Lecture 15: The Preprocessor

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/15-preprocessor.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we discuss a particular tool that is executed by the compiler during the build process of a C or C++ program, and performing textual transformations in the code: the **preprocessor**.

Introducing the Preprocessor

The preprocessor is executed automatically by the compiler right before the source code is compiled into machine code. It performs **textual transformations** on the sources, so it takes as input the C source file and produces as output a modified version of these sources, i.e. something that is still C source code. After that step the compiler transforms the modified sources into an executable containing machine code.

The preprocessor is used for doing mainly 3 things:

- **Include header files**, we have actually been using the preprocessor to do this since the beginning of the course.
- **Expand textual tokens** named macros into larger and more complex bits of source code.
- **Conditionally enable or disable some code**.

Header Inclusion

Header files are these files ending with the `.h` extension. They generally contain what is necessary to use foreign libraries/source files. They contain functions prototypes, custom types/structs definitions, etc. For example `stdio.h` contains the prototype of `printf`, amongst other things.

There are two ways to include a header:

```c
// In the .c source file, include headers like that:
#include <stdio.h>            // Use <> to include a file from the default include path
#include "my-custom-header.h" // Use "" for the local and user-supplied include directories
```

We have been using the first method extensively. It will search for the specified header file in a standard location on the filesystem. An example of such standard location is `/usr/include`.

Another method is to use the double quotes. This will search for the specified header file in the local directory, in other words the directory from which we call `gcc` when we build the program. So here in the example we assume that there is a file named `my-custom-header.h` in the local directory.

The preprocessor will include the entire content of the header in the source file before compilation. Si in our example here, `#include <stdio.h>` gets replaced by the content of `stdio.h` content, and `#include "my-custom-header.h"` gets replaced by that file's content.

Macro Expansions

Macros are textual substitutions based on tokens. They are very useful for defining things like compile-time constants. Consider this example:

```c
#include <stdio.h>

// If we want to change the array size,
// we only need to update this:
*#define ARRAY_SIZE  10

int main(int argc, char **argv) {
* int array[ARRAY_SIZE];

* for(int i=0; i<ARRAY_SIZE; i++) {
    array[i] = i;
    printf("array[%d] = %d\n", i, array[i]);
  }

  return 0;
}
```

We have a macro named `ARRAY_SIZE`. We generally put constants in capital letters in C/C++. `ARRAY_SIZE` will be replaced by the preprocessor with `10`. The macro is used both in the array definition to set its size, and in the loop header iterating over the array, to set the number of iteration. The nice thing is that when we want to update the size of the array, one just needs to update the macro. If, on the contrary, `10` was hardcoded in the source code, there is a possibility of e.g. updating just the array size in the definition and then forgetting to update the loop header, which can lead to bugs.

In general, it is important to try to have a less hardcoded values as possible. When writing a constant value more than once, the programmer should consider if using a macro is possible.

Macro functions can also be defined[1], although this can be error prone[2].

Macro Expansions and Operator Precedence

It is important to consider operator precedence when using macros. Consider the following code:

```c
#define SIZE_1  10
#define SIZE_2  10

// We can use a macro in another macro's
// definition:
#define TOTAL   SIZE_1 + SIZE_2

int main(int argc, char **argv) {
    // faulty, expands to:
    // 10 + 10 * 2
    printf("total twice = %d\n",
            TOTAL * 2);
    return 0;
}
```

We have two macros, `SIZE_1` that is `10` and `SIZE_2` that is also `10`. We define a macro `TOTAL` that is the sum of both macros. Note that we can use a macro in another macro definition. We expect `TOTAL` to be 20. In the main function we multiply `TOTAL` by 2, so we expect the result to be 40. Because the preprocessor realised a textual transformation, `TOTAL * 2` actually expands to `10 + 10 * 2`. With the multiplication operator taking precedence over the addition one, we do not obtain the expected behaviour.

[1]https://www.ibm.com/docs/en/i/latest?topic=directive-function-like-macros
[2]https://gcc.gnu.org/onlinedocs/cpp/Macro-Pitfalls.html

To avoid issues with operator precedence, parentheses should be used in the definition of the macro:

```
#define TOTAL   (SIZE_1 + SIZE_2)
```

Overall, it is advised that if a macro is made of more than a single item, parentheses should be used.

Conditional Compilation

The preprocessor allows the conditional inclusion of code. Consider the following example:

```
#define DEBUG_MODE              // controls the activation/deactivation of debug mode
#define VERBOSITY_LEVEL 4       // controls the debug verbosity level

#ifdef DEBUG_MODE
int debug_function();
#endif

int main(int argc, char **argv) {
    printf("hello, world\n");

#ifdef DEBUG_MODE
    debug_function();
#endif
    return 0;
}

#ifdef DEBUG_MODE
int debug_function() {
    printf("This is printed only if the macro DEBUG_MODE is defined\n");

#if VERBOSITY_LEVEL > 3
    printf("Additional debug because the verbosity level is high\n");
#endif /* VERBOSITY_LEVEL */

    return 42;
}
#endif /* DEBUG_MODE */
```

We define two macros `DEBUG_MODE` and `VERBOSITY_LEVEL`. `DEBUG_MODE` is defined without a particular value. Its goal is to control the inclusion of debug code. Through the use of `#ifdef` and `#endif` directives, all the code enclosed will be included in the compilation only if the `DEBUG_MODE` macro is defined. In other words, if we comment the definition of that macro, all the enclosed code will not be included in the resulting executable.

We have another macro, `VERBOSITY_LEVEL`, defined with a value, `4`. With the `#if` directive we also include some code conditionally. We can use the `>` or `<` operators, but also the equality with `==` and inequality with `!=`. Note that we use a comment to detail which `#if` the `#endif` corresponds to, as preprocessor directives are generally not indented, to avoid any confusion.