Seminar: EPC-1314

Thema: Präprozessor

Henrik Friedrichsen

February 6, 2014

Abstract

This document is the literal elaboration accompanying the presentation I have delivered in the seminar „Effiziente Programmierung in C".

It will cover a brief introduction to the C preprocessor syntax as well as the pitfalls that a developer may encounter when programming with the help of preprocessor statements.

Additionally, the performance gain achieved by inlining code with preprocessor statements is evaluated.

While the preprocessor is not restricted to a programming language, the context in this document will be limited to the programming language *C*.

Contents

1 Introduction

A *preprocessor* is a simple tool supporting a limited set of instructions to manipulate text. In this case specifically the statements are used to manipulate C code.

As the name suggests, the preprocessor is invoked *before* the actual code processor (in our case the C compiler). This is further illustrated in figure 1.

The statements accepted and interpreted by the C preprocessor are not actual C code, but instead have their own syntax.

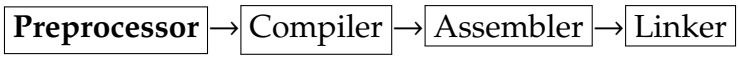

Figure 1: Typical order of compilation processes

The C preprocessor as described in this document is part of the ISO C standard [1].

Further documentation as well as the implementation in the GNU compiler collection of said standard is to be found in the *C Preprocessor manual* as part of the GCC documentation. [2]

2 Syntax

2.1 File inclusion

It is possible to include the whole content of a file with a preprocessor statement called `#include`.

For instance, the statement `#include <stdio.h>` would be replaced by the contents of the file *stdio.h* located in the include path (as specified by defaults and the -I compiler flag).

The statement `#include "localfile.h"` is analog to this, except the search path is not the include path, but the local directory of the file this statement was written in.

Paths wrapped in smaller/greater than signs instruct the preprocessor to search the include path, whereas paths wrapped in quotation marks instruct it to search the local path.

2.2 Conditional compilation

The C preprocessor supports the interpretation of simple conditions.

An example of a common case is demonstrated in figure 2. It includes *sys/socket.h* when compiled for a Linux system and *winsock.h* when compiled for a Windows system.

This is useful because Windows and Linux have different socket APIs and therefore require different headers.

```
#ifdef __LINUX__
        #include <sys/socket.h>
#elif _WIN32
        #include <winsock.h>
#else
        // other platforms
#endif
```

Figure 2: Conditional operating system dependent compilation

The identifiers `__LINUX__` and `_WIN32` are predefined depending on the platform.

Other predefined macros as part of the ISO/ANSI C standard[3, ISO Standard Predefined Macros] are:

- `__DATE__`: the compilation date (string literal)
- `__FILE__`: the name of the file compiled (string literal)
- `__LINE__`: the current line number (integer)
- `__TIME__`: time of compilation (string literal)

2.3 Compiler instructions

The set of instructions includes statements to manipulate the compilers behavior. This can be particularly useful when a developer wants to pass information to a compiler *per-file*, without changing the build system. Most of these instructions are compiler specific and **not** standardized.

The `#error` instruction (standardized) instructs the compiler to throw an error (figure 3).

Another use-case could be to turn off optimization for a specific file. When compiling with GCC this can be done with `#pragma GCC optimize ("-O0")` [4, Chapter 6.59.13].

```
#ifdef _WIN32
        #error "This program does not compile on Win32. Aborting."
#endif _WIN32
```

Figure 3: Instructing the compiler to throw an error when compiled for Windows

2.4 Macro definitions

2.4.1 Object-like macros

An object like macro will be substituted by <*replacement*> upon every reference. They are often used to store constants that are referenced multiple times in the code. In order to replace these constants throughout the whole code, only the macro definition needs to be changed.

The syntax is defined like this: **`#define`** `<identifier> <replacement>`.

An example is listed below.

```
// declare the macro object FACTOR set to the literal 5
#define FACTOR 5

int function() {
        int a = 5 * FACTOR;
        int b = a + 10;
        return b * FACTOR;
}
```

Figure 4: Macro object as a constant

Most compilers also accept a commandline parameter to declare such a macro. The switch usually used is `-D`*`identifier`*`=`*`replacement`*. [4, 3.1 Option Summary]

2.4.2 Function macro

Figure 5: Syntax to declare a macro function

```
#define <identifier>(parameters) <replacement>
```

Preprocessor macros can also be used to declare simple functions which even accept parameters.

References to such a function are replaced by the function body described in *<replacement>*. References to parameters in the function body are substituted by the expressions passed in the parameters, however such expressions are **not** evaluated.

For instance, if macro function `FOO(param)` is called with `calc(4, 5)` as the parameter *param*, every reference to *param* in the function body would be replaced by `calc(4, 5)` and not the result of `calc(4, 5)`.

This is very important to remember as it is a source of common error.

2.5 Other Operators

2.5.1 Line break

Macro definitions that are longer than one line have to be declared as such, otherwise the statements after the first line will not be interpreted by the preprocessor.

This is done by using a backslash (\).

```
#define PRINTMESSAGE printf("this is a macro definition "); \
        printf("exceeding the length of one line.")
```

Figure 6: Writing a macro definition over multiple lines

2.5.2 Stringification

An expression can be *stringified*, meaning it is converted to a string literal.

In this case the expression passed in *cond* is converted to a string literal and then printed, as well as the result of the evaluated expression.

```
#define LOG_CONDITION(cond) \
        printf("condition␣"#cond"is␣%i\n", (cond))

LOG_CONDITION((2 + 4 == 6))
```

Figure 7: Converting a macro function parameter to a string literal

2.5.3 Concatenation

To concatenate macro objects with other macro objects or literals two number signs are used.

In this example the literal `0x` will be prepended to the value passed in `v`.

```
#define HEX(v) (0x##v)

printf("value:␣0x%X\n", HEX(DEF));
```

Figure 8: Concatenating a literal with a macro object

3 Caveats

3.1 Double Evaluation

3.1.1 Description and Example

Due to the way the preprocessor is designed, parameters are *not* evaluated before they are referenced in the body of a preprocessor definition. The preprocessor also does not replace references to parameters with the result of the expression, but instead will evaluate the expression passed as the parameter every time.

This can result in undesired behavior, which may not always be that obvious.

The example below demonstrates such a case:

```
#include <stdio.h>

#define MAX(a, b) (a > b ? a : b)

int main(int argc, char* argv[]) {
        int apples = 11, kiwis = 12;

        printf("Maximum value MAX(kiwis=%i, apples=%i): %i\n",
                        kiwis, apples, MAX(kiwis, apples));

        printf("Add one more apple MAX(kiwis, ++apples): %i\n"
           ,
                        MAX(kiwis, ++apples));

        printf("Err.. what? Kiwis: %i, Apples: %i\n",
                        kiwis, apples);

        return 1;
}
```

Figure 9: Double evaluation

In this example two variables, *apples* and *kiwis*, are declared and initialized with 11 and 12 respectively.

The first `printf`-call prints the maximum of these two variables, which should be *kiwis* with 12.

In the second `printf`-call *apples* is incremented by one using the prefixed operator. *kiwis* and *apples* should now both be set to 12, however, **they are not**.

The last `printf`-call is used to print the two variables one last time.

Execution shows the following behaviour:

```
% ./double_eval
Maximum value MAX(kiwis=12, apples=11): 12
Add one more apple MAX(kiwis, ++apples): 13
Err.. what? Kiwis: 12, Apples: 13
```

Figure 10: Execution of the „double evaluation” example

3.1.2 Analysis

As shown in Figure 10 the second `printf`-call generates a result that might seem unexpected. It is not, though. As pointed out in the description parameters are not evaluated before they get referenced. Also, it will be evaluated with every reference. This is why the result might come as a surprise at first sight.

In our example we have two references to the parameter **b**, which is set to **++apples** and therefore the reason why *apples* is incremented twice. This becomes more apparent in the Figure 11 below.

```
MAX(kiwis, ++apples) in MAX(a > b ? a : b)
=> (kiwis > ++apples ? kiwis : ++apples)
```

Figure 11: Evaluation of the **MAX**-call

The condition of that ternary operator is **kiwis > ++apples**, effectively 12 > 12 (because *apples* is to be incremented by one), which is *false*. This leads to the execution of the *false*-statement, again incrementing *apples* by one.

Another problematic situation occurs when passing function calls as parameters. Just like in the example above the function call will be executed twice. If such a function modifies a global statement it will do so with every reference to the parameter, resulting in comparable behaviour.

3.1.3 Workaround

```
#define MAX(a,b) \
        ({ typeof (a) _a = (a); \
           typeof (b) _b = (b); \
           _a > _b ? _a : _b; })
```

Figure 12: Instantiation of parameters to avoid „double evaluation"

Figure 12 displays a workaround, which instantiates all the parameters in local variables. These variables are then referenced instead of the parameter-symbols. That way the statements passed as the parameters are only executed once.

Note: GCC offers an extension which introduces basic dynamic typing with a **typeof**-statement that is used in this example. It is **not** part of the C standard.

3.2 Operator Precedence

3.2.1 Description

Operator precedence is another area in which problems can occur due to the way parameters are handled in macro-functions. Consider the following definition of **CUBE**:

```
#define CUBE(x) x*x*x // also vulnerable to double evaluation
```

Figure 13: **CUBE** macro definition

Because the preprocessor only does simple text-substitution, inserting values that are more complex than a simple integer-literal can cause undesired results. Figure 14 shows two examples in which said problem occurs.

All references to **x** are simply replaced with the expression specified in the parameter without evaluating it. This results in a different order of operations than typically expected by some developers.

```
CUBE(2 + 2)   // Expected: (2+2)^3 = 64
=> 2 + 2*2 + 2*2 + 2 = 12

5*CUBE(4 - 3) // Expected: 5*((4-3)^3) = 5
=> 5*4 - 3 * 4 - 3 * 4 - 3 = -7
```

Figure 14: Two examples demonstrating problems with operator precedence

3.2.2 Workaround

To avoid such a situation a common workaround is to wrap references to the parameters in parentheses. That way the substituted references to the parameters will not interfere with the rest of the code in the body of a macro-function. This workaround and its implications are visible in Figure 15 below.

```
#define CUBE(x) ((x)*(x)*(x))

CUBE(2 + 2)   // Expected: (2+2)^3 = 64
=> ((2+2)*(2+2)*(2+2)) = 64

5*CUBE(4 - 3) // Expected: 5*((4-3)^3) = 5
=> 5*((4-3)*(4-3)*(4-3)) = 5
```

Figure 15: Workaround for operator-precedence related problems

3.3 General Forseeability

3.3.1 Description

In some cases it is not obvious what preprocessor code/definitions will be expanded to, especially when it comes to the introduction and closure of blocks (e.g. in if-statements).

Consider the following (Figure 16) example:

```
#include <stdio.h>

#define LOG_NULLPTR(x)\
        if(x == NULL) printf("is a nullptr!\n")

int main(int argc, char* argv) {
        void *foo = (void*)1;

        if(1)
                LOG_NULLPTR(foo);
        else
                printf("condition is not true!\n");
}
```

Figure 16: „Unexpected" macro expansion

The simple macro-function **LOG_NULLPTR** prints a message if the parameter *x* is **NULL**. However, the usage of *if* has implications to code where the macro is used, which is also an *if*-statement.

In this example the statement **if(1)** is always *true*, which would in turn always result in the execution of **LOG_NULLPTR**. The latter then checks whether *foo* is **NULL**, which it is not.

Therefore the program shouldn't output anything. Also the *else*-block is expected to be not executed.

Taking a look at the compiled code (figure 17) the implications of the macro become apparent.

```
void *foo = (void*)1;
if(1) {
        if(foo == NULL)
                printf("is a nullptr");
        else
                printf("condition not true!\n")
}
```

Figure 17: Code of figure 16 after macro expansion

Executing this will always yield the result *„condition not true!"*. This is because the *else*-block is now associated with the *if*-block from **LOG_NULLPTR**.

These effects are not immediately visible when looking at the code. It is therefore recommended to write macros with care so that they can be used without having effects on the environment they are included it.

3.3.2 Workaround

Below (figure 18) is a workaround to prevent such behaviour.

```
#define LOG_NULLPTR(x)\
        do {\
                if(x == NULL) printf("is a nullptr!\n"); \
        } while(0)
```

Figure 18: Workaround/safety measures

This is accomplished by enclosing the function-body in a *do-while*-loop that will only iterate once. The *if*-statement inside the loop - and the loop itself - will not affect the environment they are included in.

4 Force-Inlining

4.1 Introduction

Preprocessor macro-function calls are substituted by the code that is defined in the body of the macro-function. Additionally, references to the parameters will be replaced by the values passed in the function call.

Effectively, this means that preprocessor function-calls actually are not function calls, but instead pull in the code of the function-body. Therefore, „calling" a macro-function does not generate the overhead that a *regular* function call does. When repeatedly calling a function this overhead can have a significant effect.

Specifically, the following does **not** happen[1]:

- Pushing the parameters on the stack before changing the EIP
- Popping the parameters off the stack in the target function
- Jumping to the target function and back to the caller
- Pushing and popping the return address on/off the stack

Another way of doing this is by using the **`inline`** compiler keyword, which is also part of the C standard (since 1999).

Before that, developers had to rely on the use of macros to inline code. While the effects of the **`inline`**-keyword are similar to the usage of macro functions, they're are not entirely the same. To be precise, the **`inline`**-keyword is only a suggestion to the compiler (which it can respect or ignore), whereas the usage of macro-functions *always* inlines code, which is why this practice is also called *forced inlining*.

Since there was a separate talk on compiler keywords such as restricted, static & inline, the **`inline`**-keyword will not be covered any further in this document.

4.2 Example

In figure 19 below is a code example that displays variable behavior depending on whether it has been compiled with **`_INLINE`** defined or not.

With **`_INLINE`** defined the code that calculates the length of the vector will be inlined. If **`_INLINE`** is not defined a *real* function is declared, meaning a call to it will result in the execution of the function pro- and epilogue.

[1] It is to be noted that the generated machine code may vary depending on compiler, architecture, calling convention, etc.

The program then calculates the length of the vector (150.5, 200.5, 300.0) `INT_MAX` times, the maximum value a signed integer can hold, which is set to *2147483647* on my system.

```
#include <stdio.h>
#include <limits.h>
#include <math.h>

#ifndef _INLINE
        double veclength (double x, double y, double z)
        {
                return sqrt(x*x + y*y + z*z);
        }
#else
        #define veclength(x, y, z) sqrt((x)*(x) + (y)*(y) + (z
           )*(z))
#endif

int main(int argc, char* argv[]) {
        int i;
        for(i = 0; i < INT_MAX; i++)
                veclength(150.5, 200.5, 300.0);
        return 1;
}
```

Figure 19: Example to display performance difference between inlined and non-inlined code

4.3 Performance analysis

```
% gcc perf.c -lm
% time ./a.out
./a.out  23.50s user 0.00s system 99% cpu 23.513 total
% gcc perf.c -lm -D_INLINE
% time ./a.out
./a.out  7.92s user 0.00s system 99% cpu 7.931 total
```

Figure 20: Runtime of the inlined vs. non-inlined code

The inlined code ran approx. 7.9 seconds whereas the non-inlined code ran approx. 23.5 seconds.

It becomes apparent that the execution time has reduced considerably. The non-inlined binary runs around three times longer than the inlined. Repeated tests yielded the same results.

The reason for this is because the program does a lot of function calls to *veclength*, but since the code of *veclength* is inlined in the binary compiled with -D_INLINE, the function-call's *pro- and epilogue* are not necessary resulting in shorter execution time.

4.4 Review

As demonstrated in the example of figure 19, inlining code can shorten the execution time and/or improve the performance of a program.

It is, however, important to remember that the case demonstrated is constructed. Inlining code does not always optimize a program, it can even be destructive.

Another, possibly negative, side effect is that the size of the compiled code can increase, as the function calls are replaced by the function body.

A disadvantageous situation could arise with large functions that are to be inlined. In such a case the inlined code exceeds the instruction cache, resulting in *cache misses* and ultimately in performance decrease.

5 Conclusion

The preprocessor (when used correctly) can be a very handy tool to a developer.

By clever usage of macros/text substitution it can simplify work when writing instructions that are used repeatedly. It is even possible to optimize a program by inlining code with help of the preprocessor.

External programs can easily manipulate the code, e.g. to include or exclude bits that are (not) required by specific operating systems and/or architectures. This is commonly used by the *GNU autotools* or *Makefiles*.

References

[1] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[2] Free Software Foundation. The C preprocessor: Using cpp, the C preprocessor. `http://gcc.gnu.org/onlinedocs/gcc-4.8.2/cpp/`, 2013.

[3] Intel Corporation. Intel® C++ Compiler XE 13.1 user and reference guide, 2013.

[4] Free Software Foundation. Using the GNU Compiler Collection (GCC). `http://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/`, July 2013.