Programming Fundamentals  
Tutorial 09 - Debugging and using the Code::Blocks Debugger

### Introduction

Usually when you are developing software, the process of testing and fixing problems takes far more time than the actual writing of code. This process of finding defects in code and repairing them is called **debugging**, and it is a vital skill to have if you want to write good, solid programs.

A very basic way of determining what is happening in your code when it runs is to display values on screen or to an output file. While this is useful for basic diagnostics or when you are running a program standalone, it is quite limited in terms of what you can track and requires rewriting of code to output different variables. It is also of no help if the program logic is not functioning correctly and the display or recording code is not being called.

A better way in most cases is to use a **debugger**. This is a tool that most development environments provide that enables you to step through your code and track directly what is going on. It should also catch a lot of run time errors and allows pausing to give you the opportunity to examine the state of variables and the function call stack, amongst other things. This should help you understand and track down the source of any problems.

Another less obvious use for the debugger is as a tool that you can use in order to improve your understanding of programming. As the debugger allows you to step through code line by line, displaying exactly what each instruction is doing to the program variables, you can use it to step through unfamiliar code and get a feel for what it is doing.

Usually, even for experienced programmers, it's worthwhile single stepping through your code to ensure that it is doing what you want it to do. You will often find that you discover logical errors you’ve made that may have been hard to spot otherwise.

### The Code::Blocks Debugger

Code::Blocks provides an extremely powerful and flexible debugger. This tutorial will go over the basics, but there are a lot of things it can do that you might find useful so be prepared to invest some time in investigating its capabilities.

In order to use the debugger, first of all it's a good idea to ensure that the program has been compiled with debug information. There should be a drop-down menu in Code::Blocks that says either **Release** or **Debug**. Make sure you set it to **Debug**, this means the compiler will add extra debugging information to the executable so that the debugger can use to work out what is going on. If you don't see this toolbar item, or it is greyed out, then you may not have a project open. Code::Blocks requires you to create a project with debugging information before you can use the debugging tools.

**NOTE** - you *can* debug using a **Release** compiler executable and the debugger will try it's best to work out what is going on for you, but the information you get will be very limited. It's best to develop your programs using the **Debug** build, and on occasion, perhaps once or twice per day, try the **Release** build to ensure the optimised version is still working. Also, don't forget to switch to the **Release** build when you have completed the project.

Once the configuration has been set to enable debugging, the easiest way to start debugging is to press **F8** or click the red **'play'** button. This will automatically compile a debug version of your program and begin execution if the compilation is successful. If there is a syntax error then the debugger will stop the program, tell you where it thinks the problem is, and give you the option of looking at the code to try to determine the problem.

***Exercise 01***

1. Create a new project.
2. Add a new C file to the project called **debugtest.c.** To do this,click **New File** in the top left corner and choose **Empty File**.Name and save this new file, ensuring you use a **‘.c’** extension, and then remove **main.c** by right-clicking **-> Remove file from Project**.
3. Add the following code to **debugtest.c**:

int main()

{

int a, b, c;

a=10;

b=0;

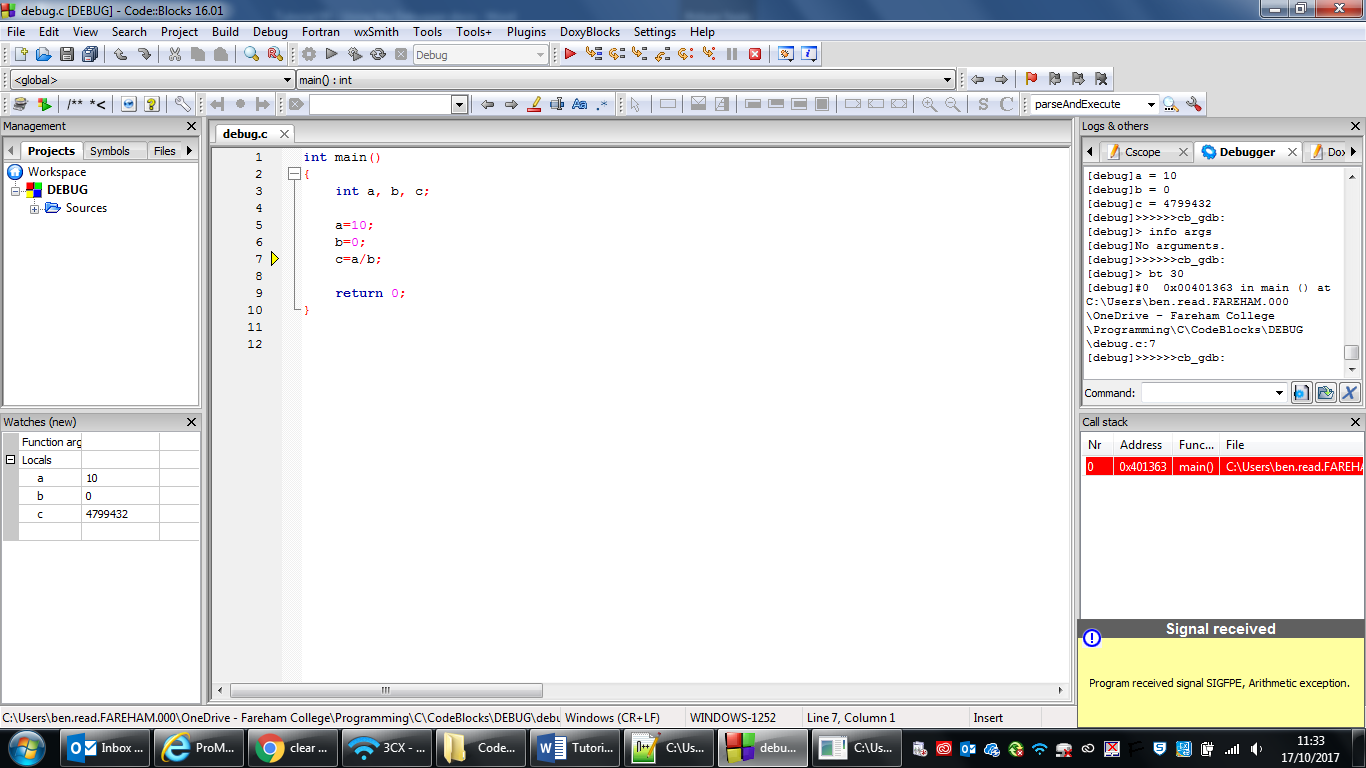
c=a/b;

return 0;

}

1. Press **F8** to compile and run the program in debug mode.

You should end up with something similar to this:



Essentially what has happened is the code is trying to divide by zero. As you probably know this result is infinite and cannot be calculated by the computer, so there is no way of giving the variable **c** a sensible value. If the debugger is being used then it is able to spot things like this and tell you what is happening using an **exception handling dialog**. Take note of what this says as it often contains useful information, in this case it's telling you an **arithmetic exception** has been received.

You can run without the debugger active to see what happens. Press **F9** and the program will run on its own. Try it and see what happens.

When an error like this occurs and the exception handling dialog appears, the IDE will jump to the code where the debugger thinks the error occurred. You can see the **yellow arrow** next to the source code above indicating the line c = a/b;

At this point you have several additional sources of useful information to help you determine the problem.

First and probably most useful is the **Watches** window which will appear when you begin debugging. If you can’t see it on your IDE then select the **Debug** menu -> **Debugging Windows** -> **Watches**, then drag the window that appears to a suitable location. This area displays all of the variables that are currently in scope, i.e. **local** variables, their datatype and their current values. In the example above, **a** equals 10, **b** equals 0 and **c** equals a large positive number. The reason **c** has such a strange value is that it has not yet been assigned a value since the **a/b** operation cannot be carried out. You’ll also notice that its highlighted in red, as its the next variable in scope which is due to be initialised.

The other important feature to discuss is the **Call Stack**. This is also accessible via the **Debugging Windows** menu. In this example it's not telling you very much as it is a very simple program with no functions, but in a more complex program you can tell from the **Call Stack** which functions have been called in what order to get to the current program instruction.

***Exercise 02***

Overwrite your debugtest.c code with the following –

#include <stdio.h>

int addition(int, int);

int main()

{

int x =5, y = 2, z;

z=addition(x,y);

printf("%d", z);

return 0;

}

int addition(int a, int b)

{

int c;

c=a+b

return c;

}

### This code simply adds together the values of two variables using a function. We can use this code to understand how to view the Call Stack and how stepping in to functions effects the Watches window.

At this point it’s worth opening both the **Watches** and **Call Stack** windows and positioning them where convenient in the main Code::Blocks window. You may have experienced unusual behaviour with these windows when debugging, i.e. as soon as you commence the debugging operation the windows disappear. To get them to stay put you need to save your perspective. Do this by clicking the **View** menu **-> Perspectives -> Save Current**.

The first thing you may notice about this code is it does almost nothing when we try to debug it using **F8**.

This is due to a lack of user input but we can slow down the debugging process using various methods in order to check exactly how the program operates.

Run to Cursor

This is the simplest form of a breakpoint. Just click on the line you would like the debugger to pause processing on and press **F4.** The debugger will begin processing your code, the **Watches** window will populate with local variables, and then the program will halt where you indicated with the cursor. There will be a yellow arrow pointing to the line the debugger has stopped on. This line has not been processed yet. **Run to Cursor** is a really useful way to quickly target an area of code to ensure its working as expected. Additionally, **F7**  will run through the code line by line.

Step Into/Out of Functions

**Step Into** is used to process the contents of functions. As they exist outside of the **main()** function the debugger will not normally operate inside of them. Often we will need to evaluate a function’s code as part of the debugging process and we can use **Step Into** for just this purpose.

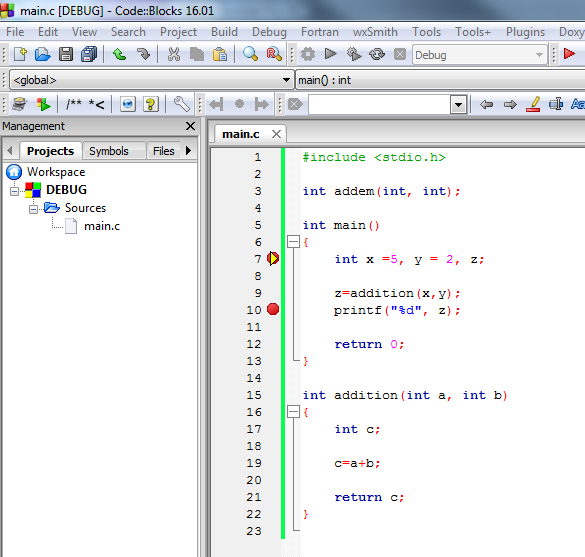
Place your cursor on **line 7**, ***int x = 5, y =2, z;*** and press **F4**. The debugger will process all of the lines up until the **addition()** function is called. Press **F7** to move down one line to the function call. In order to debug the function, press **Shift + F7** to **step into** its code. You will see the yellow arrow move down to the function area and come to rest at the first operation, in this case ***c=a+b***.

Take note of both the **Watches** and **Call Stack** windows. The contents of **Watches** has changed to represent the variables within the scope of the **addition()** function. Additionally, the **Call Stack** window indicates in red that the function currently being processed is **addition()** not **main()**. By paying close attention to these outputs you can get a much clearer idea of what is happening as your program executes.

Press **F7** to work your way through the function or press **Ctrl + F7** to **Step Out**. Stepping out will instruct the debugger to exit the function and return to the line of **main()** which it was called from.

### Breakpoints

Another useful feature that a debugger provides are **breakpoints**. Breakpoints are user defined places in code that the debugger will stop at when execution reaches them. You can set and unset them by clicking on a line of source code and pressing **F5**, or just by clicking on the grey bar to the left of the source code. Here I've set two breakpoints on lines 7 and 10 and have started debugging by pressing **F8**.



The program has stopped at the variable declaration/definition and indicates which line it’s stopped on with the yellow arrow. If I press **F8** again it will step through the code to the second breakpoint. Pressing **F8** againwill complete the program and end the debugging process.

### Conclusion

These are the basic debugging tools that you need to get by. Start using breakpoints and stepping through code to more fully understand the code you are writing, and make sure you understand how to correctly use the debugger when it stops execution due to some sort of error. I fully expect you to be able to demonstrate and explain your code to me line by line using the debugger. Additionally, if you are having a problem and want my help, I expect you to be able to show that you have attempted to solve the problem by analysing it with the debugger.

#### Advanced Debugging

The following code is not broken, however, it is somewhat more complicated than examples we’ve looked at so far. It is essential that you understand exactly how it works.

***Exercise 03***

Create a new project called Adventure and paste the code below into main.c. Use the debugger to analyse the various functions, including the pre-defined **clrscr()**, examining how they work together to create both the grid and the movement of the character. Make sure you set breakpoints and step into and out of the functions. If you get caught in a long loop, ensure you understand exactly what it’s doing and then set a breakpoint just after it so you can jump over it the next time you run the debugger. You may find that the debugger loses focus without clear instructions and use of breakpoints. Be patient, and use the techniques you’ve learned in this tutorial to keep the debugger centred on the section of code you’re analysing.

**MAKE SURE TO COMMENT THE CODE WITH WHAT YOU LEARN SO THAT YOU CAN MAKE USE OF THESE TECHNIQUES IN YOUR OWN PROGRAMS**

#include <stdio.h>

#include <stdlib.h>

const int ROOM\_HEIGHT = 20;

const int ROOM\_WIDTH = 20;

void printmap(char a[ROOM\_WIDTH][ROOM\_HEIGHT], int x, int y);

void input(char a[ROOM\_WIDTH][ROOM\_HEIGHT], int x, int y);

void clrscr();

int main()

{

int i,j;

int x = ROOM\_WIDTH / 2;

int y = ROOM\_HEIGHT / 2;

char map[ROOM\_WIDTH][ROOM\_HEIGHT];

for(i = 0; i <ROOM\_HEIGHT; i++)

{

for(j = 0; j < ROOM\_WIDTH; j++)

{

map[j][i] = '.';

}

}

printmap(map, x, y);

return 0;

}

void printmap(char a[ROOM\_WIDTH][ROOM\_HEIGHT], int x, int y)

{

int i,j;

a[x][y] = 'Î';

printf("Use 'w', 'a', 's', 'd' to move the adventurer around the grid.\n\n");

for(i = 0; i <ROOM\_HEIGHT; i++)

{

for(j = 0; j < ROOM\_WIDTH; j++)

{

putchar(a[j][i]);

}

printf("\n");

}

input(a, x, y);

return;

}

void input(char a[ROOM\_WIDTH][ROOM\_HEIGHT], int x, int y)

{

a[x][y] = '.';

char in[1];

scanf("%s", in);

switch(in[0])

{

case ('w'): y = y - 1;

break;

case 's': y = y + 1;

break;

case 'd': x = x + 1;

break;

case 'a': x = x - 1;

break;

default: printf("YOU CAN'T GO THAT WAY!\n");

input(a, x, y);

break;

}

if (x < 0 || y < 0 || x > ROOM\_WIDTH - 1 || y > ROOM\_HEIGHT -1)

{

printf("YOU CAN'T GO THAT WAY!\n");

x = (x > ROOM\_WIDTH - 1) ? ROOM\_WIDTH - 1: x;

x = (x < 0) ? 0: x;

y = (y > ROOM\_HEIGHT - 1) ? ROOM\_WIDTH - 1: y;

y = (y < 0) ? 0: y;

input(a, x, y);

}

clrscr();

printmap(a, x, y);

return;

}

void clrscr()

{

system("@cls||clear");

return;

}