SCC.111: DEBUGGING

INTRODUCTION

The purpose of this lab is to give you experience with program troubleshooting and debugging using a debugger (`VS Code` debug view and `gdb`). We will be using live code debugging on several occasions during the course to explain the underlying architecture of the operating system, and how programs run in detail.

GOALS

- [ ] Learn how to use common debugging mechanisms.
- [ ] Understand the differences between compilation and run-time errors.
- [ ] Get familiar with debugging processes.

THE TASK

The following code contains a number of different errors and problems. Copy the code into a new `.cpp` file on your local system and try to compile it (*i.e.,* `g++ -g -o dice dice.cpp`). The program contains *a series of different errors* and your goals is to make the program executable.

Easy version

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

class Dice {
    int numSides,guess;
    public:
        Dice(int sides);
        void roll();
```

```
};

D ce::Dice(int sides) {
    numSides = sides;
    return sides;
}

int roll(int numSides) {
    int rollResult = rand() % numSides;
    printf("The Dice actually rolled: ");// debugging
    printf(r llResult); // debugging
    numSides = r llResult;
    return roll(rollResult);
}

int main() {
    sr nd(time(NULL)));

    int numSides = 0; // set number of sides
    Dice sixSidedDice;

    int rolled_number = r ll() % 2;

    printf("The program thinks it rolled: ");
    printf(rolled_number);
    return 0;
}
```

Full version

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

#define DEBUG 1

class Dice {
    int numSides,guess;
    public:
        Dice(int sides);
        void roll();
        Dice(){};
};

Dice::Dice(int sides) {
    if(sides >= 2)
        numSides = sides;
    else
        printf("You can't create a Dice with less than two sides. A default value of
 ↪  6 is chosen.");
        numSides = 6;
    return sides;
}
void printResult(Dice d, int rolled_number) {
    if (&d.guess-rolled_number==0)
        printf("Better luck next time! the dice rolled %n\n", rolled_number);
    else
        printf("Lucky you! You WON..\n");
}
```

```
int roll(int numSides) {
    int rollResult = rand() % numSides;
    numSides = rollResult;
    if(!DEBUG);
        printf(rollResult);
    return roll(rollResult);
}

void makeAGuess(Dice d) {
    printf("Make a guess from 1 to %n.\n",d.numSides);
    scanf("%d",d.guess);
}

void main() {
    srand(time(NULL)));

    int numSides = 0;
    printf("Enter the number of sides: ");
    scanf("%d\n",numSides)
    Dice sixSidedDice; return;

    int rolled_number = roll();

    makeAGuess(d);

    printResult(&Dice,numSides);


    return;
}
```

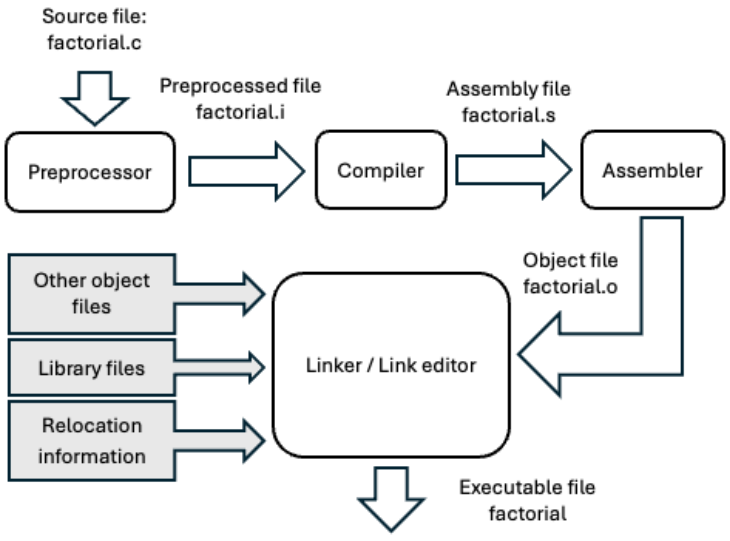
Work on the easy version first before moving onto the full one

COMPILE-TIME ERRORS

Compile-time errors and warnings are detected by the compiler as it tries to compile the source code into machine code. They usually involve syntax errors or problems with type checking. Errors will terminate the compilation prematurely and will not generate a binary executable. Warnings are suggestions to improve your code, but they will not stop compilation—*you should look at them as something is wrong though!* Compilers are becoming increasingly smart, and they will frequently detect logical errors in your code and warn you about it. It **is good practice** to use the flag `-Wall` with your gcc compiler and treat warnings as errors that stop program compilation, so you can't miss them!

Compile the program above and fix the errors and warnings reported by the compiler.

LINK-TIME ERRORS



GCC is an advanced tool that embeds a number of different steps in the compilation process. In a strict definition a compiler is a tool that translate a program written in a high-level programming language into

assembly code. Nonetheless, for ease of use when you run gcc with the default flags, a four-stage process takes place that transform your code into the final binary executable.

A common error that you will face when you compile a program are link-time errors, *i.e.,* errors that occur when the linker tries to combine various compiled source files into the final single executable file and can't find all the necessary pieces. These often happen due to missing function implementations, mismatches in how a function is declared and how it is called, or incorrect library linking.

Can you identify the link error by reading the GCC report in the previous program and fix it?

DEBUGGING

Debugging is a mechanism that allows a developer to get maximum control of a program execution and trace run-time and logical code errors. This is *your best tool* to fix your programs when they misbehave when printf and dry running have failed you.

Debugging programs using `gdb` (GNU Debugger) is a systematic process that involves several steps. Here's a guide to get you started with basic debugging using `gdb` . For this explanation, I'll assume you're using a Unix-like environment and your program is written in C++.

1.Compile the Program with Debug Information

To use a debugger, first compile your C program with the `-g` option. This tells the compiler *to include* debug information in the executable, which `gdb` can use to provide more detailed information during debugging.

```
g++ -g dice.cpp -o dice
```

2.Start the debug session

The flag `-g` will keep important debugging information in the final `dice` executable. This increases the size of the binary, but is important so you can see the program symbols and code lines as you debug. In order to debug code using VS.code follow the steps below.

VS code and plugins (optional) If you are debugging the code on your personal computer, you will need to install the following software components. **Note: lab machine already have the relevant packages already installed.**

1. **Install C/C++ Extensions**: Open VS Code, go to the Extensions view by clicking on the square icon on the sidebar, or pressing `Ctrl+Shift+X` . Search for "C/C++" by Microsoft and install it.
2. **Install a Compiler**: Ensure you have a C++ compiler installed. On Windows, you can use WSL, MinGW or Microsoft Visual Studio compiler. On Linux and MacOS, `gcc/g++` command line developer tools will need to be installed using a package manager or `xcode-select` respectively.

Setting Up Your Project

1. **Open Your Project**: Create a folder on your local filesystem and create a `dice.cpp` file inside it, containing the code at the top of this document. You should the open the folder containing your project in `VSCode` (`Ctrl+Shift+O`). Alternatively, close `VSCode` , in the terminal navigate to your new directory that has the ' `dice.cpp` ' file, execute the command `code .` to open the directory in VSCode as a workspace.
2. **Create a Configuration File**: On lefthand sidepanel, open the `dice.cpp` file by left clicking on it. Go to the Run view by clicking the play icon on the sidebar or pressing `Ctrl+Shift+D` . Click on "create a launch.json file", click on "Add Configuration.." in the bottom right corner, then select "gdb (launch)". Change the "program" line to have the name of the new program, ie "dice", without adding the ' `.cpp` ' extension.
3. **Configure tasks.json**: This file tells VS Code how to compile and run your program. Navigate back to `dice.cpp` file and click the play icon on the top right of the `VSCode` window, and click "Debug". Select the first task from the center drop-down for gcc. This will automatically create a `tasks.json` file in a `.vscode` subfolder. While the program still have compilation errors, it will show an error dialogue, select "Show errors" to show them in the bottom pane. When you have the program eventually compiling, you will see gdb output in the bottom panel. You may need to select "Terminal" in the bottom panel to see the program output and input prompt!

comment: <> (#### Writing a Task to Compile Your Code

1. **Open the Command Palette**: Press `Ctrl+Shift+P` and type "Tasks: Configure Task".

2. **Create tasks.json File**: Select "Create tasks.json file from template", then "Others". Replace the content with a script to compile your C++ code. For `g++`, it might look like this:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build my project",
            "type": "shell",
            "command": "g++",
            "args": [
                "-g",
                "dice.cpp",
                "-o",
                "dice"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

) #### Debugging Your Program

3. **Set Breakpoints**: Open the C++ file you want to debug, and click to the left of the line numbers in the editor to set breakpoints.

4. **Start Debugging**: Press `F5` or go to the Run view and click the green play button. This will compile your program and start a debugging session.

5. **Inspect Variables and Control Execution**: Use the Debug view to step through your code, inspect variables, watch expressions, view the call stack, and control the execution flow.

6. **Use the Debug Console**: The Debug Console in VS Code can be used to evaluate expressions at runtime, inspect variables, or execute commands.

Control Program execution Breakpoints can be toggled by clicking on the editor margin or using F9 on the current line. Finer breakpoint control (enable/disable/reapply) can be done in the Run and Debug view's BREAKPOINTS section. Breakpoints will trigger the debugger to pause the execution of the program at a specific location in order to inspect the state of the program. Breakpoints in the editor margin are normally shown as red filled circles. Disabled breakpoints have a filled gray circle. When a debugging session starts, breakpoints that cannot be registered with the debugger change to a gray hollow circle. The same might happen if the source is edited while a debug session is running. Right-click on a breakpoint to add a condition that must be true for the debugger to stop at that breakpoint.

A Logpoint is a variant of a breakpoint that does not "break" into the debugger but instead logs a message to the console. Logpoints are especially useful for injecting logging while debugging production servers that cannot be paused or stopped. A Logpoint is represented by a "diamond" shaped icon. Log messages are plain text but can include expressions to be evaluated within curly braces ('{}'). Unfortunately, expression evaluation does not work well for ARM Assembly.

If you want to learn more about debugging using the VS code, please visit the following article.

Tips Remember, debugging is an iterative process. You might not find the issue on your first try, but with careful examination and a bit of patience, `gdb` can be an extremely powerful tool for diagnosing and fixing problems in your code.

Your program contains logical errors that make the dice game to misbehave and report incorrect results. Can you trace the logical errors and fix them using debugging?

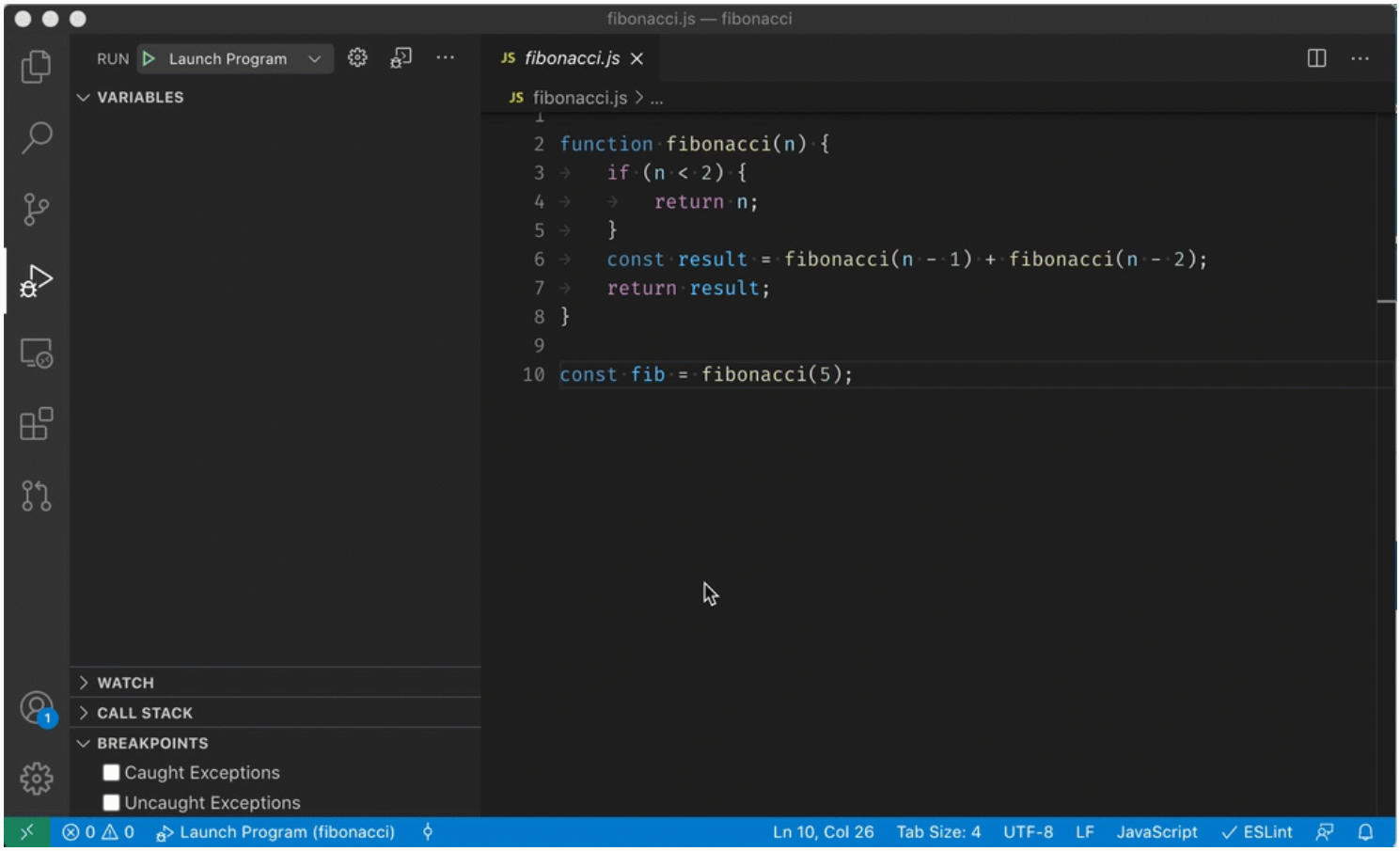


Figure 1: Adding a logpoint using vs.code

SCC.121 INTRODUCTION TO ALGORITHMS AND OPERATION COUNTING

This week's lab activities will focus on the time complexity of algorithms in terms of the number of operations executed. In later weeks we will build on this knowledge to consider asymptotic analysis of time complexity and the big-O, big-omega, and big-theta notations. These are fundamental concepts for computer scientists and software engineers and enable us to compare different algorithms theoretically in terms of their time efficiency.

The goal for today is for you to practice operation counting to determine the time complexity (in terms of the number of steps relative to the input size) of some simple pieces of code. You will also experimentally investigate the time complexity of a linear search algorithm. Note that when counting operations, we ignore the minor details such as differences between programming languages and compilers as discussed in the 2nd lecture in week 12.

The first exercises should be done using pen and paper. For the last question, you will need to experimentally investigate the time complexity using a computer.

1. How many times is print("scc121") executed for each code snippet?

```
for (i = 0; i < n; i++){
    printf("scc121");
}
```

```
for (i = 1; i < n; i++){
    printf("scc121");
}
```

```
for (i = 0; i =< n; i++){
    printf("scc121");
}
```

```
for (i = 0; i < n; i+=2){
    printf("scc121");
}
```

2. Count the number of operations (in the worst case, average case and best case) that the piece of code below executes. Assuming size of the input array n is greater than 1.

```
double func_1(int theArray[], int n ){
    int total = 0;
    for (int i=0; i<n; i++){
        total += theArray[i];
        break;
    }
    double avg = ((double) total) / ((double) n);
    return avg;
}
```

3. Given the following piece of code:

```
int findMaxArr(int theArray[], int n){
    int max_ind = 0;
    for (int i = 1; i < n; i++)
        if ( theArray[ i ] >= theArray[ max_ind ] )
            max_ind = i;
    return max_ind;
}
```

- Write an example of the input for the **worst case** scenario
- Compute the time taken by this algorithm in the **worst case** by performing operation counting. What is the overall function time T(n)?
- Write an example of the input for the **best case** scenario
- Compute the time taken by this algorithm in the **best case** by performing operation counting. What is the overall function time T(n)?

4. Given the following input, problem and output:

   Input: An array of integers (*theArray*) with length *n* and an integer element *val* to find.
   Problem: Find the index of a given element (*val*) within the array (*theArray*)
   Output: Return the **index** of the integer element being searched for in the array. Note: if the element occurs more than once in the array return the lowest index value, if the element does not occur in the array then return -1.

   - Write a function for this problem in C, starting with this:

```
int linearSearch(int theArray[], int n, int val)
```

   - What is the time complexity, T(n), in the **worst case**?
   - What is the time complexity, T(n), in the **best case**?

5. Experimentally explore how the run time of your `linearSearch` function varies in the best vs worst case by exploring the time taken to work on arrays with different lengths and different input organisations (see below for some code you may find useful in doing this).

- Plot a graph of how the run time of the linearSearch varies for arrays of different lengths in the best vs worst case. This should include repeats of each array length.
- How do your results match with the theoretical time complexity?
- What are the difficulties in using an experimental method to determine the time complexity of an algorithm?

To answer 121 Question 4 you may find the following useful:
To calculate the time a process takes to run in c you can use the time.h header file and the predefined function clock(). To get the elapsed time, we can get the time using clock() at the beginning, and at the end of the tasks, then subtract the values to get the difference. We can then divide the difference by CLOCK_PER_SEC (Number of clock ticks per second) to get the processor time. For example to determine how long a function fun() takes to run, we can use the following

```
#include <stdio.h>
#include <time.h>
// this can be used to get run time of a function in c
```

```
void fun(){
//function
}

// The main program calls fun() and measures time taken by fun()
int main()
{
   // Calculate the time taken by fun()
   clock_t t;
   t = clock();
   fun();
   t = clock() - t;
   double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
   printf("fun() took %f seconds to execute \n", time_taken);
   return 0;
}
```

To produce the plot, you can save your data to a file, as below. Then, you can plot the results using another program such as excel. The following code opens a file, writes two data columns to the file, and closes the file.

```
#include <stdio.h>
// this can be used to save date to a file in c

int main()
{
    //file name that is used to store the data
    char  *file_name="data_file.csv";

    // open the file for writing - use "w"
```

```
    FILE *file=fopen(file_name,"w");

    int j=1;

    for (j=0; j<10; j++)
    {
        fprintf(file,"%d,%f\n", j, (float)1.3*j);
    }

    fclose(file);
    return 1;
}
```

SCC.131: DEBUGGING THE MICRO:BIT

INTRODUCTION

Debugging is an indispensable skill in the world of programming and electronics, and it plays a pivotal role in ensuring the functionality and reliability of embedded systems. In this academic exercise, we delve into the exciting realm of debugging with a focus on the micro:bit platform.

This exercise offers a walkthrough on debugging processes in the context of the micro:bit. We build on the SCC.111 experience and explain the process to debug programs running on an embedded system. **This tutorial is essential in order to run the lab material that we will cover during the assembly topic in SCC.131.**

GOALS

- ☐ Use the micro:bit serial console.
- ☐ Debug a micro:bit program.
- ☐ Familiarise with debugging processes.

FIBONACCI IN THE MICRO:BIT

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1. In mathematical terms, the Fibonacci sequence is defined recursively as follows:

$$F(0) = 0$$

$$F(1) = 1$$

$$F(n) = F(n-1) + F(n-2) for n > 1$$

So, the sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding numbers. The Fibonacci sequence typically looks like this:

> 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The Fibonacci sequence has numerous applications in mathematics, science, and nature. It appears in various natural phenomena, such as the arrangement of leaves on a stem, the branching of trees, and the spiral pattern of a pine cone or a sunflower.

Below we provide a ***buggy*** implementation of a Fibonacci sequence program for the micro:bit.

```
#include "MicroBit.h"

MicroBit uBit;

// Function to compute the Fibonacci sequence up to n terms
void computeFibonacci(int n) {
    int *fibs;

    fibs = (int *)malloc(n*sizeof(int));

    uBit.serial.printf("Fibonacci Sequence (First %d terms):\r\n", n);

    for (int i = 0; i < n; i++) {
        fibs[i] = fibs[i-1] + fibs[i-2];
        uBit.sleep(500); // Delay for readability (500ms), you can comment it out, if
↪   it slows down your debugging
    }

    uBit.serial.printf("result is %d\r\n", fibs[n]);
    free(fibs);
}

int main() {
    uBit.init();

    uBit.serial.printf("Micro:Bit Fibonacci Sequence Calculator\r\n");
```

```
    while (1) {
        // print prompt for input number
        uBit.serial.printf("Enter the number of terms (0 to exit): ");
        int n = 0;

        // read user input until a newline is sent and store string in
        // variable ret.
        ManagedString ret = uBit.serial.readUntil("\r\n");
        for (int i = 0; i < ret.length(); i++) {
            n = n*10 + ret.charAt(i) - '0';
        }
        // By default the serial input will not appear on the screen,
        // so we need to print it explicitly
        uBit.serial.printf("%d\r\n", n);

        if (n == 0) {
            continue; // Exit the program if the user enters 0
        }

        computeFibonacci(n);
    }

    uBit.serial.printf("Goodbye!\r\n");
    release_fiber(); // Release the main fiber to stop the program

    return 0;
}
```

Before it is possible to debug a problem, you must convince yourself that it is possible to debug the problem.

MICRO:BIT DEBUGGING WITH VS CODE

Step 1: Reproduce the problem

As you might have noticed, your program uses the serial port to communicate with the user. The method `uBit.serial.printf` prints an integer on the serial port, while the method `uBit.serial.readUntil` will read user input from the serial port as a string.

In order to connect to the serial port, you can use the command-line program `screen`. Screen is a terminal multiplexer in Linux. Running `man screen` on your terminal will allow you to read more.

For your task, we will use screen to connect to the serial port of your device. Every time you connect your micro:bit to your Linux lab machine, a serial port file is created automatically by the operating system. To connect to the serial port you should run the command (the folder you in when you run this command is not important):

```
screen /dev/ttyACM0 115200
```

Congratulations, you are now running a serial connection to your micro:bit. By default, all keys presses are sent to the program running inside the screen session. You can control the screen session using screen shortcuts. They typically start with the control sequence `Ctrl+A`. In order to exit a screen session, you use the sequence `Ctrl+A` followed by `X` to kill the current session.

Note: You can only connect a single screen session at any point with the serial file. If your screen command returns an empty serial, or it complains that the file is busy, it probably means that you have not terminated successfully a previous serial session. You reattach to an already running session using the command `screen -rd`.

Connect to the terminal and test if the program runs correctly. You can try different inputs and check if the results are correct. At this stage, it is valid to use printf statement to get information. Nonetheless, debugging can greatly increase the accuracy.

STEP 2: RUN A DEBUG SESSION

Debugging micro:bit code using VS Code does not greatly differ from the debugging process that you used for the SCC.111 task. The biggest difference is in the debug session setup, but many of these steps are

abstracted by the build system and the project configuration. Here's a guideline to help you through this process:

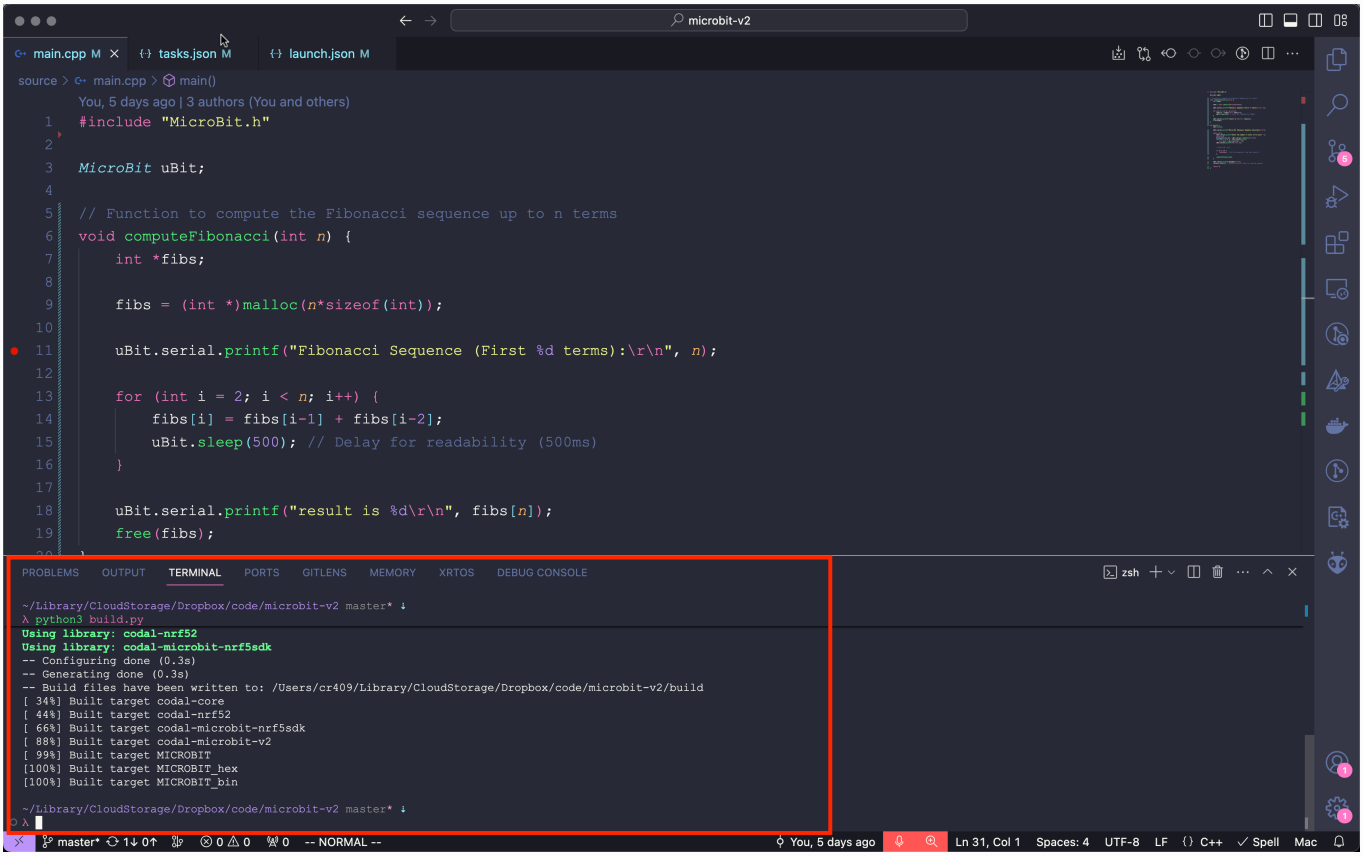
1. Your lab machine offers all the required resources to execute and debug your code, so you can safely skip this step. Nonetheless, if you want to try running things on your system, you should download the following programs and plugins.

   - Install VS Code: If you haven't already, download and install Visual Studio Code.
   - Install Extensions: Install extensions relevant for ARM Assembly and debugging, such as:
     - *C/C++ Extension*: For additional debugging capabilities.
     - *Cortex-Debug*: Specifically useful for debugging ARM Cortex-M processors.
     - *ARM Assembly Language Support:* for syntax highlighting and other Assembly-specific features.

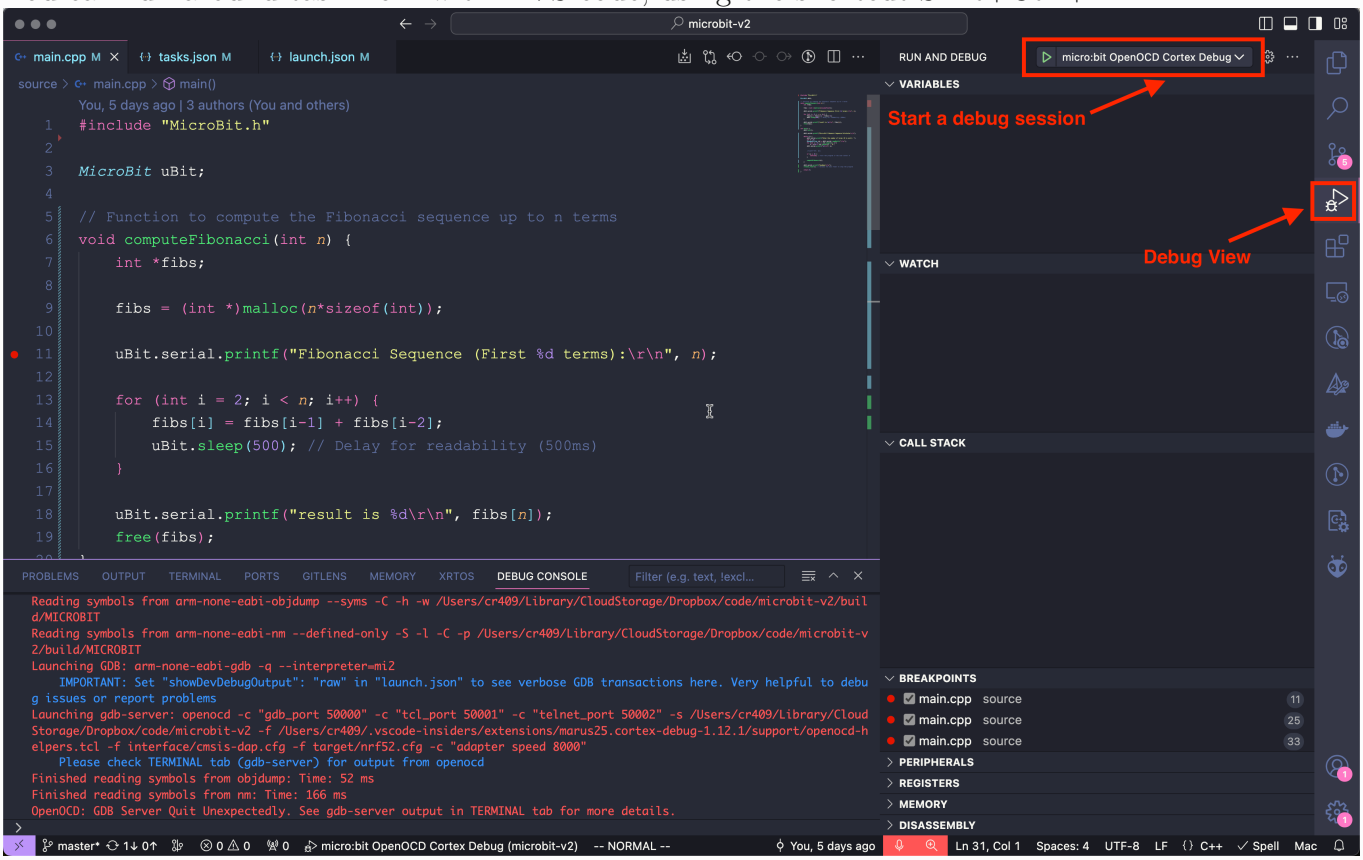
2. Open the main folder of the project from within the VS code editor (e.g., microbit-v2-samples, microbit-v2 depending on the naming you used). You can use Ctrl+O or use the menu entry `File -> Open Folder...` to open the file browser. Please note that you must open the folder of the code, not an individual file from the project. This step will force VS code to load project configuration files and allow your project to be integrated with the editor automation. If you have performed this step correctly, a number of configurations will start on your program and you will be asked to install the cortex-debug plugin.

3. In order to build your code, you have two options:

   - Run the command `python3 build.py` within the project folder. You can access a terminal using the bottom pane of your editor.



- You can run a build task from within VS code, using the shortcut Shift+Ctrl+B.

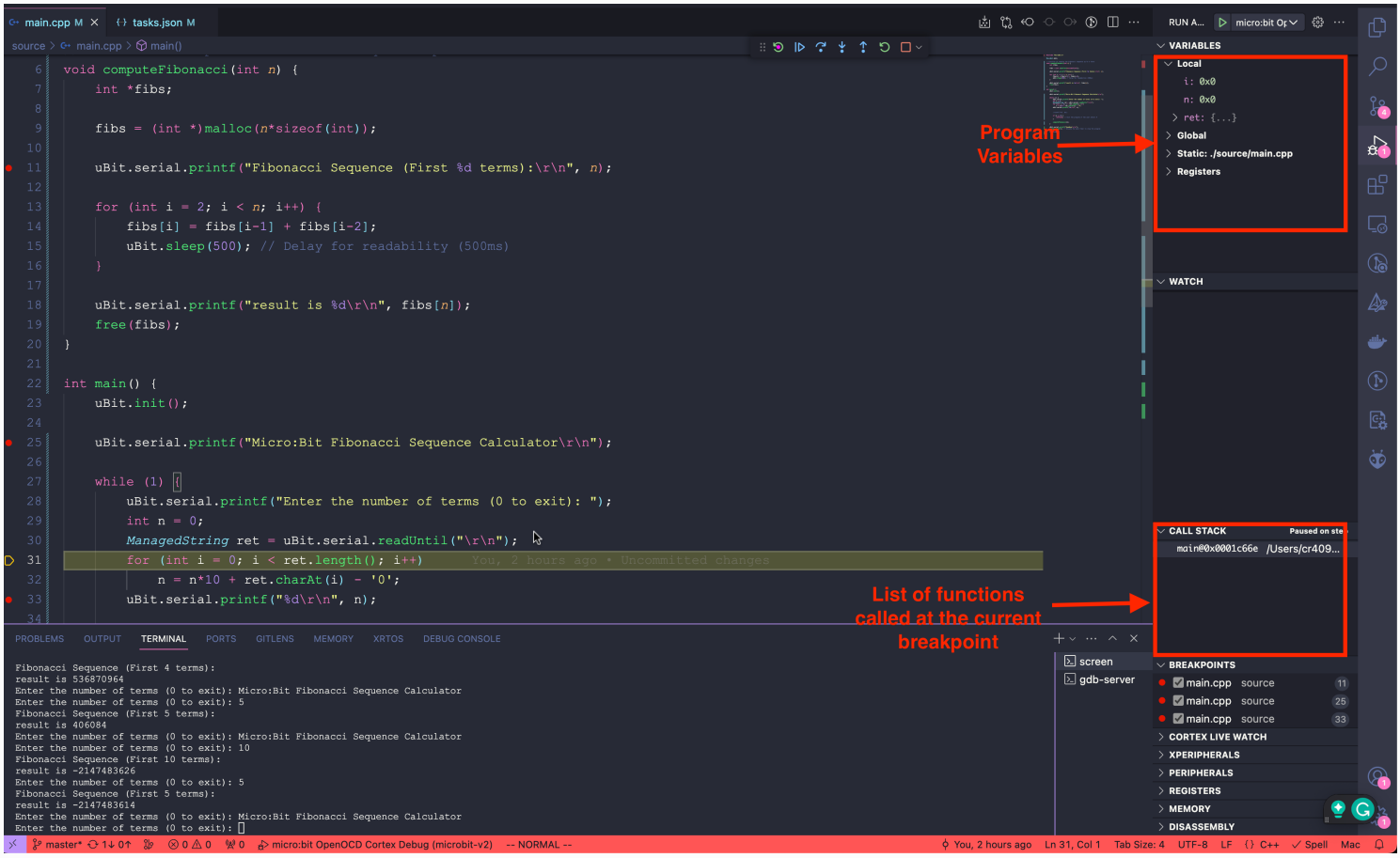


4. Open the debug view on the side panel of VS code. In order to run a debug session, you should select the *micro:bit OpenOCD Cortex Debug* and press the play button. This step will flash the latest generated hex file on the micro:bit and start the debugger. Make sure that the micro:bit is connected and that you use a data USB cable. (**Note: You do not need to move files on the MICROBIT folder to install the code. Files will be automatically transferred to the micro:bit filesystem by the debugger. Also, it might take a few trial and error attempts to get debugging working. Many times simply disconnecting and reconnecting your micro:bit could solve the problem.**)

5. We are ready now to start the debug process. Things do not greatly differ from the processes described in the SCC.111 task. You can add breakpoints, print the state of different variables and test the logic of your program. **Do not forget to start a screen session to input your parameters**

STEP 3: FIX THE PROBLEM

Execute the code step by step and inspect the state of every variable. The debug view of VS Code should provide you access to the complete state of the program.


main.cpp M
tasks.json M
source > main.cpp > main()
void computeFibonacci(int n) {
int *fibs;
fibs = (int *)malloc(n*sizeof(int));
uBit.serial.printf("Fibonacci Sequence (First %d terms):\r\n", n);
for (int i = 2; i < n; i++) {
fibs[i] = fibs[i-1] + fibs[i-2];
uBit.sleep(500); // Delay for readability (500ms)
}
uBit.serial.printf("result is %d\r\n", fibs[n]);
free(fibs);
}
int main() {
uBit.init();
uBit.serial.printf("Micro:Bit Fibonacci Sequence Calculator\r\n");
while (1) {
uBit.serial.printf("Enter the number of terms (0 to exit): ");
int n = 0;
ManagedString ret = uBit.serial.readUntil("\r\n");
for (int i = 0; i < ret.length(); i++)
n = n*10 + ret.charAt(i) - '0';
uBit.serial.printf("%d\r\n", n);
Program Variables
List of functions called at the current breakpoint
VARIABLES
Local
i: 0x0
n: 0x0
ret: {...}
Global
Static: ./source/main.cpp
Registers
WATCH
CALL STACK
BREAKPOINTS
main.cpp source 11
main.cpp source 25
main.cpp source 33
CORTEX LIVE WATCH
XPERIPHERALS
PERIPHERALS
REGISTERS
MEMORY
DISASSEMBLY
PROBLEMS OUTPUT TERMINAL PORTS GITLENS MEMORY XRTOS DEBUG CONSOLE
Fibonacci Sequence (First 4 terms):
result is 536870964
Enter the number of terms (0 to exit): Micro:Bit Fibonacci Sequence Calculator
Enter the number of terms (0 to exit): 5
Fibonacci Sequence (First 5 terms):
result is 406084
Enter the number of terms (0 to exit): Micro:Bit Fibonacci Sequence Calculator
Enter the number of terms (0 to exit): 10
Fibonacci Sequence (First 10 terms):
result is -2147483626
Enter the number of terms (0 to exit): 5
Fibonacci Sequence (First 5 terms):
result is -2147483614
Enter the number of terms (0 to exit): Micro:Bit Fibonacci Sequence Calculator
Enter the number of terms (0 to exit):
screen
gdb-server
master*
micro:bit OpenOCD Cortex Debug (microbit-v2)
-- NORMAL --
You, 2 hours ago
Ln 31, Col 1
Tab Size: 4
UTF-8
LF
C++
Spell
Mac

HACKER EDITION

SCC.111: DEBUGGING WITH GDB

VS code operates as a wrapper around a command-line tool called GDB. In this task, we will help you to setup and run a debug session using the command line.

1. Start gdb

By following the guide in the main task description, you can generate a program binary with debug information. To start debugging your program, run `gdb` followed by the name of the executable:

```
gdb ./program
```

2. Set Breakpoints

A breakpoint is a point in your program where execution will stop, allowing you to examine the state of the program. Set breakpoints at lines where you want to pause execution. For example, to set a breakpoint at line 10:

```
(gdb) break 10
```

You can also set a breakpoint at a function:

```
(gdb) break myFunction
```

3. Run the Program

Run your program within `gdb` using the `run` command. You can also pass command line arguments if necessary:

```
(gdb) run [arguments]
```

The program will execute until it hits a breakpoint, or finishes if there are no breakpoints.

You can inspect the program code and the current execution point using the command `list` .

4. Examine the State of the Program

When the program hits a breakpoint, you can examine variables, memory, and the call stack. For example:

- To print the value of a variable `n` , use `print n` or `p n` :

```
(gdb) print n
```

- To view the call stack, use `backtrace` or `bt` :

```
(gdb) backtrace
```

- To list the source code around the current line, use `list` or `l` :

```
(gdb) list
```

5. Step Through the Program

`gdb` allows you to step through your program line by line:

- `next` or `n` : Execute the next line of the program (stepping over function calls).
- `step` or `s` : Execute the next line, stepping into functions.
- `continue` or `c` : Continue execution until the next breakpoint or the end of the program.

6. Modifying Program State

You can also modify the value of variables on the fly:

```
(gdb) set n = 1
```

Tips for Effective Debugging with gdb

- Use the `watch` command to stop execution whenever the value of a variable changes.

- Utilize conditional breakpoints if you want to stop execution only when certain conditions are met.
- `info` command can be used to get information about various aspects, like `info locals` to view local variables and `info register` will print the CPU registers.

7. Quitting gdb

To exit `gdb`, use the `quit` command or `Ctrl-D`.

SCC.131: MICRO:BIT DEBUGGING WITH GDB

This task aims to familiarize you with a command-line debug toolchain. A similar mechanism is used by VS Code to support debugging, but the low level details are abstracted through the UI environment.

Step 1: Start a gdb server

The first step in this process requires you to use the openocd software, to load the hex MICROBIT file and start the debug server. In order to start this process you should run the following command:

```
openocd -f interface/cmsis-dap.cfg -f target/nrf52.cfg -c "program build/MICROBIT
↪  verify reset"
```

Step 2: Connect to the GDB server

In order to interact with the debug session, the process is similar to the debug process discussed in the SCC.111 task. You should run the following command to load the MICROBIT code with debugging information by the GDB client and to connect the client to the GDB server session.

```
> arm-none-eabi-gdb build/MICROBIT
GNU gdb (Arm GNU Toolchain 12.2 (Build arm-12-mpacbti.34)) 13.1.90.20230307-git
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=aarch64-apple-darwin20.6.0
↪  --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
```

```
For help, type "help".

(gdb)
```

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x0002c1c2 in target_wait_for_event ()
    at
    ↪  /Users/cr409/microbit-v2/libraries/codal-nrf52/source/codal_target_hal_base.cpp:41
41  }
(gdb) mon reset
(gdb)
```

3. Use GDB commands to control the debug session

Here's a table with some basic GDB (GNU Debugger) commands commonly used for debugging C and C++ programs. Commands like `run` will not work with embedded debugging.

Command	Description
`break <function>` or `b <function>`	Set a breakpoint at the specified function or line number.
`delete <breakpoint_number>`	Delete a breakpoint by its number.
`info breakpoints`	List all active breakpoints.
`list` or `l`	Display the source code around the current point of execution.
`next` or `n`	Execute the current line and stop at the next line of code.
`step` or `s`	Execute the current line and step into any function calls.
`continue` or `c`	Resume program execution until the next breakpoint is encountered.
`backtrace` or `bt`	Display a backtrace of the function call stack.

`frame <frame_number>`	Select a specific frame from the call stack.
`print <variable>` or `p <variable>`	Display the value of a variable.
`display <variable>`	Add a variable to the list of automatically displayed variables.
`undisplay <display_number>`	Remove a variable from the list of automatically displayed ones.
`set <variable>=<value>`	Change the value of a variable during debugging.
`info locals`	Display all local variables in the current scope.
`info threads`	List all threads in the program.
`thread <thread_number>`	Switch to a specific thread for debugging.
`quit` or `q`	Exit GDB and terminate the debugging session.
`disassemble`	Inspect the assembly code of the program.

Please note that GDB provides a wide range of additional commands and features for debugging, including watchpoints, conditional breakpoints, and memory examination. You can access the GDB documentation or use the `help` command within GDB to get more detailed information about each command and its options.

Please try out setting some breakpoints and inspect the state of the program. The micro:bit offers a reset button in the back side of the device, which can be used to restart the execution of your code.