# Chapter 2. Stages of C++ programme code processing

## 2.1 Preprocessor Directives

The preprocessor is a word processor that manages the text of the source code file during the first stage of translation. The preprocessor does not analyse the source text, but breaks it down into tokens to detect macro calls. Although the compiler usually calls the preprocessor on the first pass, the preprocessor can also be called separately to handle the text without compiling it.

A directive (instruction), in the way it is used, is similar to a command in that it is also used to describe some programming language constructs, that is, to tell the compiler the specifics of processing during compilation. The C++ programming language has built-in preprocessor support. The lines in the source code that must be processed by the preprocessor in the form of #define and #include are called preprocessor directives.

In the C++ programming language, the preprocessor is the part of the compiler that controls the shaping of source code into object code. The preprocessor has a set of commands called preprocessor directives, and markers where the "commands" – macros – are located. The marker divides the code into parts, and also serves as a flag to locate macros. A directive, on the other hand, is a command that forms a compiler script.

Preprocessor directives such as #define define an identifier and a sequence of characters that will replace that identifier in the program, and #ifdef will allow the contents of other files to be inserted into the source file or disable compilation of part of the file by deleting sections of text. Directives in the source file tell the preprocessor to perform certain actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or disable compilation of part of the file by deleting sections of text. Preprocessor strings are recognised and executed before macros are expanded. Thus, if a macro is expanded into something that looks like a preprocessor instruction, it is not recognized by the preprocessor. The preprocessor instructions use the same character set as the source file instructions, except that they are not supported. The character set in preprocessor statements is the same as the runtime encoding. The preprocessor also recognizes negative character values.

The preprocessor recognizes the following directives: #define, #endif, #import, #include, #pragma, #undef, #ifndef.

The lattice sign ( # ) must be the first character, not a space, in the string containing the directive. space characters may be between the number sign and the first letter of the directive. Some directives contain arguments or values. Any text following a directive (except for an argument or value that is part of the directive) must be preceded by a single-line comment delimiter ( // ) or enclosed in comment delimiters ( / \* \* \* / ). Lines containing preprocessor directives may be continued immediately before the end-of-line marker with a backslash ( \ ). Preprocessor directives can be anywhere in the source file, but they are applied only to the rest of the source file after they appear.

C++ programme text consists of tokens and spaces. A token is the smallest element in C++ that has a value to the compiler.

The C++ parsing tool recognizes the following types of tokens:

1. Keywords
2. Identifiers
3. Numeric, logical and pointer literals
4. String and character literals
5. User-defined literals
6. Instructions
7. Punctuation characters

Markers are usually separated by spaces, which may be one or more:

1. Blank values
2. Horizontal and vertical tab characters
3. Line feed characters
4. Web form feeds
5. Comments

The C++ standard specifies a basic source character set that can be used in source files. To represent characters outside of this set, additional characters can be specified using universal character names. The MSVC implementation allows the use of additional symbols. The basic source character set consists of 96 characters that can be used in source files. This set includes the space character, the horizontal and vertical tab character, the page feed and newline control characters, and the following set of graphic characters: a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 \_ \_ { } [ ] # ( ) <> % : ; . ? \* + - / ^ & | ~ ! = , \ " '

## 2.2 Basic preprocessor directives

Macros in C++ are instructions that the preprocessor processes before compiling the code. They allow textual substitution in source code, which can be useful for automating and simplifying some tasks (figure 2.1).

|  |
| --- |
|  |
| Figure 2.1 – Example of using the macro |

A token is a sequence of characters representing what we consider to be a unit of measurement, such as a number or an operator (figure 2.2).

|  |
| --- |
|  |
| Figure 2.2 - Example of C++ tokens |

The #define directive creates a macro which is an association of an identifier or parameterised identifier with a token string. Once the macro is defined, the compiler can substitute a token string for each detected identifier in the source file [11].

Syntax: #define token identifier value (if any).

And #define no token string removes occurrences of the identifier from the source file. The identifier remains defined and can be checked using the #if defined directive and #ifdef. figures 2.3, 2.4 show examples of usage.

|  |
| --- |
|  |
| Figure 2.3 – Example of using #define directive |

|  |
| --- |
|  |
| Figure 2.4 – Example of using the #undef directive |

The #if directive with the #elif, #else and #endif directives control the compilation of parts of the source file. If the expression after #if has a non-zero value, the group of lines immediately after the #if directive is stored in the conversion record [14]. Each #if directive in the source file must match the closing #endif directive. Any number of #elif directives may be used between #if and #endif directives, but no more than one #else directive is allowed. The #else directive must be the last directive before #endif.

These directives are needed for various debugging actions which should not be implemented through code writing. Figure 2.5 shows an example of their use.

|  |
| --- |
|  |
| Figure 2.5 – Example of using #if, #else, #endif directives |

The #error directive produces a user-specified error message during compilation and then terminates compilation: #error token-string.

This directive is most useful during pre-processing to notify the developer of program inconsistencies or constraint violations. The following example demonstrates processing errors during pre-processing.

The #ifdef, #ifndef directives as well as #if are used for checking, but not with a condition, but with a variable that is populated as a macro. Figure 2.6 shows an example of the construct.

|  |
| --- |
|  |
| Figure 2.6 – Example of using #ifdef, #ifndef directives |

If the DEBUG variable is recognised, information is output to the console, if no variable was found, but this block is not active.

The #include directive tells the preprocessor to include the contents of the specified file at the point where the directive is displayed.

Syntax:

#include "path – specification"

#include <path – specification>.

If the file name enclosed in double quotes is an incomplete path specification, the preprocessor first searches the parent file directory. The parent file is the file that contains the #include directive. For example, if you include a file named file2 in a file named file1, then file1 will be the parent file. #include is responsible for connecting various libraries written by other developers, which allow implementing new blocks and functionality in the code [22].

The #import directive is used to include information from the type library. The contents of the type library are converted into C++ classes, mainly describing the interfaces of the COM model.

Syntax:

#import "filename" [attributes].

#import <filename> [attributes] filename

The #line directive tells the preprocessor to set the reported compiler values for line number and filename to the given line number and filename.

Syntax: #line digit-Sequence ["filename"].

The line number and (optional) filename are used by the compiler to indicate errors it detects during compilation. The line number usually indicates the current line of input data, and the filename indicates the current input file. The line number is incremented by one after each line is processed. The value of the number sequence can be any integer constant in the range 0 to 2147483647 inclusive. macro substitution can be used for preprocessing tokens, but the result must have the correct syntax. The filename can be any combination of characters and must be enclosed in double quotes (" "). If the filename parameter is omitted, the previous filename remains unchanged. Figure 2.7 shows an example of usage.

|  |
| --- |
|  |
| Figure 2.7 – Example of using #line directive |

The #pragma directive eliminates the problem of repeated inclusion. When calling #include directives in specific two files, it is possible for two files to call each other, causing an error that is 1024 calls deep and an exception to occur. The #pragma directive solves this problem by checking to see if the file has been called before. If this #include has already been called, it will not be called again. You can also get around this problem through conditional compilation (#if, #elif, #else and #endif).

Syntax: #pragma string\_token.

The \_\_pragma keyword.

The compiler also supports the \_\_pragma keyword defined by Microsoft, which has the same functions as the #pragma directive. The difference is that the \_\_pragma keyword is a usable word embedded in the macro definition. An example is shown in figure 2.8.

|  |
| --- |
|  |
| Figure 2.8 – Example of using #pragma directive |

## 2.3 Compiling a programme

Composition of the g++ compiler:

cpp – preprocessor

as – assembler

g++ – compiler

ld – linker

A preprocessor is a programme that converts source code into code understandable to a compiler.

Assembler is a program translator that translates a programme from assembly language text into a machine language programme.

A compiler is a programme that translates source code in a programming language into machine code.

Linker is a programme that performs linking: it takes one or more object modules as input and builds an executable module using them.

What is the purpose of compiling source files?

A C++ source file is code, but it cannot be run as a programme or used as a library. Therefore, every source file needs to be compiled into an executable, dynamic or static library.

The source file to work with is driver.cpp, shown in figure 2.9.

|  |
| --- |
|  |
| Figure 2.9 – Source file driver.cpp |

Preprocessing is the very first stage of compiling a programme. A preprocessor (the term was mentioned earlier) is a macroprocessor that transforms a programme for further compilation.

A header (header file) is a file whose contents are automatically added by the preprocessor to the source text in the place where some directive is located.

Headers included in the program using the #include directive recursively go through the preprocessing stage and are included in the output file. Each header can be connected several times, so special preprocessor directives are usually used to prevent cyclic dependencies.

Let's get the preprocessed code into the output file driver.ii (C++ files that have gone through the preprocessing stage have the extension .ii) using the -E flag, which tells the compiler that it is not necessary to compile (more on this later) the file, but only to preprocess it: g++ -E driver.cpp -o driver.ii. A look at the body of the main function in the newly generated file shows that the RETURN macro has been replaced (figure 2.10).

|  |
| --- |
|  |
| Figure 2.10 – Generated driver.ii file |

In the new generated file, you can also see a huge number of new lines, these are various libraries and #iostream haser.

At this step g++ performs its main task – compilation, i.e., converts the directive-free code obtained at the previous step into assembly code. This is an intermediate step between a high-level language and machine (binary) code.

Assembly code is a human-readable representation of machine code.

Using the -S flag, which tells the compiler to stop after the compilation stage, we get assembly code in the output file driver.s: $ g++ -S driver.ii -o driver.s

You can still look at the obtained result. But in order for the machine to understand the code, we need to convert it into machine code, which will be obtained in the next step.

Since processors execute commands in binary code, it is necessary to translate assembly code into machine code using an assembler.

The assembler converts assembly code into machine code by storing it in an object file.

An object file is an intermediate file created by the assembler that stores a piece of machine code. This piece of machine code, which has not yet been linked together with other pieces of machine code to form the final executable programme, is called object code.

It is then possible to save this object code into static libraries so that this code does not have to be compiled again.

Let's get the machine code using assembler (as) into an output object file driver.o: $ as driver.s -o driver.o

But at this step we need to link the object files into a single executable file using a linker. So, let's move on to the next stage – linking.

The linker links all the object files and static libraries into a single executable file that we can run later. To understand how linking is done, we should tell you about the symbol table.

The symbol table is a data structure created by the compiler itself and stored in the object files themselves. The symbol table stores the names of variables, functions, classes, objects, etc., where each identifier (symbol) corresponds to its type and scope. The symbol table also stores addresses of references to data and procedures in other object files. It is with the help of the symbol table and the references stored in them that the linker will be able to further build links between data among many other object files and create a single executable file from them.

Let's get the driver executable file: $ g++ driver.o -o driver // you can also add other object files and libraries here.

The last stage to be passed by the program is to call the loader to load our program into memory. It is also possible to load dynamic libraries at this stage. The final result of all the steps is shown in figure 2.11.

|  |
| --- |
|  |
| Figure 2.11 – Final result |

## 2.4 Namespaces

Namespaces are used to logically group adverts and restrict access to them. The larger the programme, the more relevant the use of named spaces. The simplest use case is to separate code written by one person from code written by another. When using a single global scope, it is very difficult to form a programme from separate parts because of possible overlap and name conflicts. Using named scopes prevents access to facilities that are not needed. A namespace declaration has the format shown in figure 2.12.

|  |
| --- |
|  |
| Figure 2.12 – Announcement format |

Once a namespace is declared, its elements can be accessed. The name of the namespace can be omitted if the name is declared in the same namespace. The way a namespace is accessed is shown in figure 2.13.

|  |
| --- |
|  |
| Figure 2.13 – Announcement format |

An example of namespace declaration and usage is shown in figure 2.14.

|  |
| --- |
|  |
| Figure 2.14 – Example of namespace declaration and usage |

In order to improve the readability of a programme and avoid constantly specifying the name of a namespace when accessing its elements, the using directive is used. This directive can be used in two cases, which are shown in figure 2.15.

|  |
| --- |
|  |
| Figure 2.15 – Use cases of using directive |

In the first case, the entire namespace is connected. You can then use all elements of this namespace without explicitly specifying a scope.

In the second case, an individual namespace element is connected. After that, you can use the name without explicitly specifying an area.

An example of accessing all elements of a namespace is shown in figure 2.16.

|  |
| --- |
|  |
| Figure 2.16 – Example of access to all elements |

An example of accessing an individual element in a namespace is shown in figure 2.17.

|  |
| --- |
|  |
| Figure 2.17 – Example of access to an individual element |

The result of the programme execution: z = 2.

In the above example, the methods are connected by the using directive using the following lines: using Operations::Sub2;

These methods can then be called directly without using the operations:: prefix.

If you try to call an unconnected Add2() method, the compiler will generate an error message: identifier Add2 is undefined.

The global namespace defines the highest-level scope. Global variables, types, functions, etc. can be declared in this scope. To access the global namespace, you must use the scope extension operator ::. If there are matching names in the global namespace and in some local namespace, it is mandatory to use the :: operator to access the global name, figure 2.18 shows an example of using the operator.

|  |
| --- |
|  |
| Figure 2.18 – Example of access to an individual element |

The result of the programme execution: Pi = 3.1415, MathItems::Pi=3.1415.

If you try to implement the ShowPi() call in the above code, the compiler generates an error: ambiguous call to overloaded function. This is logical because it is unknown which function ShowPi() is being called to.

A namespace may be declared more than once, with subsequent declarations being treated as extensions of the previous ones. Thus, a namespace can be declared and modified outside of a single file, figure 2.19 shows an example of a repeated namespace declaration.

|  |
| --- |
|  |
| Figure 2.19 – Example of namespace re-declaration |

A named scope declaration may contain both declarations and definitions. It is logical to put only declarations in it, and to define them later using the area name and the scope access operator ::, an example is shown in figure 2.20.

|  |
| --- |
|  |
| Figure 2.20 – Example of a variable definition in a namespace |

In classical C++, the basic namespace is the std namespace. This namespace implements the tools of the C++ standard library. It implements thousands of functions that are useful when developing any programs. In previous versions of C++, the entire standard library was in the global namespace. Putting the standard library in the std namespace significantly reduces the possibility of name conflicts. To connect the std namespace the following line is used: using namespace std;

It is also possible to refer to std namespace elements by specifying their full name, for example: double a = std:: sqrt (9);

Namespaces can be nested. This means that one namespace can include another namespace. In this case, the nested namespace and its components can be accessed in the usual way using the :: operator. igure 2.21 shows an example of nested namespaces.

|  |
| --- |
|  |
| Figure 2.21 – Nested namespaces |

## 2.5 Localization

Localization is the process of adapting a product to a specific region or country.

Localization can affect:

1. translation
2. images, icons, colors
3. brand name
4. numbers
5. currencies and prices
6. time and dates
7. measurement values
8. telephone numbers
9. postal codes

An example of localization is shown in figure 2.22.

|  |
| --- |
|  |
| Figure 2.22 - Localization of the program |

A locale is a set of parameters that defines regional settings for the user interface.

A locale contains information about how to interpret and perform certain input/output and conversion operations in a geographically and language-specific manner in a particular environment.

It can be defined:

by user-defined data,

by geolocation,

through system configuration/

So, to set the locale in C++, there is a function setlocale, which sets or retrieves the runtime language standard.

It is defined by two parameters: category and locale.

The category parameter specifies the parts of the programme language standard information that are affected.

The locale parameter refers to a location (country, region, language) for which certain aspects of the programme can be configured.