Preprocessor

Raul P. Pelaez

October 9, 2025

Contents

1 Introduction

Today we will learn about the C preprocessor, a tool that allows us to manipulate our code before it is compiled. The preprocessor can be used to define macros, a use of the preprocessor that we can leverage to create compile-time rules (functions), and conditionally compile code. We will also learn about the `constexpr` keyword, which allows us to define constants that can be evaluated (and stored) at compile time.

The preprocessor is a separate program that the compiler runs on our code before proceeding to compilation.

Info

The compilation stages
Going from a source code to an executable (a source with a main function) consists of three stages that are run one after the other:

1. Preprocessing
2. Compiling
3. Linking

Advanced

When we run the compiler ourselves, like this:

```
$ clang main.c
```

The compiler is running these stages automatically for us. We can invoke each one separately if we want. For instance, we can ask clang (or gcc) to run the preprocessor like this:

```
$ clang -E main.c
```

The result will be a very long file in which there are no preprocessor directives (lines starting with `#`).

Compilation stages can be run separately like this:

1. Preprocessing: `clang -E main.c -o main.i`
2. Compiling: `clang -c main.i -o main.o`
3. Linking: `clang main.o -o main`

2 Preprocessor directives

Info

Preprocessor directives

Any line starting with `#` is a line that will be interpreted by the preprocessor. These lines are called preprocessor directives. We use them to define macros, include other files, and conditionally compile code.

Keep in mind that preprocessor directives are not part of the C language, they are interpreted by a separate program before compilation.

Some common preprocessor directives are:

- `#include <file>` or `#include "file"`: includes the contents of another file. The angle brackets are used for system headers, while the quotes are used for user headers.
- `#define NAME value`: defines a macro. The macro `NAME` will be replaced by `value` everywhere it appears in the code.
- `#undef NAME`: undefines a macro.

Example

```c
// File pre.c
#include <stdio.h>
#define PI 3.14159
#define print my_printf

void my_printf(const char* msg, float value){
  printf("Custom print: ");
  printf(msg, value);
}

int main(){
  print("Value of PI: %f\n", PI);
  print("Value of PI squared: %f\n", PI*PI);
  #undef PI
  // printf("Value of PI: %f\n", PI); // This would cause a compilation error
  return 0;
}
```

Output

```
Custom print: Value of PI: 3.141590
Custom print: Value of PI squared: 9.869588
```

In this example, we define a macro `PI` with the value `3.14159` and a macro `print` that expands to `my_printf`. Both definitions are simply text substitutions, which is why we are able to define `print` as `my_printf` even before we define the function itself. After defining `print`, every time the preprocessor

sees the text `print` used as a symbol, it will replace it with `my_printf`. This is why we can call `print` in the main function, and it will actually call `my_printf`. Lets call the preprocessor explicitly and inspect the last lines of the output:

```
clang -E pre.c | tail -n 13
```

Output

```
void my_printf(const char* msg, float value){
  printf("Custom print: ");
  printf(msg, value);
}

int main(){
  my_printf("Value of PI: %f\n", 3.14159);
  my_printf("Value of PI squared: %f\n", 3.14159*3.14159);


  return 0;
}
```

Inspect this output closely. We can see that the preprocessor has replaced all occurrences of `print` with `my_printf` and all occurrences of `PI` with `3.14159`.

Insight

This is but my personal opinion but, in general, I would avoid using the `#undef` directive. It can lead to confusion and bugs if we are not careful. If we need to change the value of a macro, it is better to use a different name.

Advice

Prefer using const variables over macros
In general, it is better to use `const` variables instead of macros for defining constants. This is because `const` variables are type-safe and have a scope, while macros are not.

2.1 Conditional compilation

We can use preprocessor directives to conditionally compile code. This is useful for including/excluding code based on certain conditions, such as the target platform or the presence of certain features. We can use the following directives for conditional compilation:

- `#if condition`: compiles the following code only if the condition is true (non-zero).
- `#ifdef NAME`: compiles the following code only if the macro `NAME` is defined.
- `#ifndef NAME`: compiles the following code only if the macro `NAME` is not defined.
- `#elif condition`: like `#else if`, used after an `#if` or `#ifdef` directive.
- `#else`: compiles the following code if none of the previous conditions were true.

All these directives must be closed with an `#endif` directive.

Here is an example of conditional compilation:

```
#include <stdio.h>
#ifndef DEBUG
#define DEBUG 1
#endif

int main() {
  #if DEBUG
    printf("Debug mode is on\n");
  #else
    printf("Debug mode is off\n");
```

```
  #endif
  return 0;
}
```

In this example, if the macro `DEBUG` is not defined, it will be defined to 1. Then, the code inside the `#if DEBUG` block will be compiled, printing "Debug mode is on". If we change the value to 0, the else block will be compiled instead, printing "Debug mode is off".

2.2 Defining preprocessor values during compilation

The previous example states that if the macro `DEBUG` is not defined, it will be defined to 1. But how can it be defined before that line? We can define macros during compilation using the `-D` flag. For instance, we can compile the previous code like this:

```
$ clang -DDEBUG=0 main.c -o main
```

This will define the macro `DEBUG` to 0, and the else block will be compiled.

2.2.1 CMake integration

If we are using CMake, we can define macros using the `target_compile_definitions` command:

```
add_executable(my_program main.c)
target_compile_definitions(my_program PRIVATE DEBUG=0)
```

This will define the macro `DEBUG` to 0 when compiling the target `my_program`. The `PRIVATE` keyword means that the definition is only used when compiling the target, and not when compiling other targets that link to it.

Info

Conditionals in CMake

CMake also has its own conditional directives that can be used in `CMakeLists.txt` files. Their syntax is eerily similar to the C preprocessor. For instance:

```
if(DEFINED DEBUG)
    message("Debug mode is on")
    # Set the DEBUG macro for the target my_program to 1
    target_compile_definitions(my_program PRIVATE DEBUG=1)
else()
    message("Debug mode is off")
endif()
```

We can set the variable `DEBUG` in CMake using the `-D` flag when invoking CMake:

```
$ cmake -DDEBUG=ON -B build
```

This will set the variable `DEBUG` to true, and the message "Debug mode is on" will be printed.

3 Macros

Macros are a way to define compile-time functions using the preprocessor. They are defined using the `#define` directive and can take arguments. When the macro is used, the preprocessor replaces the macro call with the macro definition, substituting the arguments. Here is an example of a simple macro that squares a number:

```
#define SQUARE(x) ((x)*(x))
int main() {
  int a = 5;
  int b = SQUARE(a);
  printf("%d\n", b); // Prints 25
  float c = 1.0/SQUARE(2.0);
  printf("%f\n", c); // Prints 0.25
  return 0;
}
```

In this example, we define a macro `SQUARE(x)` that takes one argument and returns the square of that argument. When we call `SQUARE(a)`, the preprocessor replaces it with `((a)*(a))`, and when we call `SQUARE(2.0)`, it replaces it with `((2.0)*(2.0))`.

Info

A value is also called a macro
In general, a macro is any definition made using the `#define` directive. This includes both macros that take arguments (like `SQUARE(x)`) and macros that do not take arguments (like `PI`). The latter are often called "values" or "constants", but they are still macros.

Warning

Beware of the preprocessor
It is tempting to overuse it, as at first it might seem like a quick solution to many problems. However, it is a dangerous tool which, in essence, allows us to do text substitution before compilation and "skip" the rules of the C language (such as the type system). As such, it is easy to shoot ourselves in the foot and create mega nasty bugs.
Imagine that you anticipate the need of squaring values of many different types (say `int`, `float`, `double`, etc.). The rules of the language do not allow to create a "generic" function that works for any type. We are left with two options: write an almost identical function for each type, or use a macro, which is simple text substitution.
See this simple example:

```c
#define SQUARE(x) x*x
int main() {
  int a = 5;
  int b = SQUARE(a + 1);
  // Expands to a + 1 * a + 1 = a + (1 * a) + 1 = 5 + 5 + 1 = 11
  printf("%d\n", b); // Prints 11 instead of the expected 36
  float c = 1.0/SQUARE(2.0);
  // Expands to 1.0/2.0*2.0 = (1.0/2.0)*2.0 = 0.5*2.0 = 1.0
  printf("%f\n", c); // Prints 1.0
  return 0;
}
```

The macro `SQUARE` is defined without parentheses around the argument and the whole expression, leading to unexpected operator precedence when used. The correct definition should be:

```c
#define SQUARE(x) ((x)*(x))
```

A function would have been type-safe and would not have suffered from this problem. We skipped the rules of the language to overcome a limitation, but that comes with some additional responsibility.

3.1 Special macros and definitions

Compilers make some special macros and definitions available to us. Some of the most common ones are:

- `__FILE__`: expands to the name of the current file as a string literal.
- `__LINE__`: expands to the current line number as an integer literal.
- `__DATE__`: expands to the current date as a string literal in the format "Mmm dd yyyy".
- `__TIME__`: expands to the current time as a string literal in the format "hh:mm:ss".
- `__func__`: expands to the name of the current function as a string literal.
- `NDEBUG`: if defined, disables the standard `assert` macro.

Here is an example of using some of these macros:

```c
#include <stdio.h>
#define error_msg(msg) \
  printf("Error in function %s, line %d: %s\n", __FUNCTION__, __LINE__, msg)
```

```c
int main() {
  printf("This file was compiled on %s at %s\n", __DATE__, __TIME__);
  error_msg("Something went wrong in main");
  return 0;
}
```

Output

```
This file was compiled on Sep 29 2025 at 14:56:23
Error in function main, line 17: Something went wrong in main
```

Advanced

Variadic macros

We can define macros that take a variable number of arguments using the `...` syntax. The special identifier `__VA_ARGS__` is used to represent the variable arguments in the macro definition. Here is an example of a variadic macro that prints an error message with a variable number of arguments

```c
#define log_error(fmt, ...) fprintf(stderr, "Error: " fmt "\n", __VA_ARGS__)

int main() {
  log_error("Failed to open file %s", "data.txt");
  log_error("Some numbers %d %d %d %d", 1,2,3,4);
  return 0;
}
```

In this example, the macro `log_error(fmt, ...)` takes a format string and a variable number of arguments. When we call the macro, the preprocessor replaces it with a call to `fprintf`, passing the format string and the variable arguments.

3.2 Multiline macros

We can define macros that span multiple lines using the backslash (\) character at the end of each line. Here is an example of a multiline macro:

```c
#include <stdio.h>
#define VERY_LONG_MACRO(x) {                          \
    printf("This is a very long macro\n");            \
    printf("It takes one argument: %d\n", x);         \
    printf("And it does multiple things\n");          \
  }
int main() {
  VERY_LONG_MACRO(42);
  return 0;
}
```

Output

```
This is a very long macro
It takes one argument: 42
And it does multiple things
```

Advice

Macros vs functions

In general, prefer using functions over macros. Functions are type-safe, have a scope, and are easier to debug. Macros should be used sparingly, only when we need to do something that cannot be done with functions, such as defining compile-time constants or creating generic code that works with multiple types.

4 Constexpr

The `constexpr` keyword is used to define constants that can be evaluated at compile time. This is useful for defining values that we want to use in contexts that require compile-time constants, such as array sizes or template parameters. Here is an example of using `constexpr`:

```c
#include <stdio.h>
constexpr double pi = 3.141592653589793;

int main() {
  printf("Value of pi: %f\n", pi);
  printf("Value of pi squared: %f\n", pi * pi);
  return 0;
}
```

Output

```
Value of pi: 3.141593
Value of pi squared: 9.869604
```

Use `constexpr` instead of `#define` for defining constants. This is because `constexpr` variables benefit from all the features of the C type system, while macros are simply text substitutions.

Insight

The `constexpr` was introduced in C++11 and was not available in C until C23. This is not the first instance of C++ features being added to C. As usual in these cases, the C version is a limited subset of the C++ version. In this case, C++ allows us to define `constexpr` functions, while C only allows us to define `constexpr` variables.

Advice

Why bother with constexpr if we have const?
In general, we want to give as much information as we can to the compiler and the readers of our code. The `const` keyword tells the compiler that a variable will not be modified after its initialization, but it does not guarantee that the value is known at compile time. The `constexpr` keyword, on the other hand, guarantees that the value is known at compile time. This allows the compiler to perform optimizations and use the value in contexts that require compile-time constants. For instance, knowing that the variable is known at compile time, the compiler might decide to replace all occurrences of the variable with its value, avoiding the need to allocate storage for it and making the use of that variable faster.

5 Exercises

5.1 Preprocessor 101

Goal

A language within a language
You have now been introduced to the idea that C is actually two languages: the C language and the preprocessor language. Lets play around with this preprocessing step.
Lets go step by step. We will start from a simple C file that we will compile in the command line to a project with CMake.

Milestone

Create a C file that includes the standard header `stdio.h` and has a main function that prints "Hello, World!" to the standard output. Compile and run this file using the command line.

hint

- Use the `printf` function to print to the standard output.
- Compile the file using `cc hello.c -o hello` and run it using `./hello`. Remember to activate a conda environment first.

Milestone

Your first macro

Instead of printing "Hello world", lets assume our program must be compilable to print messages in other languages. Lets start by defining a macro called `HELLO_EN` that contains the message to be printed. Use this macro in the `printf` function to print the message.

hint

- Define the macro using the `#define` directive. Place it below your includes.

Insight

In real life, it is probably a bad idea to deal with internationalization using macros in this way, so take this as just an exercise to practice with the preprocessor.

Milestone

Adding a second language with conditional compilation

Now, add a second macro called `HELLO_ES` that contains the message in Spanish ("Hola, Mundo!"). Add a second `printf` call that prints the message in Spanish.

Now the program will print both messages, but we want to be able to choose which message to print at compile time. In order to do that, we will use conditional compilation, so that if the macro `LANGUAGE` is defined to `EN`, the program will print the message in English, and if it is defined to `ES`, it will print the message in Spanish. If the macro is not defined, the program should print an error message and exit with a non-zero status.

- Define a default value for the `LANGUAGE` macro if it is not defined.
- When compiling the program, define the `LANGUAGE` macro using the `-D` flag. For instance, to compile the program to print the message in Spanish, use `cc -DLANGUAGE=ES hello.c -o hello`.

hint

- Use the `#if`, `#elif`, and `#else` directives to implement the conditional compilation.
- Use the `#ifndef` directive to check whether a macro is NOT defined.

Milestone

From demo to project 1

Now you have a small "proof of concept" program that uses the preprocessor. Lets turn this into a proper project.

- Start by moving the message choosing code to a function called `get_welcome_msg`. Make sure that the program still works as before. You should be able to call this function like this in the main function:

```
const char* msg = get_welcome_msg();
printf("%s\n", msg);
```

  Insight

  We could had called the function `print_welcome`, but then we would not be able to use the message for anything else, such as logging it to a file. More importantly, by designing the function in this "do one thing" manner, we are making it easier to test.

- Now, separate the function declaration and definition into a header file and a source file, respectively. Create a directory called `include` for the header file and a directory called `src` for the source files. The header file should be called `welcome.h` and the source file should be called `welcome.c`, the file with the main function should be called `main.c`.
  For instance, these would be the contents of `welcome.h`:

```
#pragma once
const char* get_welcome_msg();
```

  Info

  The `#pragma once` directive is a non-standard but widely supported way to prevent multiple inclusions of the same header file. It is simpler and less error-prone than using include guards.

- Make sure you can compile the program using the command line, like this:

```
$ cc -Iinclude src/welcome.c src/main.c -o welcome
```

  Try it also with the `-D` flag to define the `LANGUAGE` macro.

  Info

  The `-I` flag tells the compiler to look for header files in the specified directory.

Milestone

From demo to project 2

Instead of relying on the command line to compile our program, lets use CMake. Create a `CMakeLists.txt` file in the root of the project. It should add welcome as a library and main as an executable that links to the welcome library.

- Look up how to define macros in CMake and use that to define the `LANGUAGE` macro when compiling the welcome library. You should now be able to compile the program using CMake, like this:

```
$ cmake -DLANGUAGE=EN -B build # configure
$ cmake --build build # build
$ ./build/main # run
```

Milestone

From demo to project 3

Before we can call this project done, we need to add a few finishing touches:

1. Add a `.gitignore` file to ignore the `build` directory.
2. Add a `README.md` file with a brief description of the project and instructions on how to compile and run it.
3. Add an `environment.yml` file with the dependencies (`cmake`, `c-compiler` and `make`).
4. Add unit testing using `utest`. You can just copy the `utest.h` file from the course example repository and place it into the `test` folder. Integrate it into CMake.

5.2 Super assert

Goal

`supper_assert`

This second exercise will be much less guided. We will create a new version of `assert`, called `super_assert`. This macro enhances the standard `assert` by adding some configurable features:

1. When the macro `SUPER_ASSERT_TRACE` is defined to 1, the macro will always print, even if the condition is true.
2. When the macro `SUPER_ASSERT_PERMISSIVE` is defined to 1, the macro will not abort the program when the condition is false. Instead, it will print a warning message and continue execution.
3. If the macro `NDEBUG` is defined, the macro will do nothing, regardless of the other macros.
4. If the macro `SUPER_ASSERT_VERBOSE` is defined to 1, `super_assert` will print, before each message, the file name, line number, and function name where it was called from.

Here is an example of how the macro should be used:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "super_assert.h" // This is where you must define the macro

int main(){
  int x = 5;
  super_assert(x == 5, "x should be 5");
  super_assert(x != 5, "x should not be 5");
  return 0;
}
```

Depending on the macros defined, the output of this program should change. For instance, if we define `SUPER_ASSERT_TRACE` to 1, the passing assertion will also print a message. If we define `SUPER_ASSERT_PERMISSIVE` to 1, the failing assertion will print a warning message and continue execution instead of aborting the program. If we define `NDEBUG`, nothing will be printed, and the program will run without any checks.

Advanced milestone

Tweak the filename that you get with `__FILE__` so that only the base name is printed (i.e., without the directory path).

hint

- You can use the `strrchr` function from `string.h` to find the last occurrence of a character in a string.