NWEN 241 Exercise 2

(Intermediate C Program Development)

Release Date: **16 March 2021**

Submission Deadline: **26 March 2021, 23:59**

Objective:

The objective of this exercise is to familiarise you with an intermediate C development tool, and write and debug programs using structures and pointers.

At the end of this exercise, you should submit the required files to the Assessment System (`https://apps.ecs.vuw.ac.nz/submit/NWEN241/Exercise_2`) on or before the submission deadline. You may submit as many times as you like in order to improve your mark before the final deadline. Submissions beyond the deadline will not be marked and will receive 0 marks.

1. Exercise Requirements

For NWEN 241, it is highly recommended that you undertake all development using the computers in CO246. The computers in this lab use the Linux operating system. *This guide is written with the assumption that you are in CO246 lab.*

If you are not able to go the lab, you can remotely access similar computers via secure shell (ssh). Consult one of the remote study guides (see `https://ecs.wgtn.ac.nz/Courses/NWEN241_2021T1/RemoteStudyGuides`) and follow one that suits you the most.

2. GDB

In the previous exercise, you have learned basic debugging by looking at compiler messages. There are many cases where this form of debugging is not sufficient. For instance, a program may compile without errors and warnings, but during runtime, it crashes or behaves unexpectedly on certain inputs. This situation requires the use of a more advanced tool.

The tool that you will learn in this exercise is the GNU Debugger or `gdb`. Below are the key features of `gdb`:

- It can print stack traces, stop program at predefined breakpoints, or step through line by line
- It allows us to see values of each variable at each line

Start by writing this simple C program:

```c
#include <stdio.h>

int main(void)
{
    int i = 4;
    while (i >= 0) {
        printf("20/%d=%d\n", i, 20/i);
        i--;
    }
}
```

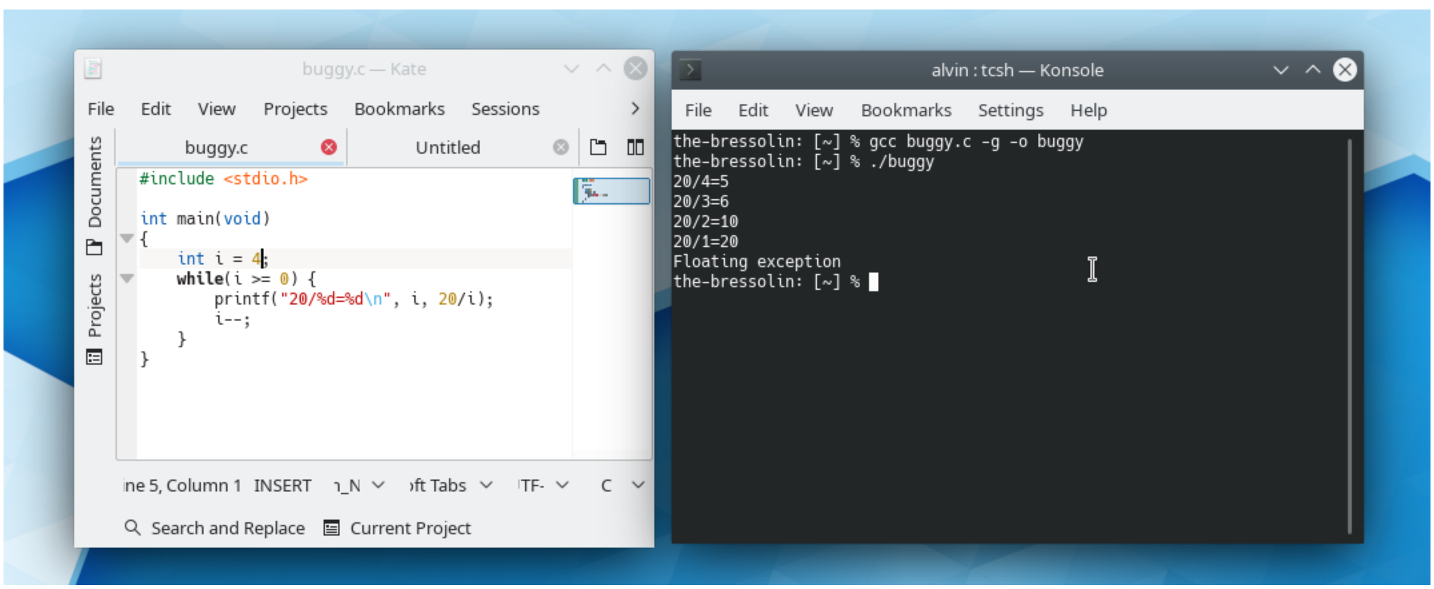
Save the file as `buggy.c` and compile with the `-g` option as follows

```
gcc buggy.c -g -o buggy
```

The `-g` option allows the executable file to be run by `gdb`. Before using `gdb`, execute `buggy` by typing

```
./buggy
```

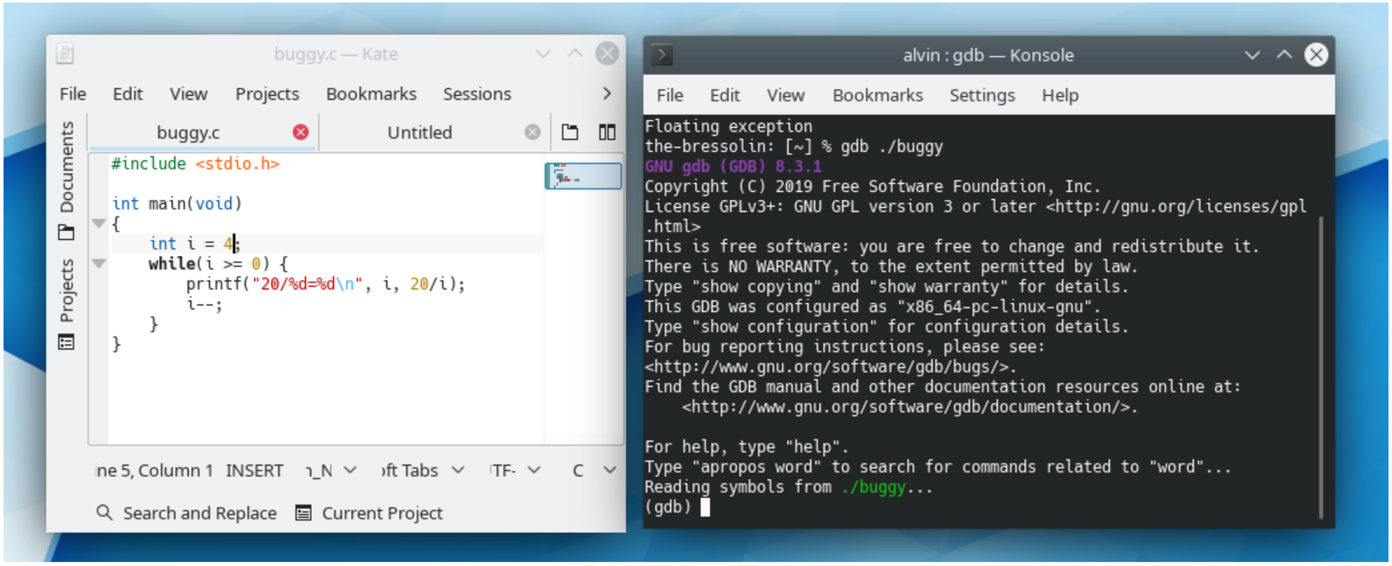
on the terminal. The program will terminate abnormally with an exception:



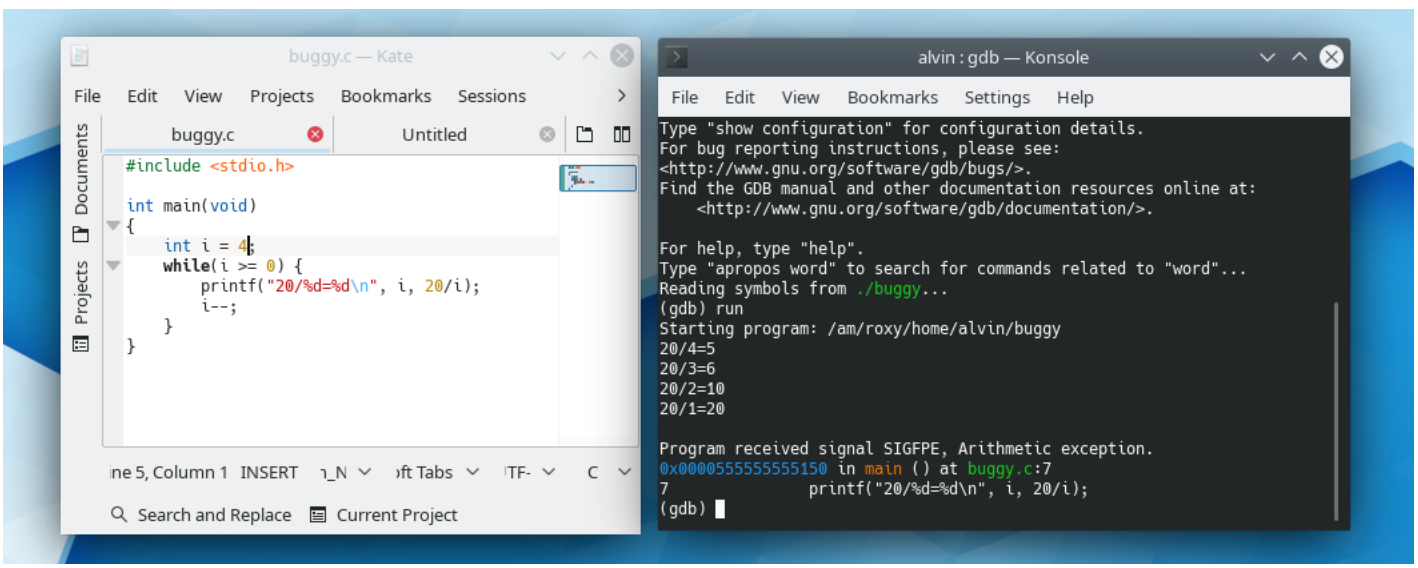
As you can see, the exception message is not helpful at all. We will now use `gdb` to investigate this issue. To invoke `gdb` to load the executable file `buggy`, just type

```
gdb ./buggy
```

on the terminal. You should see something like this on the terminal:



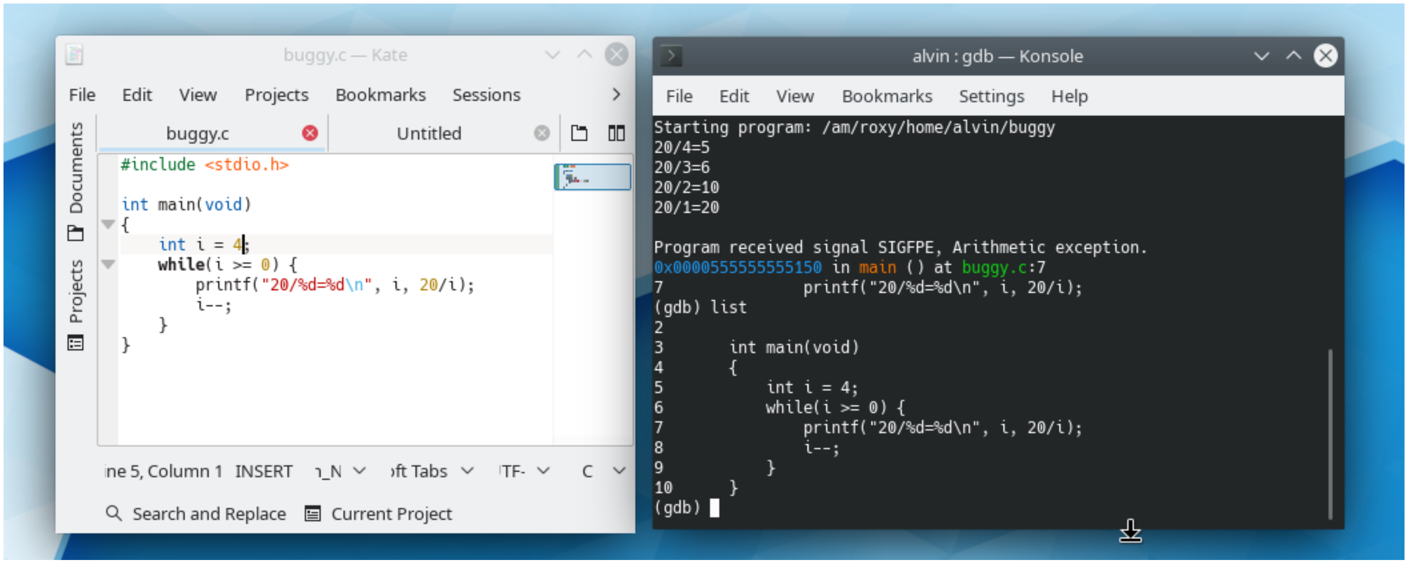
When you see the `(gdb)` prompt, that means `gdb` is ready to accept commands. You may now run the program within `gdb` by typing `run` on the `(gdb)` prompt. You should see something like this:



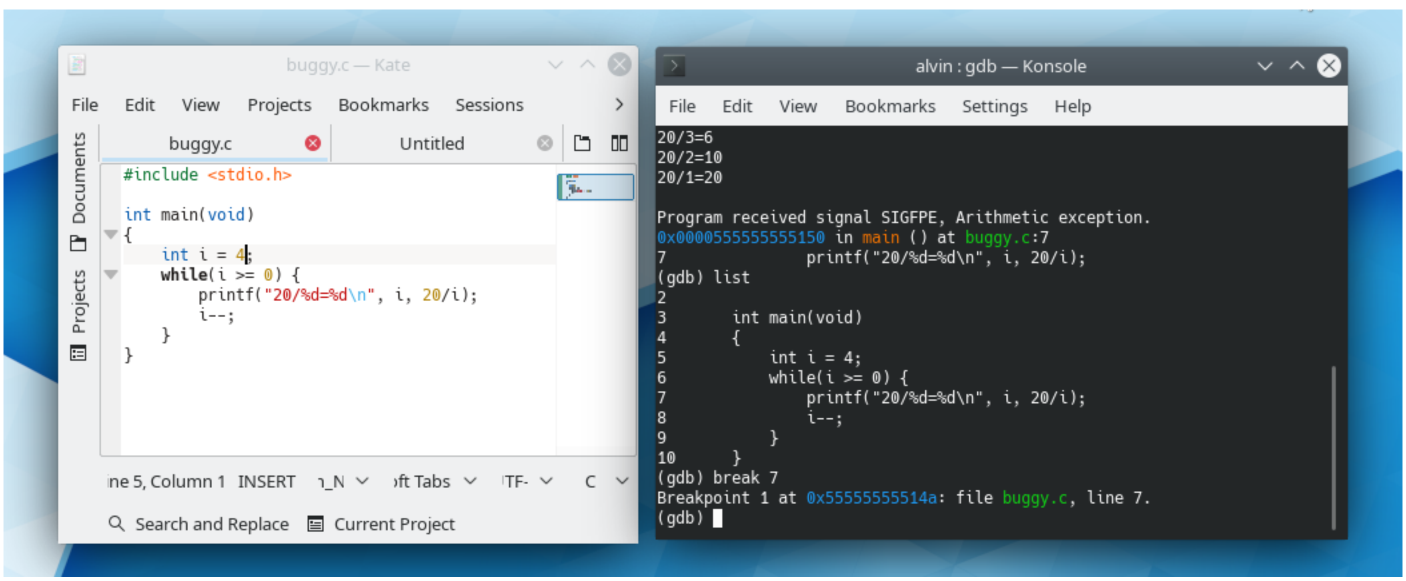
You can see the program output prior to the exception, and more importantly, the details of the exception and the exact line that caused the exception.

From the above output, you should see that the problem (**arithmetic exception**) happens at line 7. Since there is only one arithmetic operation in that line (the operation `20/i`), you are certain that this is the issue. In fact, the problem occurs when `i` becomes `0` causing the operation `20/i` to throw the arithmetic exception because division by 0 is illegal. One possible fix is to change the condition in the while-loop on line 6 from `i >= 0` to `i > 0`.

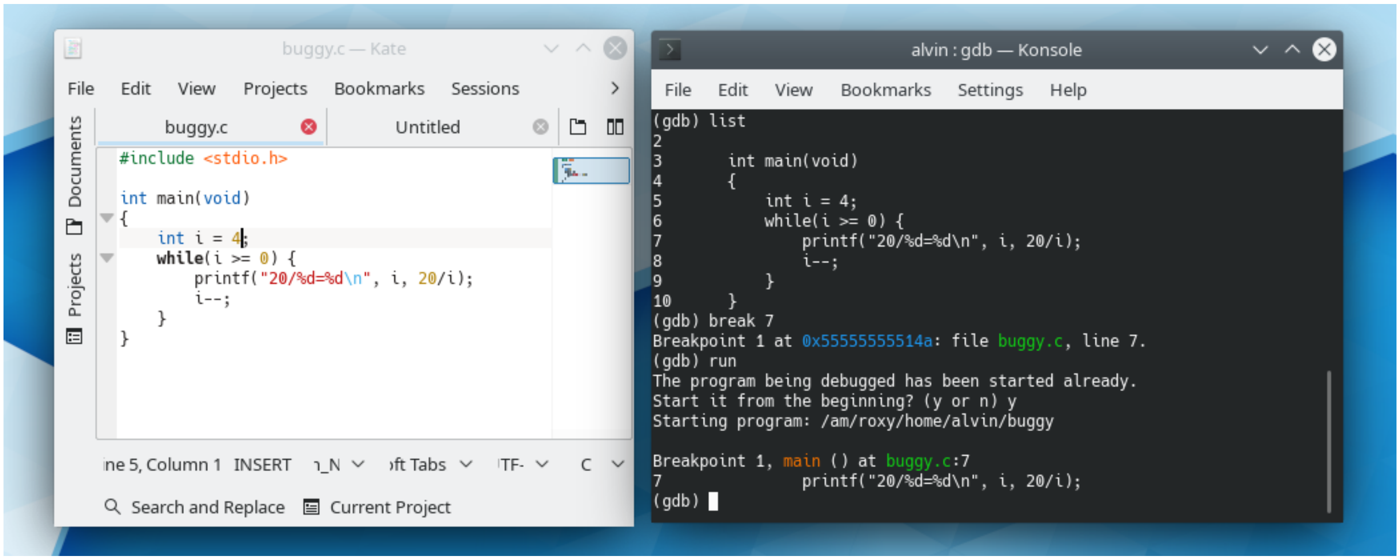
If you want to know more about the context of the exception, you can use the command `list`. This command shows the code around the exception.



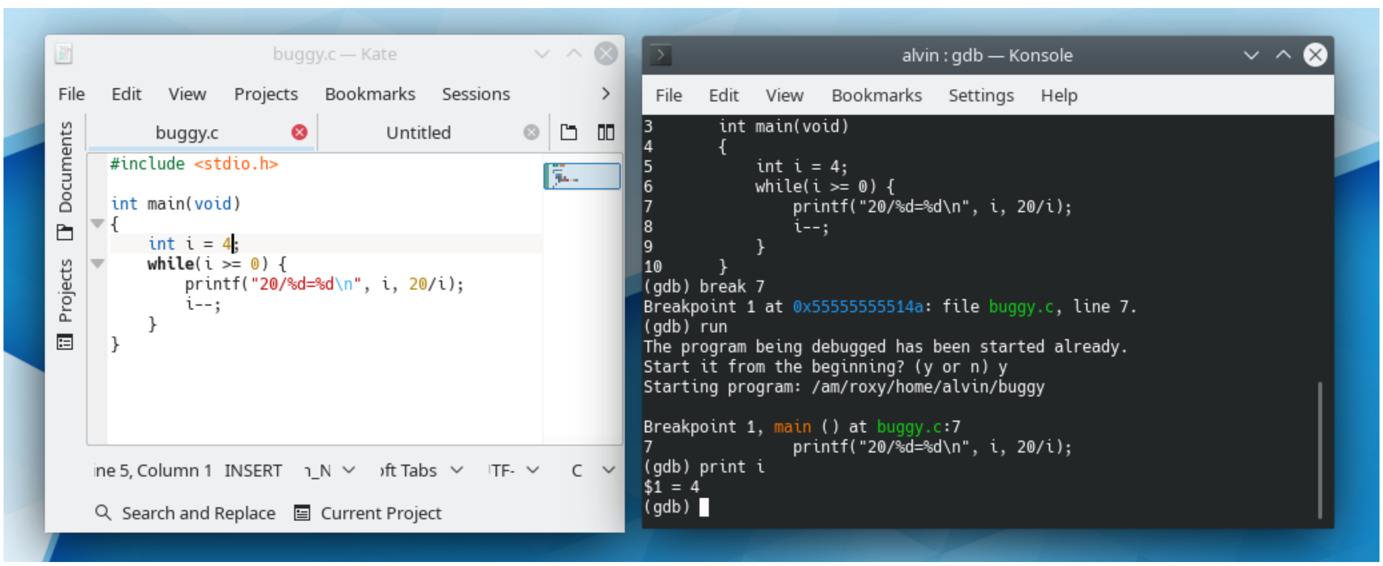
Let's do more advanced stuff with `gdb`. Let's begin with breakpoints. If you want to insert a breakpoint at line 7, just type the command `break 7`.



To run the program again, just type `run`. Answer yes when `gdb` asks whether you want to start from the beginning. You should see this:



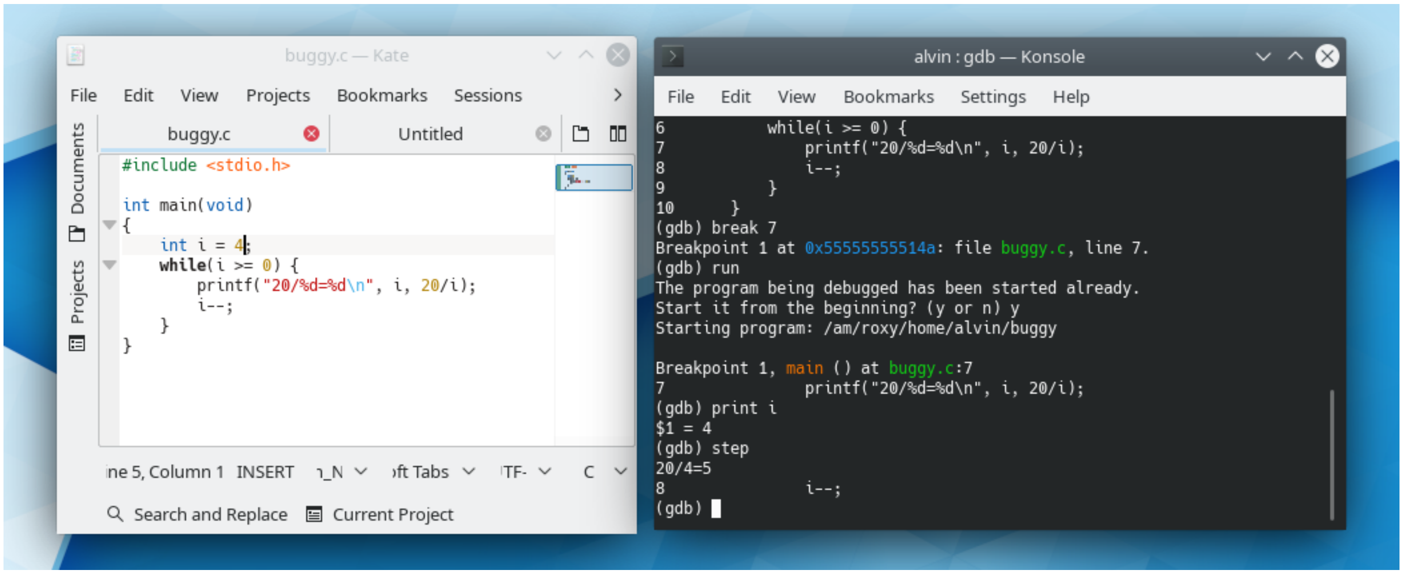
At this point, program execution paused at line 7 because of the breakpoint. You probably want to view the value of some variables in the program at this point. You can use the `print` command to do this. For instance, to view the value of `i` in our program, just type `print i`.



The output tells you that the current value of `i` is `4`. To let the program run execute the current line, you can use either the `step` or `next` command. There is a subtle different between these commands:

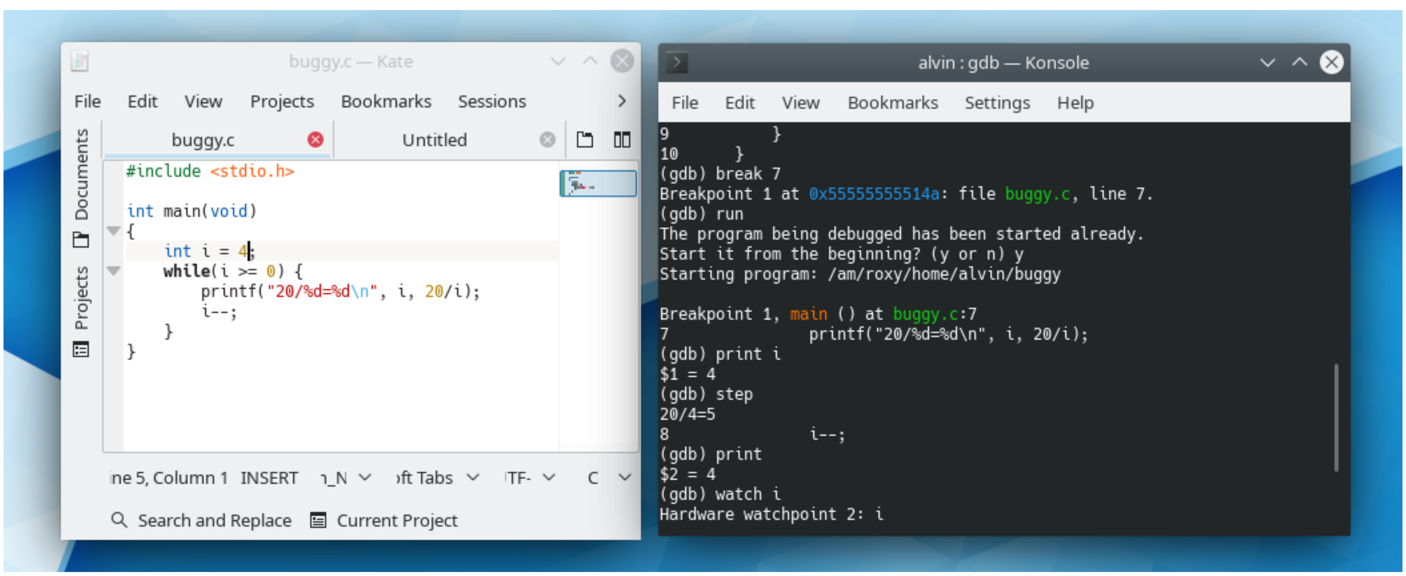
- `step` will step over functions
- `next` will actually step into functions

In our case, if we use `next`, gdb will enter into the `printf()` function and execute the function line by line. We don't want this. What we want is for `printf()` to be executed as a single line, i.e., we want to step over `printf()`. The appropriate command for this case is therefore `step`.

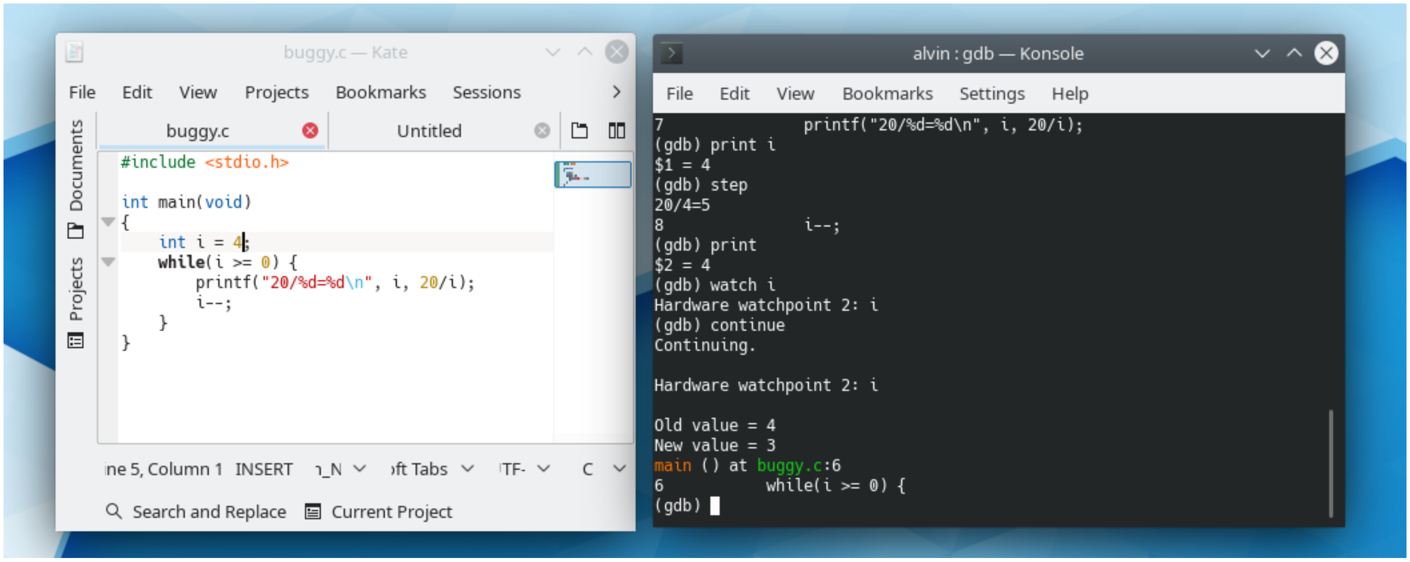


As you can see, line 7 is executed, generating the output `20/4=5`. The program is now halted at line 8. Try using the `print` command to display the value of `i`. Since line 8 is not yet executed, `i` should still be `4`.

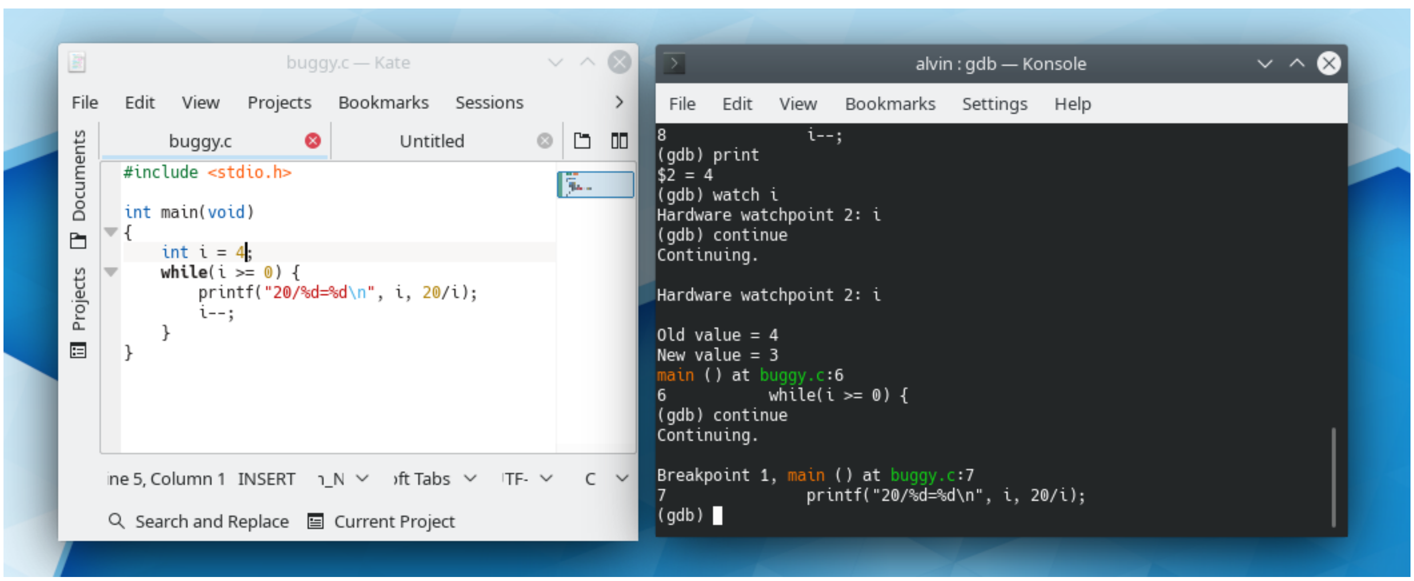
Note that using `print` to track a variable is cumbersome. A better way is to use the `watch` command. To watch variable `i`, just type `watch i`. This command will cause program execution to be paused everytime `i` is modified.



If instead of stepping you want the program to continue execution until the next breakpoint, just use the command `continue`.



Note that program is paused after the change of variable `i` from `4` to `3` because of our watchpoint on `i`. Type `continue` to proceed.



If you want to delete all breakpoints, use the command `delete`. To delete a specific breakpoint, specify the number, for instance, to delete the first breakpoint, type the command `delete 1`. Try doing this followed by `continue`.

Finish the debugging until the exception is encountered.

To quit `gdb`, just type `q`.

Below are other useful `gdb` commands:

Command	Purpose
`bt`	Show stack trace
`until`	Run until end of current loop
`finish`	Run until end of current function
`info break`	List all breakpoints
`clear <fun>`	Delete breakpoint set at `<fun>`
`x <addr>`	Print content at `<addr>`

Exercises

You may download a copy of the base source files used in the activities from `https://ecs.wgtn.ac.nz/foswiki/pub/Courses/NWEN241_2021T1/Exercises/nwen241_exercise2_files.zip`.

Activity 1: Using GDB [50 Marks]

Copy and paste the following C program to your favorite text editor:

```
1  #include <stdio.h>
2  #include <ctype.h>
3
4  void capitalize(char *str)
5  {
6      int i=0;
7      while(str[i] != '\0') {
8          if (islower(str[i])) str[i] = toupper(str[i]);
9          i++;
10     }
11 }
12
13 int main(void)
14 {
15     char *s = "ABC123 is the most common password";
16     capitalize(s);
17     printf("%s\n", s);
18 }
```

Save the file as `activity1.c`. Compile and run the program. The program should crash, with the rather terse message `Segmentation fault`.

Now, you will need to find out why the program is crashing. Recompile the program with the `-g` option. Load the program in `gdb` and run. What line does the program crash? What is the value of `i` when the program crashes? Why to do you think the program crashes?

Fix the program so that it will not crash. You must only fix one line (line 15 to be exact.) Submit the fixed program to the Assessment System for marking.

Activity 2: Structure [15 Marks]

Copy and paste the following C program to your favorite text editor:

```c
#include <stdio.h>

// Define structure record

// Implement print_record() function

int main(void)
{
    struct record rec;

    scanf("%s %d %f", rec.name, &rec.age, &rec.height);
    print_record(rec);
    return 0;
}
```

Save the file as `activity2.c`. Study the source file and do the following within the file:

1. Define a structure named `record` with the following fields:
    - `name`: a string variable that can hold 40 characters (including null terminator)
    - `age`: a short integer
    - `height`: a single precision floating point number
2. Implement a function named `print_record()` which accepts a single argument of type `struct record` and does not return anything. The function should use `printf()` to display the `name`, `age` and `height` fields. The output must follow this format:

    ```
    Name   : name
    Age    : age
    Height: height
    ```

    The height should be displayed with precision of 2 decimal places. As your submission is going to be auto-marked, do not add any extra character in the output, including `'\n'` after the last line.

Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking.

Activity 3: Passing Pointer to a Structure [10 Marks]

One disadvantage of the code in Activity 2 is that the entire structure is copied to the function. This can be highly inefficient, especially when the structure being passed is large. An approach to address this issue is to pass pointer to a structure.

In this activity, you will repeat Activity 2, except that you will be passing a pointer to a structure in the function invocation. To begin, copy and paste the following C program to your favorite text editor:

```c
#include <stdio.h>

// Define structure record

// Implement print_record_ptr() function

int main(void)
{
    struct record rec;

    scanf("%s %d %f", rec.name, &rec.age, &rec.height);
    print_record_ptr(&rec);
    return 0;
}
```

Save the file as `activity3.c`. Study the source file and do the following within the file:

1. Define a structure named `record` with the following fields:
   - `name`: a string variable that can hold 40 characters (including null terminator)
   - `age`: a short integer
   - `height`: a single precision floating point number
2. Implement a function named `print_record_ptr()` which accepts a single argument of type pointer to `struct record` and does not return anything. The function should use `printf()` to display the `name`, `age` and `height` fields. The output must follow this format:

   ```
   Name  : name
   Age   : age
   Height: height
   ```

   The height should be displayed with precision of 2 decimal places. Do not add any extra character in the output, including `'\n'` after the last line.

Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking.

Activity 4: Pointers [25 Marks]

In this activity, you will use pointers to traverse an array and access array elements. Begin by copying and pasting the following C program to your favorite text editor:

```c
#include <stdio.h>

#define MAX 100

// Implement find_max() function using pointer
int *find_max(int *a, int alen)
{
    // These variables should be enough. Do not declare any other variable.
    int *max = a, *p = a;

    // Use pointer p to iterate over arrays and access arrays elements
    // Use max to point to the latest maximum value found
    // Use either while-loop or for-loop to iterate


    // max should point to the maximum
    return max;
}

int main(void)
{
    int n, array[MAX];
    scanf("%d", &n);
    for(int i=0; i<n; i++)
        scanf("%d", array+i);
    int *max = find_max(array, n);
    printf("%d", *max);
    return 0;
}
```

Save the file as `activity4.c`. Study the source file and implement the `find_max()` function using pointers. The function accepts two input arguments: the first argument is a pointer to the first element of the array, and the second argument specifies the number of elements in the array. The function must return a pointer to the element with maximum value.

Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking.