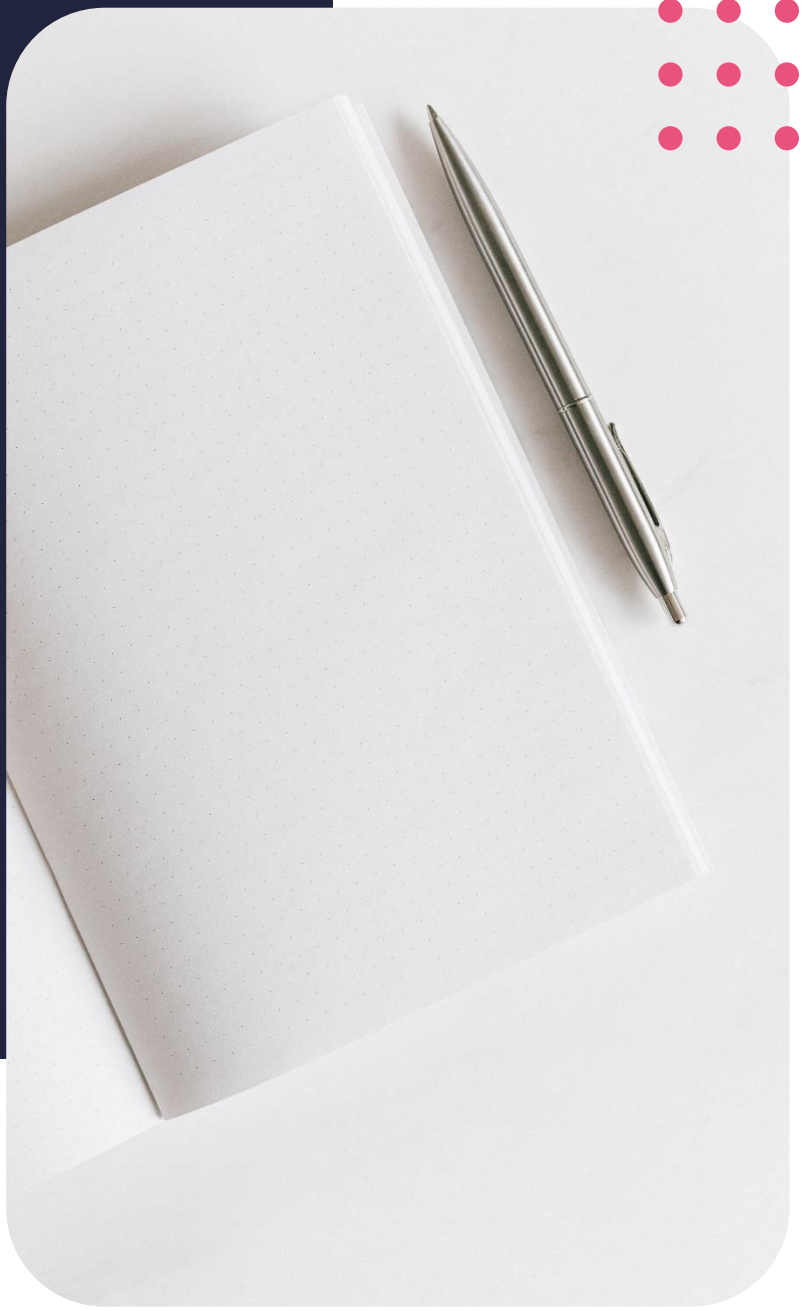
Diploma in Computer Science

Variable arguments

Contents



Lesson outcomes

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

- Recall the role of arguments in functions
- Understand the concept of variable arguments
- Explain the role of arguments in a program
- Use the functions contained in the stdarg header file

Arguments

In our previous lessons, we discussed how functions can work together to breakdown a large problem and two small bits and pieces and solve it from the ground up. But this wouldn't be of much use if the functions cannot communicate. This is where parameters and arguments come in., they allow functions to pass data to each other so is to be able to modify the same copy of data, and thereby solving the problem. You recall that in C, there are two types of function arguments, namely actual arguments and formal arguments. We are going to visit this just now.

A step back

To fully understand what we are trying to get at here, let's take a step back and look at function arguments in more detail. As we just mentioned, function arguments basically take two forms, that is, actual arguments and formal arguments. This has to do with how the arguments are passed. The variables declared in the function prototype or definition are known as formal arguments. These are demonstrated in this code snippet.

```
Int calculation(int x, int y);
```

In this code snippet, X and Y are the parameters. Although formal parameters are always variables, it doesn't really mean that they are always variable parameters, it's not strictly set that they're variables. You can use numbers, expressions, or even function calls as actual parameters.

The other type of arguments is called actual parameters. Actual parameters are not in the form of declarations as are their formal counterparts.

Passing arguments to functions

Just now we were looking at the methods that are used to pass arguments to functions, but why do we need to do that in the first place? Well, a function is just a bunch of instructions. These instructions aren't really useful if they do not have data to work with. This data can be passed to function using the methods that we outlined above, or the variables can be declared in the function itself and used within the function.

Ways of passing arguments

A function can receive arguments in two ways. The first way in which this can be done is receiving the argument as a value, which means that a copy of whatever the function is going to be working on is passed on to the function. The other way in which it is done is by passing the address of the variable that the function is going to be working on. This means that the function would directly access whatever is contained in that address, and also directly modify it. This is in contrast to passing by value, where the computer only works on its copy of the variable and does not affect the actual variable that is stored in the address. This has varying results on the final state of the variable after the function is completed execution.

Variadic functions

Whenever you're writing a C program, or any program for that matter, whenever your function requires data from somewhere else, you typically provide this data in the form of arguments. This is a fairly straight forward process for most of the time, but have you ever stopped to think what would happen if you do not know the number of arguments that your function will take at runtime? In all the programs that we have written so far, once we write the code and compile it, it is pretty much set in stone and we can only edit it by terminating the running program and going back to the source code to modify it, adding whatever it is that we want to add. This is a problem, mainly because in the real world, where typically don't really know how much data the program will encounter at runtime. Also, you won't always be around to alter the source code when the user wants to use the program. Luckily, C has a built-in mechanism that will allow us to send a variable number of arguments to a function. Let's take a look at this more closely.

Why do we need variadic functions?

As with quite a lot of things in computing and programming, we do not know the number of things that we are working with. These are only known at runtime and this Pause is a problem for the computer. This is because we normally assign values to variables and the number of variables that we need during programming and these are set in stone during compilation. It is not normally a problem when you set the upper limit whenever an idea of how many elements you will be working with.

When you define an ordinary function, and at a specific number of arguments to it, it forces you to supply that specific number of arguments whenever you call that function. This works very well for functions that have a fixed number of arguments, such as our quadratic equation solver from one of our demos. In that example, if we had to create a function, we know it will always take the three arguments that represents the values of A, B and C. Pretty simple, right.

Contrary to what you may think, there are actually functions that perform meaningful routines and expected variable number of arguments. Some of these functions can handle any number of values by operating on all of them as a block. In our last lesson, we learnt about the malloc function. This function brings about so many possibilities, but also brings problems. You can allocate memory dynamically during runtime, but how will you access all the additional memory slots that are created? Variadic functions solve this problem by allowing you to supply the function with a variable number of arguments. This means that, for example, you can have one function that creates any number of array elements, and the function that works on that array becomes a very attic function. In this way, you can be sure that if free memory is not that is allocated by the malloc function has a corresponding argument to it. Problem solved!

A much easier and more relatable example is that of the print function. You have seen in all the lessons that we have done so far, touch the printf function can handle just about any amount of information that you throw at it. This is an example of variable arguments. Anything that you put into the printf function in between the parentheses is an argument. It gets passed to the printf function, that then works whatever wizardry it does so that the information is displayed on the screen. You can have anything between just one character to a long line of text and output from variables without running into any errors! Let's take for example these code snippets that have variable numbers of arguments in them.

```
Int a=0, b=1, c=2;

Printf("%d , %d ,%d", a,b,c);

Printf("%d ,%d", a,c);
```

In this code snippet, the printf function takes 3 arguments on the 1st instance. In the second instance, the printer function takes 2 arguments. Both of them are perfectly correct and will not produce any compilation or run time error.

The scanf function also works in pretty much the same way. Remember when we read a number of different inputs from the keyboard and stored them in different variables, it was exactly the same concept that we just applied here. Valuable arguments offer you greater flexibility what regards to what you can do with your function.

Syntax

The syntax of a variadic function is almost the same as what we could call an "ordinary" function. The only difference here is that its formal parameter list ends with an ellipsis. Although it is pretty much straight forward, here is a code snippet that highlights exactly what we just said

```
Int functionname (int firstargument, ...);
```

At least one named formal parameter must appear before the ellipsis parameter. To get a clearer picture of what is going on here, let's go back to the function that we just talked about, the printf function. Should dig deep into the standard library header file, you will come across this code:

```
Int __printf ( const char *format, ... )
{
va_list arg;
int done;
va_start (arg, format);
done = vfprintf (stdout, format, arg);
va_end (arg);
return done;
}
```

Surprise! Surprise! The printf function is not the one that does all the magic that we see on the screen! The printf function just passes everything on to the vfprintf function and all the magic and spells happens there! That is not exactly our point of interest, so we are not going to dig much deeper than that. In both functions, which refers to the printf function and the vfprintf functions,three macros and one special data type are used to implement the variadic functions, namely va_start, va_arg, va_end and va_list.

Let's now look at the skeletal version of the function that handles these parameters.

```
 #define __va_argsiz(t) (((sizeof(t) + sizeof(int) - 1) / sizeof(int)) * sizeof(int))

// several lines are skipped

#if defined __GNUC__ && __GNUC__ >= 3

Typedef __builtin_va_list va_list;

#else

Typedef char* va_list;

#endif

// several lines are skipped

#ifdef __GNUC__

#define va_start ( ap, pn ) ((ap) = ((va_list) __builtin_next_arg(pn)))

#else

#define va_start ( ap, pn ) ((ap) = ((va_list) (&pn) + __va_argsiz(pn)))

#endif

// several lines are skipped

#define va_arg(ap, t) (((ap) = (ap) + __va_argsiz(t)), *((t*) (void*) ((ap) - __va_argsiz(t))))

// several lines are skipped

#define va_end(ap) ((void)0)
```

That looks pretty cryptic, but the good news is it's only just there so that the rest of the explanation makes sense. Let's not take a closer look at the data types and macros.

First up, we have typedef va_list. This is a list of variable arguments, a char* variable which will be initialised by the address of the first variable argument. It’s an opaque variable that holds information about

Which argument we’re going to get next with va_arg(). The computer just keeps calling the function va_arg() repeatedly. The va_list variable is like a monitor, that tracks where we’re at in progressing with the list. In spite of that, we still need to initialise that variable to a value that makes sense. Now that’s where va_start() comes into the picture. When the computer calls called va_start(va, count), it was initialising the variable va to point to the variable argument that is immediately after count.” Interesting! It’s becoming even more clearer why we need to have at least one named variable in our argument list.

With that pointer pointing to the initial parameter, the computer can easily get all the following argument values by calling va_arg() repeatedly. When you want to do the same, you have to pass your va_list variable together with the type of argument you're about to copy.

Next, we have void va_start(va_list ap, pn);. This is a macro, which takes a va_list variable, ap, and the last-named parameter, pn, as its input, and initialises ap by adding the size of pn to its address. After calling va_start, the pointer ap will point to the address of the first unnamed parameter.

After this initialisation process, we then have a macro retrieving the value of the next unnamed parameter. It takes the va_list variable ap and the type of the next unnamed parameter T as input and returns the value of the next unnamed parameter. The assumption here is that an actual argument is always passed. This means that if the va_arg()function is invoked, and there are no more arguments, it results in undefined behaviour.

Stdarg

This name probably looks familiar to you. And yes, it's not just some hullaballoo from a dream that you had last night. It's the standard argument header file. We've covered this already in previous lessons, but it is of particular interest today because it enables us to use the functionality battery have been discussing earlier. Its contents are the key drivers of variadic functions as we are going to see shortly. Quite a lot of what we have just discussed in the previous section is actually the contents of this header file. Interesting, isn't it? Well let's take a closer look.

What is this now?

The primary focus of this lesson was to show us how to write programs that accept an ina null pointer is definite number of arguments. This is exactly what the standard argument header file does. We took a peek at it in one of our previous lessons, but we kind of just glided on top and didn't really get much into it.

Typically, the size of the unnamed argument list is generally an unknown value. Because of this, the calling conventions that are used by most compilers do not allow you to determine the size of the unnamed argument block pointed at by va_list inside the receiving function. There is also no reliable, universally functional and fool proof way to pass these unnamed arguments into another variadic function. Even if your device some genius way of determining the size of the argument list indirectly, like parsing the format string of fprintf()),it will just hurt the portability of your code because when you pass the dynamically determined number of arguments into the inner variadic call, as the number and size of arguments passed into that call will still need to be known at compile time. You could come to some extent, get away with this restriction by using variadic macros instead of variadic functions. In addition to that, most standard libraries provide an alternative way to access the unnamed argument list (i.e. An initialised va_list variable) instead of the unnamed argument list An example of this is vfprintf() which is an alternative to the fprintf() function. It expects a va_list instead of the actual unnamed argument list. A user-defined variadic function can, following this logic, initialise a va_list variable using va_start, then proceed to pass it on to the relevant standard library function, which has the indirect effect of passing the unnamed argument list by reference instead of passing it by value. The library function then has direct access to the list. Because there is no reliable way to pass unnamed argument lists by value in C, providing variadic API functions without also providing equivalent functions accepting va_list instead is considered a bad programming practice.

Some other versions of C have built in C extensions that allow the compiler to check for the proper use of format strings and sentinels. If the compiler does not make use of these extensions, as usually is the case, the compiler cannot check whether the unnamed arguments passed are of the type of the function expects or convert them to the required type. It is then left up to you, the programmer, to make sure that everything is correct, because if anything doesn't match, it results in undefined behaviour. If, for example, if the expected type is int *, then a null pointer should be passed as (int *) NULL. Writing just NULL would result in an argument of type either int or void *. Both forms are incorrect and will result in undefined behaviour. You should also take into consideration the typecasting conversions that we discussed in previous lessons. Type promotions and demotions will also result in undefined behaviour. Since we already know how values are promoted and demoted, the function that is receiving the arguments must be programmed in such a way that it expects the new types.

Demo

This lesson is a bit on the deep end, so instead of spending more time talking and looking at theoretical concepts, it is better if we spend most of our time in a demo, looking at how it actually works. Today's demo is on a program that accepts a variable number of arguments and displaying the list after processing the data. I would like to request you to pay special attention to the syntax that we are going to be using in the demo, is this is what you will be using whenever you want to use variable arguments

Our little project.

Now comes the time that most of us have been waiting for. Remember in module 1, we promised that we were going to create a small project in reach we were going to build code as we went along, At this point in our course, we have sufficient knowledge and information to use to develop a small but functional app that actually does some work. Starting from our next lesson, we will develop a flow chart for the project. As we learned from module 1, a computer program starts its life as a problem statement. When you, as a computer scientist, read this problem statement, your head is already bubbling with potential solutions to the problem. But we also learned in module one that you do not just jump into writing code. If you do so, chances are, you end up with a hot mess that either works as intended but is incredibly slow and clunky, or you just simply end up with a hot mess of spaghetti code that doesn't work at all. We also learn that to avoid this, you have to plan the solution using a series of steps. Let's just jog our memories a bit.

When you encounter a problem, the first thing to do is to think of the possible ways in which you can solve the problem. It can help to write down the solutions and look at them to find which one is probably the best one to follow. After writing down these solutions, you then pick out the one that you think is the best one. From then you flesh out the probable solution in all the ways that you can think of right off the bat. This includes any functions that you might think are necessary for the solution to function. Before you even get to functions, it would be helpful to think of the strategy that the program will follow. This makes it easier for the next step, which is writing pseudo code. Of course, this is not set in stone as you can start with the flow chart instead of the pseudo code. My personal preference is starting with pseudo code as you can easily alter it without making too many changes to the entire program, so for now we will go with that.

Next up is the pseudo code. Pseudo code, as you remember from module 1, is fake code. You also remember that's no compiler on earth can compile and run pseudo code. It doesn't conform to the strict syntax and semantics of programming languages. This gives you a great deal of flexibility in strategising how the program will work. We need to flesh out the pseudo code as extensively as possible, and this gives us a rough idea of what the flow chart will look like. And of course, we can always go back to the pseudo code and alter it as needed at any time in the development process.

After we finish with the pseudo code, we will be ready to draw our flow chart. At this point our program will be in suggested that you can follow the logic and see how the solution will work. Like the pseudo code, the flow chart needs to be as detailed as possible. This makes it a lot easier to change the flow chart and the pseudo code to actual code. It is important here to try and stick to what you wrote in your pseudo code so as to keep things consistent. When you're done with that, the next step will be direct the actual code.

In our next lesson, we will look at the software development process and this will give us a very good idea of how we should handle the development process. In case you are one of those who are interested in sneak previews, our small project will be on a system that accepts student details from the user, calculates average Marks and totals according to the school’s specifications, then displays the marks on demand to the user. Our program will also employ a command line menu from which the user can pick which functionality of the program they want to use. As we go along, we will add functionality that we will have learned about and integrate it into the existing code. By the time we get to our 8th lesson, we will have a fully functional program that runs in command line and is able to fulfil all the requirements that are specified by the user. Interesting times ahead!

Conclusion

In this lesson, we learned about one of the subtle but useful features in C programming. We learned that it is not always that we know the number of arguments that we want to pass through a function, and so it might not make sense in those instances to pass a certain, predetermined number of arguments. This is where we employed the concept of variable

arguments. We learned that this allows us to send any variable number of arguments to a function during runtime. We will no longer be bound by the strict rules of see that dictate that the number of arguments must be known at compile time. This allows us to create programs that are much more useful in the real world. In our next lesson, we will amalgamate all the knowledge that we have amassed so far and integrate it into what software development in computer science is in industry. We will look at the software development life cycle, which entails the steps in which a program follows from the time that it is just an idea in a client's head, till the time that you deliver a finished product to the client and beyond. We will also learn that software development doesn't end when you deliver a product to the client, it goes way beyond that, open till the software is no longer being used! That's it for me today, see you in the next one!

References

Beej, B. And Hall, J. (n.d.). Beej's Guide to C Programming. [online] . Available at: https://beej.us/guide/bgc/pdf/bgc_a4_c_1.pdf

Chuan Zhang (2020). Variadic Functions - The Startup - Medium. [online] Medium. Available at: https://medium.com/swlh/variadic-functions-3419c287a0d2

Codescracker.com. (2021). C Variable Arguments. [online] Available at: https://codescracker.com/c/c-variable-arguments.htm

Duke.edu. (2021). The C Book — Variable numbers of arguments. [online] Available at: https://webhome.phy.duke.edu/~rgb/General/c_book/c_book/chapter9/stdarg.html

Gwu.edu. (2021). Variable number of arguments. [online] Available at: https://www2.seas.gwu.edu/~rhyspj/spring06cs143/lec4/lec453.html

Ibm.com. (2018). IBM Knowledge Center. [online] Available at: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.4.0/com.ibm.zos.v2r4.bpxbd00/stdlibh.htm

W3Schools | Tutorialspoint | W3Adda. (2016). C Variable Arguments | W3Schools | Tutorialspoint | W3Adda. [online] Available at: https://www.w3adda.com/c-tutorial/c-variable-arguments