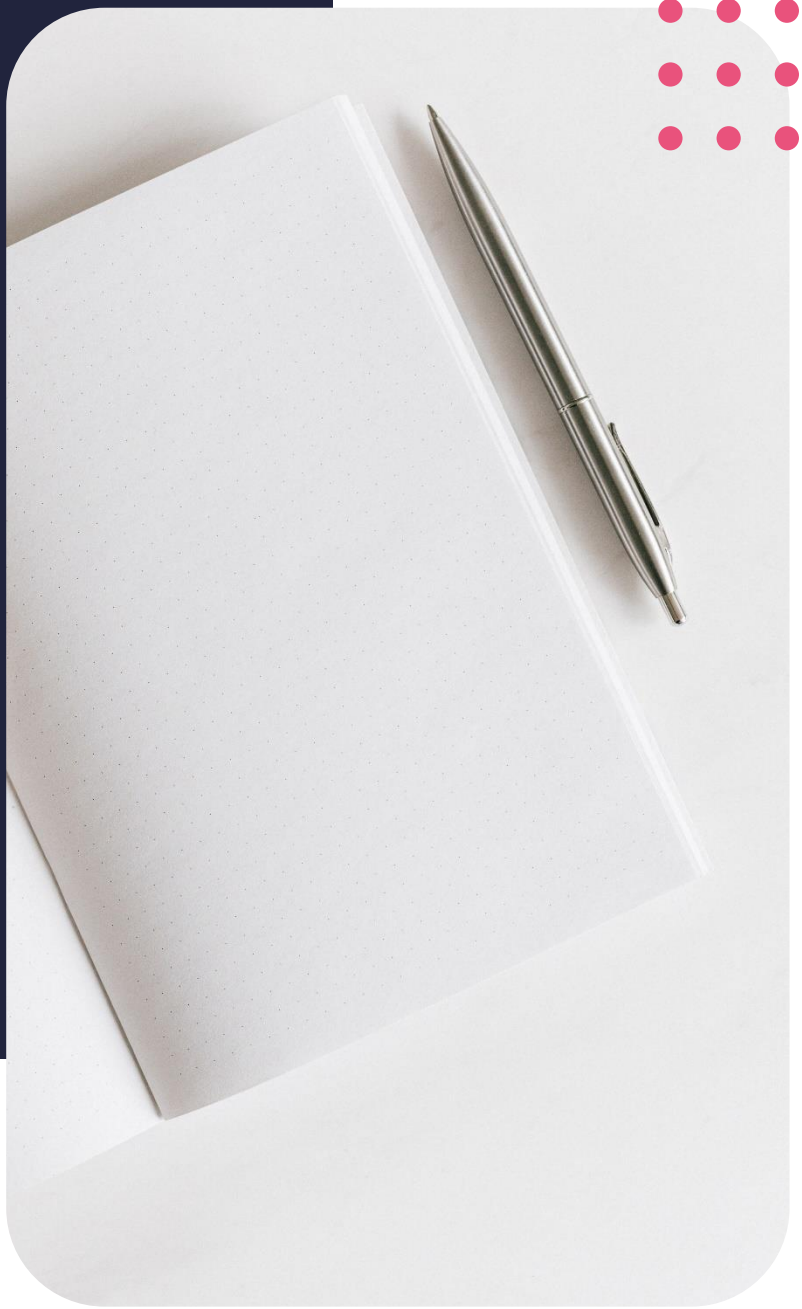
Diploma in Computer Science

Program Structure

Summary Notes

Contents



Lesson outcomes

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

- Understand the basic structure of a C program
- Discuss the importance of each part of the program
- Explore the basics of program readability
- Explain the significance of functions as the body of the program

Parts of a C Program

In our previous lesson, we wrote our very first “Hello world” program. As short as it was, the program covered the basic structure of a C program! The C programming language. In case you’re wondering why we wrote hello world; it has grown over the years to become a standard of “initiation” into the programming world. It also serves as a simple test to see if all is set and running correctly

Documentation section

The documentation section in a C program is the topmost part of the program. This is used to give information about the program, along with other vital information.

The documentation section will typically contain several comment lines. In this section, that is where you would put your name, along with the names of others you would have worked with if applicable. Other information such as the date on which the program was created, the version number. If this is an update to a previous version, this should be specified, along with the date on which the update is created.

It is also important to include a brief description of the purpose of your program. This gives anyone (including yourself) a clear picture of what the program does without having to browse through the whole source file.

Here you may also make reference to a user guide, a chapter in a development report or reference to a help website.

Using another programmer’s code without permission and/or acknowledgement is considered plagiarism and may qualify as copyright infringement. If you start a project that builds on a project for which several key programs have been written, or if you find really useful code during your research, you need to look at the copyright of that code to see how you can legitimately include it in your project. Of course, this is usually not an issue if you are using the code for educational purposes, such as now, when you are still learning to code, but if you are building commercial software, you definitely need to cite the source. For commercial projects, these acknowledgements and copyright conditions need to be followed strictly.

There are general guidelines to copying code from other sources

If you are including the copied method as part of your project, or a function, include a reference in the documentation lines. It is good practice to include it, even if it is not requested by the original programmer. Programmers usually specify the format in which it should be referenced. If not, you can use conventional citing methods.

If you are using the copied code as a standalone program, make sure that the original programmer's conditions of use are met and correctly documented and referenced. Also, make sure that the program is clearly referenced as someone else's work and not mistaken as your own.

It is always better to contact the developers if anything pertaining to copying any part of their code is not clear. Typically, contact details are included in the documentation section, and you should include yours in your own programs too!

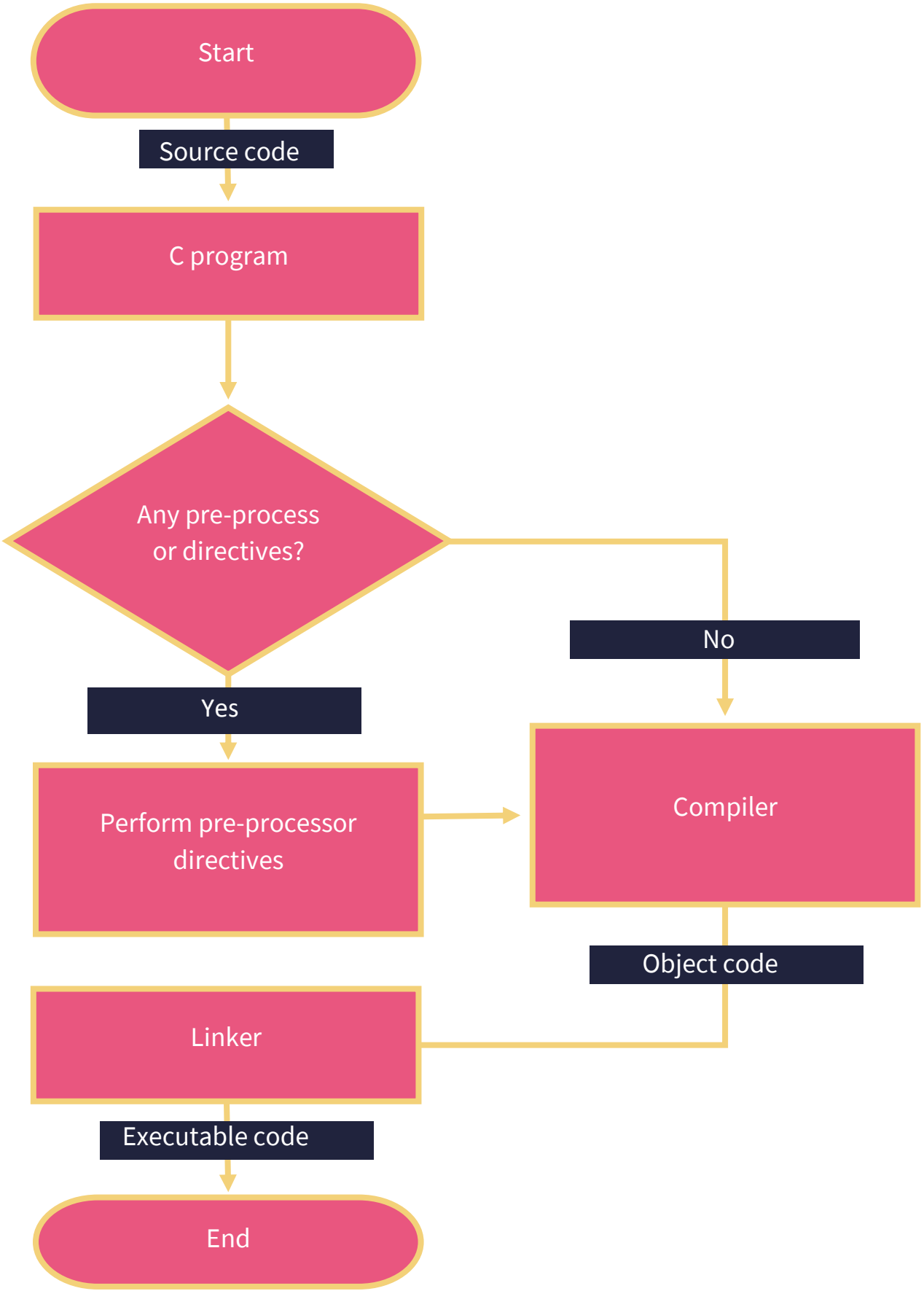
We spent quite a bit of time on this section because copyright battles sometimes end up in court and cost a lot of money. You might have come across this in the news, where companies pay millions in fines for copyright infringement. It's kind of easy to plagiarise code, it's easy to convince yourself that you can get away with it as long as your program works, but it is certainly unacceptable.

Pre-processor directives

This section contains 2 main sections, namely the

- link section and the
- definition section.

As the name suggests, these commands are carried out before processing the rest of the program. This applies to the compilation process that we looked at in module 1. Here is a simplified version of the process, showing where the pre-processor directives fit in.


Start
Source code
C program
Any pre-process or directives?
No
Yes
Perform pre-processor directives
Compiler
Object code
Linker
Executable code
End

The pre-processor directives will be checked first before any compilation is done, and if there are any, those are dealt with first, then the computer moves on to compilation.
The link section is used to include the external header files which contain predefined functions that enhance the capabilities of your program. In our previous lesson, we used the stdio header file, which provides the functionality we need to display text on a terminal. This is how we were able to use the printf function, without having to manually map characters onto the screen.

There are four main types of pre-processor directives, that fall into the two sections stated above, namely macros, file inclusion, conditional compilation, and miscellaneous directives.
The 'include' directive does exactly what it implies. It tells the computer to include the specified file in the compilation process. The angular brackets tell the computer to search for the specified file in the system area. In this case, the computer will look for the stdio.h file in the system area.

You can also include your own files by using quotes instead of angle brackets. When writing a large program, it is easier to split it into several files, which is what you will do anyway if there are several people working on the project at the same time. Also, also programs become larger, it becomes increasingly tedious to have to forward declare every function you want to use that is defined in a different file. It's easier to have those functions in one place, then include those files whenever they are needed. If you use quotes, then the pre-processor will look for the file in the project/current directory

Definition section

The definition section is still part of the pre-processor directives, but here, instead of linking to existing files, we are actually creating new objects.

Macros are essentially a piece of code that has been given a name. whenever the compiler comes across that name, it copies the code into that location. Macros can be a simple label that can be replaced by a value, or a bit more complex, such as performing calculations and passing arguments. It can also be an expression, basically anything within the confines of the C language. For example, we can create a macro that will calculate the area of a triangle. The computer is presented with a label that has arguments, and these arguments are passed to the copied code that performs the calculation and returns a value. Pretty cool right? We are going to see this shortly in a piece of code. To define a macro, the pre-processor directive #define is used. A macro is not affected by block structure. It will last until it is undefined by the # undef pre-processor directive.

A point to note here; because pre-processor replacements take place before any C syntax check, they might result in unexpected behaviour. They seem really handy, but if you overuse them, your code might end up really hard to read. It's easy for the computer to simply replace the label with code, but a human can't be jumping back and forth through the code from time to time, trying to refresh their memory with the code defined in the macro. When defining a macro, there is no termination symbol, or full stop or anything. It's just the statement, then we move to a new line.

There also exist predefined macro names, which always begin with 2 underscores. These are primarily diagnostic and documentation oriented.

Conditional definitions are only honoured if the specified condition is met. these directives can direct a computer to compile a specific portion of code or to skip compilation of a certain section of code. This is done using the pre-processor directives #ifdef, #ifndef, #if, #endif, #else and #elif

The error directive #error will halt compilation if an error is found, and it displays an error as defined in its parameters.

Other compiler specific directives and other less common directives exist and are used in specialised programs. You might encounter these in advanced C programs, and you can always refer to C documentation to figure out what they are, what they do and how they are defined.

Although header files are processed before compilation, they too can sometimes cause errors. (for example) If you include a header file twice, it means that the code will be included twice; you probably can already imagine the confusion that that would cause, given that the computer locates things using absolute addresses. When you include the same header file twice, your code will not compile. Header files are often paired with code files, with the header file providing forward declarations for the corresponding code file

Declaration Section

The declaration section is an essential part of any program. This is where a good number of variables are brought into being. This is where most of the program lives, that is the user defined functions and the main function.

Function Declaration

Within global declaration, you can also declare your own functions. If you recall we mentioned in previous lessons that a program runs from beginning to end, following the sequence of actions defined in the code. This means that for functions to be visible to the rest of the program, they have to

be declared above the main function. When the compiler parses the source code, it will make reference to these functions, so when they are called, it knows exactly where to find them.

The main function

The main function is, well, the main function of the program. your program needs to have a main function because this is where execution takes place. Your code might be written elsewhere but ultimately it is called from within the main function in order to be executed. The main function

consists of 2 sections, the declaration section, and the execution section. In the declaration section, any variables and constants that are going to be used in your program are declared and initialised, and these are visible from within the main function. The execution section contains statements of code that your computer will execute line by line, taking note of any references to other areas within the program.

Just in case you're wondering, a function is a self-contained block of code, essentially just a bunch of commands tied together and given a name. as we mentioned before the main function is mandatory. When the computer begins to execute your program, the first thing it will look for is the main function.

All the tings above the main function would have been integrated into the program and woven in during compilation, so technically, when your program has been compiled, the first point of entry into the program is the main function. It's probably safe to say that it is the only point of entry into the program!

Once the main function has executed all the statements within it, it needs to exit. This is done using return statement. As mentioned in the previous lesson, the return statement tells the computer to end the function and return a value from the executed function if applicable. In the main function, the computer typically needs a way to determine if the program has executed successfully. A program that returns zero will have executed successfully. If the function returns a value other than zero, it means there was a problem with the function, and the programmer needs to look into it.

User defined functions

You can define your own functions in this section. These are referred to as user defined functions. We are going to look closely at user defined functions in our next lesson when we cover syntax. User defined functions will allow you to split your program into sections without having to save it in different files. This is also useful if you have routines that you want to use in different places in your program. When you have defined functions, it means that when you want to use the procedure defined in that function, all you have to do is call the function, and yes, it's literally calling the function! You type in the name of the function where you want it to be used, along with any arguments, if your function takes any. The computer would have filled in the function contents into that portion of the program, so it executes the code contained in your function before moving to the next line.

User defined functions are also useful when you want to move specific routines into functions to either keep things neat, make your program more readable or make writing the code more efficient. You can define all your routines in functions, and in the main function, you only call the functions and pass relevant arguments to them. of course, readability here depends on how you name your functions.

For example, if you are writing a program to calculate the area of various shapes, you would want to name your functions something like areasqr, areatri, areacircle, which will have the effect of not only making your code shorter but making it a lot readable.

Explaining your code

One of the most important aspects of good code is readability. The people who created programming languages thought of this and included a genius way of addressing this.

High level languages contain functionality referred to as comments. Comments allow the programmer to explain a line of code or a block of code that is not really clear at first glance. It's not just reserved for cryptic code, you are at liberty to use comments as you see fit, they serve to explain every bit of your code. The good news does not end there; you can put comments just about anywhere in your code; when declaring functions, when declaring variables, when performing a calculation, when defining macros, when defining constants, anywhere! You may be wondering what happen to all that text when you execute your program.

The answer might not be what you are expecting-: nothing! Because comments are really just meant for humans to understand the program and have nothing at all to do with the execution of the program, when the compiler compiles the code and produces machine code, it chucks comments right into the

trash before the compilation process has even started! You remember from module 1 that even spaces count to nothing when the computer is compiling machine code, all that matters at machine code level are symbols that actually do something, like represent a value, assign values, perform calculations, or carry out a command. Just so you know, comments are not some random junk that some people just thought of adding to code, it is actually part of coding standards and constitutes good programming practice to include comments in your code.

Let's now look at how you add comments to your code. There are two types of comments, namely single line comments and multi-line comments.

Inline comments

Inline comments are comments that are written in just one line. By one line, it means the comment will be in the same line as the code that it is describing. A single line just goes on and on until the end of the line. This type of comment is suitable for when you don't want to say much; it's one line, it wouldn't look good to have a line that extends way beyond the screen. There's really no need to do that because there's a type of comment for that, called multiline comments.

A single line comment is recognisable by the double forward slash //

Multiline comments

A multiline comment is one that starts with a forward slash and an asterisk /* and ends with an asterisk and a forward slash */. Everything in between /* and */ will be ignored by the compiler, and that includes code, even if you accidentally comment it out. Multiline comments are the more

suitable option if you want to describe a line of code or a block of code in a bit more detail. It allows you to write an extensive explanation for your code. This is particularly useful for explaining what, say, a block of code does.

When writing comments that cover more than one line, which is probably what you will be doing anyway while using multiline comments. The conventional way is to add an asterisk at the beginning of each line. If you take a look at one of the many C source files lying around on the internet, you will see that it's almost always done this way, again this is done to improve readability. It's just easier to tell apart lines of code from lines of comments because of the asterisks. If you ignore this rule, the reader can easily get lost in the comment, because the symbols that identify the comment will only be at the beginning and at the end of a pretty long paragraph. This is especially true if you have commented out a block of code.

Comment conventions

A person reading a large chunk of code can easily get lost in it if you do not comment it.

Comments ensure that the code is readable. I'm probably saying for the 100th time. It's important that it really sinks in. comments will also help you, yes, you read your own code when you read it later!

Comments are written using readable, narrative language. This is typically what you should do wherever possible, especially when explaining large chunks of code. It is also a good practice to put comments after the block of code that they are explaining rather than before the block of code. Generally, it's best to avoid shorthand. If you have picked up somebody else's classroom notes and found that they are written in shorthand, it's easy to pick up what I am getting at here. Shorthand is kind of a personalised language slash writing style, so that would defeat the purpose of trying to explain the code in the first place.

When using a comment on a pretty self-explanatory block of code, it is good programming practice not to put the comment in the same line. The comment should rather be on its own line.

It is also good practice not to put a blank line between code and comments. The reason here is pretty clear, you wouldn't write a story with each sentence as its own paragraph, right? The same thing applies here, and it's the same anthem we have been singing throughout the entire lesson: readability!

We're not completely shunning blank lines, however. They have their place. It is good practice to put a blank line whenever you write a block of comments unless the comment is at the beginning of a code structure.

Just like normal sentences, comments should be in sentence case. You really don't have an excuse not to, remember, we are aiming for readability here. Also, check the spelling and grammar in your comments. When you comment out a block of code, when you are now compiling the final version of the code, you should remove that code and leave only the comment.

In case you do leave the commented-out code, there has to be a valid, documented reason for doing so. This is important, because code might be accidentally commented out, and you will spend hours and hours trying to debug your program because of this rookie error. In case you are wondering,

sometimes you do need to comment out sections of your code during debugging. This is an easy way to test the behaviour of code without having to completely remove it.

More content available in the demo.

References

Aℓιєи (2018). Basic Structure OF C Program with Example. [online] CyboWap | Leaen Programming Easily. Available at: https://cybowap.blogspot.com/2018/05/basic-structure-of-c-program.html

GeeksforGeeks. (2018). Comments in C/C++ - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/comments-in-c-c/

Joomla! Developer NetworkTM. (2020). Coding Standards Manual. [online] Available at: https://developer.joomla.org/coding-standards/inline-code-comments.html#:~:text=Inline%20comments%20are%20all%20comments%20not%20included%20in%20doc%20blocs.&text=Comments%20that%20are%20written%20in,the%20entire%20block%20of%20code.

OverIQ.com. (2020). Basic Elements of a C Program. [online] Available at: https://overiq.com/c-programming-101/basic-elements-of-a-c-program/

SPLessons. (2017). Structure of C Program - SPLessons. [online] Available at: https://www.splessons.com/lesson/structure-of-c-program/

Suryateja Pericherla (2016). Structure of a C Program - My Blog. [online] My Blog. Available at: https://www.startertutorials.com/blog/structure-c-program.html

Ucl.ac.uk. (2013). Documenting your programs. [online] Available at: https://www.ee.ucl.ac.uk/~mflanaga/Documenting.html#:~:text=date%20on%20which%20the%20program,in%20a%20report%2C%20web%20page