# Appendix B

# The C Preprocessor and Useful Debugging Techniques

## B.1 Overview

This laboratory covers a number of different topics that are useful to know about when constructing, and particularly, troubleshooting (debugging) your C programs.

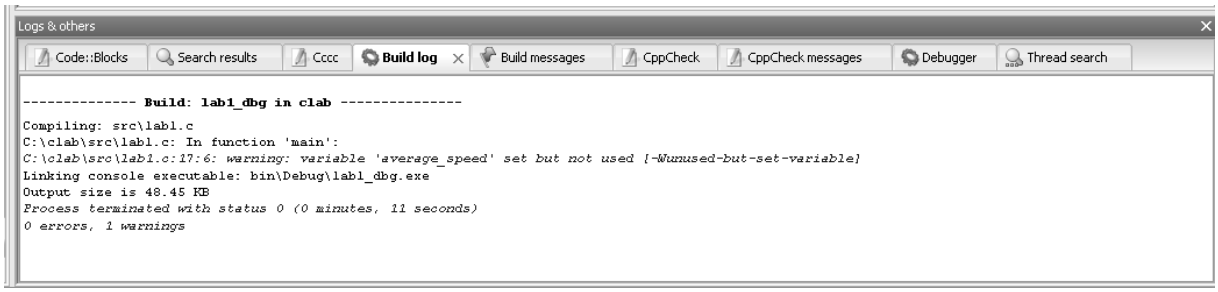
We will look at:

- a part of the compiler called the *preprocessor*, which deals with all of the lines in your code that begin with a hash (or sharp) symbol (#);
- how to use preprocessor commands to stop parts of your program from compiling;
- how to use conditional compilation for debugging;
- setting *breakpoints* and using them for debugging your program;
- finding the executable file that corresponds to your program and running it without the IDE.

Some of the tasks you do will be different depending on whether you are following the graphics or music option, but there are no special sections for the different options in this laboratory.

## B.2 Stages of Compilation

When you build a prgram, the output window shows messages generated by the compiler. If everything goes well, the output window in the Code::Blocks IDE looks like this:




From this window you can see the two major stages of compiling your programs (this is a reminder from the first lab):

- **compiling**, which produces an intermediate file (called an *object file*) for every source file related to your program;

- **linking**, which combines all of the object files with *library files* which contain the code for pre-existing functions such as `printf` to form an executable file: your program.

We are going to look at the compiling stage in a little bit more depth.

What we have been called the compiling stage can actually be split into two major parts:

- the **preprocessor** which processes the text in your source file, before it is passed on to...
- the **compiler** which does the work of translating your text written in C syntax into machine code object files.

The preprocessor takes, as input, a text file, which contains the C program that you have written. It produces, as output, another version of that text file, still in text, but expanded and altered slightly. For example, an important job that the preprocessor usually does is to remove all of your comments from the file so that the compiler never sees them.

The next few sections of this laboratory are going to focus on the preprocessor, to understand what it does and to be able to use it to our advantage.

## B.3 Introducing the C Preprocessor

Most of the time the C preprocessor silently goes about its work every time you build your program, for example stripping out comments as mentioned above. Sometimes you might want to communicate with the preprocessor directly. This is done by including preprocessor commands, or *directives*, in your program. All preprocessor directives begin with the hash (or sharp) symbol, #. Preprocessor directives are **not** statements, so they are not followed by a semicolon.

You have already used one preprocessor directive many times: `#include`. For example:

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

You might have noticed that there are two kinds of `#include` directive:

```
#include <filename>
```

and

```
#include "filename"
```

We will see what the difference is between these forms in a moment. In common with most preprocessor directives, the `#include` directive does not do anything very complicated. When the preprocessor sees a `#include` directive it looks for the file that have specified, for example `stdio.h`. When it finds the file it simply takes all of the text out of the file an inserts it into your source file in place of the `#include` directive. It does not try and process anything in the file, or try to understand it in any way. That is the compiler's job. The difference between enclosing the filename in angle brackets (like this `<stdio.h>`) and quotation marks (like this `"stdio.h"`) is in where the preprocessor looks to find the file you have specified.

- If the filename is enclosed in quotation marks then the preprocessor looks for the file in the current folder (where your source file is stored) before looking in any places that you have specified.
- If the filename is enclosed in angle brackets then the compiler assumes it is a standard library header file. Standard library header files describe functions that are part of the the C *standard library*: functions that are specified as part of the C language. This includes functions like `printf` and `scanf` in `stdio.h`, and `getch` in `conio.h`. It does not include any of the graphics or music functions.

So the preprocessor handles a `#include` by finding the file (looking where the angle brackets or quotation marks tell it to look) and substituting the *entire* contents of the file for the `#include` line.

**Syntax: The `#include` Directive**

`#include <filename>` or `#include "filename"`

The `#include` preprocessor directive is completed as part of the preprocessor, before the compiler is run. The directive identifies a file (*filename*), which is usually a header file, but can be anything. The directive takes the text from the header file and inserts it in place of the `#include` directive.

The angle brackets and quotation marks determine where the preprocessor looks for the file. As a general rule, angle brackets are for header files that are part of the C standard library (such as `stdio.h`) and quotation marks are for header files that are not (such as the graphics and music libraries). With most modern compilers (including the one you are using), using quotation marks instead of angle brackets (or vice-versa) will not stop your program from working.

## B.4 Using the Preprocessor for Text Substitution

A lot of Windows programs, like Microsoft Word, have the facility to search the text of a document to find a particular bit of text that you specify. This is called "Find". You may also have come across the facility (also in Word) to find bits of text and automatically replace them with another bit of text that you specify. This is usually called "Replace" and is an example of *text substitution*.

The C preprocessor has a facility for doing text substitution for you. It allows you to define keywords of your own which will get substituted for (usually more complicated) bits of text, which you specify. This is done using the `#define` preprocessor directive.

Open the "appendixa" program. This is a very simple example of how to use the `#define` directive for text substitution. The source code for "appendixa.c" is shown below.

```
/*
 *  A program to demonstrate the use of the #define directive
 *  C Programming laboratory appendixa
 */

/*  These lines allow the compiler to understand the
 *  printf and getch functions
 */
#include <stdio.h>
#include <conio.h>

/* An example use of the #define directive */
/* The preprocessor will replace HELLO_STRING, */
/* wherever it appears, with "Hello, World!\n" */
#define HELLO_MESSAGE "Hello, World!\n"

/* A numeric example, wherever PI appears */
/* it will be replaced by 3.14159265 */
#define PI 3.14159265

int main(void)
{
    /* Print a message to the screen */
    printf(HELLO_MESSAGE);
```

```
    /* Display pi */
    printf("Pi = %lf (roughly)\n", PI);

    /* Wait for a key press */
    getch();

    return 0;
}
```

If you build and execute the program you should find that it displays the following text on the screen:

```
Hello, World!
Pi = 3.141593 (roughly)
```

Remember that the preprocessor runs through the source code *before* it gets to the compiler. So the code that the compiler sees looks a bit like this:

```
...
int main(void)
{
    printf("Hello, World!\n");

    printf("Pi = %lf (roughly)\n", 3.14159265);

    getch();

    return 0;
}
```

Notice how the code was changed by the preprocessor.

Like the `#include` directive, a `#define` is also very simple. It is structured like this:

```
#define tag replacement
```

The `#define` directive tells the preprocessor: "from now on, everywhere you see the text *`tag`*, replace it with the text *`replacement`*."

### Syntax: The `#define` Directive

```
#define tag replacement
```

Starting from where the `#define` directive is written the preprocessor replaces an occurrence of the text *`tag`* with the text *`replacement`*.

For example, the directive:

```
#define E_CONST 2.71828
```

replaces any occurrence of the text `E_CONST` with the text `2.71828`.

It is generally considered good programming practice to use UPPERCASE names for `#define` tags to visually distinguish them from variable and function names.

`#define` directives are used to define replacement tags for two main reasons:

- To make your program more readable. For example:

```
circle_area = PI * radius * radius;
```

is a lot easier (and quicker) to read and understand than:

```
circle_area = 3.141592653 * radius * radius;
```

- To make your program easier to change. For example, it is common to use a `#define` directive for your program's version number. This allows you to put a `#define` directive clearly at the beginning of the source code, like this:

  ```
  #define PROGRAM_VERSION 2.3
  ```

  Somewhere in your source code, perhaps when the program starts, you might write:

  ```
  printf("Welcome to myprog Version %lf\n", PROGRAM_VERSION);
  ```

  When you make a change to your program, you don't have to go looking for all the places where you display the version number, you simply change the number in the `#define` directive at the top of the source file.

### Exercise B.1: Using the `#define` Directive

Add to the "appendixa" example program to:

- prompt the user for a number, the radius of a circle;
- store the number in variable of type `double`;
- calculate the area of a circle with the radius they entered using `PI`;
- display the result.

### Exercise B.2: Problems with `#define` Directives

You are now going to change the `#define PI` directive so that it is **wrong**. Change the line to be something like this:

```
#define PI error 3.14159265 error
```

This means that the text that will be substituted for the tag `PI` will be `error 3.14159265 error`, which is obviously not going to be valid C! Try compiling your program.

The compiler will give you an error message. Which line does the compiler think the error is on? Why is this?

If you don't understand this, ask one of the demonstrators to explain it to you. Remember to put the `#define` directive back to how was before you continue.

Choose one of the following two exercises depending on which option you have been doing.

### Exercise B.3: Adding a `#define` Directive: Graphics Option

Open the graphics program you have been working on, which should be called "graphics1". The horizontal location of the stick person is controlled by the user. The vertical location is fixed. Use `#define` directives to define tags for:

- the width and height of the graphics window. You could call these something like `WINDOW_HEIGHT` and `WINDOW_WIDTH`.
- the vertical location of the stick person. You could call this something like `PERSON_POS_Y`.
- the key number for the 'Enter' key (13). You could call this something like `ENTER_KEY`.

Have a look to see what other parts of your program you think would be better substituted for a `#define` tag. Good programming practice says that you should consider using `#define` tags whenever you have a fixed value in your program that you might want to change at compilation time (like the window size).

### Exercise B.4: Adding a `#define` Directive: Music Option

Open the music program you have been working on, which should be called "music2". Add a `program_change` function call for each channel you are using. The instrument number used should be defined as a `#define` tag. For example, the tag `PIANO` might be defined to be replaced by the number 1, like this

```
#define PIANO 1
```

Once you have done this add one or more `#define` directives like the following:

```
#define FIRST_CHANNEL_INSTRUMENT PIANO
```

You would then change the `program_change` function call to be:

```
program_change(1, FIRST_CHANNEL_INSTRUMENT);
```

Notice how you are using one `#define` tag to define another. This technique can be very useful in making your program easier to read, understand and change.

Have a look to see what other parts of your program you think would be better substituted for a `#define` tag. Good programming practice says that you should consider using `#define` tags whenever you have a fixed value in your program that you might want to change at compilation time (like instruments, maybe even the musical key).

## B.5 Using `#define` Directives for Conditional Compilation

In laboratory 2 we saw how to get parts of your program to execute only under certain conditions using an `if` statement. There is a similar preprocessor directive that allows you to conditionally *compile* parts of your program under certain conditions. Logically, the directive is called `#if`.

Close whatever program you have open. Change the target to "appendixa-a" and open "appendixa-a.c" in the program editor. You should see that the code is very similar to the example we looked at in the

first laboratory. It has some additional lines at the end to display the results, it also has this bit of code:

```
#if defined(DEBUG)

    printf("time_to_fly = %d, time_to_drive = %d\n", time_to_fly, time_to_drive);

#endif
```

What do you think this will do? Try building and executing the program to find out.

### Exercise B.5: Investigating Conditional Compilation (1)

Make the compiler ignore the line:

```
#define DEBUG
```

by enclosing it in a comment (enclose it between `/*` and `*/`). Build and execute the program. What has happened?

You should have seen that the code inside `#if...#endif` directives is only compiled under certain conditions.

The `#if` directive allows you to set conditions under which parts of your program will be removed by the preprocessor and therefore ignored by the compiler. For example:

```
#if 0
    ...
#endif
```

will **never** compile the code between the `#if` and `#endif` directives because 0 is treated as *false*. On the other hand, in this example:

```
#if 1
    ...
#endif
```

the code between the `#if` and `#endif` directives will **always** be compiled because a non-zero value is treated as *true*. It is not common to use `#if` directives with a constant value after them like this. It is more common to use them like this:

```
    #define ADD_EXTRA_FUNCTIONALITY 0
        ...
    #if ADD_EXTRA_FUNCTIONALITY
        ...
    #endif
```

This means that the compilation of the section of code between the `#if` and `#endif` directives is dependent on the value that the `ADD_EXTRA_FUNCTIONALITY` tag has defined as its replacement. In the case above, the code between the `#if` and `#endif` directives would **not** be compiled.

Rather than test the value of a `#defined` tag, it is more common to test whether a tag has been `#defined` *at all*. You have already seen how to do this in the example program.

### Syntax: The #if and #endif Directives

```
#if condition ...#endif
```

The `#if` directive tells the preprocessor to remove all the parts of your program until the next `#endif` directive, only if the *`condition`* is *false*. When testing the condition, a value of 0 is interpreted as *false*; a non-zero value is interpreted to be *true*.

The most common way to use the `#if` directive is with the special preprocessor syntax `defined(`*`tag`*`)`. This tests to see if the specified tag has been previously specified as part of a `#define` directive. It is used in the following way:

```
#if defined(SOME_TAG)
    ...
#endif
```

You can form more complicated tests using the logical operators `&&`, `||` and `!`. The *not* operator (`!`) is perhaps the most useful. For example:

```
#if !defined(SOME_TAG)
    ...
#endif
```

In this case the guarded code would only be compiled if the tag `SOME_TAG` is **not** `#defined`.

There are two very useful abbreviations for these simple operations:

```
#ifdef SOME_TAG    is short for   #if defined(SOME_TAG)
#ifndef SOME_TAG   is short for   #if !defined(SOME_TAG)
```

### Exercise B.6: Investigating Conditional Compilation (2)

Change the `#if` directive so that the code is compiled only if `DEBUG` is **not** defined. You will need to use the information in the syntax box about the `#if` and `#endif` directives to do this.

`#if` directives are very useful for including code (such as `printf` statements) which you only want to use for finding problems in your code (debugging). It means you can add lots of extra statements to your program and stop them from being seen by the compiler by removing a single `#define` line from the top. Even better, if you find similar problems again, you can add the lines back into your program again by putting just one `#define` line back.

Using `#if` directives for including special lines for debugging is a very useful technique in situations where stepping through your code is tedious. For example, inside a loop you could add an `if` statement that displays the value after the tenth iteration. The code could be removed from the final program by ensuring that you do not define a `DEBUG` tag.

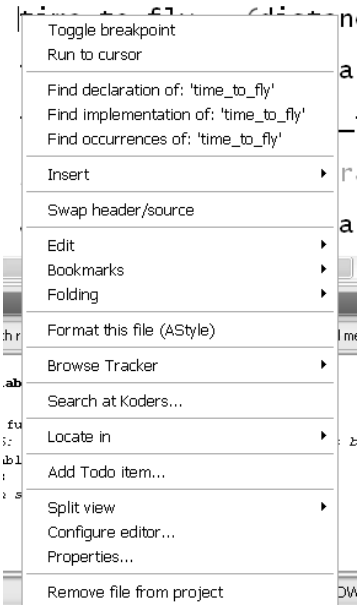
## B.6 Using Breakpoints for Debugging

Right from the beginning of this course we have used the IDE's debugging facility for stepping through your programs. If you are confident that all of the code at the beginning of you program works then stepping through it to get to the bit you are worried about can be very boring. If you have loops with a large number of iterations then it could take all day!

Fortunately, the debugger has a facility for you to be able to specify a line in your source code that you want to be able to stop on. To do this you mark a line in your code with a *breakpoint*.

Open a project you have been working on in previous labs. Choose something with a fairly substantial amount of code in it; a project from your option (either graphics or music) would be a good idea.

Pick a line of source code that will execute (i.e. not a comment) and click on it with the right-hand mouse button. You should see as menu like this one:




Select the option "Toggle Breakpoint".

You should find that the line in the source code that you clicked on now has a red dot at the right-hand side of the line.

This indicates that the line in the source code has a breakpoint on it.

Start the program running in the debugger by selecting "Debug" → "Start"

You should see that the program starts running, until it reaches the breakpoint. It then stops the program, but you have the chance to continue the program from where it is using the step commands or to continue running using the "Continue" option on the "Debug" menu. When the debugger stops on a breakpoint, it is exactly as if you had used the step commands to get there, just a lot quicker!

You can remove a breakpoint by clicking on the line of source code with the breakpoint on it using the right-hand mouse button. Then choose the "Toggle Breakpoint" option from the menu.

**Exercise B.7: Trying Out the Breakpoint Feature**

Now is your opportunity to try out breakpoints. Try setting them, removing them and using them for debugging. Breakpoints are very useful and you will probably find them invaluable when debugging your programs for your assessments.

## B.7 Locating the Programs You Have Written

All the programs you have produced so far do not `have` to be run in the IDE. In fact, they do not need the C source code or the C compiler to run once you have built them. If you sent your program to a friend you would only have to send them one file: the executable. Just to prove this point you are now going to try executing some of your programs without using the IDE.

First close the IDE; we won't be using it now. Now we have to find the executable file that was built by the compiler and linker from your C source code.

Open up the "clab" folder that you created in the first lab. You should now see a number of folders: "bin", "include", "lib", "obj", "src" and ".objs". Go into the "bin" folder. You will see two folders: "Debug" and "Release". Double click on the "Debug" folder to view its contents. You will see many files which has the same name as the targets in the IDE. It will have an icon a bit like this:


(This example is for the "graphics1" project). This is the executable that was produced when you built the "graphics1" or "music2" project. If you double click on it, the program will run. Try it.

**Exercise B.8: Running Your Programs from Executables**

Prove that some of your other programs can be executed this way by finding their executable files and double clicking on them.

It is important to note that if you were to give your program to someone else to try out, **this is usually the only file they would need**. Actually, if it is a midi program you may need the file portmidi.dll. The executable file is exactly the same type of file that executes when you run any program on your computer. You could copy this file onto any medium, like a memory stick or a CD, or even make it available over the internet. Anyone with access to just this file could run your program (on a MS Windows machine). Note: The program will only run if the operating systems is the same as the one you used when you compiled the program.

## B.8 Summary

Now that you have finished this laboratory you should understand the basic functions of the C preprocessor including the `#include`, `#define`, `#if` and `#endif` directives.

You should be able to use `#define` directives, both on their own, and in combination with `#if` and `#endif` directives for debugging.

You should have seen what a breakpoint does, and understand how to use them to debug your programs.

Finally, you should have seen where to find the executable file that corresponds to each of the programs you have written.