer to the base index. This also means that you can access the area elements using p++. It is not considered good programming practice to use the increment operator and the decrement operator in the same statement.

Not only does it hurt the readability of your code, but also has the potential to create undefined behaviour, and should be avoided whenever possible, which is 100% of the time! I mean why play with fire when you can avoid it?!

To make this a little easier to visualise let's look at this diagram of an array.



When you assign the address of the array to p, the base address if the array part_number is immediately assigned to the pointer p. now if you perform the addition operation p+=14, the pointer will now point to the 14th location in the array. Here, you need to take care not to go beyond the range of the array, which will result in your computer exploding, your house catching fire and your cat disowning you! Of course, it wouldn't actually happen that way, but what you can be guaranteed of is that your program will crash. Back to our array, the usual pointer arithmetic rules apply. The computer needs to multiply the increment by the size of the data type in question in order to calculate the address.

There are also a few rules regarding the precedence of operators. This table neatly summarises these rules.

Syntax	Operation	Example

P++
*p++
*(p++)
	Post-Increment Pointer	New = *(p++);
		is the same as:
		new = *p;
		p = p + 1;

(*p)++	Post-Increment data pointed to by Pointer	New = (*p)++;
		is the same as:
		new = *p;
		*p = *p + 1;

If you remember the precedence table, the ++ operator and the * operator both have the same precedence. This is a problem. But wait, this is where we use associativity. These operators have right to left associativity, which means that effectively, the ++ operator has precedence over the * operator. In plain English, this means that the increment applies to the pointer and not the data pointed to by the pointer.

There are also similar rules to pre-increment operators. Here's another small, neat table that summarises these rules.

Syntax	**Operation**	**Example**

++p
++*p
*(++p)
	Pre-Increment Pointer	New = *(++p);
		is the same as:
		p = p + 1;
		new = *p;

++(*p)	Pre-Increment data pointed to by Pointer	New = ++(*p);
		is the same as:
		*p = *p + 1;
		new = *p;

The same precedence also causes a problem here, and we use right to left associativity, which means that the value that the pointer points to is incremented.

In short, this syntax is used to modify the pointer:

```
Pre-inc: *(++p) or *++p or ++p
```

```
Post-inc: *(p++) or *p++ or p++
```

and this syntax is used to modify the value that the pointer points to

```
++(*p) and (*p)++
```

Pointer to pointer

Up to this point, we have been referring to pointers as holding the address of a data entity. But what would happen if the pointer points to another pointer? This type of pointer is known as a double pointer.

The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer.

If, for example, we want to create a double pointer pointing to a variable a which contains the value 5 the syntax would look like this:

```
Int a = 5;

Int *ptr = &a; //ptr references a

Int **dptr = &ptr; //dptr references ptr
```

Note here that the double pointer has a double asterisk, which immediately tells it apart from a usual pointer. If we are to print the value of the double pointer, the value we get is the address of the pointer pointing to a.

Pointers to pointers are typically used for creating 2 dimensional arrays and also as function arguments if you want to modify the pointer itself rather than the data. Also, when writing code, you might want to interface with APIs from other programs. Some APIs have output parameters that return to a pointer. Sometimes the code may not be completely compatible with your program, so using a pointer to a pointer would be necessary to ensure maximum compatibility.

Linked lists

Linked lists are the simplest dynamic data structure that use pointers.

Another popular use of pointers is the creation of linked lists. We would have done ourselves a serious injustice if we talked about pointers and didn't include linked lists. They are particularly important because they are the simplest dynamic data structure that use pointers in their implementation. This concept sort of forces you to understand how pointers work since it is solely based on pointers. This serves as a solid foundation for more complex data structures, that you will encounter in either C or other programming languages in your journey as a computer scientist.

Linked lists are very similar to arrays in that they are both linear data structures. However, linked lists are not necessarily contiguous in memory, rather, data elements are connected together using pointers.

Linked lists have dynamic size which means they can grow and shrink as required. It is easy to insert additional data elements in a linked list. Unfortunately, it's not all roses and champagne.

Linked lists do not allow random element access. This means that elements have to be accessed sequentially starting with the 1st.

Their nature is not suitable for caching - since the elements are not stored in contiguous locations; there's a chance that they may be far away from each other, and this causes problems for the computer's caching system.

Also, since it deals with addresses, extra memory space for the pointer is required for each element in the list. This makes link lists a bit less efficient than arrays, which is not really a problem on modern computers.

A linked list is represented by a pointer that points to the first node in the list. A node is made up of two parts, first the data element then a pointer that points to the next node.

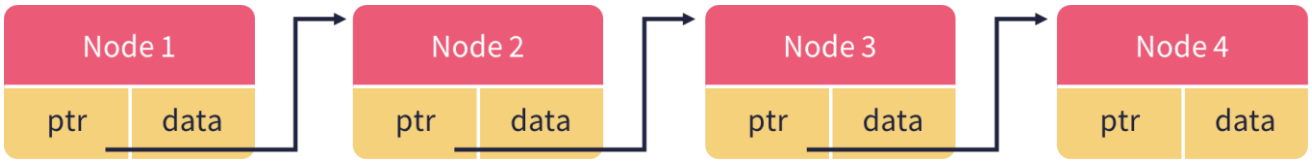
To define a linked list in C we use a structure. If you remember from our lesson on data types, a structure is a data type that is defined using the keyword struct.

Structures allow you to represent several data items of different types.

Just to jog your memory let's look at the basic syntax of a structure again…

```
Struct [structure name {
   Member;
   Member;
   ...
   Member;
} [structure variables];
```

This diagram is a visualisation of how a linked list looks in memory. The pointer in each node is connected to the pointer in the next node up until we get to the end of the linked list. In our next module, we will learn about memory allocation and deallocation and we will implement a few linked lists using those concepts.



Things to note when using pointers

When using pointers, you have quite a lot of control over memory and how it is utilised. So, it is pretty easy to crash a program. Pointers need to be used with care, just as you would use all other low-level tools in C. Let’s look at some of the ways in which pointers can become dangerous accidents waiting to happen.

Sometimes a pointer can become a dangling pointer. This happens when the memory location to which the pointer is pointing has been deleted or freed. There are two different ways in which a pointer can become a dangling pointer. If memory is deallocated, and the pointer in question happens to be pointing at the memory that was deallocated, it means the pointer can no longer be dereferenced.

If you attempt to reference any static variable within a function the pointer will also become a dangling pointer. Another way in which a pointer can become a dangling pointer is when a variable goes out of scope.

In all the above cases attempting to resolve the memory location will produce a garbage value.

Another point to note is that void pointers that we talked about earlier cannot be dereferenced. This is because a void pointer does not have a specific type. Pointer arithmetic is also not possible on void pointers for the same reason. It only makes sense because you cannot perform at automatic on an object that doesn't have a specific type.

It's the action part of the lesson! Today's demo is going to be on pointers. Let's jump right in!

We have a fun challenge for you today. Remember what we did using for loop arrays?

This time we want to do the same thing using pointers. So your task is to fire up your IDE and try to create a programme that traverses an array using only pointers. So here's your chance to go and play with pointers.

Conclusion

On that note we have come to the end of our fifth lesson on pointers. Our adventure doesn't end here, as in our next lesson, we will be looking more closely at structures, and another important component of programming and computer science, that is typecasting. This will give us even more superpowers as we learn how to change the data type of expressions in our codes! Remember to keep programming as practice makes perfect! See you next time!

References

Cprogramming.com. (2019). Linked Lists in C - Cprogramming.com. [online] Available at: https://www.cprogramming.com/tutorial/c/lesson15.html

Dinesh Thakur (2013). What is Pointers in C - Computer Notes. [online] Computer Notes. Available at: https://ecomputernotes.com/what-is-c/function-a-pointer/what-is-pointers

Duramecho.com. (2013). Why C has Pointers. [online] Available at: http://duramecho.com/computerinformation/whycpointers.html

K-state.edu. (2021). 6.1. Basics of pointers — Applications in C for Engineering Technology. [online] Available at: http://faculty.salina.k-state.edu/tim/CMST302/study_guide/topic5/pointers.html

Learn-c.org. (2021). Linked lists - Learn C - Free Interactive C Tutorial. [online] Available at: https://www.learn-c.org/en/Linked_lists

Oregonstate.edu. (2021). Chapter 10: Why are pointers useful? [online] Available at: http://web.engr.oregonstate.edu/~rookert/cs162/ecampus-video/CS161/template/chapter_10/pointer.html

Puneet Sapra (2018). No more fear of Pointers - The Mighty Programmer - Medium. [online] Medium. Available at: https://medium.com/the-mighty-programmer/variable-and-pointer-fb637566bfd9

Weber.edu. (2021). 4.10 Uses For Pointers. [online] Available at: https://icarus.cs.weber.edu/~dab/cs1410/textbook/4.Pointers/uses.html