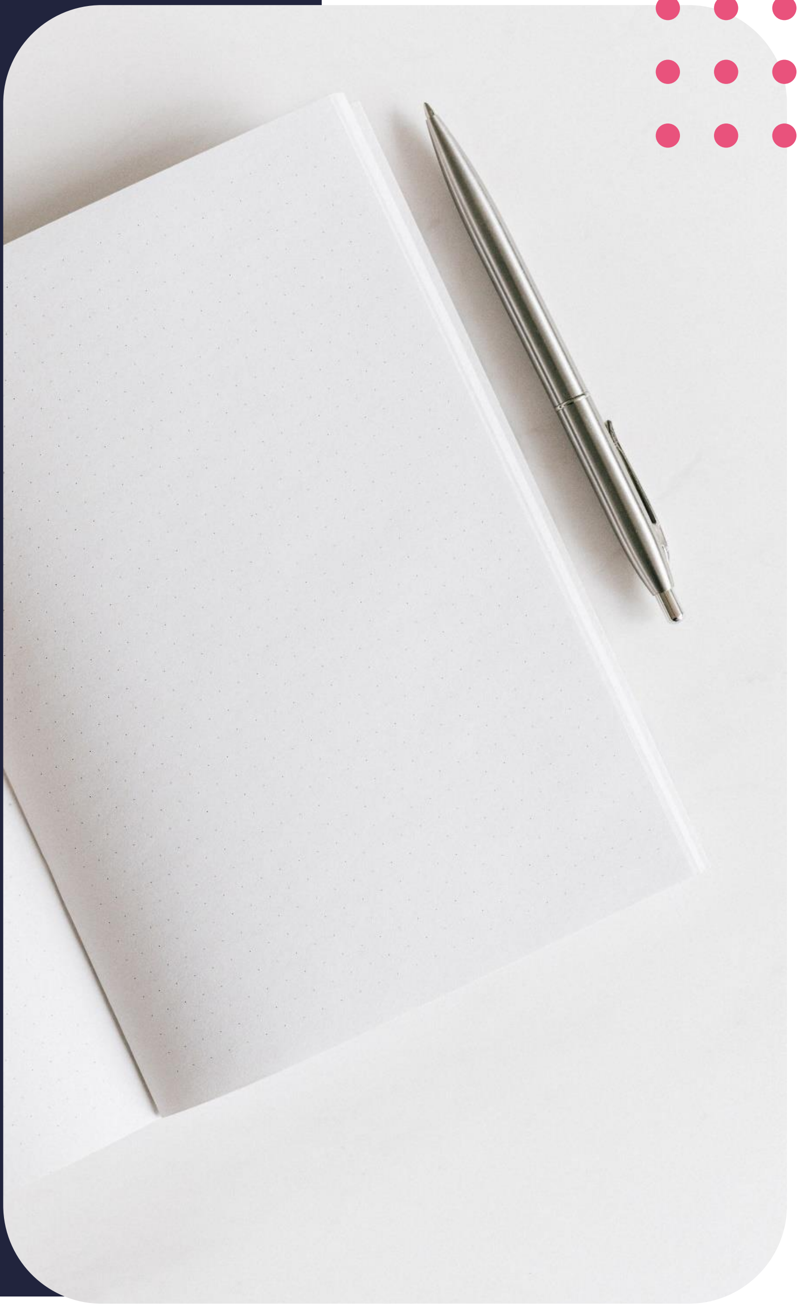
Diploma in Computer Science

Memory management

Contents



Lesson outcomes

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

- Appreciate the relationship between memory management and pointers
- Understand how the operating system handles memory and how this relates to processes
- Explore the built-in functions that allow allocation in the allocation of memory
- Write programs using memory management techniques

Memory

We have been talking about memory from module one but what exactly is it? What is it that is contained in those chips that are sitting on top of your motherboard? Are they just mystery black squares that work magic and make your PC work? Do they contain billions of tiny brains that your PC users to store information? To answer all these questions that are buzzing in our heads, let's take a step back and go right back to module 1 to examine exactly what memory is. It might seem like it's way, way back and totally unnecessary, but it really gives you a very good picture of what you will be doing in C. You will discover that when you write code after looking at the physical aspect of memory, it'll make so much more sense.

A step back

You recall from module 1 that's memory is the area in a Von Newman computer which stores data. Data is all the unprocessed characters in the form of binary numbers that the computer works on and produces information. There are also different types of memory, and each one serves its own somewhat unique purpose. Memory is made up of transistors and capacitors that represent either a 1 or a 0 using charge. The exact way in which that is done is a whole topic for another day. Memory in a PC is categorised in a hierarchy, which is roughly according to how far from the CPU that type of memory is and how large or small the said memory is. CPU registers are the closest to the CPU and are very small but extremely fast. You also recall that the closer we get to the CPU; the smaller memory gets and the more expensive it becomes. Next up we have cache. This is still fairly close to the CPU but not as fast as registers. This is used to store data that the CPU is likely going to need in the foreseeable future. It's worth noting that “foreseeable future” in terms of CPU time it's just a few seconds or less! Cache is also quite expensive and even in modern computers, cache rarely ever goes beyond a few megabytes in size! The largest and slowest of all the various types of working memory is RAM. The computer can and actually sometimes does work from main memory. It is not as terribly slow as data cached on the hard drive but does introduce some performance impact on the program.

As we mentioned earlier, speed, size, cost, and position relative to the CPU are all important. Speed is often measured by the amount of latency the type of memory has. Latency is the time taken for a component to respond to a request. In terms of memory, this means the time taken for a read or write operation to be completed. The greater the latency value of the memory, the slower it is.

Now, back to the main story of the day, variables aren't just thrown into memory, they need to be arranged in a way that uses memory efficiently. Think of it this way, what happens in your kitchen cupboard if you just throw things in there without looking? It will always look full and yet there will be astonishing gaps where you can fit a soccer ball, but somehow you won't be able to fit a saltshaker or a tiny little spoon! You're also 100%

guaranteed that all your chinaware will be broken by the end of the week! The exact same thing happens with pcs, and the results are equally catastrophic and produce amusing results! Sounds hard to believe? Hang on, in the last section of the lesson, we will look at the technical consequences of poor memory management.

Operating system memory management

When a program starts running, it requests a block of memory from the operating system. This is done by a program known as the memory manager. The memory manager has a subcomponent called the allocator. The process is automatic, that is why you have to declare variables during coding and not at runtime, although using dynamic memory allocation, more memory can be requested when needed at runtime. We will look at this closely in a bit.

The operating system handles movement of data and programs between memory and the hard drive during execution. The memory manager keeps track of all the memory locations in the main memory, and this refers to the entire RAM, whether it is occupied or not. Think of what a warehouse manager would do. The manager has a list of everything in the warehouse, along with its specific location. The manager knows whether there is enough space to receive a certain amount of stock or not. We could liken the warehouse itself to RAM. The warehouse can only hold a certain amount of stock, and it should be organised in a way that uses space as efficiently as possible. Back to computers, the memory manager decides which process gets memory, how much and at what time. It also checks how much memory has been deallocated and reclaims it for reallocation to other processes that need it.

C program structure

When a process (a fancy word for a program) is first launched, it gets a place in the process address space, which I'm sure you still recall from module 1. When you are writing your code, the variable names, constants, and instruction labels that you use ultimately all become memory addresses. The compilation process will convert these to a computable address, which the computer will use during runtime. The computer neither cares about nor does it have any use for all the fancy variable names that we use in our code. All it gets, from, say, int age=0; is that a memory location that is 4 bytes (well, at least on my PC) needs to be reserved and the value zero needs to be stored in it. The variables that you create are stored in what is called a stack. Stack variables are freed when they go out of scope. A stack is managed automatically, but it is not without its own drawbacks. Firstly, the compiler needs to know how big the variables are, and this should be known in advance. Secondly, space in the stack is somewhat limited. Typically, at compile time it is hard to tell how large the variable is going to be. This is where a structure called a heap is employed. A heap has only one problem. The compiler doesn't know when you are done with it and no longer need it. This means that if it is not handled correctly, all that memory is going to be wasted, because it will be "occupied" (Nina: pronounce the quotes) but not used. This means that the program will consume much more memory than it needs to. This is referred in computer science as a memory leak. In this situation, the computer cannot recover that memory, and it cannot be used for anything else until your program terminates. And wait there's more bad news! There's an even worse situation where the heap is released before you are actually done with it. You've already guessed that it creates a whole lot of problems that had not easy to solve at runtime.

A typical C program contains five sections, namely the

1. Text segment

2. Initialized data segment

3. Uninitialized data segment

4. Stack

5. Heap

Notice the numbering, as it tells us how high in the address space a particular segment is. Let’s look at each of them in detail.

The text segment is where all the executable code that you write goes. The text segment is sharable, and this means that only a single copy is kept in memory for frequently executed programs, such as text editors, the C compiler, or the shell. The text segment is typically read-only, and this is done to prevent a program from accidentally modifying its own instructions and destroying itself.

The initialised data segment is a is a portion of virtual address space of a program, which contains the global variables and static variables that you initialised while you were writing your code. In this section, variables are not read only, since you technically should be able to alter them at runtime, otherwise they wouldn’t be variables.

Next, the uninitialised data segment contains the variables that you left unitialised. This segment starts where the initialised data segment ends. These variables typically don't contain any value and will typically have garbage values if you try to display them or use them.

Next up the data hierarchy is the stack. This is where automatic variables are stored together with data that is saved each time a function is called. The stack is the last in first out structure and as we saw in the memory hierarchy it is located high up in the memory structure. Notice that we said last in first out. This is commonly abbreviated as LIFO. This means that the last element to be added to the structure is the first to be taken out. Each time a function is called, the return address is saved on the stack, along with information on the calling function’s environment. This new function then creates room for its automatic and temporary variables on the stack. Recursive functions demonstrate the operation of the stack very well. In fact, this is how recursive functions work. Each time recursive function calls itself; a new stack is used so that variables from one function to not interfere with variables from another.

All the variables that are stored in a stack can only be accessed locally. This means that any variables that are declared globally will not be stored in a stack. Variables allocated on the stack are stored directly to the memory and access to this memory is very fast. This is partly due to the fact that allocation of addresses is done during compilation. Memory management in a stack is a pretty simple exercise because the element that is removed from memory is always the last element in the stack. Freeing a block of memory is as simple as adjusting a pointer.

The heap section begins where the stack section ends. This is the section that we are most interested in because it is managed by the built in C functions that we want to look at today. The memory spaces allocated in the heap can either be contiguous or non-contiguous. This is also an interesting memory space because it is allocated dynamically. This means that whatever is in the heap can be shrunk or expanded as required. This is where we can declare the dynamic areas that we were talking about before. One key difference between a heap and a stack is that a stack is a linear structure whereas a heap is a hierarchical data structure. Variables allocated on the heap have their memory allocated at run time and accessing this memory is a tad bit slower, but the heap size is much larger and is only really limited by the size of virtual memory. This makes it ideal if you are not sure of the size of data you will be working with at runtime.

Thankfully, in C, you are responsible for cleaning up your variables. Some other programming languages do this automatically using a process called garbage collection. While it is a nice thing to have, it has a fair share of its own drawbacks that cause system overheads and take a toll on system performance. As we mentioned at the beginning of module 2, C provides tools that you can use to allocate and reallocate memory as required. These are quite powerful tools and help you work as reasonably close to hardware as possible. This has the direct result of creating small, efficient, fast programs with a very small system footprint.

Memory management in C

We have already mentioned that she gives us the ability to manage memory. This is done using special functions that are built into the C standard library. This way of handling memory is more commonly known as dynamic memory allocation. It works hand in hand with arrays, as these data structures are commonly used where you do not know the size of the data at runtime. Dynamic memory allocation allows us to give the program more memory during runtime as and when needed. This allows us to overcome the two biggest limitations of arrays that we have seen so far, that is, the size of the array must be known while you are still writing your code and the size of the array is pretty much set in stone after you finish writing a code. Also, setting a definitive number of elements in an array is also a disadvantage because if you only use two slots after declaring an array of 10 elements then that means the rest of the space is just being wasted. Let's now look at how memory allocation is done in C.

Memory management functions

As we mentioned before, C provides ways in which you can manually allocate and deallocate memory. These functions are defined in both this standard library header file and in the malloc header file depending on which operating system you are using. If you're using Windows, then you are likely going to be using the standard library header file. It is worth noting here, though, that malloc.h contains only the definitions for the memory allocation functions and not the rest of the other functions. These still exist in the standard library header file. There are four functions provided in C that are used for dynamic memory allocation. These are malloc(), calloc(), free(), and realloc(). Like most other things in C, the names are pretty much self-explanatory. Malloc() is used for memory allocation, while calloc is used for contiguous allocation of memory. Free() is used for freeing memory, while realloc() is used for re allocating memory. Simple as that!

The malloc function

Memory allocation is done using a function called malloc(). Which is a built-in function found in the C standard library. This function can dynamically allocate memory during runtime. The malloc() function allocates the requested memory and returns a pointer to it. The general syntax of the malloc function code looks like this:

```
Void *malloc(size_t size)
```

The malloc function takes the size of the memory block in bytes as a parameter and as we said previously, it will return a pointer to the memory location if the operation is successful. If the operation is not successful, then it will return NULL. In today's demo we are going to look at an example that uses this function. The malloc function is thread-safe: it behaves as though only accessing the memory locations visible through its argument, and not any static storage.

The calloc function allocates the requested memory and returns a pointer to it. You may be wondering what the difference is between the calloc function and the malloc function. Well, the malloc function does not set the memory to 0 where is the calloc memory sets the allocated memory to 0. Setting the memory location to 0 is essentially an initialization, just like what you would do if you were allocating a value to a variable on declaration. The basic syntax of the calloc function looks like this

```
Void* calloc( size_t num, size_t size );
```

Just like the malloc function the calloc function takes size as a parameter and returns a pointer to the memory location if the operation is successful. If the operation is not successful, just like the malloc function, it will return null. Similar to the malloc function,

Calloc is thread-safe: it behaves as though only accessing the memory locations visible through its argument, and not any static storage. It's pretty easy to see that the only difference between the calloc function and the malloc function is that the malloc function does not initialise the memory location while the calloc function will initialise the memory location.

The free function is probably the most straight forward one. It is used to deallocate memory which was previously allocated by a calloc, malloc or realloc. The basic syntax for the free function looks like this:

```
Void free(void *ptr)
```

The parameter of a free function call is a pointer, which points to the memory block that was previously allocated my anyone of the three memory allocation functions. If another pointer is passed as an argument, then nothing will happen. Unlike all the other functions, this function does not return any value. An attempt to free a memory location that has already been deallocated results an undefined behaviour. It also causes undefined behaviour if the memory area referred to by ptr has already been deallocated, that is, free() or realloc() has already been called with ptr as the argument and no calls to malloc(), calloc() or realloc() resulted in a pointer equal to the pointer in question afterwards.

The last function that we are going to look at is the realloc. Just like the others, the name suggests that the function reallocates if given area of memory. There is a catch though; the block of memory must have previously been allocated by the malloc calloc or realloc function and not yet freed using the free function. If you just go ahead and use it anyway, it results in undefined behaviour. We have seen in earlier lessons that undefined behaviour is not a good thing to have in your program! Reallocation of memory can be done in two ways. The first way is expanding or contracting the existing area pointed to by a pointer if possible. The contents of the area remain unchanged up to the lesser of the new and old sizes. If the area is expanded, the contents of the new part of the array are undefined.

The second method is by allocating a new memory block of the required new size, copying memory area with size equal the lesser of the new and the old sizes, and freeing the old block.

If there isn't enough memory, then the old block of memory is not freed and a null pointer is returned. If you attempt to use the reallocate function on an all pointer then the behaviour is exactly the same as what you would get if you call the malloc function with a new size parameter. The basic syntax of the realloc function looks like this:

```
Void *realloc(void *ptr, size_t size)
```

The function takes 2 parameters. The first parameter is the pointer to a block memory that was previously allocated by the any one of the three memory allocating functions. The second parameter is the new size of the memory block in bytes. If it is zero and the points are points to an existing block of memory, then the block of memory that is pointed to by the pointer is deallocated and null pointer is returned.

Fragmentation

This problem is kind of unfair to programmers because it can still happen even if they did everything right. External fragmentation happens when allocated memory within other allocated memory is deallocated. Only something of that exact size or smaller will be able to fit into the newly deallocated memory region. This can cause a program to run out of memory even if there's lots available. An example is shown on the right.

It's up to allocator to be able to avoid this kind of problem. Sometimes, custom memory managers are used to avoid problems like this, though this problem is really just out of your hands.

Problems in memory

It's not all roses and champagne when it comes to memory management. We have mentioned it countless times that if you do not do things correctly, then expect a spectacular mess when you execute your code. The exact same mess that you will get if you do not put your belongings in a cupboard in an orderly manner is exactly the same results that you would get if you do not use memory allocation and the allocation functions correctly. C provides granular control to the dynamic memory section of the program, which leaves it entirely up to you, yes, you, the programmer, to do it correctly. Having this much control in a program also means that C will happily sit and watch while you douse yourself in petrol then go on to light a gas stove! Heck, your program will even compile successfully with those suicidal errors! Let's look at some of the pitfalls that you can encounter in memory management.

Dangling pointers

When you create a pointer, you must always keep track of it. If you create a pointer then proceed to deallocate the memory that it points to, you are left with what is a called a dangling pointer, which is a pointer that points to nothing. Not only will dangling pointers most certainly make your program crash, but they are also used in security attacks in software. Dangling pointers have been used quite a lot of times in zero-day attacks. Another not so dangerous type of error is a wild pointer. Wild pointers are created by not initialising the properly on 1st use. This means that we have absolutely no idea what the pointer is pointing to, which poses problems for any routine that attempts to access the pointer. The simplest technique to avoid dangling pointers is to implement an alternative version of the free function which guarantees reset of the pointer.

Another pitfall is creating garbage with no way of collecting it. Imagine what would happen if you never emptied your trash can. Things will keep piling up and pretty soon, you will not have space for any other new things. The same thing happens if you keep allocating memory and provide no way of reclaiming it when you're no longer using it. Keep in mind that the heap section of the programme is managed by you the programmer and the program has no way of knowing if you no longer need a portion of memory unless you tell it, your program will keep growing and growing and you will not have any idea what is going on. It is not really problem on systems that have a lot of RAM, but if you're working on a constrained system such as an embedded system, sooner or later the entire thing will just crash.

This one is pretty obvious, and it seems like a no brainer. You should not attempt to free a memory area that was never allocated. This results in undefined behaviour. It is particularly difficult to find and you will spend hours and hours wondering why your program is behaving mysteriously. To solve this problem, it is always a good practice to initialise pointers when you declare them. You can use the malloc function and set the pointer to null.

Demo

We have come to our favourite part of the lesson in which we are going to use the four functions that will learn about today in a demo. We are going to create an array that uses the malfunction and can grow on demand. I'm sure it's dizzying just to look at code snippets and a bunch of rules that govern them, so an actual programme that does stuff would definitely help. Let’s jump right in!

Conclusion

In this lesson, we learned about all the fabulous ways in which you can control dynamic memory in C. We also looked at the dangers of not doing this correctly, and possible ways of avoiding this. This brings us to the end of our third module and we are preparing to go into our 4th and final module for this course. There are still so many exciting things that are awaiting us, and we also want to write our own fully functional program. Our first lesson in the fourth module is entirely dedicated to that, and we will be looking at planning a full-fledged programme that does a handful of tasks. Exciting times ahead! That's it for me today and remember to keep coding. See you next time!

References

BBC Bitesize. (2021). Memory - CPU and memory - GCSE Computer Science Revision - BBC Bitesize. [online] Available at: https://www.bbc.co.uk/bitesize/guides/zmb9mp3/revision/6

Kjell, B. (2021). Symbolic Address. [online] Programmedlessons.org. Available at: http://www.programmedlessons.org/assemblytutorial/Chapter-15/ass15_14.html

Net-informations.com. (2020). Differences between Stack and Heap. [online] Available at: http://net-informations.com/faq/net/stack-heap.html

Oxford Reference. (2021). Symbolic addressing. [online] Available at: https://www.oxfordreference.com/view/10.1093/oi/authority.20110803100546743

Programiz.com. (2021). C Dynamic Memory Allocation Using malloc(), calloc(), free() & realloc(). [online] Available at: https://www.programiz.com/c-programming/c-dynamic-memory-allocation

Techtarget Contributor (2017). Memory management. [online] whatis.com. Available at: https://whatis.techtarget.com/definition/memory-management

Uah.edu. (2021). CS 221. [online] Available at: http://www.cs.uah.edu/~rcoleman/Common/C_Reference/memoryalloc.html