Universidad de Alcalá

Departamento de Automática

COMPUTER SCIENCE

Assignment 3.
Program debugging.

GIEyAI
Fall Semester, 2022-2023
v2.7(07.09.22)

Debugging.

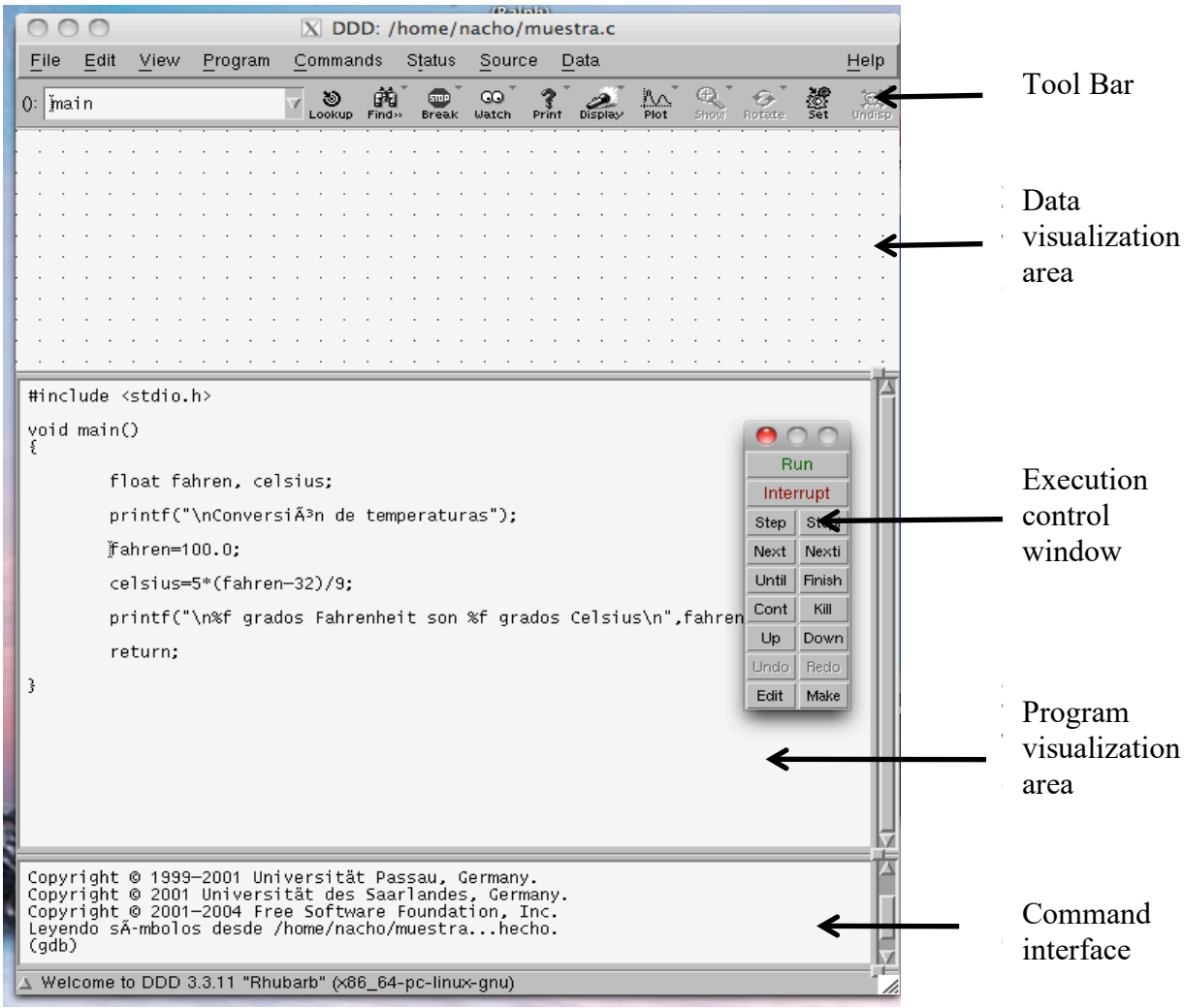
Beginning programmers (and more experienced ones, as well) frequently make design or coding errors when creating programs, so one of the more useful tools for them is the debugger. A debugger is a program which can execute another program (yours) step by step (instruction by instruction) and let you analyze the values that variables take, so that you can find the spot where it starts doing what it should not. It is appropriate at this time to remember the maxim:

"A program does what you tell it to, not what you want it to"

The first debugger that we will cover here is called Data Display Debugger (or `ddd`). It can be launched from the desktop, or from a command prompt, typing:

```
bash-ln.05$ ddd muestra&
```

The debugger window will be something similar to the one shown here:



Once our program has been loaded, the first thing is to start it running. To do that, we set a breakpoint in one line of code (for instance, in the first call to `printf`) by clicking in the corresponding line of code, at about the beginning of the line. This will set the cursor in this line. Next, we click the *'Break'* button, in the toolbar. This sets the breakpoint. In case we need to unset it, we would just click on the *'Break'* button again. With the breakpoint on, we start the program by clicking the *'Run'* button in the execution control window. The program will stop at the line where we set the breakpoint.

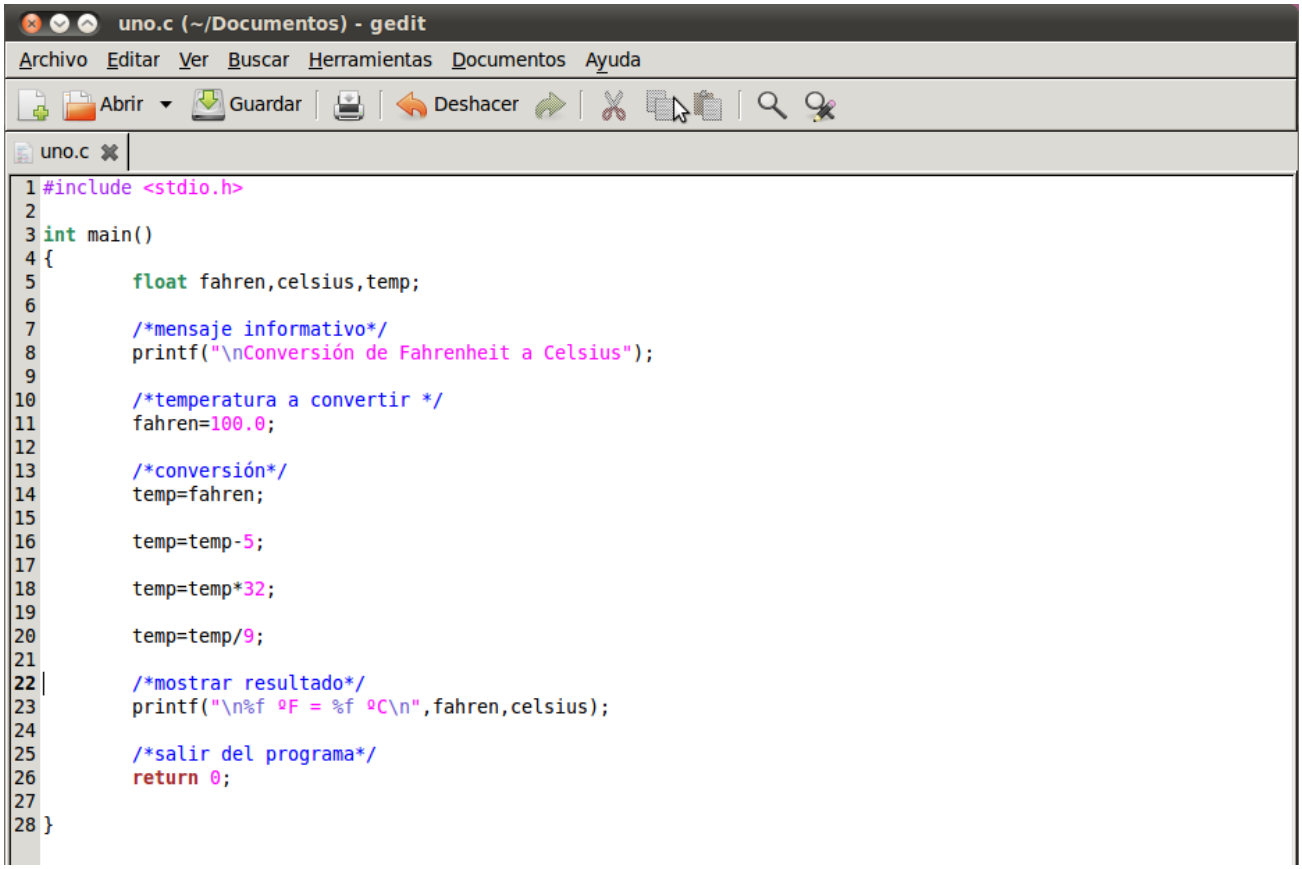
It is possible to see the values of variables by selecting them with the mouse (clicking on them in the program visualization zone) and, next, clicking the *'Display'* button. This will show them in the data visualization zone, along with their current value.

In order to execute one line of code, one can use either the *'Step'* button or the *'Next'* button, both of them in the execution control window. The difference between both commands is how they deal with functions. When executions comes to a function call, *'Next'* executes the whole function in one step, whereas *'Step'* executes just the first instruction in the function and then stops.

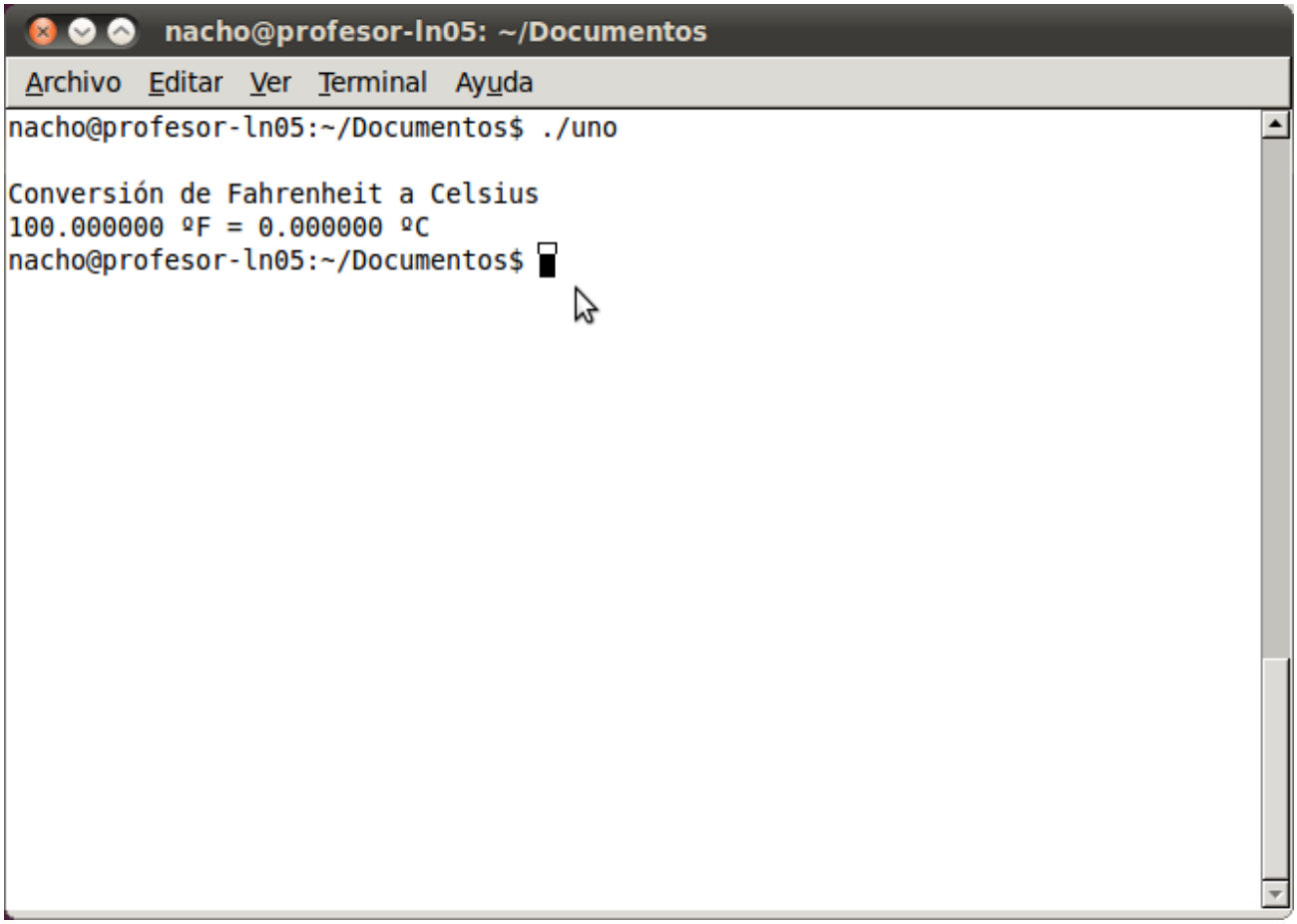
Besides, it is possible to execute nonstop whole regions of code simply setting a breakpoint on the instruction after the desired region, and running the program with the *'Cont'* button. The execution control window has some other interesting options. To learn more, consult the references at the end of this document.

Using ddd to debug a program (sample debug session).

This section shows how to use ddd to find the errors in a program. The starting point is the following program:

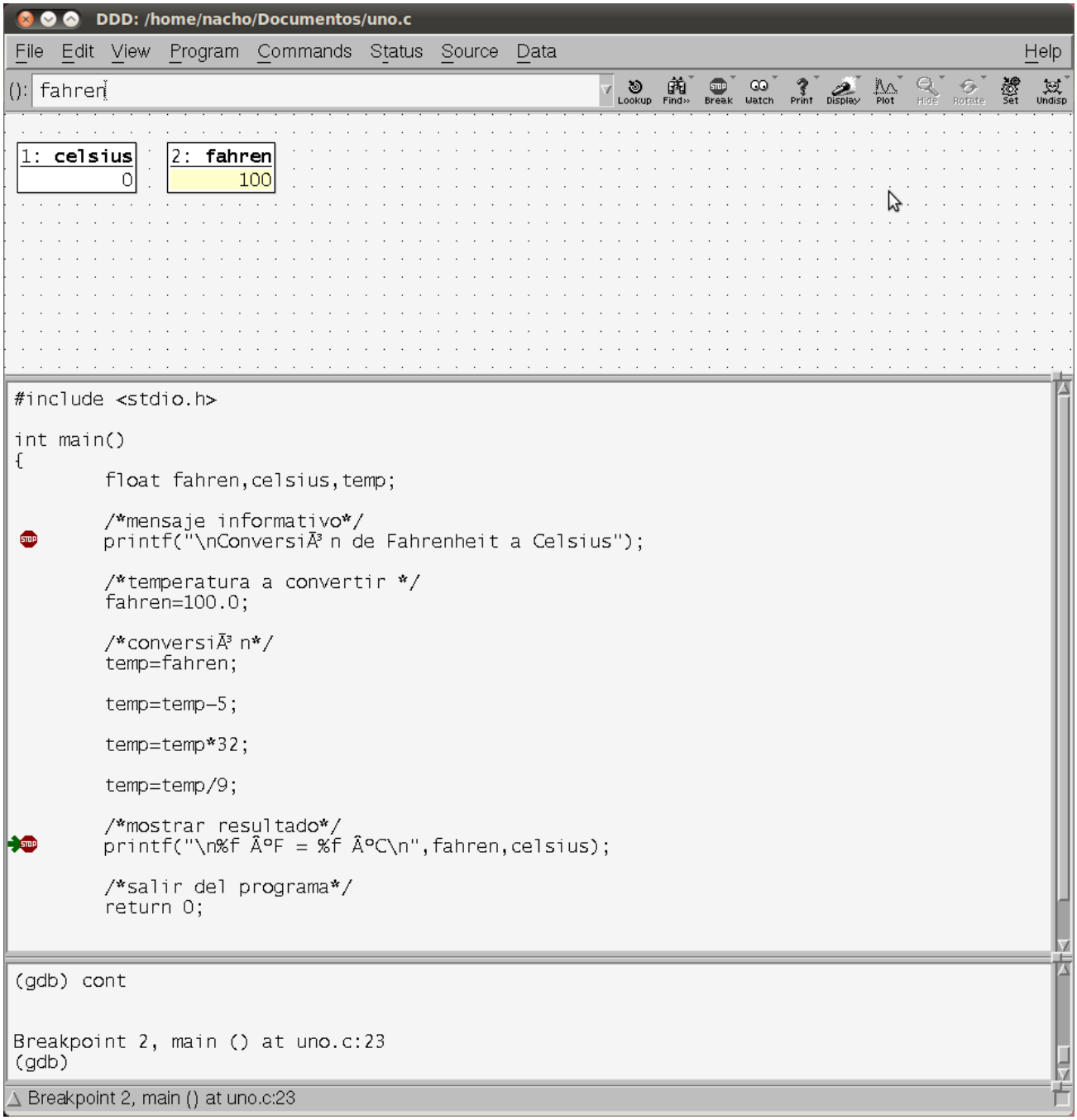


The program should convert the temperature in degrees Fahrenheit, contained in variable fahren, to Celsius, and show the result on the screen. When compiled and run, the result is:

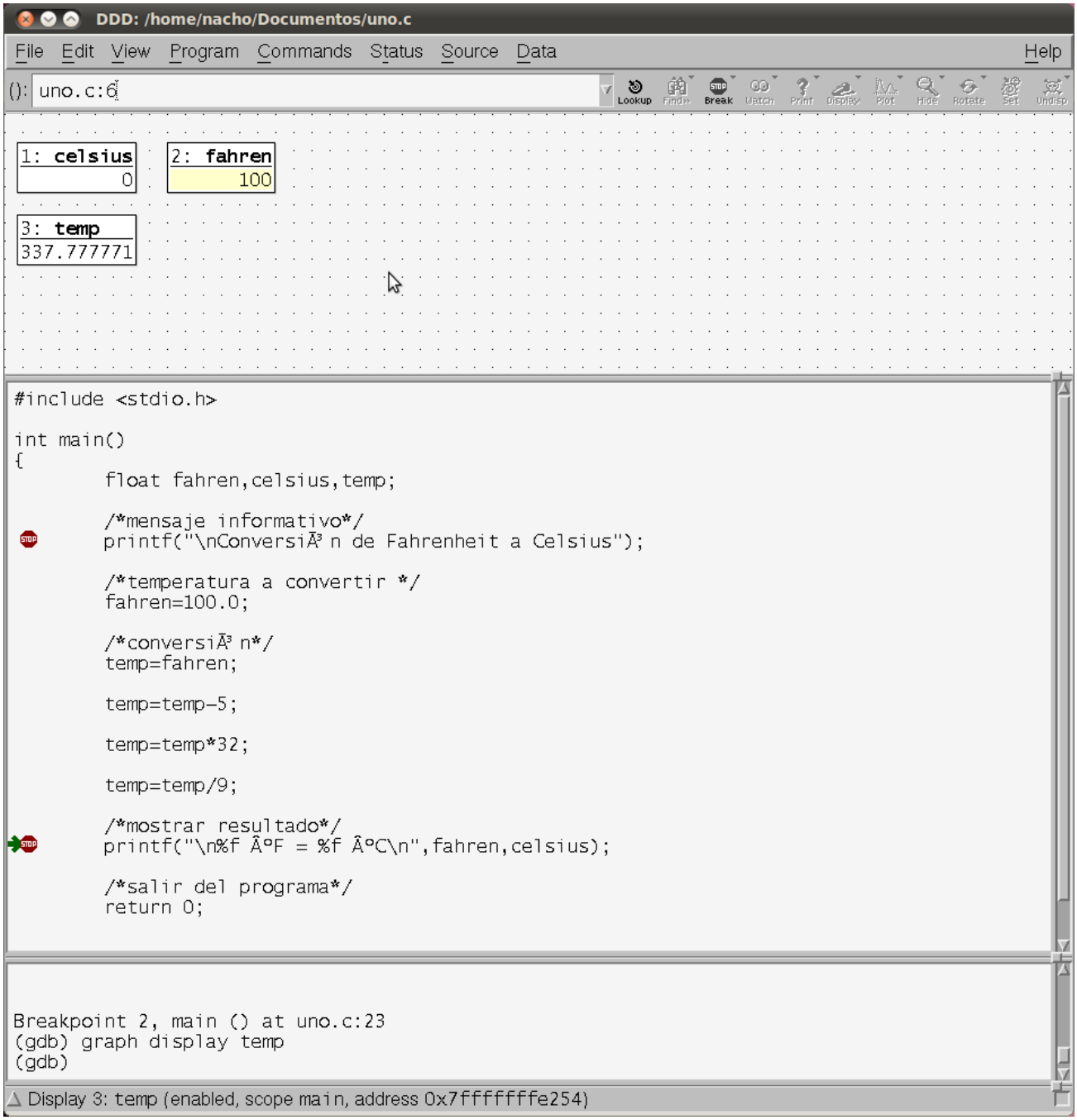


As you can see, the result is not correct, since 100ºF are not 0ºC, so the program is not working well. To find the errors, first we have to launch the debugger. In the following picture, we already have two breakpoints, one in the first sentence of the program, and the other on the call to `printf` which shows the results. Besides, the values of variables `fahren` and `celsius` are visualized. The instructions to do all this have already been discussed above.

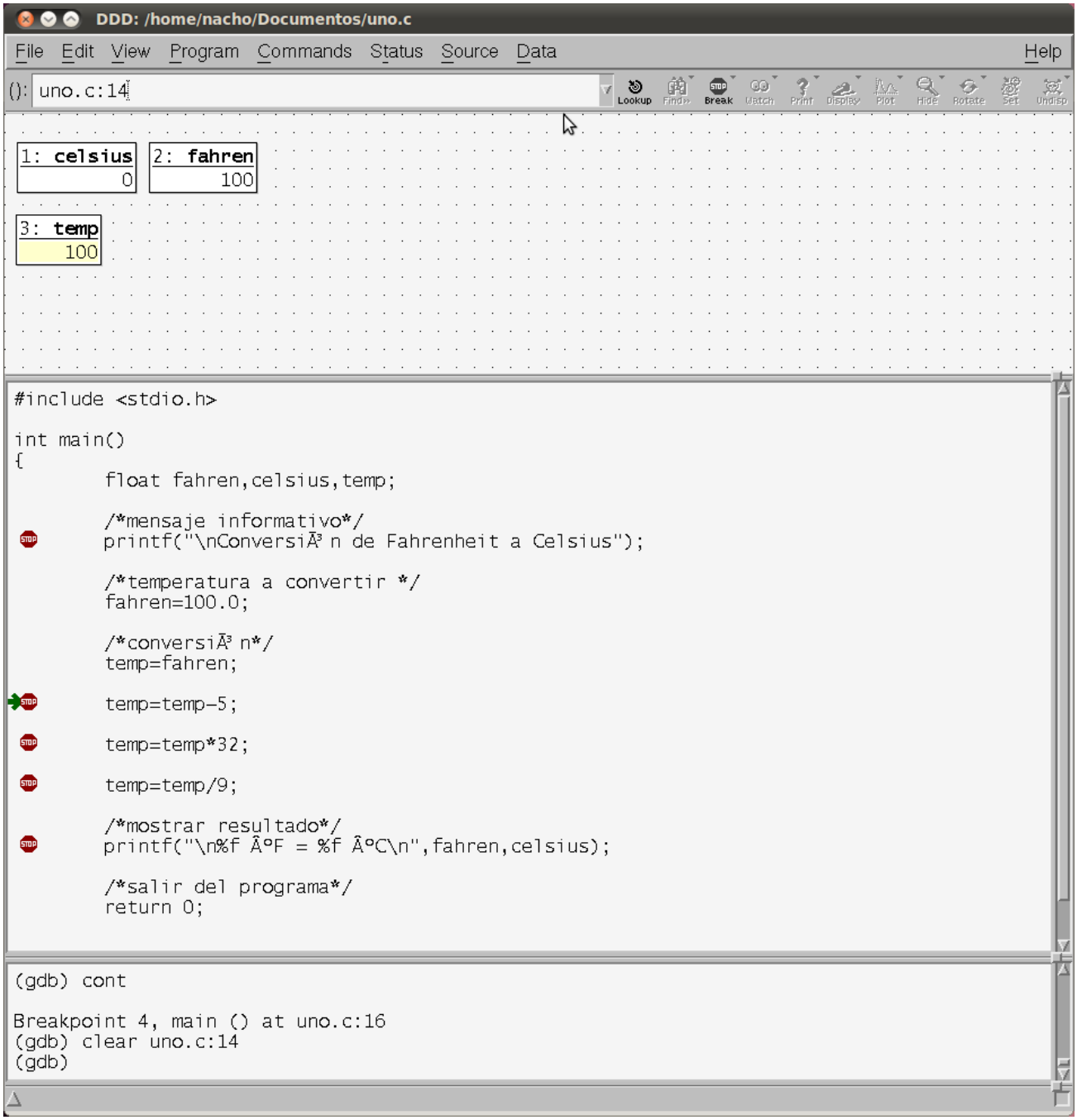
The program has run until the second breakpoint, and you can see that variable `celsius`, which is shown, has value `0`, so the call to `printf` is working correctly (as library functions usually do). Why does `celsius` equal `0`? The answer is plain to see: the conversion is done using `temp` as a temporary variable, and the final result is not moved to `celsius`, which, thus, keeps its original value.



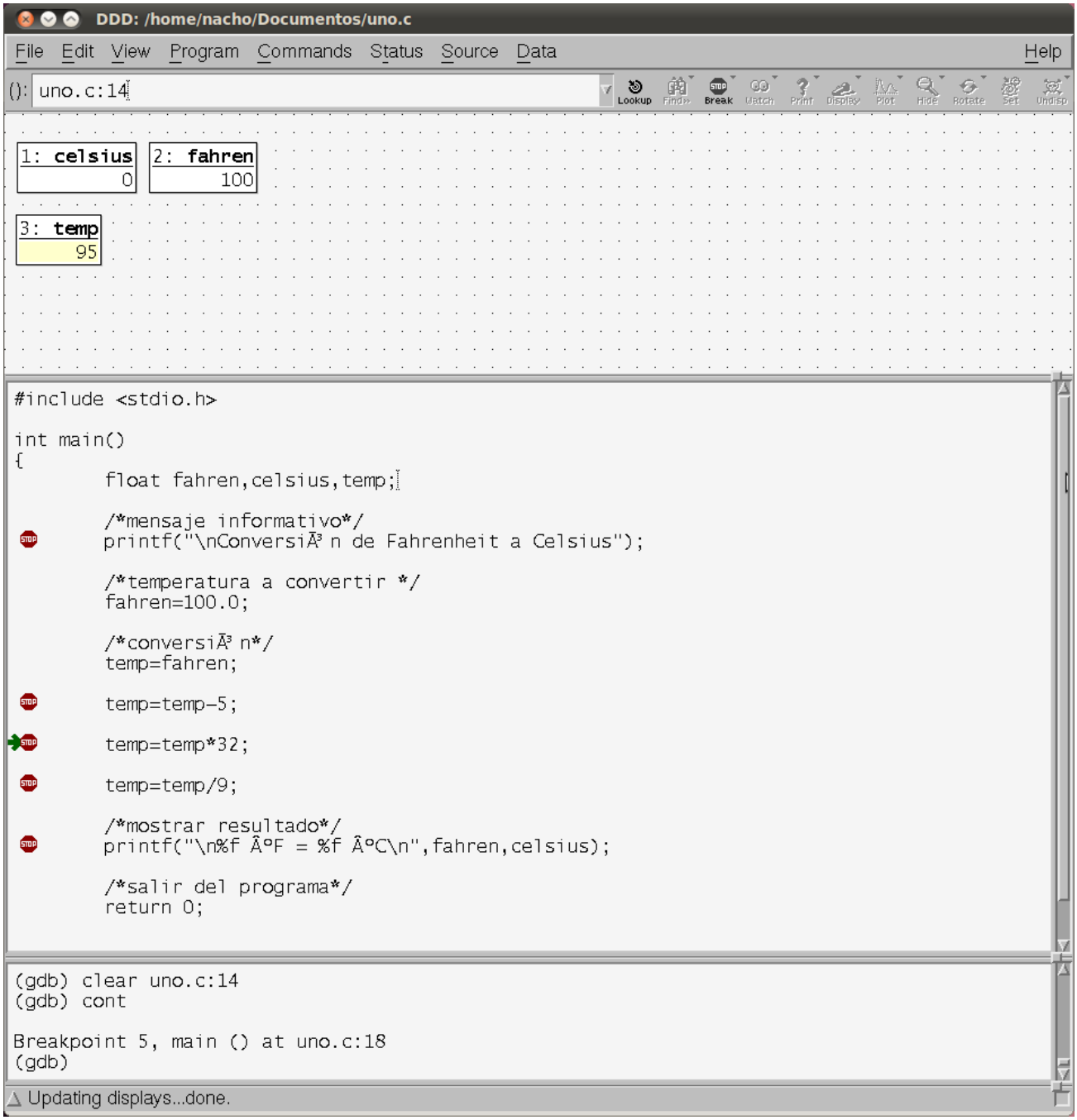
To see if variable `temp` has the right value and the conversion was correct, we show its value in the debugger:



The value of `temp`, 337.777771, is not the right one, so the conversion has another error. It is necessary to debug the conversion process. We can do this by restarting execution of the program (the *'Run'* button), and setting breakpoints in each sentence of the process. This is not strictly necessary, since we can use the *'Next'* button to execute one sentence at a time. Both methods are correct (although the latter one is a little less tedious).



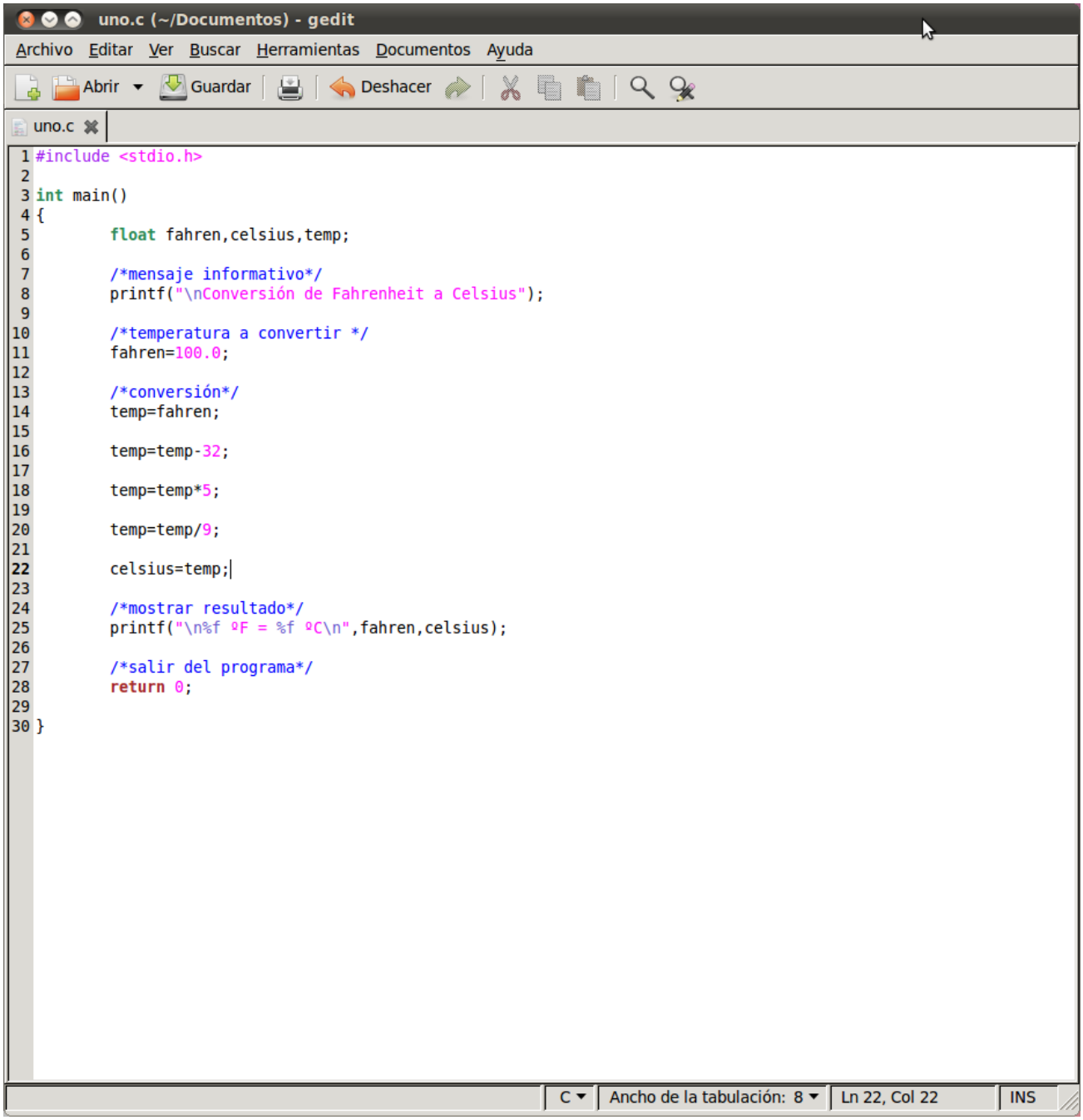
Now we execute the program to see the conversion process step by step. If you use breakpoints, to jump to the next you should use the *'Cont'* button. The following capture shows one of the intermediate steps of this process:



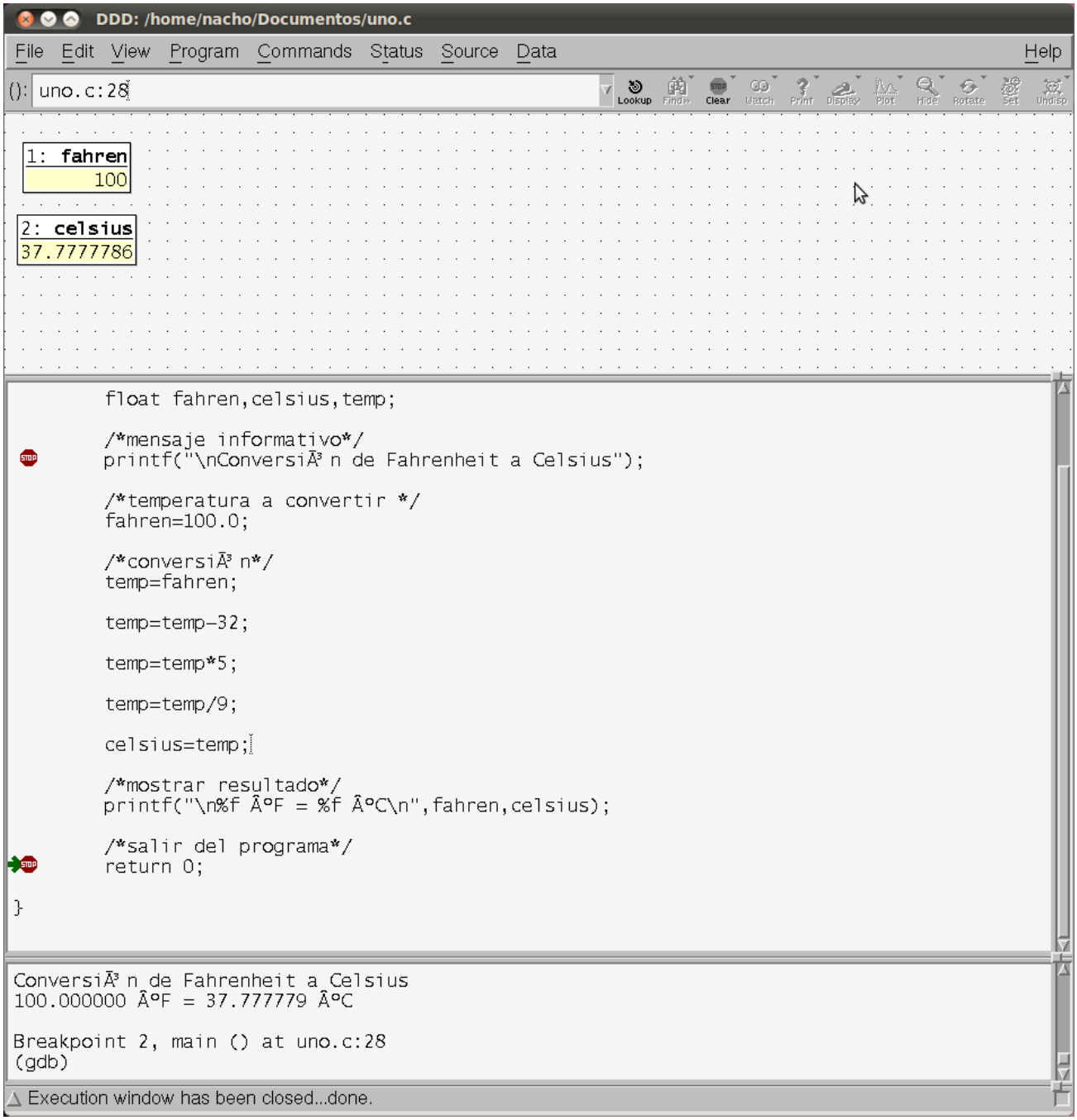
At some point, the attentive student will notice that the conversion formula is not correct. The correct one is:

```
celsius=5*(fahren-32)/9;
```

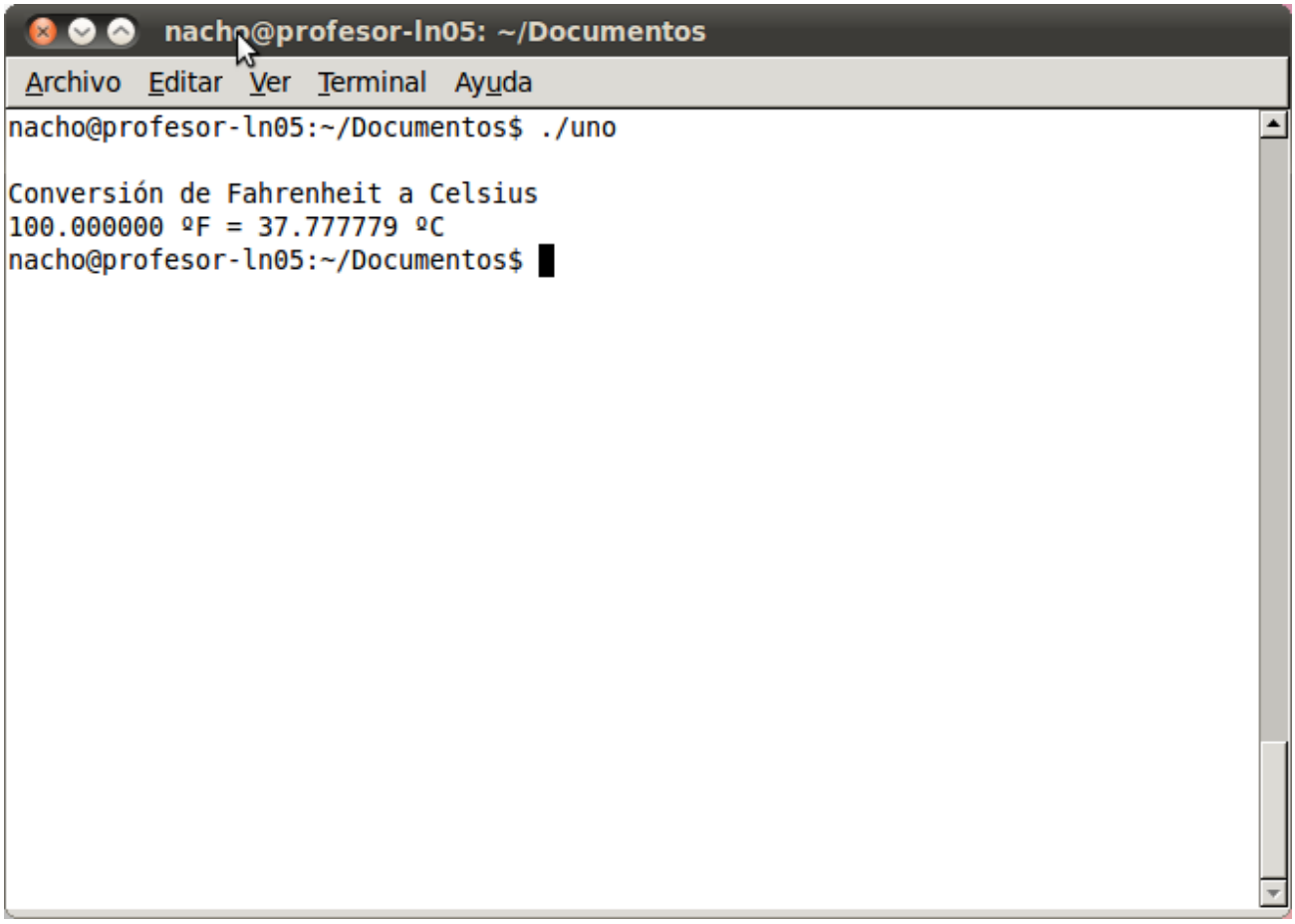
The next figure shows the code with the correct conversion formula, and the transfer of variable `temp` to variable `celsius`, so `printf` shows the correct value on the screen.



When this new version of the program is compiled and executed, the result is correct. The next figure shows the result of the execution. In the command interface panel (the one at the bottom), we see that the call to `printf` shows the correct value this time. This is what we would see from the command interface window if the program were run directly from the prompt. Besides, variables `fahren` and `celsius` have the right values.



The next figure shows the command interface with the execution of the program and the correct result.


nacho@profesor-ln05: ~/Documentos
Archivo Editar Ver Terminal Ayuda
nacho@profesor-ln05:~/Documentos$ ./uno
Conversión de Fahrenheit a Celsius
100.000000 ºF = 37.777779 ºC
nacho@profesor-ln05:~/Documentos$

Debugging with `gede`.

In the previous section, one of the most widespread debuggers available, `ddd`, was described. In this one we show another one, which is also freely available, as an alternative. It is called `gede`, and the way to use it is very similar (most debuggers will do more or less the same). It will let you run step by step, set breakpoints, run nonstop, and visualize variables, which is basically what you need to do to debug a program.

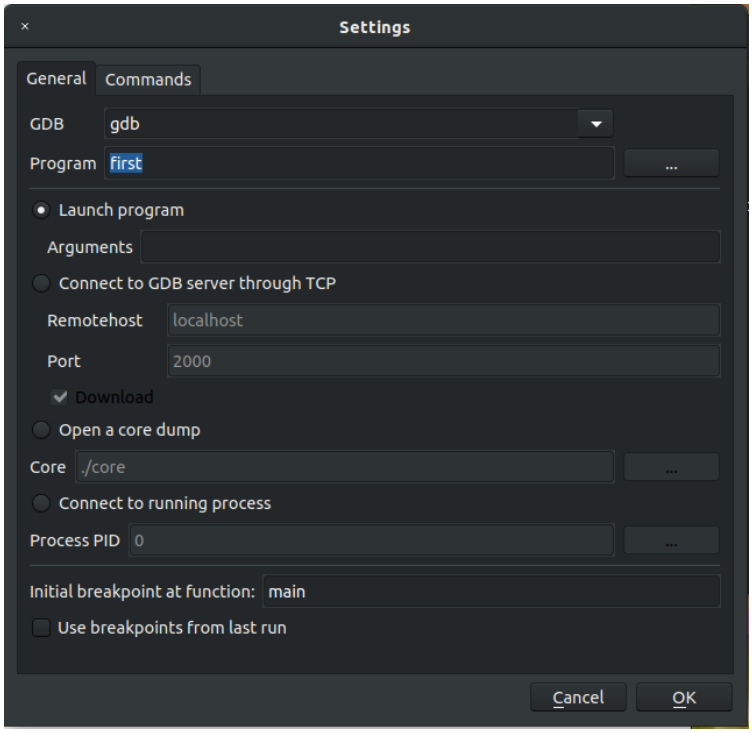
The first thing to do is call the debugger. You can do this and tell it which program you want to debug (in this case, `first`):

```
bash-ln.05$ gede --args first&
```

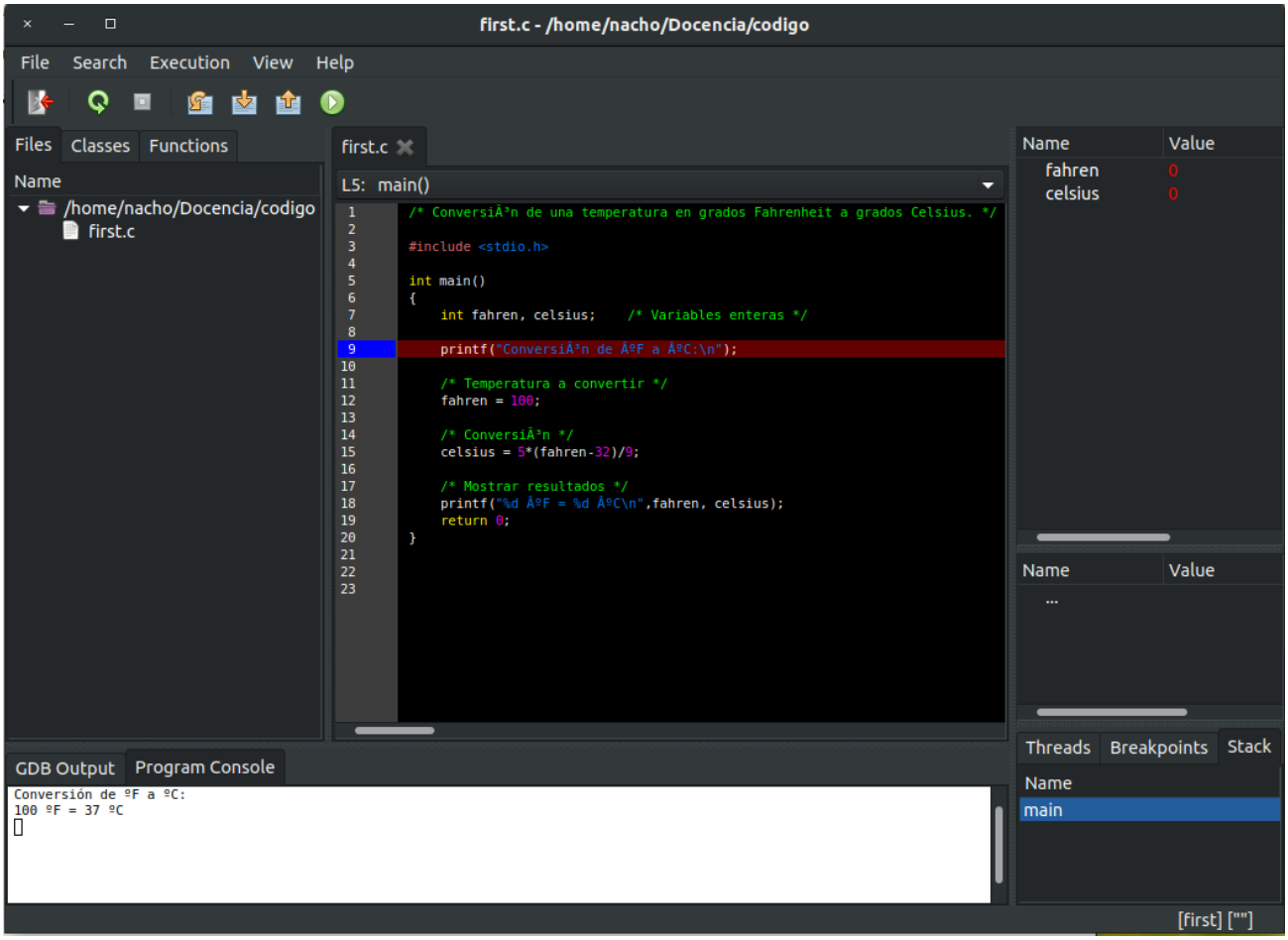
or just call it and tell it later:

```
bash-ln.05$ gede&
```

The first window you will see is the following:



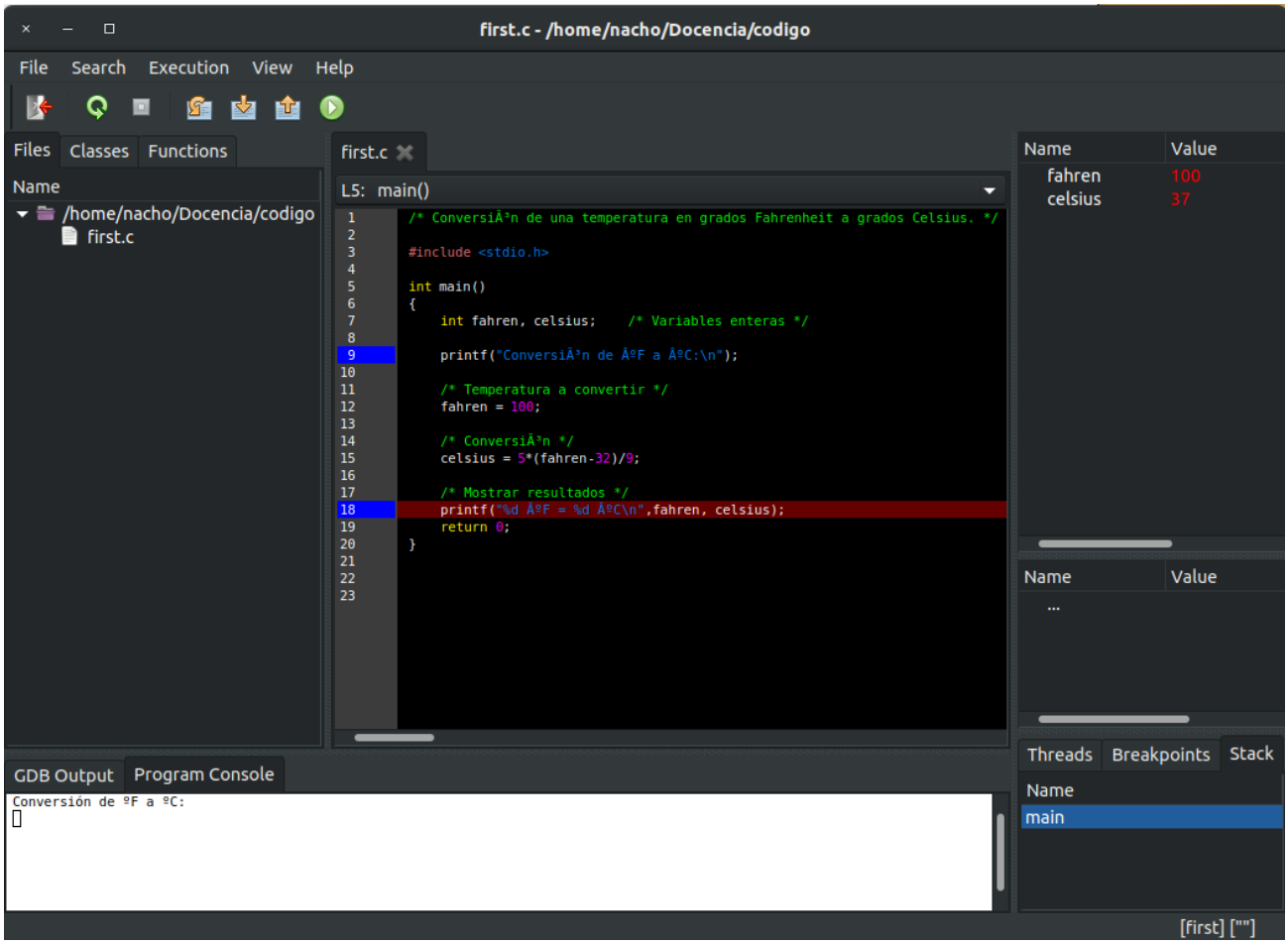
Highlighted you can see the name of the program to debug (`first`). If you told this to the debugger as you called it, the name will be there. Otherwise, you can tell it now. When you click on *'OK'*, you go to the main window:



The central part in this window shows the code of the program being debugged. The debugger automatically inserts a breakpoint in the first instruction of the program (highlighted in the image). Beside the main window, on the right, there is a pane to visualize the variables (variables local to the function being debugged appear automatically, like `fahren` and `celsius` here, there is no need to add them, as you had to in `ddd`). The window also has, close to the top, a tool bar with buttons to control execution, analogously to what happened in `ddd` with the program control panel. There is a button for *'Next'*, another one for *'Step'*, and another one for *'Cont'*. When you hover the mouse cursor over them, `gede` shows which is which. Beside those mentioned, there is one for restarting execution and another one for exiting the program.

On the lower part of the window there are two tabs for output: one is the program's output and the other is the debugger's output (GDB Output). The former (Program Console). is what you use to type in data when your program reads from keyboard. When your program is expecting input, you click on this pane and type in the data. Also, look here for what your program sends to the screen.

Lastly, management of breakpoints is done directly with the mouse. Double-click on a line number to set a breakpoint on that line, and double-click again to remove it. In the following picture, a breakpoint has been added to line 18, and execution has progressed until that line (with the *'Cont'* button).



The debugger offers many possibilities to the programmer. It greatly reduces the time to generate more complex programs, so the time spent in learning its features will greatly pay off in later assignments. Some of the features are not necessary for this class (like thread debugging or stack analysis, etc), but it is good to know they are there as you advance in your studies.

Debugging. The `gdb` debugger.

Both DDD and `gede`, the debuggers from the previous sections, are in reality only a graphical user interface for the real debugger running under the hood, `gdb`. `gdb` is the one really doing all the debugging work, while the user interface (the frontend) just acts as an intermediary between it and the user, transferring information and commands to and fro. Sometimes it is more interesting to use the `gdb` debugger directly, and this is what this section is about. We will see, in a somewhat superficial way, how to use `gdb`.

Since DDD and `gede` resort to `gdb` to do the debugging work, obviously everything that was done in the previous sections can also be done now: execute one instruction at a time, see the values that variables take, manage the program's input/output, set breakpoints, etc. The only difference (and what makes it a little more tedious) is that `gdb` is used from the terminal, it has no graphical interface. In other words, the debugging is carried out by means of commands given at the terminal.

The first thing to do is start the debugger. In a terminal, type:

```
bash-ln.05$ gdb
```

and this causes the debugger to start. The prompt changes, to indicate that we are no longer interacting with the command interpreter, but with the debugger. And from here on a basic pattern is repeated over and over: type a command and get a result. The first command should tell `gdb` which program we wish to debug. In our case, `muestra`. Since, in order to debug it, we need to refer to instructions in the program, it is advisable to open it in a text editor and activate the option to show the line numbers. The command to load our program is `file`.

```
(gdb) file muestra
```

This makes `gdb` ready to process our program, `muestra`. The debugging process in `gdb` is identical to what we did before. The only thing we need is to know the `gdb` commands that correspond to the buttons in DDD or `gede`. For instance, if we want to set a watchpoint in sentence `return 0` (line 28 in the corrected version of `muestra.c` above), the command is:

```
(gdb) b 28
```

To run the program we have 4 commands: the `run` command, which executes it nonstop from the beginning, the `step` command, which executes a single instruction, the `next` command, which executes one line of code (if that line is a function call, it will execute the whole function in one step) and, lastly, the `cont` command, which resumes execution nonstop from where it stood stopped. As can be seen, they are the same possibilities we already know. The continuous execution will stop at the first breakpoint found (or at the end of the program).

When execution stops, the debugger shows the line number and the next instruction to execute. For instance, stopping at the breakpoint in line 28 would show:

```
Breakpoint 1, main() at muestra.c:28
28          return 0;
(gdb)
```

The first line shows that execution stopped at the first breakpoint in the program, set in function `main()` at line 28 of file `muestra.c`. Next, the line in question, and finally, the prompt waits for the next command.

It is also possible to visualize variables. In this case we have two options: to see a value once, or to follow the evolution of a variable. For the first options we have the `print` command, and for the second, the `display` command. The `print` command shows the current value of a variable only once, whereas the `display` command will show the variable every time execution is stopped. For instance, if, before running the program, we had used the `display` command to follow the evolution of variables `temp` and `fahren`, when stopping at line 28, `gdb` would show:

```
Breakpoint 1, main() at muestra.c:28
28          return 0;
2: temp = 37.77777778
1: fahren = 100.0
(gdb)
```

You can also visualize strings of characters, integers, single characters, etc.

Finally, the `kill` and `quit` commands terminate execution of a program. `kill` stays within the debugger (to start debugging again, for instance), and `quit` exits the debugger (and returns to the command interpreter).

The `gdb` debugger is a very complete tool with a lot of useful options. It is very advisable to look into its manual page. Besides, the web page for this class has a link to a quick-reference sheet with a summary of the commands and a sort description of each one.

EXERCISES.

Find the errors in the following programs, possibly with the help of one of the debuggers from the previous sections:

Program 1: *The program should compare two variables and show the bigger one on the screen. In the case that they are equal, it should also indicate so.*

Code:

```
#include <stdio.h>

int main()
{
   int var1, var2;

   var1=10;
   var2=5;

   if (var1>=var2)
        printf("\nThe bigger variable has a value of: %d",var2);
   else
        printf("\nThe bigger variable has a value of %d",var1);

   if(var1=var2)
        printf("\nBoth variables have the same value:%d\n",var1);

   return 0;
}
```

Program 2: *The program should calculate the arithmetic mean of the values of an array of* `int`.

Code:

```
#include <stdio.h>

int main()
{
   int list[10]= {3,5,2,1,6,2,3,1,5,9};
   int sum,i,j;
   float mean;

   for(i=1;i<=10;i++)
        sum=sum+list[j];

   mean=sum/2;

   printf("\nThe arithmetic mean of the list is: %f\n",mean);

   return 0;
}
```

Program 3: *The program should create an array of integer numbers from another, given one, such that every position of the new array should be equal to the sum of all the elements of the first array up to that position (inclusive).*

Example:

if the original array is:

```
{2,1,4,7,2}
```

then, the new one would be (the operations between parenthesis shown as indication only):

```
{2,3(2+1),7(2+1+4),14(2+1+4+7),16(2+1+4+7+2)}
```

Code:

```
#include <stdio.h>

int main()
{
    int original[10]= {3,5,2,1,6,2,3,1,5,9};
    int new[10];
    int sum,i,j;

    printf("\nOriginal\tNew\n");

    for(i=1;i<=10;i++)
          printf("%d\t",original[i]);
          sum=sum+original[i];
          printf("%d\n",sum);
          new[i]=sum;

    return 0;
}
```

References:

- The DDD page: https://www.gnu.org/software/ddd/manual/html_mono/ddd.html
- A DDD tutorial: http://knuth.luther.edu/~leekent/tutorials/ddd.html
- DDD Manual: http://www.gnu.org/software/ddd/manual/pdf/ddd.pdf
- The gede page: http://gede.acidron.com/
- The gdb page: https://www.gnu.org/software/gdb/
- A gdb tutorial: https://web.eecs.umich.edu/~sugih/pointers/summary.html