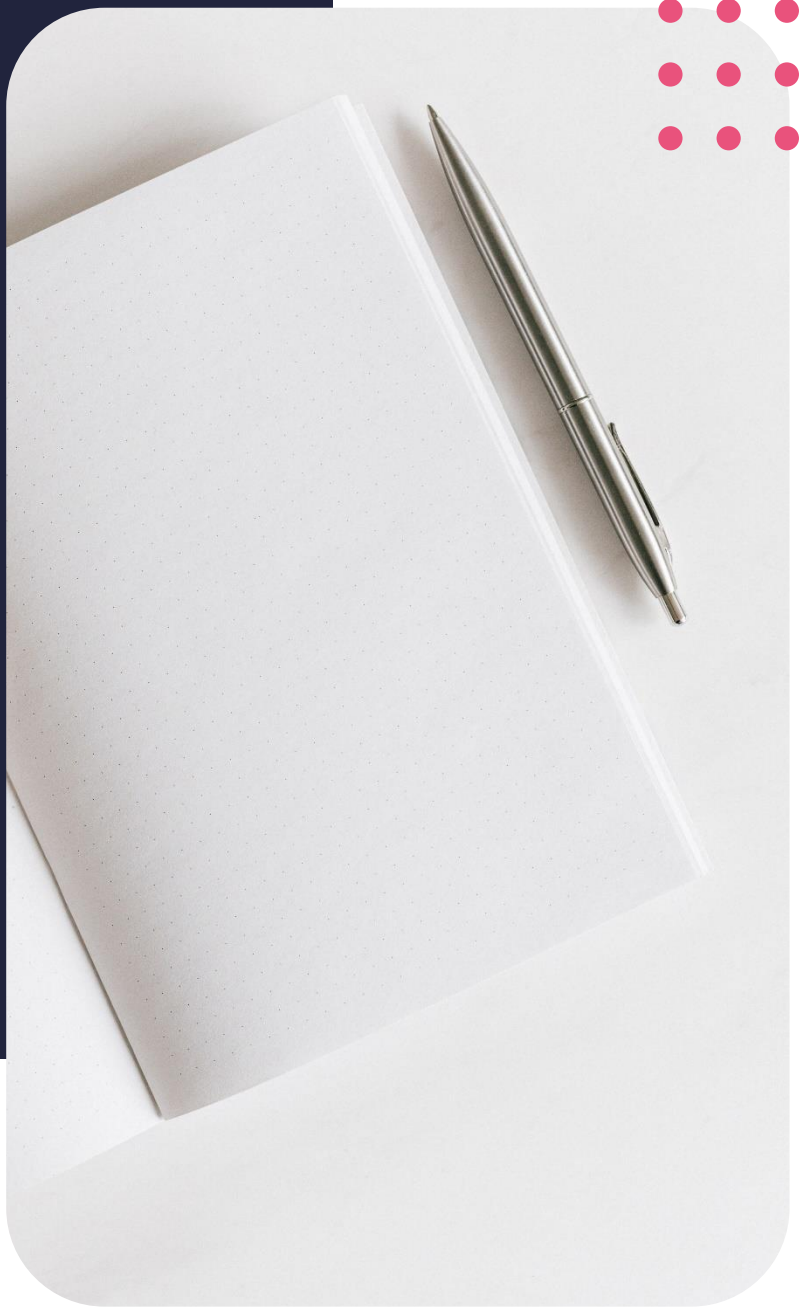
Diploma in Computer Science

Preprocessors, header files and libraries

Contents

Contents



Lesson outcomes

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

- Explore the contents of header file
- Appreciate what a library is as used in programming
- Understand the concept of APIs
- Appreciate the role of the preprocessor

The compilation process (again!)

To kick start our lesson today, we are going to have another look at the compilation process. Why again are we doing this? Well, since we want to get a closer perspective of header files, we need to know their place in the compilation process. this helps us understand why they behave in a certain way and why some rules apply to them and not to the rest of the code. The compilation process is where your code is “reviewed” by the computer to make sure that everything is in its place and works correctly. It follows a certain procedure, which always starts with the pre-processor directives. that portion of the process is what we are keenly interested in today.

Preprocessor directives

In our first module, we mentioned that whenever you compile a programme, the computer takes the high-level language code, runs it through the pre-processor and update is pure high level language code. This is then passed through the compiler and the process produces relocatable code. The relocatable code is then passed on to the assembler which will produce assembly code. The assembly code is then passed on to the linker and loader which then produces machine code that is ready for execution by the machine. The process is quite a lengthy one, and each stage produces its own type of code. The very first part of process it's carried out by a special piece of software called the pre-processor. Most of what is processed by the pre-processor is indicated in the code by the hash symbol (#). these are known as pre-processor directives. It means exactly what it sounds like. The hash symbol directs the pre-processor to process whatever comes after it. This list shows some pre-processor directives that are available in C. the most common one by far is #include. we have already used this quite a lot in our programs to include the standard header file, which provides most of the functionality that we need in most programs.

the pre-processor is actually a separate program from the compiler, which does all the preparatory work for the compilation process. The pre-processor enables this to include header files, micro Expansions, conditional compilation, and line control. The grammar of C is not strictly implemented in the pre-processor, and so other kinds of text files can also be handled by the pre-processor when needed. Pre-processing is defined by 4 main stages. Let's take a closer look at each one of the stages.

The first stage is called trigraph replacement. In this stage the pre-processor replaces trigraph sequences with the characters that they represent. Trigraphs are sequences of three characters that

appear in source code, which should be treated as if they were a single character. These are used when there is no keyword to represent that particular set of characters although it performs a specific function. When the pre-processor is done with this it then moves on to the next stage.

The next stage in the pre-processor's sequence is line splicing. This is where physical source lines that way continued with escaped newline sequences during coding are spliced to form logical lines.

the next step it's called tokenisation. in this step call mother pre-processor breaks the resulting code enter tokens and white space. You recall tokens from our first module as bits of characters that are meaningful to the compiler. Comments are completely removed and replaced with white space. again, you recall here that comments are of absolutely no use do the compiler and are only really there for you the programmer.

The last stage is where macros and directives are handled. Any lines that contain pre-processor directives including inclusion and conditional compilation are executed. the pre-processor expense macros and handles pragma operator. Esther mentioned inclusion, this is where any files that are not exactly part of the code that you wrote are important and lined up for compilation.

Types of directives

Probably the most common and the one that you are most used to is the #include directive. It is used when we want to include a header file, either from the standard set or from the set of our own defined files. We are going to look at this more closely in a bit. These files that are added to the program using the include keyword together with a pair of angle brackets are part of a standard library, which is a repository of code that is available alongside every C compiler that conforms to the C standard. The exact location of the files that are included in this library is system dependant.

The pragma directive, which is short for pragmatic information, provides a way of requesting special behaviour from the compiler. In your early days as a programmer, you will probably not come across this directive, as it is used for programs that are larger than usual and require the compiler to carry out compilation in a certain way. By "certain way" we refer to the special features that are peculiar to that compiler and are perhaps only available on request. The pragma request is usually followed by a single token, as shown in its general syntax in this code snippet.

```
#pragma token
```

Only tokens from the set of permitted tokens will be recognised in this directive. The set of commands is different for each compiler, so it's a good idea to check with your compiler's documentation to see which commands it allows you to use.

From c99 onwards, there's a _pragma operator, which allows pragma to be produced as a result of a macro Expansion. This means that from this version of c, you can embed the _pragma operator, much like the sizeof operator, in a macro. This code snippet shows how you can use a pragma

```
#pragma exit
```

When included in a program, the function specified in the pragma must run before the program ends.

Pragma directives are compiler dependent and if your compiler does not support pragma directives, it will simply ignore the code. To test it out for yourself, try writing this code on your computer and see.

```
#include<stdio.h>
int pragmaTest();

#pragma startup pragmaTest
#pragma exit pragmaTest

int main() {
  printf("\nI am in the main function");
  return 0;
}
int pragmaTest() {
  printf("\nI am in the test function");
  return 0;
}
```

Where do header files fit in

Like we mentioned one of the functions of the header file is to include files in text that are not explicitly laid out in the code that you have been writing. This is one of the most common uses for the pre-processor. If for example, the pre-processor encounters #include<stdio.h>, it knows immediately that the file that is enclosed in angle brackets has to be included in the code. He recalled that when we talked about header files, we mentioned that if the filename is enclosed in angle brackets, the computer automatically knows that it has to look for the specified file in the standard C include source path. Nearly every ide has a way of configuring the location of the standard include file path or directory. You can even go into the settings of your ide and you will see where these files are stored. This can also be introduced using a command line flag, which can be parameterised using a makefile. This is important if a source file is to be compiled on different operating systems or platforms. Back to our process, when the computer encounters this file, it replaces the tag #include how long will the filename with the contents of the file in question. This means that, for example, when the pre-processor encounters stdlib.h, it replaces that with the text that is contained in the standard library header file. In the next stage, which is compilation, this text is then treated as though it is code that you the programmer wrote. It is compiled and executed in exactly the same way as your own typed out code. Saves the programmer at a lot of time and effort as they do not have to type out, or copy paste the contents of standard header files. Instead, the programmer only needs to know the name of the relevant header file and include it in their code.

By convention as we have already seen in previous lessons, a header file is characterised by a ".h" as its extension. Sometimes you might encounter header files that carry the dot hpp extension. Some of the files carried the dot def extension and this is usually used for files that are designed to be included multiple times. You might also encounter dot xbm files, which are image files. Header files should only be included once and will generate an error if you attempt to include it more than once. This is in line with the general syntax The C programming language which does not allow the same names to be used more than once. You will get the hang of this once we get to the section where we are going to explore the contents of header files.

Header files typically contain all the functions and expressions that you need to perform certain actions. When the header file is included in your code, this functions expressions and procedures become part of your code. These are then used and referenced as though you wrote this code yourself.

Standards in libraries and header files

C was primarily developed to work in the development of operating systems. During the early days, it what is a lot different from the standard C programming language that we know today. As C programming became increasingly popular, the need to standardise the way things were done in the programming language became more apparent. Around 1984, the first C standard library appeared. Of course, before this, libraries existed but were not standardised as was done around this time. A standard library made it's easier for a programmer to know what they could do and what they could not do. If the library was not standardised, it would be very hard to tell whether he'll programme would work on other systems or not. The libraries that you mentioned in your code may not be available on the system on which the code is eventually compiled.

The C standard Library

The C standard library is basically just a collection of header files that contain functions and procedures that perform various tasks. Functions in the standard library group pretty much organically from communities of programmers that were sharing ideas on how to implement functionality that the C programming language didn't natively provide. The stemmed largely from the 1970s where many people used see on Unix across multiple architectures. By the 1980s, a group called the “/usr/group” organisation set out to work to combine the files that had been built into a library of sorts. Around this time, it was still pretty much informal, and no standard had been established. The standard was then established in 1984, if you recall from our previous module. The American national standards Institute (ANSI) formed a committee to establish a standard specification for the C programming language. The committee drew on a lot of experience, but with an inclination towards clarifying existing code rather than innovating new sets of code. This resulted in a small, lean standard library.

To make it easy to understand the role of the standard library, let's think of a function that ref been using for quite a long time , the printf function. How does it send text to the screen and display it? I bet you have no idea how that's even done. This is 1 example of how a library is used. You do not need to know the intricate details of how a task is carried out, rather you just need to know the syntax of the particular function or routine that you want to carry out, along with any parameters that needs to be passed and what kind of information will be returned. Libraries save programmers so much time since they do not have to code every little single thing that they want the computer to do. Instead, allot of the common tasks I already partially solved by other programmers and made available in the form of header files. Nearly every programming language has a library. The sea standard library it's not as large as the libraries and other programming languages, which is an advantage for us because you want to learn the intricate details of how a computer works. Like we mentioned in our first lesson in the second module, there's really no incentive for you to learn how functions and algorithms work if they are already provided and standard library. Just like we just mentioned, why on earth would you bother yourself what learning how to display characters on the screen when you can just use the printf function?!

Libraries are an integral part of programming as they shave off a substantial amount of development time and I love you to use that time on other tasks. Libraries also allow interoperability of code among

various systems. As long as the libraries that you used in your code are available on that particular system, she can be guaranteed that your code will work as intended.

Syntax
The general syntax for using libraries is simple. It is as simple as adding a pre-processor directive such as #include<math.h>. As soon as you do this, you already have access to all the functions that are contained in that header file. You recall from our demo that used the math dot H header file that we could use new functions such as the square root function just by adding that header file.

What is really in there

Compared to libraries from other languages the C library could be described as skinny. Doesn't have much in the way of containers and general algorithms. Well, this might sound like a disadvantage, but it actually helped see to be ported easily to other platforms and architectures. This is also the reason why C is a relatively low-level language but manage is to retain high portability. It's very rare to encounter a C program that won't run as intended on a different platform.

Let's now run through the header files that are found in the C standard library.

Assert.h

The assert.h provides a simple way to hold a programme with debugging information if a provided assumption does not hold. This is a very useful file for quick test runs. Rather than calling the exit function they assert header file calls the abort function to quit and dump the core if the operating system allows this. If supported, you can then load the file in a debugger and inspect the programmes full state at the time of the assertion. This header file it's one of the special ones as it can be used multiple times without causing an error. In fact, it is the only one that can be used multiple times without causing errors. By including its multiple times, you can enable and disable the assert macro. Using the search header file is a good way to keep track of bugs in a large program.

Ctype.h

This header file contains functions that are useful for testing and mapping characters. All functions accept it as a parameter, but the value must be presented as an unsigned character. All functions return true if the argument satisfied the condition described and false if not. The header file contains functions that can compare many types of characters ranging from digits, hexadecimal digits, lower case and uppercase letters, alphanumeric letters, punctuation and graphics characters and many other types of characters.

Errno.h

You may have guessed that this name is short for error number. It is a sort of global integer variable. If value, which is the error number, is stored in errno by certain library functions whenever they detect an error. When the program start, the value of errno will be 0. The functions will only ever store values that are greater than 0 . This header file doesn't do much other than this.

Float.h, limits.h, and math.h

These header files essentially just scream mathematics! In fact, that is the very reason why we combined the mental one section. You probably remembered the math dot H file from one of our demos. Each of these have a set of mathematical functions that allow you to carry out special mathematical operations, such as calculating square roots. Because we are sticking to only the standard C header files, you'll notice that we skipped the complex dot h file. It is one of the films that allows you to perform special mathematical operations. I would like to note at this point however, that using non-standard header files may save you a lot of coding but it is detrimental to the portability of your program. This cannot be emphasised enough.

Float, as the name suggests, is used for floating point operations. The limits header file header file is used to determine the maximum and minimum values of various data types. The math header file is used for performing advanced mathematical operations.

Locale.h

Often, when writing code, you may want to tell the computer how to handle regional customs such as language date time and currency. This is quite important as you may have noticed that if you change the language on your keyboard sometimes the keys don't even correspond. You do not want this to happen to your program after you deploy it. Some of the settings in this hit the file I meant for interpretation by your own By default, C users this C locale which is ASCII text and American formatting.

Setjmp.h

This header file allows you to use a sort of saveable go to statement in order to jump to places that you've already been. It also allows you to jump from one function to another. Since C does not have built-in exception handling combat this is a sort of exception handling for a C programmer. It must be noted that you must use this file with care as it is rather complicated and might lead to problems in your program.

Signal.h

This file is primarily a Unix file. It utilises the Unix technique for Inter process communication that causes a process to asynchronously call a handler function, or else take a default action. As we already mentioned, this file provides primarily Unix functionality and it's not really portable. This means that the file does very little outside the Unix system.

Stdarg.h

When C spread to other architectures, C benefited from creating a portable way to access variable numbers of arguments. The stdarg file carries a number of f assumptions regarding portability:

The variadic function must declare at least one fixed argument The function must call va_end before returning (for cleanup on some architectures) va_arg can deal only with those types that become pointers by appending “*” to them. Thus register variables, functions, and arrays can’t be returned by va_arg

If a type widens with default argument promotions, then va_arg should request the widened type.

Stddef.h

The stddef.h header defines various variable types and macros. Many of these definitions also appear in other headers because C can be compiled in either a “hosted” or a “freestanding” environment. The freestanding environment is reserved for embedded programming where there isn’t enough space to store the entire standard library. An implementation must include all the standard library headers to be considered a hosted environment, while a freestanding environment must include only float.h, limits.h, stdarg.h, and stddef.h.

Stdio.h

Judging by the sheer number of times that you have used this header file it is probably one of the most useful. Back in the days when C was mostly used for Unix development, things were very clean and simple. When C started expanding into other operating systems come on problem started popping up in the input output system. Developers needed a way to standardise the input outputs populations on Unix and other systems. As the name suggests, this file provides the standard functions that you need for input and output on the host system. This means that you do not need to know how this is implemented, rather these functions will interface with the operating system and do whatever magic needs to be done for the input output operations to take place. The important outputs operations that are implemented in this header file are not just limited to display characters from the screen and grabbing them from the keyboard but includes files and other peripherals.

Stdlib.h

We've seen the standard library header file quite a lot of times, but we have never really stopped to look at what it contains. The standard library header file defines four variable types, several macros, and various functions for performing general functions. A quick example of what might easily ring a bell is those string manipulation functions that we looked at when we were discussing arrays. Important functions for allocating and deallocating memory are contained in this header file. For this reason, we're going to use it quite a lot when we look at memory management.

String.h

The string header file, as you guessed already after looking at the name, is mainly used for string manipulation. But wait, didn't we say that these are contained in the standard library? Turns out, both files contain different sets of string manipulation functions and both work in slightly different ways. The string at a file contains most of the basic functions like string comparisons, string copying and concatenation.

Time.h

You have already seen by now that most of the names of the header files in the standard library are pretty much self-explanatory. This is the same story with the time header file. It has all sorts of functions that I used to handle conversions of time and date. One interesting function that is contained in this header file is the Clock function. It can be used to measure the time elapsed since the programme started running. It must be noted however but this library is limited to nothing smaller than seconds of precision.

User defined header files

We mentioned previously that C allows you to create your own header files. This means that you can effectively create your own library that caters for your needs. This is actually what some developers do to save 1 development time on later versions of the software. Since header files are just a collection of functions and procedures nothing really stops you from creating your own. The only change comes when you are referring to these in your programme. Instead of using angle brackets, you would enclose the name of the header file in quotes “”. This tells the computer, rather, the pre-processor not to look for the set a file in the system area call but rather to look for it in the project directory. Other than that, this user defined files work exactly the same way as standard library files.

C standards

When C started getting popular, a lot of different versions of it started sprouting. People were creating library files that were suited for their needs and this hurts the portability of the programs around 1984, the first sees standard was defined. Today, there are three main standards for C, namely ANSI, POSIX and ISO. these standards define the way in which things are done in the programming language, from syntax, semantics, keywords right up to the names and functions of the header files that are included in the standard library. You'll see that different standards contain different types of files and behave slightly differently from each other. Let's briefly look at each of them.

ANSI

this was the first C standard to be developed as the world moved towards standardising C this was done in 1983 Pi 3 American National standards Institute, which produced its first draught in 1986. ANSI C is supported by in most widely used compilers such as GCC and Clang. Any source code written only in standard C and without any hardware dependent assumptions is virtually guaranteed to compile correctly on any platform with a conforming C implementation. C85 and C90 Where developed under ANSI C. As C became more popular, in 1990 fantasy was eventually merged into ISO C, which is the major international see standard today. Nearly old major compilers support the standard and we're going to look at it closely in a bit.

ISO

in 1990 ansi C was adopted by the international standards organisation ISO. from then on, this became the most widely accepted C standard. this standard is revised quite regularly and has had several iterations to date. From 1990 iterations such as see 99 see 11 and c18 where release test revisions to this standard. These introduced minor changes to the way see syntax is interpreted. C18 is the latest revision of the C standard. The standard specifies 5main aspects covered in the standard, namely the representation of C programs.

- the syntax and constraints of the C language.

- the semantic rules for interpreting C programs.

- the representation of input data to be processed by C programs.

- the representation of output data produced by C programs.

These same rules apply to all standards.

POSIX

The POSIX standard library contains quite a lot of header files that are not available in the C standard library. The POSIX specification includes header files that are used for multithreading regular expressions and networking. Most of these header files at developed with POSIX compliant systems in mind. in case you're wondering, the acronym POSIX stands for portable operating system interface, which is a family of standards specified by the Institute of electrical and electronics engineers (IEEE) for maintaining compatibility between operating systems. We are not going to go far in the direction as that requires an entire course to explain! Back to our topic of interest, pose X is a super set of the standard C library. The POSIX standard was developed at around the same time as the ANSI standard. A lot of effort was made to make the post extended compatible with this standard C, which was widely viewed to be the ANSI standard. The general aim of most of the head files in the POSIX standard is to provide lower levels support for the particular kind of operating system environment. The POSIX standard is typically used on Linux and Unix like environments which is where most of the header files included standard work. In fact, most of the header files in the POSIX standard are designed and are highly optimised full Linux and Unix like systems and have little or no use on Windows and other systems. For this reason, very few of the POSIX header files ever made it to the standard C library. A lot of people get confused when they see the exact same files from the see standard library defined in the POSIX standard. These files are known to overlap. This is because these files are specifically written with POSIX compliant systems in mind and during development whenever there was too much of a similarity, compatibility was built into files belonging to the C standard library. This means that code written in the international standard C can execute just fine on a POSIX system, but POSIX compliant code may have a hard time executing on ISO C.

Conclusion

In this lesson, we looked at the libraries that are included in the C programming language. We also looked at what’s really inside the header files, as well as the standards that C complies to. These standards are critical to the way in which your program will be compiled. As we mentioned, certain header files are included in each of the standards according to the target systems for each standard . in our next lesson, we will look at variable arguments, engineers our very own many project that we will be working on up until the end of the course . exciting news! until then remember, practice makes perfect. See you next time!

References

Pluralsight.com. (2020). Preprocessor Directives in C# | Pluralsight. [online] Available at: https://www.pluralsight.com/guides/preprocessor-directives-in-c [Accessed 21 Jan. 2021].

Tenouk.com. (2018). The C and C++ standards information: ISO/IEC, ANSI, POSIX, Single Unix Specification, ANSI C, GNU and cert.org Secure C/C++ coding. [online] Available at: https://www.tenouk.com/cplusstandard.html [Accessed 21 Jan. 2021].

ISO - International Organization for Standardization (2018). ISO/IEC 9899:2018. [online] ISO. Available at: https://www.iso.org/standard/74528.html [Accessed 21 Jan. 2021].

Saks, D. (2003). Abstract types using C - Embedded.com. [online] Embedded.com. Available at: https://www.embedded.com/abstract-types-using-c/ [Accessed 21 Jan. 2021].

Gnu.org. (2021). Pragmas (The C Preprocessor). [online] Available at: https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html [Accessed 21 Jan. 2021].

Narkive.com. (2013). [Dev-C++] #pragma directive. [online] Available at: https://dev-cpp-users.narkive.com/AFGf6dty/dev-c-pragma-directive [Accessed 21 Jan. 2021].

Blekhman, A. (2008). What Every Computer Programmer Should Know About Windows API, CRT, and the Standard C++ Library. [online] Codeproject.com. Available at: https://www.codeproject.com/Articles/22642/What-Every-Computer-Programmer-Should-Know-About-W [Accessed 21 Jan. 2021].

Msspace.net. (2021). APIs, POSIX, and the C Library. [online] Available at: http://books.msspace.net/mirrorbooks/kerneldevelopment/0672327201/ch05lev1sec1.html [Accessed 21 Jan. 2021].

Cprogramming.com. (2011). c preprocessor output issue with comments after #include. [online] Available at: https://cboard.cprogramming.com/c-programming/128021-c-preprocessor-output-issue-comments-after-sharpinclude.html [Accessed 21 Jan. 2021].