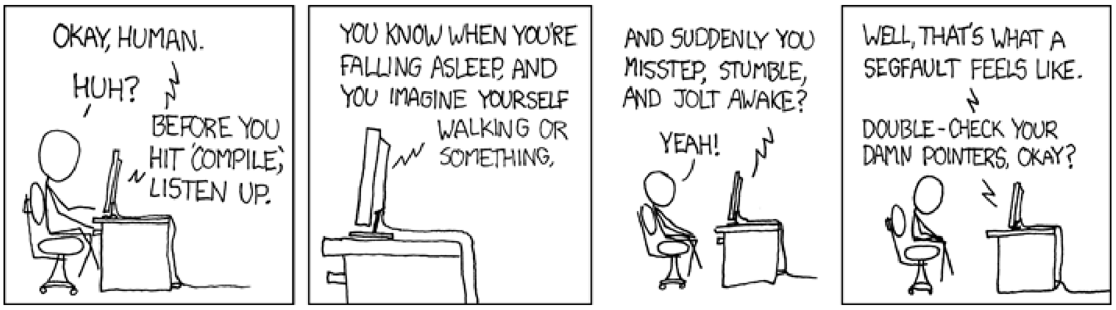
GROUP T LEUVEN CAMPUS

Operating Systems
Programming with C: lab 2



OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Bachelor Electronics/ICT

Course coordinator: Bert Lagaisse

Lab coaches:

- Ludo Bruynseels
- Toon Dehaene
- Jonas Verbruggen

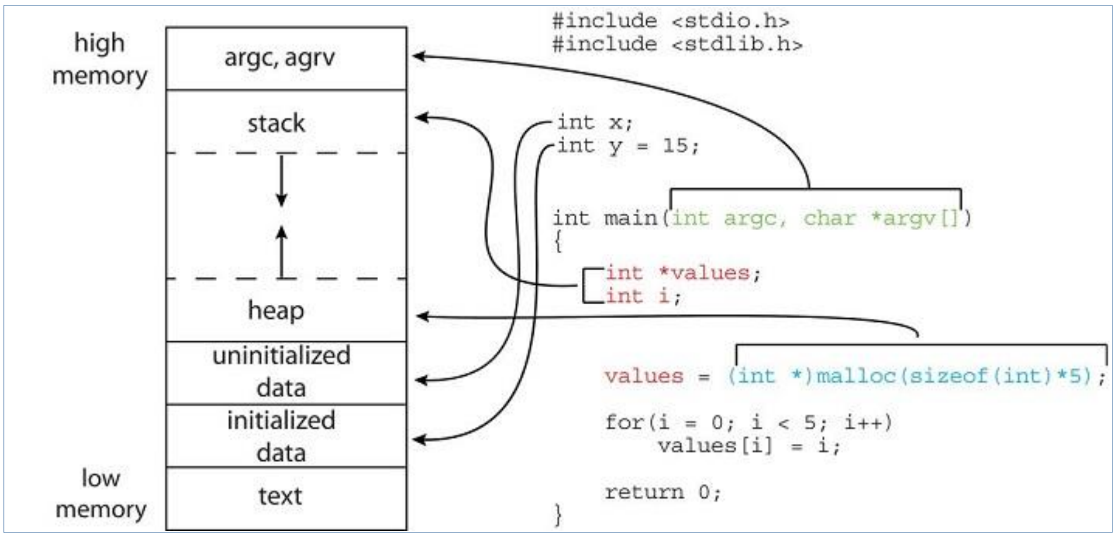
Last update: October 13, 2023

```
 -----------------------------------------------------------------
| 1.   Programming with C: Lab 2                                  |
 -----------------------------------------------------------------
```

Lab target:

- *Understand scope and lifetime of variables.*
- *Understand the difference between data on the heap and on the stack.*
- *Understand how parameters are passed to functions.*
- *Understand pointers in C.*
- *Understanding the function stack in C*
- *Debugging with GDB and CLion.*

```
 -----------------------------------------------------------------
| 2. Organisation of computer memory for C-based processes        |
 -----------------------------------------------------------------
```



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

int x;
int y = 15;

int main(int argc, char *argv[])
{
      int *values;
      int i;

      values = (int *)malloc(sizeof(int)*5);

      for(i = 0; i < 5; i++)
          values[i] = i;

      return 0;
}
```

- Text, initialized data and uninitialized data are loaded by the OS when you start the program. To address them, absolute addresses are used. Therefor these data are accessible from anywhere in the program.
- Data on stack are addressed relative to the stack pointer [SP + n]. These data are only accessible if you know the value of the SP. Consequently, these data are accessible within a function (= local variables)
- Argc and argv are copied when the program is loaded. However, the addresses are not known. These data are passed as parameters to the main function.

```
 -----------------------------------------------------------------------
| 3.   Exercise: parameter passing and the function stack              |
 -----------------------------------------------------------------------
```

The following code is incorrect. **Draw the function stack** before, during, and after the function call to `date_struct()` to indicate where the problem appears.

```
typedef struct {
    short day, month;
    unsigned year;
} date_t;

//function declaration
void date_struct( int day, int month, int year, date_t *date);

//function definition
void date_struct( int day, int month, int year, date_t *date) {
    date_t dummy;
    dummy.day = (short)day;
    dummy.month = (short)month;
    dummy.year = (unsigned)year;
    date = &dummy;
}
int main( void ) {
    int day, month, year;
    date_t d;
    printf("\nGive day, month, year:");
    scanf("%d %d %d", &day, &month, &year);
    date_struct( day, month, year, &d);
    printf("\ndate struct values: %d-%d-%d", d.day, d.month, d.year);
    return 0;
}
```

And what if we rewrite the code such that the function `date_struct()` returns a pointer to date? Draw the function stack before, during, and after the function call to `date_struct()` to find out what really happens. Explain why the code might work if the function f is not called.

```
typedef struct {
 short day, month;
 unsigned year;
} date_t;

void f( void ) {
 int x, y, z;
 printf("%d %d %d\n", x, y, z );
}
date_t * date_struct( int day, int month, int year ) {
 date_t dummy;
 dummy.day = (short)day;
 dummy.month = (short)month;
 dummy.year = (unsigned)year;
 return &dummy;
}
int main( void ) {
 int day, month, year;
 date_t *d;
 printf("\nGive day, month, year:");
 scanf("%d %d %d", &day, &month, &year);
 d = date_struct( day, month, year );
 //f();
 printf("\ndate struct values: %d-%d-%d", d->day, d->month, d->year);
 return 0;
}
```

Solve the problem by allocating dummy on the heap instead of the stack. Now, use `date_t *d` to experiment with memory leaks. Validate your memory leak with *valgrind*. Use `date_t * d` to clean up in the end. Again, validate your clean-up with *valgrind*.

```
   ------------------------------------------------------------------
 | 4.   Exercise: parameter passing                                  |
   ------------------------------------------------------------------
```

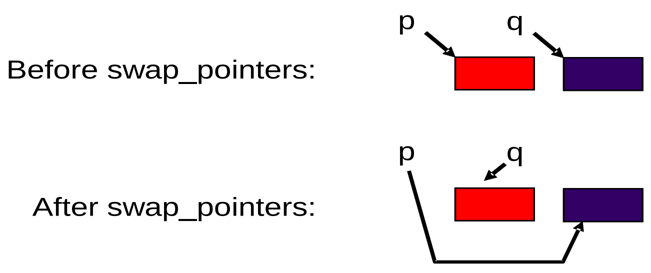
Implement the function '`swap_pointers`'. This function takes two arguments of type void pointer and has no return value. The functions 'swaps' the two pointers as illustrated below. **Draw the function stack** before, during, and after the function call to swap_pointers(). What are **`the addresses when a and b are allocated on the heap?`**

```
int a = 1;
int b = 2;
// for testing we use pointers to integers
int *p = &a;
int *q = &b;

printf("address of p = %p and q = %p\n", p, q);
// prints p = &a and q = &b

swap_pointers( ?p , ?q );

printf("address of p = %p and q = %p\n", p, q);
// prints p = &b and q = &a
```



```
   ------------------------------------------------------------------
 | 5.   Exercise: random numbers, the time() and sleep() functions   |
   ------------------------------------------------------------------
```

Implement a program that simulates a sensor node measuring the outdoor temperature. Use the pseudo-random number generator to simulate temperature readings and use the `sleep` function to generate temperature readings at a pre-defined frequency. The temperature values should be realistic outdoor values (not too cold, too hot – e.g. between -10 and +35°C). Use

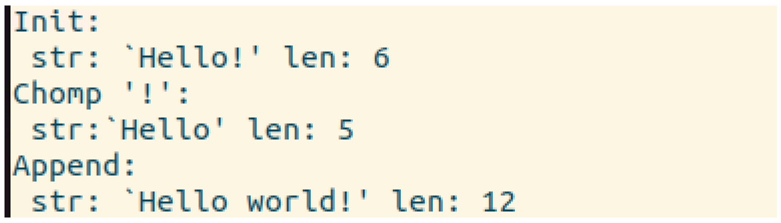
`#define` to set the frequency, min. and max. temperature values. Print every reading as a new line on screen as follows: `Temperature = <temperature> @<date/time>`

In this format, <temperature> should be printed with 1 digit before (= width) and 2 digits after (= precision) the decimal point, and <date/time> is the date and time as returned by the Linux 'date' command.

Hint 0: All functions that you could use have very informative man pages.
Hint 1: use the library function `srand`() to initialize the pseudo-random generator with the result of `time(NULL)`. You should call `srand()` only once.
Hint 2: Printing the temperature in the correct format can be easily done with the format specifier `%1.2f` i.s.o. `%f`.
Hint 3: `printf()` followed by `sleep()` could delay the output to screen due to buffered output. To avoid this use the statement `fflush(stdout)` just after `printf()`.
Hint 4: the `time.h` header file defines the functions `time(), asctime(), localtime()`. These could be of use for your implementation.

```
 ------------------------------------------------------------------
| 6.   Exercise: debug challenge                                  |
 ------------------------------------------------------------------
```

a) Expected output:



```
Init:
 str: `Hello!' len: 6
Chomp '!':
 str:`Hello' len: 5
Append:
 str: `Hello world!' len: 12
```

b) Source code:

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

typedef struct {
   char *str;
   int  len;
} CString;

int main(void)
{
   CString *mystr;
   char c;

   mystr = Init_CString("Hello!");
   printf("Init:\n str: `%s' len: %d\n", mystr->str, mystr->len);
   c = Chomp(mystr);
   printf("Chomp '%c':\n str:`%s' len: %d\n", c, mystr->str, mystr-
>len);
   mystr = Append_Chars_To_CString(mystr, " world!");
   printf("Append:\n str: `%s' len: %d\n", mystr->str, mystr->len);

   Delete_CString(mystr);

   return 0;
}

CString *Init_CString(char *str)
{
   CString *p = malloc(sizeof(CString));
   p->len = strlen(str);
   strncpy(p->str, str, strlen(str) + 1);
   return p;
}

void Delete_CString(CString *p)
{
   free(p);
   free(p->str);
}

// Removes the last character of a CString and returns it.
//
char Chomp(CString *cstring)
{
   char lastchar = *( cstring->str + cstring->len);
   // Shorten the string by one
   *( cstring->str + cstring->len) = '0';
   cstring->len = strlen( cstring->str );

   return lastchar;
}

// Appends a char * to a CString
//
CString *Append_Chars_To_CString(CString *p, char *str)
{

   char *newstr = malloc(p->len + 1);
   p->len = p->len + strlen(str);

   // Create the new string to replace p->str
   snprintf(newstr, p->len, "%s%s", p->str, str);
   // Free old string and make CString point to the new string
   free(p->str);
   p->str = newstr;

   return p;
}
```

c) Assignment:

- Author a Makefile with one target : all. You will need it when working with Clion.
- Compile with `$ gcc -g -W -Wall cstring.c -o cstring`
- Find and correct all the errors. Do no add any '`printf`' statements, use GDB.

d) Useful GDB commands:

- run : start program
- run *arglist* start program with arguments.
- kill : kill running program
- set a breakpoint: b + linenr or function name. (b or break)
- bt : backtrace. When you hit a seg fault. Bt unwinds the stack so you can see at which point in the execution the seg fault occurred.
- p (or print) + expression: get the value of the expression.
- n : step 1 line, step over function calls
- s : next line, step into function calls.
- gdb *program* : start gdb and load *program*
- *quit : leave gdb*

There are many more commands that can help you. See the lecture slides or:

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/documents/gdbref.pdf

Why are we doing this? Modern IDE's offer you plenty of tools to debug your program. However, these IDE's are a overwhelming: they tend to be like a Swiss army knife that integrates all tools you might possibly need.

e) Tips

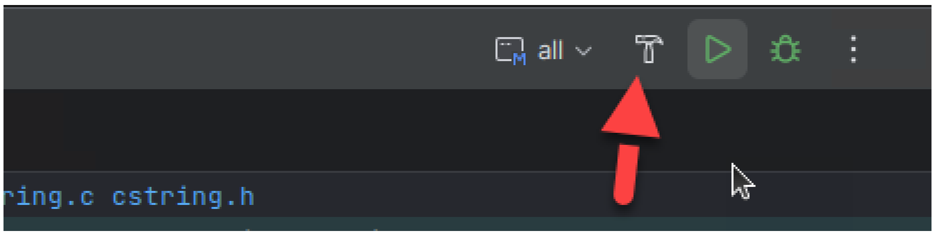
Tip 1: this program is a mess. It doesn't even compile. To direct you in the right direction: a function declaration must precede the function's first use. And the declaration must match the definition on return type and parameters. The implicit declarations are wrong most of the time; don't count on it. Roll your own. Eventually you can put all declarations in a separate header file.

Tip 2: gdb halts when a seg fault occurs. Type bt (backtrace) to find the location where it happened.

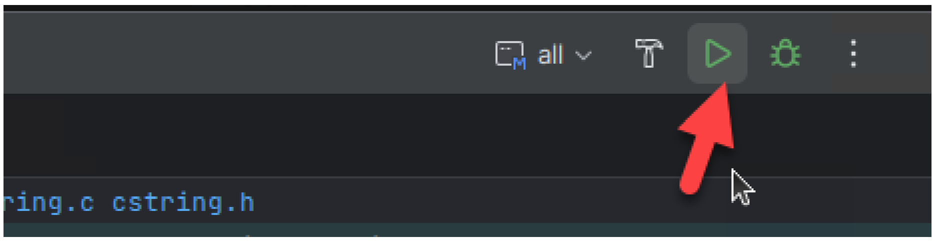
f) Debug challenge revisited: do the same exercise with Clion.

- Launch Clion
- Open the Makefile that you created in the previous exercise as a project.

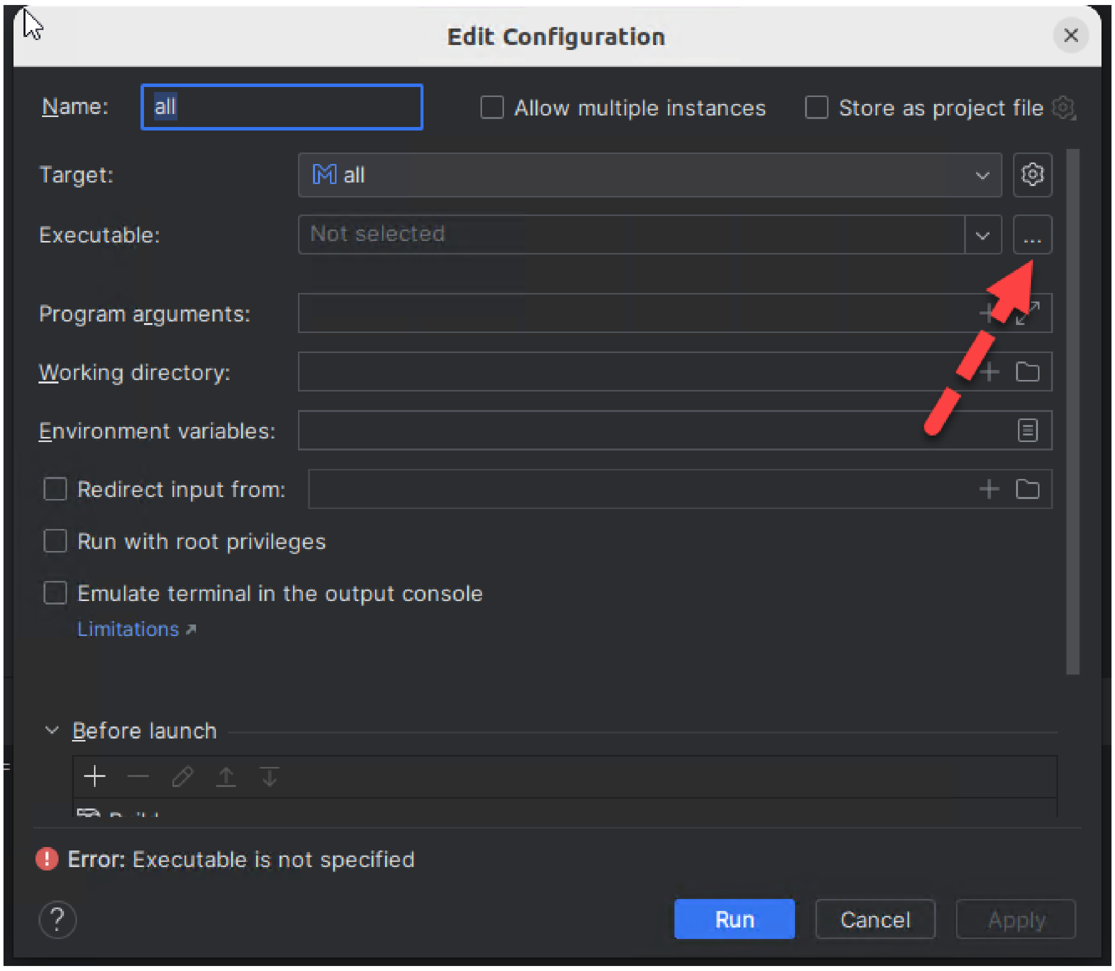
- To build the project:



- To run the project:



- Clion will prompt you for the executable to run:



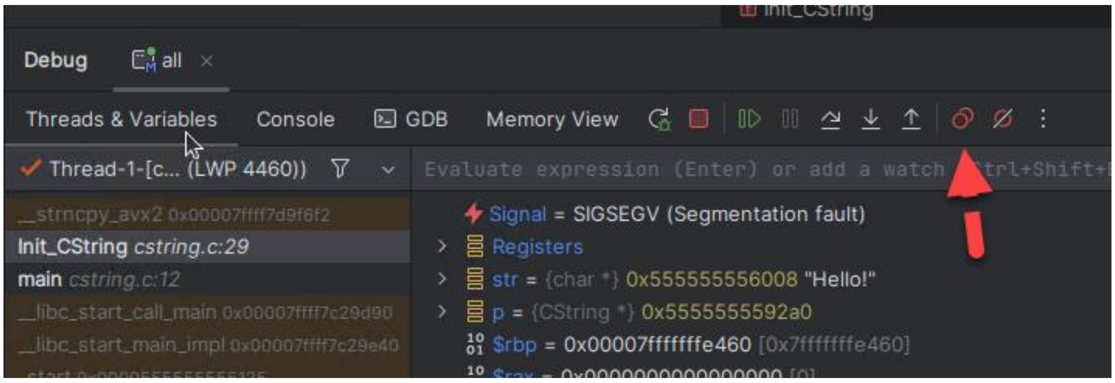
Edit Configuration

Name: all
Allow multiple instances
Store as project file
Target: all
Executable: Not selected
Program arguments:
Working directory:
Environment variables:
Redirect input from:
Run with root privileges
Emulate terminal in the output console
Limitations ↗
Before launch

Error: Executable is not specified

Click the ellipsis and browse to the executable file you just build.

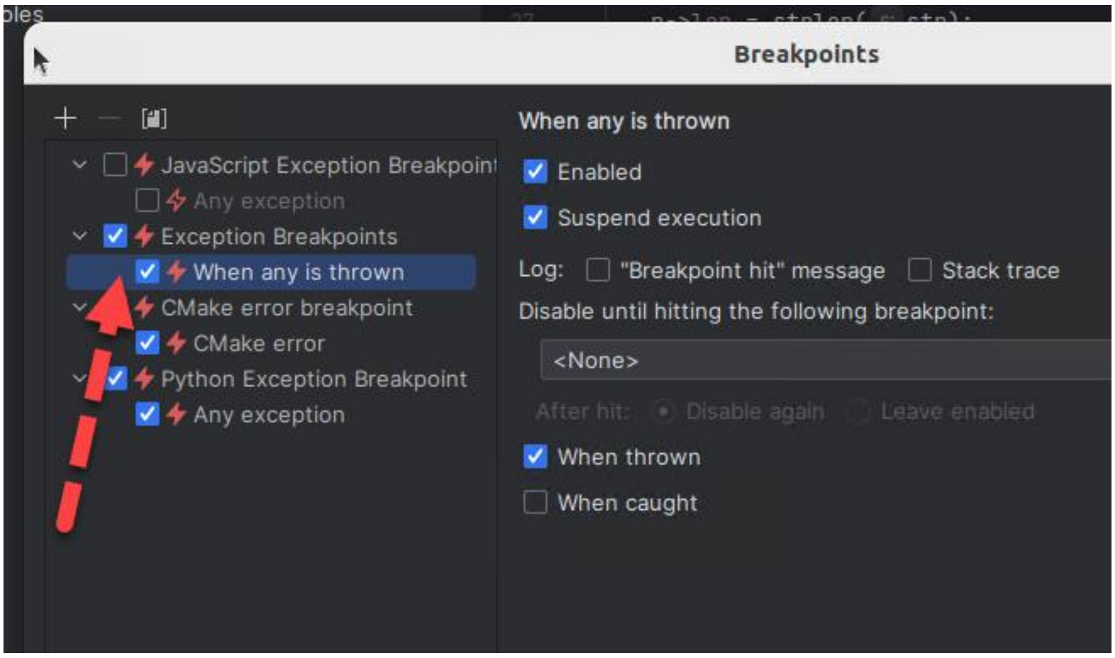
One more tip: to help with debugging seg faults, you can break as soon as the fault occurs. To do that, start a debug session by clicking on the little bug. In the debug window, click on the double ring:



```
Thread-1-[c... (LWP 4460))
__strncpy_avx2 0x00007ffff7d9f6f2
Init_CString cstring.c:29
main cstring.c:12
__libc_start_call_main 0x00007ffff7c29d90
__libc_start_main_impl 0x00007ffff7c29e40

Signal = SIGSEGV (Segmentation fault)
Registers
str = {char *} 0x555555556008 "Hello!"
p = {CString *} 0x5555555592a0
$rbp = 0x00007fffffffe460 [0x7fffffffe460]
```

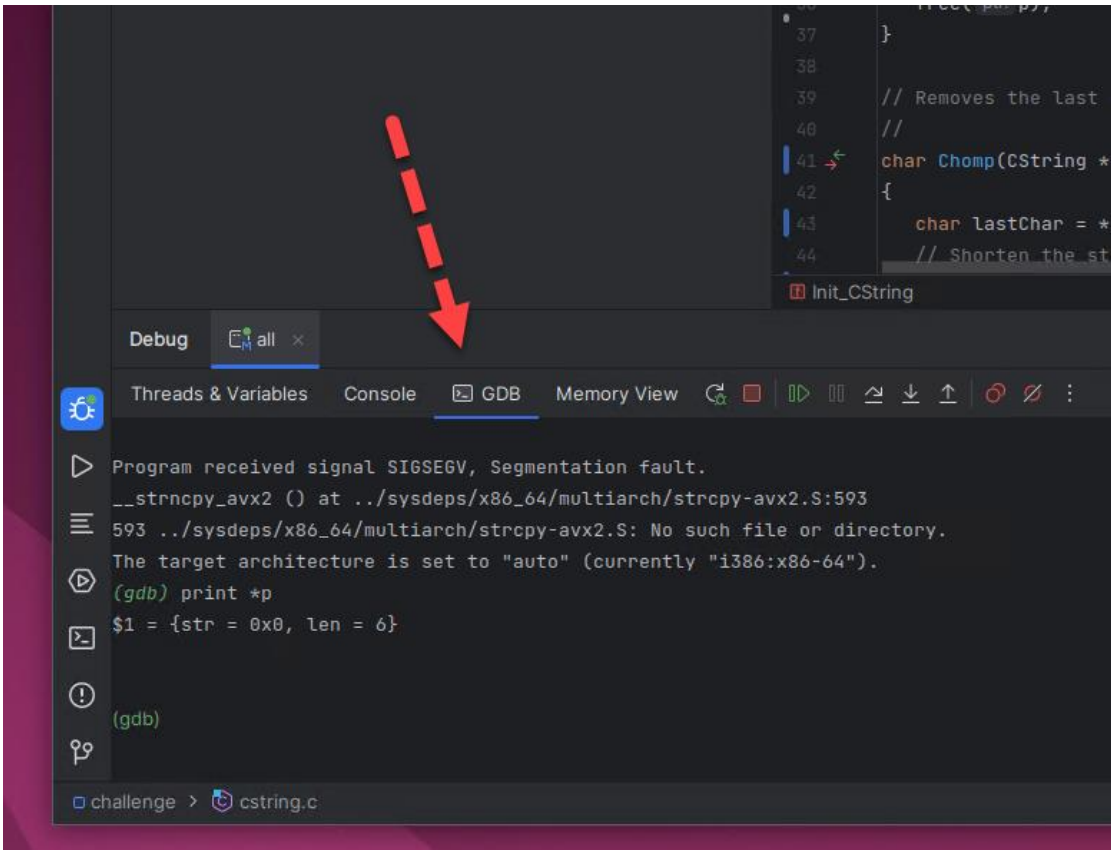
Then:



When any is thrown
- [✓] Enabled
- [✓] Suspend execution
- Log: [ ] "Breakpoint hit" message [ ] Stack trace
- Disable until hitting the following breakpoint: <None>
- After hit: Disable again Leave enabled
- [✓] When thrown
- [ ] When caught

This will help you resolving seg faults but we are not saying that resolving seg faults becomes easy. The program will be suspended and you can inspect your variables to find the null pointer or the pointer that points to an area where it should keep out.

Also remark that CLion is tightly integrated (in fact, under the hood CLion dispatches the real work to gcc and gdb). There is one window that directly talks to gdb:



```
}

// Removes the last
//
char Chomp(CString *
{
    char lastChar = *
    // Shorten the st
```

```
Program received signal SIGSEGV, Segmentation fault.
__strncpy_avx2 () at ../sysdeps/x86_64/multiarch/strcpy-avx2.S:593
593 ../sysdeps/x86_64/multiarch/strcpy-avx2.S: No such file or directory.
The target architecture is set to "auto" (currently "i386:x86-64").
(gdb) print *p
$1 = {str = 0x0, len = 6}


(gdb)
```

```
 -----------------------------------------------------------------
| 7.   Valgrind                                                   |
 -----------------------------------------------------------------
```

The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct.

'faster' is not an issue in this course but 'more correct' is a big issue. The use of pointers and thread synchronization are very prone to errors. Some of those errors do not happen all the time which make them very difficult to simulate, repeat and resolve.

Valgrind will help us to find memory (pointer) errors. For instance, every malloc() should be matched by a free(). If not, your program is leaking memory.

For detailed information see valgrind.org.

To install valgrind:

```
$ sudo apt install valgrind
```

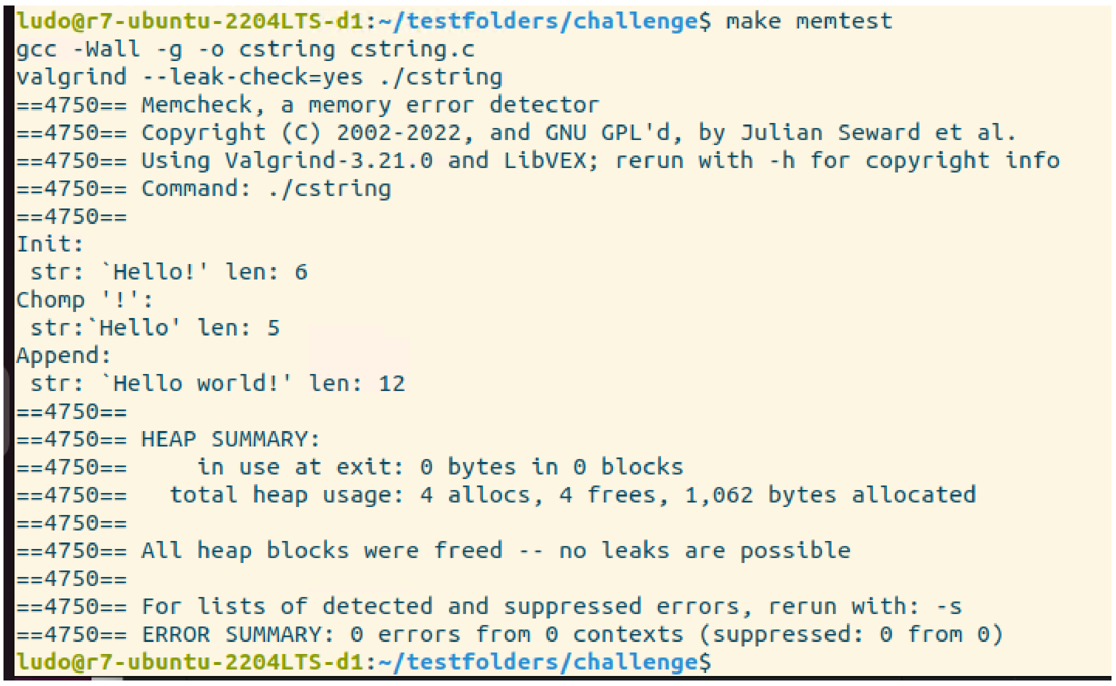
In your Makefile, include the following target:

```
memtest: all
        valgrind --leak-check=yes ./cstring
```

In a terminal you can now type:

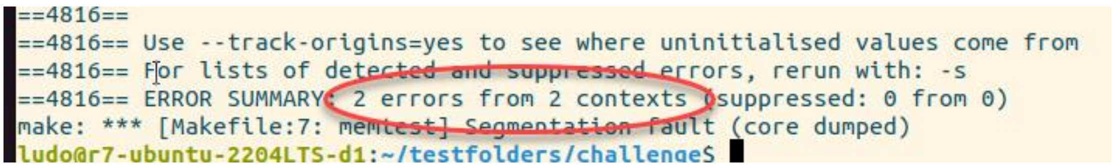
```
$ make memtest
```

If your program uses memory correctly, you should see following output:



```
ludo@r7-ubuntu-2204LTS-d1:~/testfolders/challenge$ make memtest
gcc -Wall -g -o cstring cstring.c
valgrind --leak-check=yes ./cstring
==4750== Memcheck, a memory error detector
==4750== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4750== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==4750== Command: ./cstring
==4750==
Init:
 str: `Hello!' len: 6
Chomp '!':
 str:`Hello' len: 5
Append:
 str: `Hello world!' len: 12
==4750==
==4750== HEAP SUMMARY:
==4750==     in use at exit: 0 bytes in 0 blocks
==4750==   total heap usage: 4 allocs, 4 frees, 1,062 bytes allocated
==4750==
==4750== All heap blocks were freed -- no leaks are possible
==4750==
==4750== For lists of detected and suppressed errors, rerun with: -s
==4750== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ludo@r7-ubuntu-2204LTS-d1:~/testfolders/challenge$
```

With errors:



```
==4816==
==4816== Use --track-origins=yes to see where uninitialised values come from
==4816== For lists of detected and suppressed errors, rerun with: -s
==4816== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
make: *** [Makefile:7: memtest] Segmentation fault (core dumped)
ludo@r7-ubuntu-2204LTS-d1:~/testfolders/challenge$
```

It is your task to work until you have no errors. Your understanding of pointers and memory management will deepen and this will help you in the next exercises.

This session covers a lot of debugging techniques that go beyond simple printf() statements.

May this force be with you to successfully complete the next installments of these lab sessions**.**