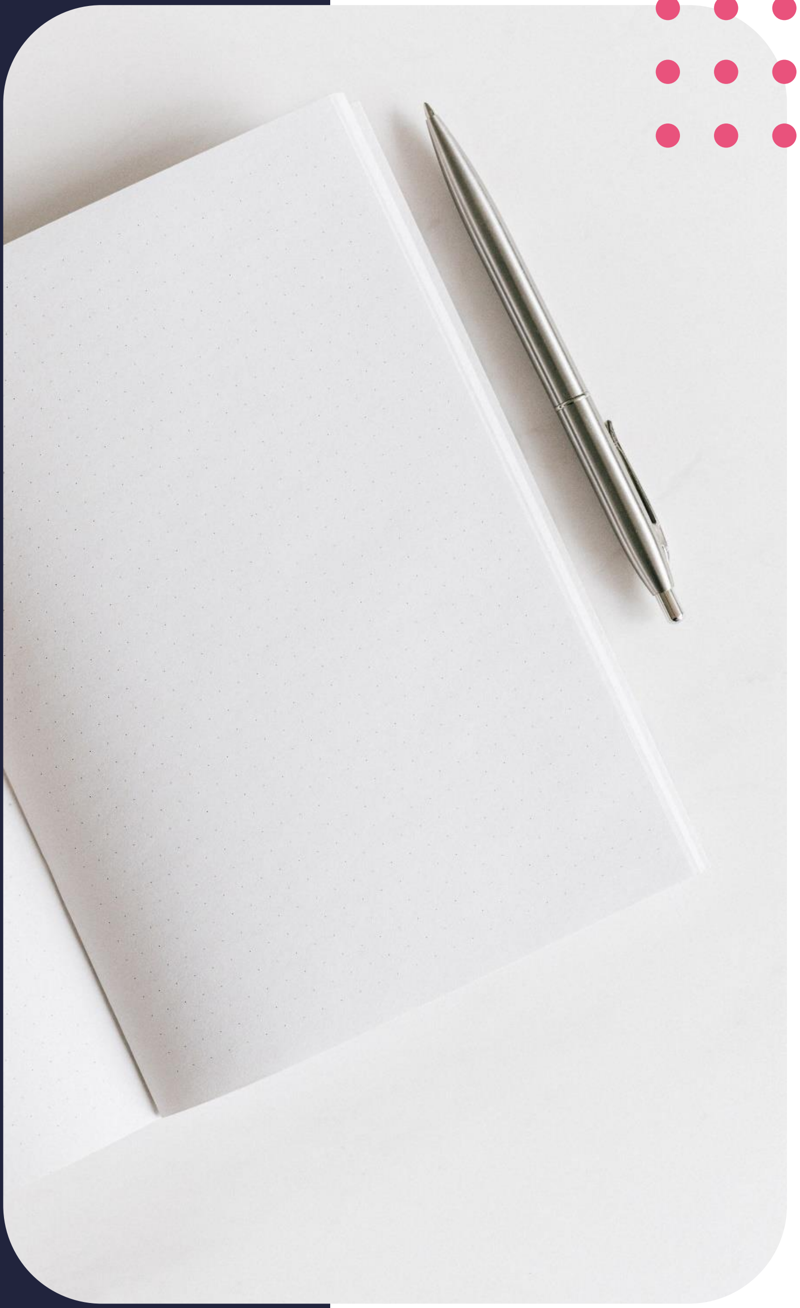
Diploma in Computer Science

Operators

Contents



Lesson outcomes

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

Understand what operator is and what it does

Explore increment and decrement operators in programs

Use logic operators appropriately in code

Discuss the various ways of assigning data to variables using operators

Welcome to the last lesson in module 2. Things escalated really quickly, and we are already writing functional C programs! In this lesson, we are going to wrap up the basics by looking at the tools that we use to move data around, "cook" shape it and produce information. These are known as operators. We will look at how they are used, as well as the rules that you should follow to ensure that your programs always produce accurate results. We are going to be using quite a bit of maths principles here so dust off and sharpen your maths skills and buckle up!

Arithmetic and logic operators

The first thing that probably crossed your mind is "hey, I've seen this before!" Well, you actually have seen this before! This is the exact same arithmetic that you have encountered in maths classes. Arithmetic in C is done in exactly the same way as you would do it in any other field of study, except that instead of always using x, y, a, b and all those other letters as variables, you will be using the variable names that you defined yourself.

Addition and subtraction

This will probably the simplest topic in this lesson. We are simply going to take the addition and subtraction that you already know and sprinkle a little C programming on top. When performing addition or subtraction in C, we use the + operator for addition and the – operator for subtraction. As simple as that! The only difference with the usual mathematical convention is that instead of having the result on the right, the variable in which the result is stored is always on the left.

If, for example, we have a variable in which we want to store the result of a calculation, the correct syntax for performing that calculation would be

```
total=2+3;
```

Notice that unlike in ordinary mathematics, the "result" is on the left as opposed to the left. The exact same applies to subtraction. In c, you can do actual arithmetic, of the type we just mentioned above, or do the arithmetic using variables. The latter allows for much greater flexibility at runtime because you can ask the user for input and do the arithmetic using those values.

Incrementing and decrementing

In some statements, you might want to increase the value of a variable in steps, typically when dealing with repetitive content. You probably remember this from that program where we read ten numbers from the keyboard. There are two ways in which you can do this. The first being a simple addition operation as shown in this code snippet

```
a=a+1;
```

This is pretty self-explanatory, but what if I told you there's an even cooler way of doing this? You could increment the variable by using

```
a++;
```

It's really as simple as that, and works for subtracting too, so if you are starting with a=10 and you want to decrement it notch by notch until it is a=0, then you could type. You have probably figured that we can make our code from the previous lesson even shorter than where we left it.

```
a--;
```

In the portion of your code that does the repetition.

There are two types of the increment/ decrement operation, namely pre increment and post increment

In the pre increment variation, the value of the variable is incremented before it is used in whatever operation that the computer will be performing.

An example of this is shown in this code

```
#include <stdio.h>
int main()
{
            int a=0;
            while(++a < 10)
            {
                 printf("%d ",a);
            }
            return 0;
}
```

In this program, the value of a is first modified. On the first cycle, for example, the value of a will be 0, then before the comparison a<10 is done, the value of a will be incremented, so effectively, the computer will compare a<10 where a will be 1. The output of this code will be

```
1 2 3 4 5 6 7 8 9
```

For post increment, the variable is first incremented before the operation on the variable is carried out. We just need to change the code a little bit to see this in action.

```
#include <stdio.h>
int main()
{
            int a=0;
            while(a++ < 10)
            {
                 printf("%d ",a);
            }
            return 0;
}
```

Notice here that we just changed one little thing, instead of ++a, we used a++. You may have already guessed, pre- means before, and post- means after. Here we are using it to describe the position of the ++ operator. Back to our code, the value of a is first compared to 10. In the first cycle, the computer will compare the value of a then increment it. This means at the time of comparison the value of a will be zero. When we execute this code, we will get the sequence

1 2 3 4 5 6 7 8 9 10

As much as these increment operators come in handy when writing repetitive code, they must be used with care to avoid unexpected code behaviour. You might, for example, end up with a loop that is not reachable or an infinite loop if you misuse these operators. You could even end up in a much worse situation where your code will simply refuse to compile!

Multiplication and division

You may have already probably already guessed that you can do multiplication and division in C. This is done using both actual values and variables. Multiplication in can be performed in 2 ways. You can use parentheses () or the asterisk *. There isn't really anything special going on here, the same way that you have always carried out multiplication in mathematics is the same way in which you would carry out multiplication in C. The only exception here is that when you use parentheses, you need to actually specify the multiplication operation using an asterisk, since brackets mean a lot of different things in C. Let's take a look at this snippet of code that multiplies a and b.

```
a =a*b;
a=a*(b+c);
```

The second expression will be evaluated using the usual BODMAS rule from mathematics. The expression in brackets is evaluated first, followed by multiplication by the variable a. Just like in mathematics, parentheses, or in more common language, brackets, are used to group variables and/ or elements so that they are evaluated together. The reason why we avoid the word brackets is that in C, as you may have already seen, we use different types of brackets, from angular brackets to curly brackets, so the word "parentheses" comes in as a welcome distinction.

The multiplication operator has another important use as a special operator as we shall see very soon.

Division is carried out using the forward slash. If we want to divide a by b and store it in a, then our code would look like this:

```
a=a/b;
```

The rules of mathematics apply here; the same BODMAS rule from mathematics.

Another important point to note is that when performing division, the result is typically not an integer. If you store your result in a variable of type int, you will certainly lose the fractional part of the result and your calculation will become inaccurate. This is usually a big problem, especially when accuracy is important or hen the result is used to perform further calculations. To get around this, the variable that stores results of a division operation is typically declared as float. This preserves the accuracy of the result as it stores the entire value, including the fractional part. This doesn't restrict you to using float for the entire expression; you can, for example, perform a calculation on two variables of the type int and store it in a variable of type float.

Logical operators

Logical operations are typically used when we want to execute a block of code based on a condition or a set of conditions, as illustrated by this table, the code requires the input of at least 2 variables in order to perform an operation

Operators	Example/Description
&& (logical AND)	`(a>5)&&(b<5)` It returns true when both and and b are true
|| (logical OR)	`(a>=10)||(b>=10)` It returns true when a or b or both are true
! (logical NOT)	`!((a>5)&&(b<5))` reverses the state of the operand "((a>5) && (b5))" If "((a>5) && (b<5))" is true, logical NOT will return false

This table might ring a bell if you've done a bit of elementary physics or electronics. If so, then you're in luck because these logic operators work on those exact same principles.

The && (logical AND) operator will return true if both operands are true. If any one of the operands does not satisfy the condition, then the expression will return false.

The || (logical OR) operator will return true if any one of the operands, or both of them, satisfy the condition. If not, then it will return false.

The ! (logical NOT) operand returns the alternate output of an expression. If we put it before the first expression, which returns true, the ! Operator will flip the output and return false. If the output of the expression is false, the ! Operator will flip it and return false.

Logical operators are very useful when you want to perform an operation that depends on more than one condition, for example if you are creating an automatic plant watering system, instead of watering the plants every day at 7 AM, you could add a check for the moisture level in the soil, so that the system would only water the plants if the time is 7AM or moisture level is less than 30%. Or both. This would be a smarter system since if for example it rains at 5AM, it can automatically determine that watering is not necessary, or, if it is unusually hot and dry during the day, it will automatically water the plants even though it's past 7AM. Clever!

More operators

There's a lot more that we can do with operators. In actual fact, the versatility of programming languages lies in what we can do with operators, they are the "handles" with which we move data around, compare it with other data, replicate it, “grow” it or “trim” it. Were now going to look at the more advanced data types in C. Let’s jump right in.

Relational operators

These are some of the things that we have been using in programs and you were wondering what on earth they are.

This table summarises what each of them does.

Operator	*Example*
>	a > b (a is greater than b)
<	a < b (a is less than b)
>=	a >= b (a is greater than or equal to b)
<=	a <= b (a is less than or equal to b)
==	a == b (a is equal to b)
!=	a!= b (a is not equal to b)

We have used some of these in while loops and in if statements, which is where they are mainly used. They allow us to test a variable against a certain condition to see if we can go ahead and execute a block of code.

Assignment

We have been using the = sign to assign values to variables. That, quite interestingly, is not the only way to do it. There are two types of assignment operators in C, namely simple assignment operators and compound assignment operators.

A simple assignment operator will assign a value to a variable, end of story. Compound assignments will do a bit more gymnastics before assigning a value to a variable such as adding a value to it. This table shows what you can do with assignment operators.

Operator	**Description**
=	a = 10; 10 is assigned to the variable a.
+=	a += 10; the value 10 will be added to whatever is contained in a, and the result is assigned to a, just like what a = a + 10 would do.

-=	a -= 10; the value 10 will be subtracted from whatever is contained in a, and the result is assigned to a, just like what a = a - 10 would do.
*=	a *= 10; the value 10 will be multiplied with whatever is contained in a, and the result is assigned to a, just like what a = a * 10 would do.
/=	a /= 10; the value in a will be divided by 10, and the result is assigned to a, just like what a = a +/10 would do.
%=	a %= 10; the value in a will be divided by 10, and the modulus is assigned to a, just like what a = a % 10 would do.
^=	a ^= 10; it will raise a to the power of 10 and the result is assigned to a, just like what a = a + 10 would do.

Learning to use the compound assignment operators will have the direct effect of shortening your code and make it more readable. Besides making your code readable, it actually results in less operations for the computer, therefore making your code more efficient.

If we consider the code:

```
x=x+5;
```

The resultant assembly code during compilation would be something like this:

```
MOVE A (X)
ADD A 5
MOVE (X) A
```

That is three lines of assembly code, which are roughly 3 operations in the CPU. Now if we change the code and write it as:

```
x+=5;
```

The resulting assembly code will look like this:

```
ADD (X) 5
```

We have saved the computer 2 operations! It might not seem like much, but imagine you have a program with one hundred thousand lines of code, and we have ten thousand such operations. It means that the computer, using the first method, would generate 30 000 lines of assembly code. Now if we use the second method, we will save the computer 20 000 operations with one little change! Now imagine the program loops 10 000 times. That means the computer would have been saved 2 million operations, just by changing a small portion of code! As we proceed with the course, you will start to see the great importance of writing efficient code. It is particularly important in C as it is used on a lower level of abstraction than other programming languages and is typically used to build platforms on which other programs, such as PHP. Nobody would have been using PHP if it were a slow and clunky system riddled with inefficient code!

Special operators

C has special operators that are concerned with the variable’s attributes in the computer’s hardware. The sizeof and the & operator allow you to request the size of a variable and the address if a variable, respectively. The * operator, when used in prefix to a variable, is used as a pointer to a variable.

Operators	Description
&	Requests the address of the variable. &a will return address of a.
*	This is used as pointer to a variable. * a, * points to the variable a.
Sizeof ()	This gives the size of the variable. Size of (int) on my PC returns 4

Now wait a minute, you remember that & that we used with scanf that we had no idea why we were putting it there? Well, that means you are telling the computer to save the input to the address location specified in the statement! If we look at the code snippet

```
Scanf(“%d”,&a);
```

We will see that the scanf function is directed to the actual address of the variable.

This is a particularly important instance of where the & operator is used. In C, arguments are always passed by value, unless otherwise specified. Now you recall that scanf is actually a function that is called from the stdlib header file, and that means the arguments that we pass to it, that is, the stream from the user and the variable to which we want to pass the stream, are passed by value. This isn’t really a problem for the input stream, but it is a big problem for the variable to which we want to store the input stream. You see, the variable ‘a’ in the snippet that we just looked at is a copy of the actual variable. That is always the case whenever you pass a parameter by value, a copy of the variable is made accessible to the function. This means that our input stream is stored in the copy, and once we navigate away from the scanf function, we no longer have access to the input stream-oops! This is where the & operator comes in. It tells the computer to store the input stream at the address of ‘a’. This bypasses the variable copy and modifies the actual variable. So, when we navigate away from the scanf function, we have the input stream stored int the variable.

To make it easy to remember, you could describe this code snippet one entity at a time: “scanf integer and store at address a”.

The sizeof operator will return the size of the variable in question. As an example, this code snippet

```
printf("%d”,sizeof(a));
```

Will output the size of the variable a in bytes, if I run this on my computer, it shows that the size of a is 4 bytes. Operator is especially useful when you want to dynamically allocate memory, for example, when you want to allocate memory that can hold 10 integers, but the program is targeted at many different machines, so it’s not possible to tell the size of int on the target system right off the bat. Remember we mentioned that the size of data types depends on the hardware of the target system. The sizeof operator allows you to get the size of the data type from the system at runtime and allocate the required space accordingly. This saves your program from crashing, because if you hardcode the size, if a computer has a smaller int size, your program will most certainly crash on that system.

The sizeof operator can also be used to calculate the number of array elements. Instead of relying on the array pointer to get the array size, which might turn into a nasty bug if the array pointer is modified elsewhere in the program, the sizeof operator allows you to determine the size in a fool proof way. We will take a closer look at this when we look at arrays in our next module.

Operator precedence

Expressions are the fundamental way in which a computer carries out computations. In fact, this is why programming languages were developed to begin with. If you look closely at any line of code, you will discover that there is always an operator. Even when it's not immediately apparent, the statement will probably have operators tucked away in a function or header file. As we will see shortly, operators can either be unary, binary, or ternary, referring to the number of operands that the operator requires in order to complete a task. The order of precedence of operators is very important, it affects the meaning of your statements the very same way it would affect a mathematical expression. If you are presented with an equation which involves a bracketed expression and you think to yourself, "duh, who cares about brackets" and you proceed to evaluate the expression whichever way you want, be prepared for major disappointment. Similarly, the compiler will always evaluate expressions using mathematical and logical means. Remember BODMAS/BOMDAS (Brackets, Division, Multiplication, Addition, Subtraction).

Arithmetic

Operator precedence when dealing with arithmetic expressions is pretty straightforward. If you're confident with the BODMAS rule from mathematics, then this is a breeze for you. This table summarises arithmetic operator precedence:

Operator	**Operation(s)**	**Position**
( )	Parentheses	Evaluated first. If the parentheses are nested, you evaluate inside out. If there are a number of pairs of parentheses on the same level that occur one after the other, they are on the same level and are evaluated from left to right.
*/%	Multiplication, Division, Remainder	Evaluated second. If there's more than one, they're evaluated left to right.
+-	Addition, Subtraction	Evaluated third. If there's more than one, they're evaluated left to right.
=	Assignment	Evaluated after evaluating everything else

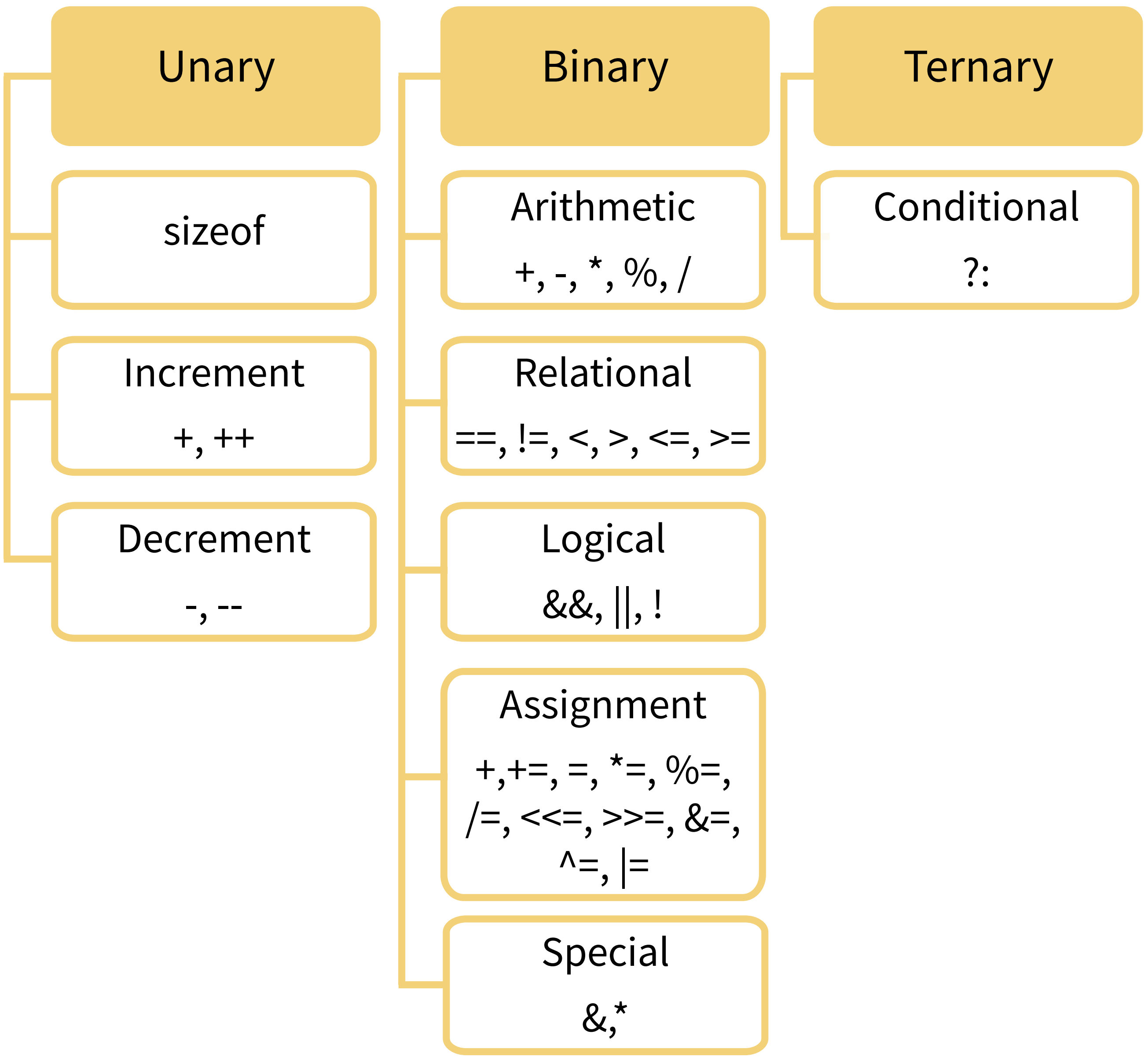
To demonstrate how arithmetic precedence works, we will look at a short demo of a program that calculates the roots of a quadratic equation. You probably remember the equation that solves equations that are in the form ax2+bx+c=0 can be solved using the quadratic formula, x=(-b±b2-4ac)/2a.

We want to write a program that ask the user for the value of a, b and c then calculate the two values of x and display them as x1 and x2. And just because we can, we will show the results for a+b+c, a-b-c, a*b*c, and a/b/c. Let's fire up our IDE and get coding!

[Quadratic equation solver demo]

Unary, binary, and ternary

Another way of classifying operators is looking at the number of operands that they require to complete an operation. Using this method, there are three types of operators, namely unary, binary, and ternary.



The good thing here is that the name of the classification suggests the number of operands that is requires. Unary operators, from the word uni-, require one operand to complete an operation, for example a++

Binary operators require 2 operators, as we learnt in our first module that binary means 2

Ternary operators, as has already become apparent, require 3 operators. Here, you require the condition, the expression for when the condition is met, and the expression for when the condition is not met.

If you have this rule on your fingertips, you will be able to immediately tell when you have used an operator correctly. This is also particularly useful when debugging your code.

Logical

Operator	**Precedence**
!	First precedence, always evaluated first
&&	Second precedence, evaluated after the ! operator if present
||	Last precedence. Evaluated after all other logical operators have been evaluated.

Logical operators are a bit tricky, but well, there are only 3 of them so it shouldn't be too much on your plate. In any expression, it's always NOT (!) Then AND (&&) then OR (||). Typically, you would use parentheses to group operands with their operators so that they are evaluated in the correct order. Remember we mentioned previously that we first evaluate whatever is inside parentheses.

Bitwise operations, as the name suggests, are done at bit level. You may be wondering why they are even used in programming when you can just program using base 10 numbers and save yourself all the effort of thinking like a computer. Turns out, bitwise operations are fast, in fact, they are many times faster than multiplication/division and can make code that rely on such calculations extremely fast. This is especially apparent when using a system with constrained resources. Although many processors can compute regular arithmetic just as fast as bitwise operations using longer pipelines and great architectural designs, bitwise operations still rule because they use less resources and consequently use less power. This is really good news for devices such as mobile phones, which rely on battery power. A study by Globalwebindex shows that 35-43% of users in various markets are consider the battery life of a device when shopping for one. You can quite easily see why writing programs that are power efficient is such a big deal. These are some of the things that you as a computer scientist will be addressing in your code as you optimise it after releasing the first version of an applications. We're just going to glide over them and look at them in later modules.

Operator associativity

When evaluating long expressions, we now know that we use operator precedence, right? But what if er encounter multiple operators at the same level of precedence? A new method works here, called associativity. Each level of precedence has its own associativity, summarised in the table of reference as we shall see shortly. When the computer encounters multiple parentheses, for example, the computer will evaluate them from left to right, and the same is applied to all other operators except those on the second level, ternary and assignment operators, which are evaluated from right to left.

It's really important to understand precedence and associativity because it can quite easily give you wildly incorrect results. To better understand the consequences, let's look at a simple example:

```
a=4+5*6;
```

If you evaluate the statement as it appears, then addition would be applied first, and the result would be 54. That is definitely incorrect! According to the precedence table, we should apply multiplication first. If we do so, then the result would be 34. That's more like it! You would have worse results if you incorrectly applied associativity where there are parentheses. Its's worth noting though, that the computer will always use the correct associativity rules, no matter what you type, so it's more on you, the programmer, to understand what you want to do than for the computer to understand what you want to do. And well, the computer is always right, it will always do what you tell it to do, and as long as your expression is syntactically and semantically correct, it will produce a result, even if it's far from what you intended for it to do!

This table from the official C reference guide beautifully summarises all that we have been talking about in the last section of this lesson

Precedence	**Operator**	**Description**	**Associativity**
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	

	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-left
14	= += -= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

(Source cppreference.com)

There's a few more things that we need to add to this table.

The operand with the prefix ++ and __ cannot be type cast. We are going to learn more about what this means in our next module.

The operand of `sizeof` cannot be a typecast.

The expression in the middle of the conditional operator, that is between **?** And `:`, is parsed as though it were in parentheses. Its precedence relative to `?:` will be disregarded

Operands to the left of assignment operators should always be unary.

Whenever you encounter a long expression, the order specified in the table that we just looked at will always be your friend. No matter what happens, the contents of a pair of parentheses will be evaluated first, followed by any increment or decrement operators. Next, BODMAS kicks in and evaluates any arithmetic in the expression. Logic follows and evaluates according to the precedence of logic operators. Ternary operators are then evaluated, assignment operators, in order of their precedence. Finally, any commas that separate expressions are then evaluated. In short, it goes like this:

Parentheses→prefix increment/decrement→postfix increment/decrement→unary plus/minus→negation/complement→cast/address/size→BODMAS→relational→Bitwise→ternary→ assignment→comma.

Quite a mouthful, but you will get the hang of it with practice. In any case, nothing stops you from having a reference table while you write your code, unless of course when you're writing an exam.

Conclusion

We have covered most of the basics that we need to write a functional C program. While there's still a lot more to learn, you can now pretty much fire up your IDE and write a small functional C program that can accomplish a few simple tasks. We are exactly halfway through our course, and we are now going to venture into the deeper waters of C programming and learn to harness the full power that the computer provides. In our next module, we will learn how to control the flow of our programs, which is a giant leap from the simple programs that we have written so far. We will also explore more powerful forms of variables such as arrays and structures, which will allow us to manipulate sets of data. That’s it from me, happy coding, and remember, practice makes perfect!

References

Buckle, C. (2019). Which Smartphone Features Really Matter to Consumers? - GWI. [online] GWI. Available at: https://blog.globalwebindex.com/chart-of-the-week/smartphone-features-consumers/

C Operator Precedence Table. (n.d.). [online] Available at: http://web.cse.ohio-state.edu/~babic.1/COperatorPrecedenceTable.pdf

Chapter 7 Expressions and Assignment Statements Chapter 7 Expressions and Assignment Statements. (n.d.). [online] Available at: https://www2.southeastern.edu/Academics/Faculty/kyang/2014/Fall/CMPS401/ClassNotes/CMPS401ClassNotesChap07.pdf

Cppreference.com. (2011). Increment/decrement operators - cppreference.com. [online] Available at: https://en.cppreference.com/w/c/language/operator_incdec

fresh2refresh.com (2020). Relational operators in C | C Operators and Expressions | Fresh2Refresh. [online] fresh2refresh.com. Available at: https://fresh2refresh.com/c-programming/c-operators-expressions/c-relational-operators/

https://www.facebook.com/webencyclop (2017). C Operators | Types of operator: unary, binary, ternary - WebEncyclop. [online] WebEncyclop Tutorials. Available at: https://tutorials.webencyclop.com/c-language/c-operators/

OverIQ.com. (2020). Operator Precedence and Associativity in C. [online] Available at: https://overiq.com/c-programming-101/operator-precedence-and-associativity-in-c/

Software II: Principles of Programming Languages Lecture 7 -Expressions and Assignment Statements Why Expressions? (n.d.). [online] Available at: https://home.adelphi.edu/~siegfried/cs272/272l7.pdf.

Super User (2015). C Operator Precedence and Associativity table with examples. [online] A4academics.com. Available at: http://a4academics.com/tutorials/77-c-programming/688-c-operator-precedence

Weber.edu. (2020). 2.1 Operators and Operands. [online] Available at: http://icarus.cs.weber.edu/~dab/cs1410/textbook/2.Core/operators.html