

# Algorithmic complexity. Code profiling. Pointer errors.

**Ștefan-Adrian Toma**

Military Technical Academy “Ferdinand I”

# Contents

- The C programming language
- Algorithmic complexity
- Big “O”
- Examples
- Big O – hands on
- Valgrind and code profiling
- Valgrind – hands on
- Pointers – common errors
- Valgrind and finding pointer errors – hands on

# The C programming language

- C was developed at Bell Labs, in the 1970s, by Dennis Ritchie.
- C is a general-purpose, high level, structured language.
- The instructions are similar to algebraic expressions. Keywords are in English (e.g., *if*, *else*, *for*, *do* and *while*).

```
int main()
{
    FILE* fid;
    int** mat=NULL;
    int N = 3;

    createBinaryFile(N, "binFile.bin");

    mat = (int**)malloc(N * sizeof(int*));
    for (int i = 0; i < N; i++)
        *(mat+i) = (int*)malloc(N * sizeof(int));

    fid = fopen("binFile.bin", "rb");
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            fread(( * (mat + i) + j), sizeof(int), 1, fid);
        }

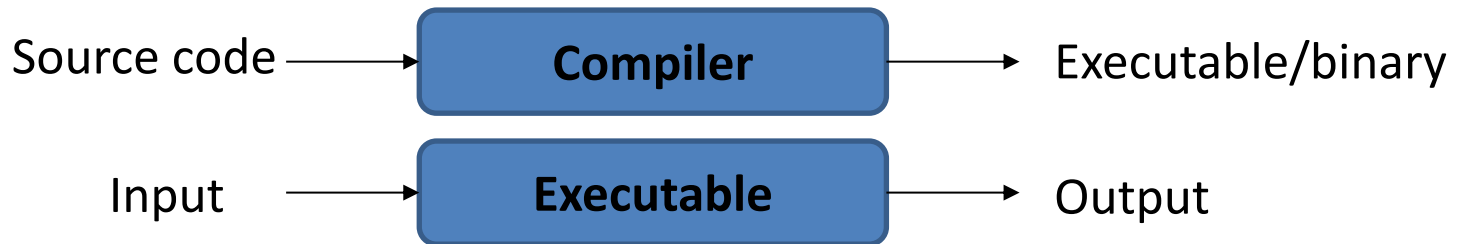
    printMatrix(mat, N);
    createTextFile(mat, N, "textFile.txt");

    fclose(fid);

    for (int i = 0; i < N; i++)
        free(mat[i]);
    free(mat);
    return 0;
}
```

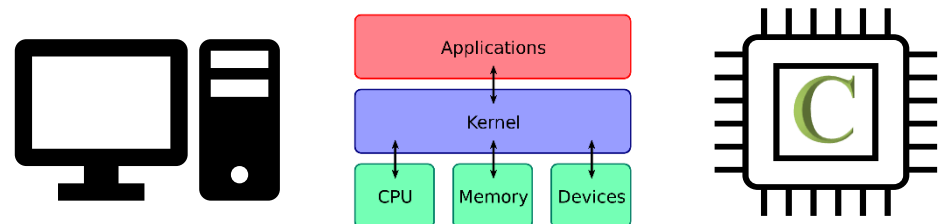
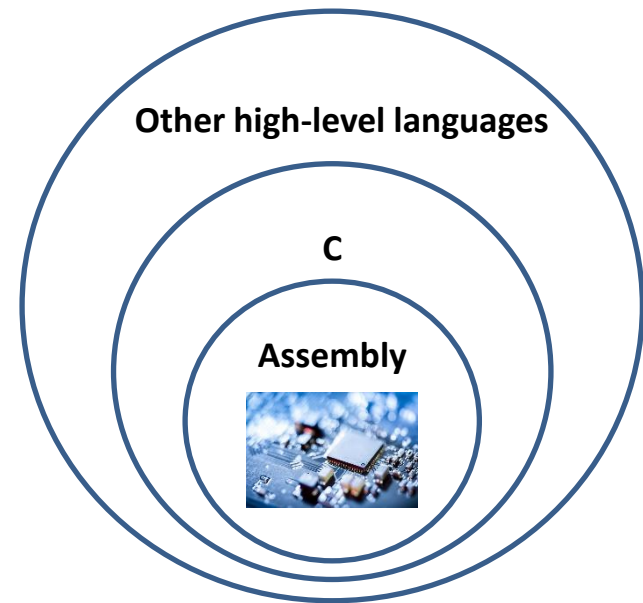
# The C programming language

- Learning C helps understand how computers work.
- The language is standardized: ANSI C, C99, C11, C17, C2x.
- It is a **compiled language**. This means that if the programs are written appropriately, it is very fast.



# The C programming language

- The C programming language has some characteristics that make it a bridge between machine language and high-level languages. Hence, one can use C for both general purpose programs (e.g., *Doom*, *git*) and system programs (e.g., *the Linux kernel*, *device drivers*)
- It widely used in **embedded systems** (automotive, communications, radars etc.)



# Algorithmic complexity

*An algorithm is a finite sequence of rigorous instructions, typically used to solve a class of specific problems or to perform a computation. (Wikipedia)*



1 apple pie – 30 minutes	10 apple pies – $10 \times 30$ minutes
1 cheesecake – 60 minutes	10 cheesecakes – $10 \times 60$ minutes
1 bread – 10 minutes	10 breads – $10 \times 10$ minutes

It is important to know how does the time it takes preparing pies or cakes changes with the increase of their number.

This is important not only in a restaurant, but also in computing. We want to know how much time processing my data will take. And, if I increase the data size, how will processing time increase.

# Big O

- Big O notation is a mathematical notation.
- It describes the limiting behavior of a function when the argument tends towards a particular value.
- **In computer science, it is used to classify algorithm – how running time or storage depends on the size of the processed data.**

**$O(1)$ ,  $O(N)$ ,  $O(N \cdot \log N)$ ...**

# Big O

- Example – computing the discrete Fourier transform (DFT)
- How many computations (additions, multiplications) are needed to compute the direct form of the DFT.

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}$$

- For each  $k$  we have  $N$  complex multiplications and  $(N-1)$  complex summations. For all  $k$  values (i.e.,  $N$ ), we have  $N^2$  complex multiplications and  $N \cdot (N - 1)$  – complex additions.
- A complex multiplication requires 4 real multiplications and two real additions. A single complex addition is made of two real summations. Hence, the total number of operations is  $4 \cdot N^2$  real multiplications and  $[2 \cdot N + 2 \cdot (N-1)] \cdot N$  real additions.
- Therefore, the complexity of the direct form of the DFT is  $O(N^2)$ .
- Example: for  $N = 1024$  we need  $4 \cdot 1024 \cdot 1024 = 4194304$  real multiplications.



# Big O

- Therefore, the complexity of the direct form of the DFT is  $O(N^2)$ .
- Example: for  $N = 1024$  we need  $4 \cdot 1024 \cdot 1024 = 4194304$  real multiplications.
- Simplified analysis of the computation efficiency.
- Analysis in terms of the size of the input data. We are not interested in the nature of the data (int, float) or what it represents.
- It is machine independent.
- We count basic computing steps (sums, multiplications, comparisons, swaps ... )
- Tries to express time and/or space.

# Big O

- The efficiency of an algorithm can depend on the input data. Example – sorting an already sorted array.
- Big O measures the **worst-case scenario**.

# Big O – general rules

- $O(1)$  – it does not matter the size of data
- Ignore constants.

$$O(5N) \rightarrow O(N)$$

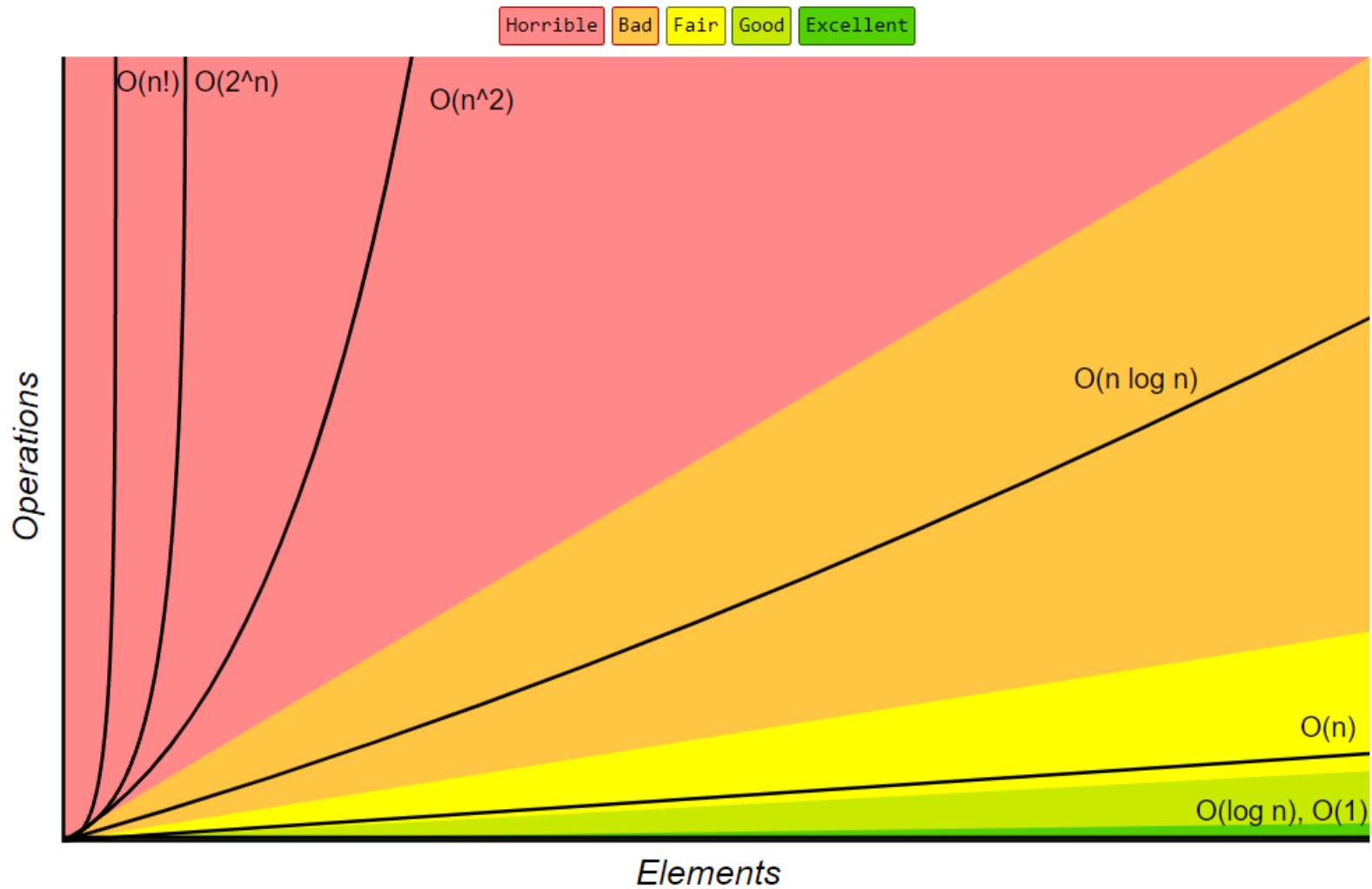
$$O(5+N) \rightarrow O(N)$$

- Some terms dominate others.

$$O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(2^N) < O(N!)$$

$$O(1) + O(\log N) \rightarrow O(\log N)$$

# Big O – hands on



<https://www.bigocheatsheet.com/>

# Big O – other examples

## Constant time

```
5
6  int main()
7  {
8      int x;
9      x = 12 + 1 * 5;
10     return 0;
11 }
```

$O(1)$

```
6  int main()
7  {
8      int x=0,y=0;
9      x = 12 + 1 * 5;
10     y = 10 + 3;
11     return 0;
12 }
```

$O(1) + O(1) \rightarrow O(1)$

The operations are independent of input data size.

# Big O – other examples

## Linear time

```
6  int main()  
7  {  
8      int N = 1000;  
9      for (int i = 0; i < N; i++)  
10     {  
11         printf("Hello");  
12     }  
13     return 0;  
14 }
```

$$N \cdot O(1) \rightarrow O(N)$$

```
6  int main()  
7  {  
8      int N = 1000, x=0;  
9      x = 10 + 30;  
10     for (int i = 0; i < N; i++)  
11     {  
12         printf("Hello");  
13     }  
14     return 0;  
15 }
```

$$O(1) + N \cdot O(1) \rightarrow O(N)$$

# Big O – other examples

## Quadratic time

```
6  int main()  
7  {  
8      int N = 100;  
9      for (int i = 0; i < N; i++)  
10     {  
11         for (int j = 0; j < N; j++)  
12         {  
13             printf("%d\n", i * j);  
14         }  
15     }  
16     return 0;  
17 }
```

$$N \cdot N \cdot O(1) \rightarrow O(N^2)$$

# Big O – other examples

```
6  int main()
7  {
8      int N = 100, x = 0;
9      printf("Start program ... \n");
10     for (int i = 0; i < N; i++)
11         x = i + 100;
12     for (int i = 0; i < N; i++)
13     {
14         for (int j = 0; j < N; j++)
15         {
16             printf("%d\n", i * j);
17         }
18     }
19     return 0;
20 }
```

$O(1)$

$N \cdot O(1) \rightarrow O(N)$

$N \cdot N \cdot O(1) \rightarrow O(N^2)$

worst case scenario



# Big O – other examples

```
6 int main(){
7     int N = 100, x = 0;
8     scanf(" %d", &x);
9     if (x > 0){
10         printf("Start program ... \n");
11     }
12     else if (x == 0){
13         for (int i = 0; i < N; i++)
14             x = i + 100;
15     }
16     else{
17         for (int i = 0; i < N; i++)
18         {
19             for (int j = 0; j < N; j++)
20             {
21                 printf("%d\n", i * j);
22             }
23         }
24     }
25     return 0;
26 }
```

$O(1)$

$O(N)$

$O(N^2)$

worst case scenario

# Big O – hands on

## Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
<u>Array</u>	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>Stack</u>	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
<u>Queue</u>	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
<u>Singly-Linked List</u>	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
<u>Doubly-Linked List</u>	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
<u>Skip List</u>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
<u>Hash Table</u>	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>Binary Search Tree</u>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>Cartesian Tree</u>	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
<u>B-Tree</u>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>Red-Black Tree</u>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>Splay Tree</u>	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>AVL Tree</u>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
<u>KD Tree</u>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

<https://www.bigocheatsheet.com/>

# Big O – hands on

## Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

<https://www.bigocheatsheet.com/>

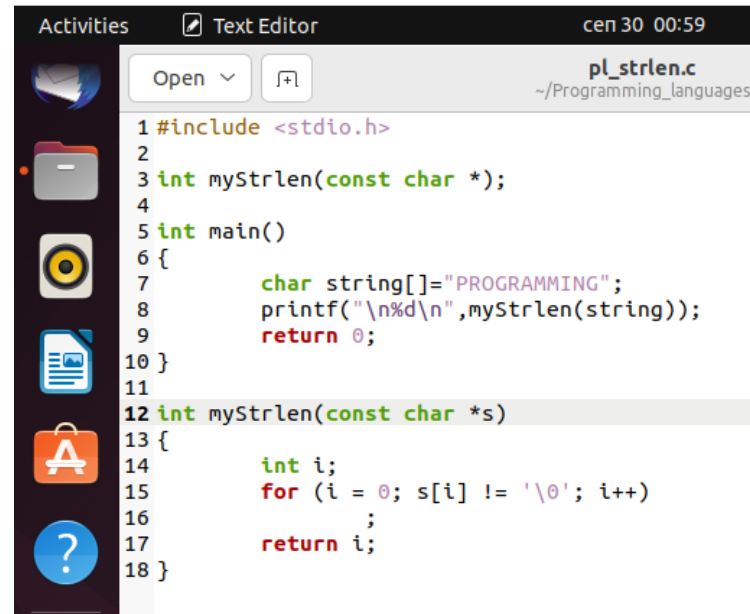
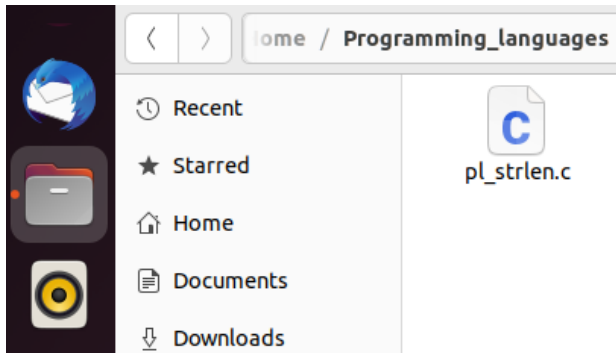
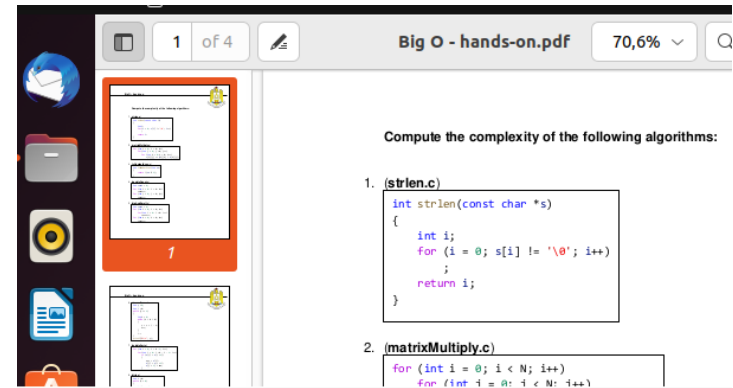
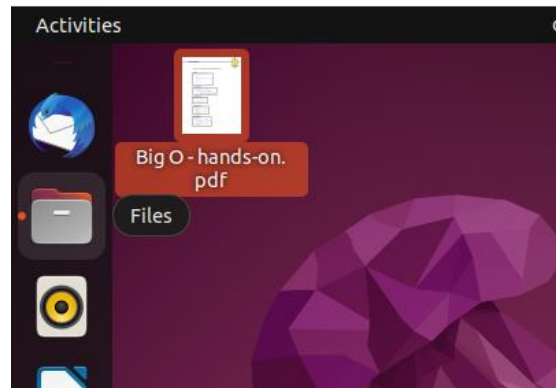
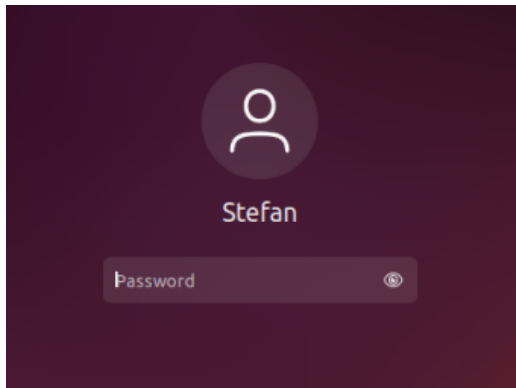
# Big O – hands on

1. Find the computational efficiency of the following code snippets, using big O.
2. For each code snippet, measure running time for different values of the size of the input data. Write your findings in the following table. Compare the results with the ones computed using big O.

N	1	10	100	1000	10000	100000	1000000
Time							
Time -O3							
O(N)							

Use the VM provided by the teacher, to measure code efficiency.

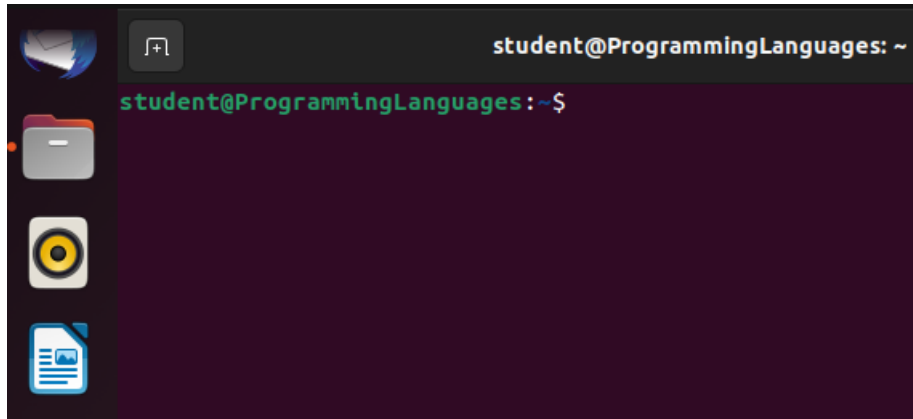
# Big O – hands on



Code snippets from:

[courses/Enunt-laborator03.docx at master · cristianchilipirea/courses · GitHub](https://github.com/cristianchilipirea/courses/blob/master/courses/Enunt-laborator03.docx)

# Big O – hands on



CTRL + ATL + T

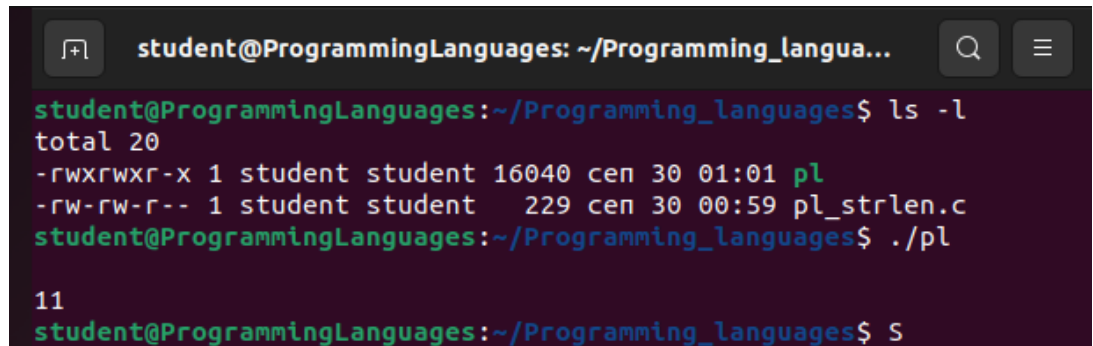
```
student@ProgrammingLanguages:~/Programming_languages$ gcc -O0 -o pl pl_strlen.c
student@ProgrammingLanguages:~/Programming_languages$
```

`gcc -O0 -o pl pl_strlen.c`

`-O0` – no optimization

`pl` – executable file

`pl_strlen.c` – source code



# Big O – hands on

```
student@ProgrammingLanguages: ~/Programming_languages$ time ./pl
11
real    0m0,001s
user    0m0,001s
sys     0m0,000s
student@ProgrammingLanguages: ~/Programming_languages$
```

```
7 int main()
8 {
9     char *string;
10    int N=100000, length=0;
11    double time_spent=0.0;
12
13    /*Initialize array*/
14    string=(char*)calloc(N, sizeof(char));
15    for(int i=0; i<N-1; i++)
16        string[i]='A';
17    /*End*/
18
19    clock_t begin= clock(); //start time
20    length=myStrlen(string); //do important stuff
21    clock_t end =clock();    //end time
22
23    time_spent=(double)(end-begin)/CLOCKS_PER_SEC;
24    printf("\nTime spent: %f\n", time_spent);
25
26    free(string);
27
28    return 0;
29 }
```

```
student@ProgrammingLanguages: ~/Programming_languages$ time ./pl
Time spent: 0.000200
real    0m0,001s
user    0m0,001s
sys     0m0,000s
student@ProgrammingLanguages: ~/Programming_languages$ ./pl
Time spent: 0.000200
```

# Big O – hands on

```
student@ProgrammingLanguages:~/Programming_languages$ gcc -O3 -o pl pl_strlen.c
student@ProgrammingLanguages:~/Programming_languages$ ./pl
Time spent: 0.000000
student@ProgrammingLanguages:~/Programming_languages$
```

<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>



# Code profiling. Valgrind

How do we find the bottleneck in an algorithm implemented by someone else?

Or we don't foresee a bottleneck in our own code?

Answer:

Read the code and try to understand what and how it does what it does.

Or we can do **code profiling**.

# Code profiling. Valgrind

Code profilers are programs that help identify performance problems without modifying the code.

- The number of times a method/function is called
- Time spent in each function/method
- Track memory allocations and garbage collection.

.....

# Code profiling. Valgrind

```
gcc -O0 -g -o pl pl_strlen.c
```

```
valgrind --tool=callgrind --dump-instr=yes --  
simulate-cache=yes --collect-jumps=yes <executable>  
[args...]
```

```
valgrind --tool=callgrind --dump-instr=yes --  
simulate-cache=yes --collect-jumps=yes ./pl
```

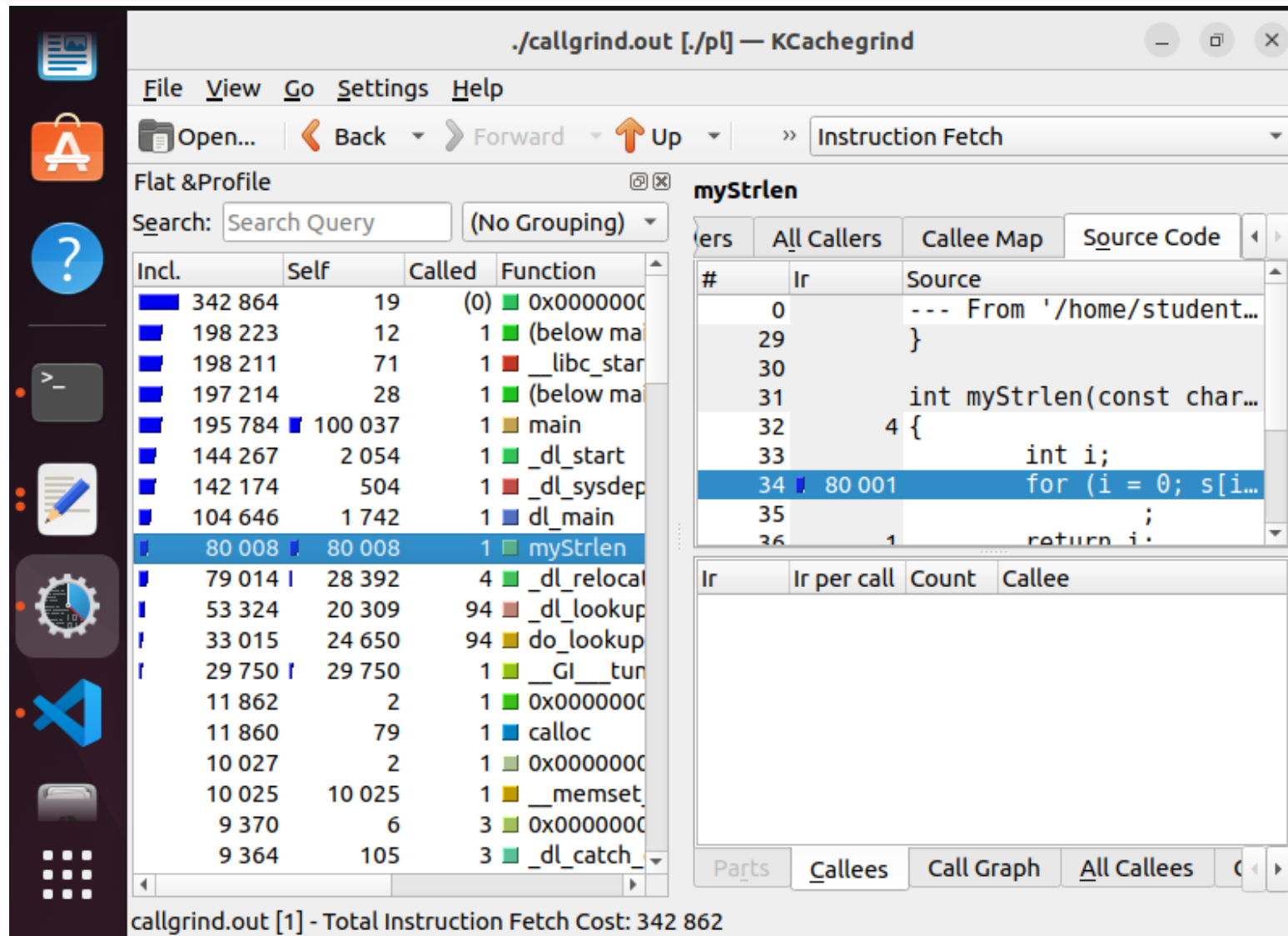
```
kcachegrind
```

# Code profiling. Valgrind

```
student@ProgrammingLanguages:~/Programming_languages$ gcc -O0 -g -o pl pl_strle
n.c
student@ProgrammingLanguages:~/Programming_languages$ valgrind --tool=callgrind
--dump-instr=yes --simulate-cache=yes --collect-jumps=yes ./pl
==5495== Callgrind, a call-graph generating cache profiler
==5495== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==5495== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5495== Command: ./pl
==5495==
--5495-- warning: L3 cache found, using its data for the LL simulation.
==5495== For interactive control, run 'callgrind_control -h'.

Time spent: 0.000911
==5495==
==5495== Events      : Ir Dr Dw I1mr D1mr D1mw ILMr DLMr DLMw
==5495== Collected : 342862 105648 53955 1438 1608 799 1409 1327 755
==5495==
==5495== I    refs:      342,862
==5495== I1  misses:      1,438
==5495== LLi misses:      1,409
==5495== I1  miss rate:    0.42%
==5495== LLi miss rate:    0.41%
==5495==
==5495== D    refs:      159,603 (105,648 rd + 53,955 wr)
==5495== D1  misses:      2,407 ( 1,608 rd +   799 wr)
==5495== LLd misses:      2,082 ( 1,327 rd +   755 wr)
==5495== D1  miss rate:    1.5% ( 1.5% + 1.5% )
```

# Code profiling. Kcachegrind



./callgrind.out [./pl] — KCachegrind

File View Go Settings Help

Open... Back Forward Up Instruction Fetch

Flat & Profile

Search: Search Query (No Grouping)

Incl.	Self	Called	Function
342 864	19	(0)	0x00000000
198 223	12	1	(below ma
198 211	71	1	__libc_star
197 214	28	1	(below ma
195 784	100 037	1	main
144 267	2 054	1	_dl_start
142 174	504	1	_dl_sysdep
104 646	1 742	1	dl_main
80 008	80 008	1	myStrlen
79 014	28 392	4	_dl_relocal
53 324	20 309	94	_dl_lookup
33 015	24 650	94	do_lookup
29 750	29 750	1	__GI__tun
11 862	2	1	0x00000000
11 860	79	1	calloc
10 027	2	1	0x00000000
10 025	10 025	1	__memset
9 370	6	3	0x00000000
9 364	105	3	_dl_catch_

myStrlen

ers	All Callers	Callee Map	Source Code
#	Ir	Source	
0		--- From '/home/student...	
29		}	
30			
31		int myStrlen(const char...	
32	4	{	
33		int i;	
34	80 001	for (i = 0; s[i...	
35		;	
36	1	return i;	

Ir	Ir per call	Count	Callee
----	-------------	-------	--------

Parts Callees Call Graph All Callees

callgrind.out [1] - Total Instruction Fetch Cost: 342 862

# Code profiling. Kcachegrind

Activities KCachegrind cen 30 02:08

/home/student/Programming\_languages/callgrind.out.5541 [./pl] — KCachegrind

File View Go Settings Help

Open... Back Forward Up Instruction Fetch

Flat & Profile

Search: Search Query (No Grouping)

Incl.	Self	Called	Function
6 499	36	2	openaux
283	253	1	open_verify.
882	633	10	open_verify.
1 502	623	1	open_path
6	6	1	munmap
24	24	4	mprotect
108	108	12	mmap
334	334	7	memset
24	24	1	memset
21	21	1	memchr
405	405	29	memcpy
22	22	1	memcpy
44	44	2	memmove
22	22	1	memcpy@@
203	203	16	memcpy
23	23	1	memchr
2 698	15	1	map_doit
1 813	55	1	malloc
5 370	16	1	main

main

ers All Callers Callee Map Source Code

#	Ir	Source
31		int myStrlen(const char ...
32		{
0		--- Inlined from '/usr/i...
109		_fortify_function int
110		printf (const char * __re...
111		{
112	4	return __printf_chk ( _...
5 280		1 call(s) to '0x000000000001...
113		}

Ir	Ir per call	Count	Callee
5 280	5 280	1	0x00000000000109070
74	37	2	0x00000000000109060

Parts Callees Call Graph All Callees

callgrind.out.5541 [1] - Total Instruction Fetch Cost: 151 346

# Count no. of operations using the MSVS debugger

sourceVA.c

Consultatii\_23.08.2022 (Global Scope) main()

```
3  #include <string.h>
4
5
6  int main(){
7      int N = 3, x = 0;
8      scanf(" %d", &x);
9      if (x > 0){
10         printf("Start program ... \\\n");
11     }
12     else if (x == 0){
13         for (int i = 0; i < N; i++)
14             x = i + 100;
15     }
16     else{
17         for (int i = 0; i < N; i++)
18         {
19             for (int j = 0; j < N; j++)
20             {
21                 printf("%d\\n", i * j);
22             }
23         }
24     }
25     return 0;
26 }
```

109 % No issues found

Autos

Search (Ctrl+E) Search Depth: 3

Name	Value	Type
N	3	int
x	102	int

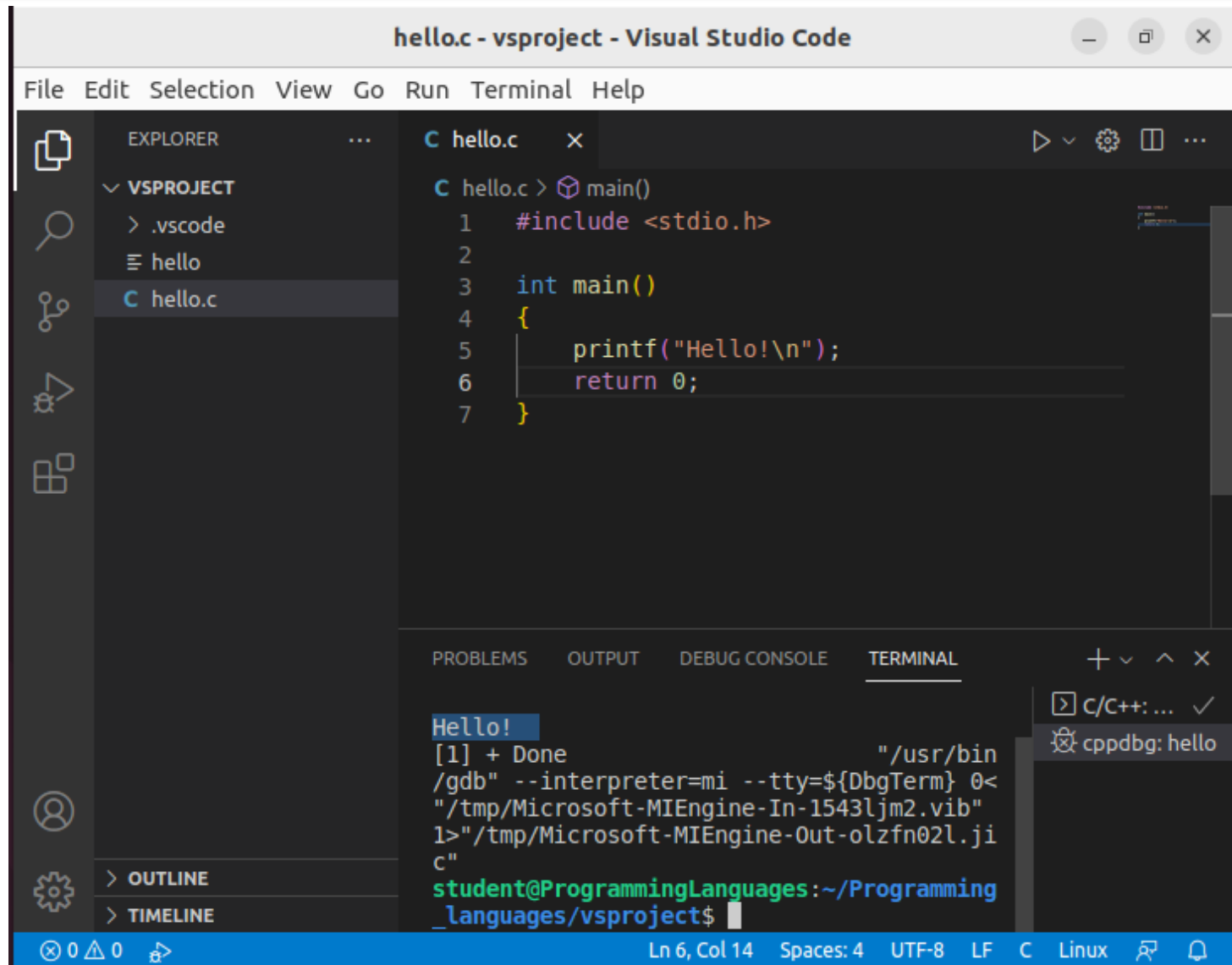
Call Stack

Name
Consultatii_23.08.2022.exe!main(...) Line 15
[External Code]

Autos Locals Watch 1

Call Stack Breakpoints Exception Settings Command Window

# Count no. of operations using VSCode



The screenshot displays the Visual Studio Code interface with a project named 'hello.c - vsproject'. The Explorer sidebar on the left shows the project structure with files '.vscode', 'hello', and 'C hello.c'. The main editor window shows the source code of 'hello.c', which includes a C program that prints 'Hello!' and returns 0. The TERMINAL panel at the bottom shows the execution output, including the prompt 'Hello!', the message '[1] + Done', and the command line used to run the program. The status bar at the bottom indicates the current line and column (Ln 6, Col 14) and the file encoding (UTF-8).

```
hello.c - vsproject - Visual Studio Code
File Edit Selection View Go Run Terminal Help

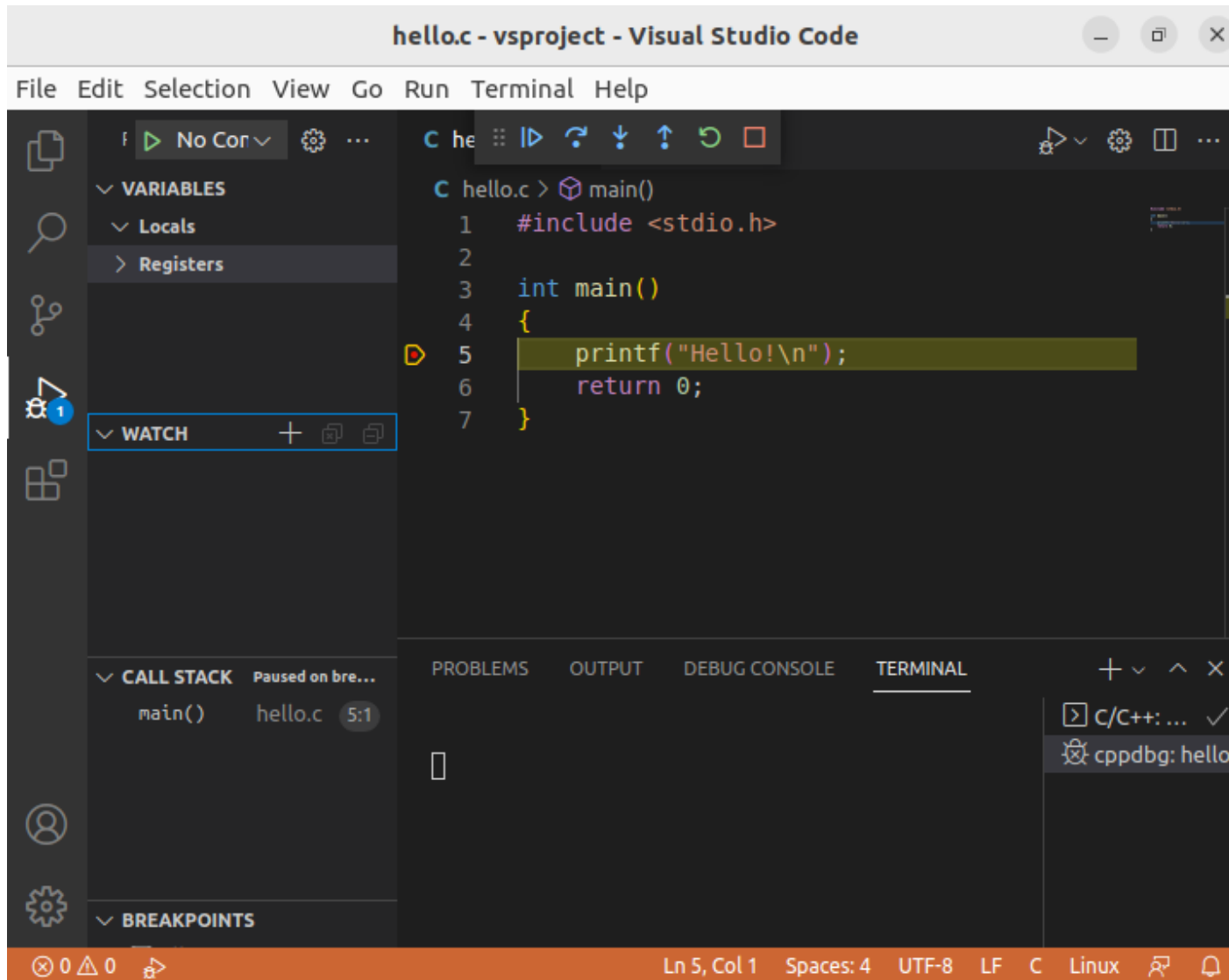
EXPLORER
  VSPROJECT
    .vscode
    hello
    C hello.c

C hello.c
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("Hello!\n");
6      return 0;
7  }

TERMINAL
Hello!
[1] + Done
"/usr/bin
/gdb" --interpreter=mi --tty=${DbgTerm} 0<
"/tmp/Microsoft-MIEngine-In-1543ljm2.vib"
1>"/tmp/Microsoft-MIEngine-Out-olzfn02l.ji
c"
student@ProgrammingLanguages:~/Programming
_languages/vsproject$
```



# Count no. of operations using VSCode



- The DFT is computed with the FFT.

# Pointers – common errors

- wrong or no initialization

```
int main()
{
    int *p=12323428933;
    *p = 100;
    printf("Hell "):
    return 0;
}
```

Exception Thrown

Exception thrown: write access violation.  
**p** was 0x2DE889A45.

[Copy Details](#)

Exception Settings

☒ Break when this exception type is thrown

```
//int main()
//{
//    char *p;
//    p = NULL;
//    p = (char*)
//
```

```
student@ProgrammingLanguages: ~/Programming_languages$ gcc -o badPointer badPointer.c
student@ProgrammingLanguages:~/Programming_languages$ ./badPointer
Segmentation fault (core dumped)
student@ProgrammingLanguages:~/Programming_languages$
```

# Pointers – common errors

- Memory leaks
- Use Valgrind to detect memory leaks

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

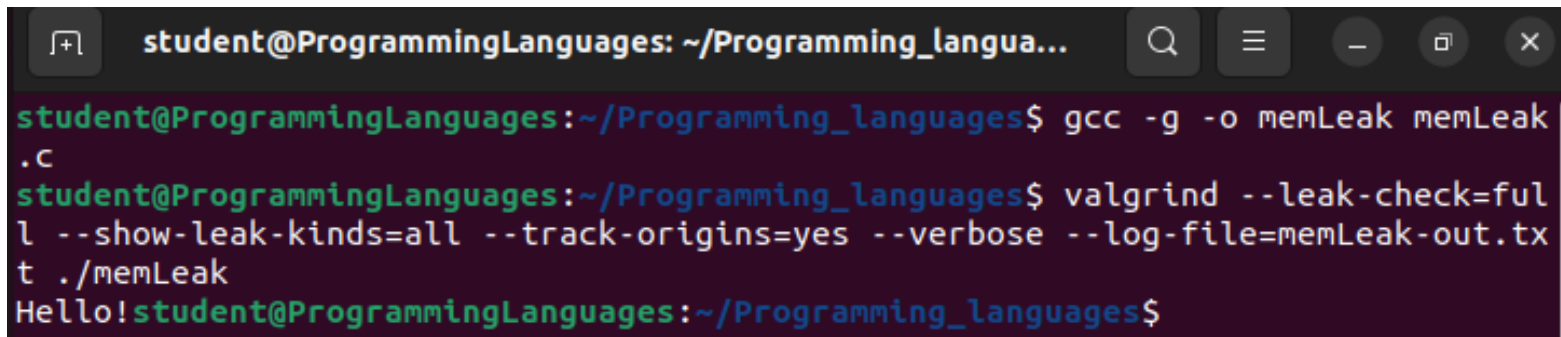
int main()
{
    int *p;
    p = (int*)malloc(3 * sizeof(int));
    p[0] = 1;
    p[1] = 1;
    p[2] = 1;
    int c = 0;
    p = &c;
    printf("Hello!!!!");
    return 0;
}
```

```
student@ProgrammingLanguages:~/Programming_languages$ gcc -o memLeak memLeak.c
student@ProgrammingLanguages:~/Programming_languages$ ./memLeak
Hello!student@ProgrammingLanguages:~/Programming_languages$
```

# Pointers – common errors

```
valgrind --leak-check=full --show-leak-kinds=all --  
track-origins=yes --verbose --log-file=valgrind-out.txt  
./executable exampleParam1
```

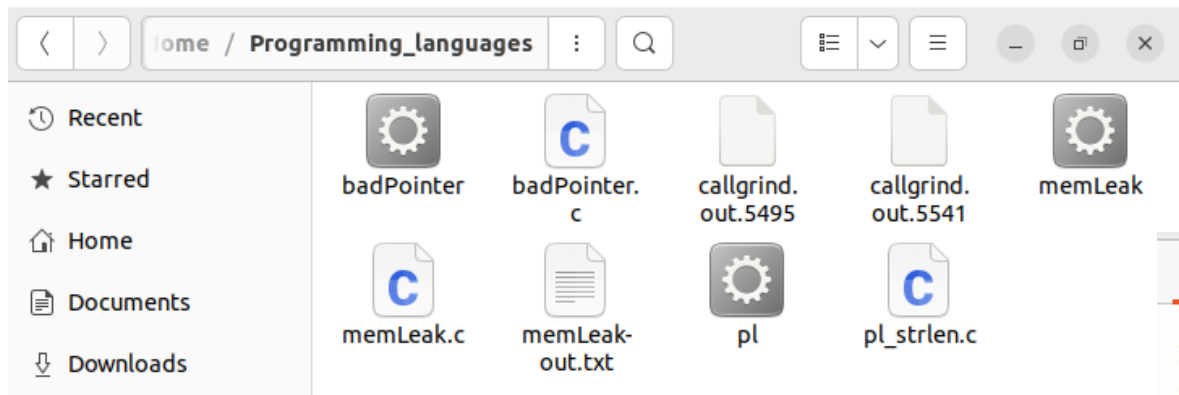
```
valgrind --leak-check=full --show-leak-kinds=all --  
track-origins=yes --verbose --log-file=memLeak-out.txt  
./memLeak
```



A terminal window screenshot showing the compilation and execution of a program. The window title is "student@ProgrammingLanguages: ~/Programming\_langua...". The terminal shows the following commands and output:

```
student@ProgrammingLanguages:~/Programming_languages$ gcc -g -o memLeak memLeak.c  
student@ProgrammingLanguages:~/Programming_languages$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose --log-file=memLeak-out.txt ./memLeak  
Hello!student@ProgrammingLanguages:~/Programming_languages$
```

# Pointers – common errors



```
117 ==5801==
118 ==5801== 12 bytes in 1 blocks are definitely lost in loss record 1
119 ==5801==    at 0x4848899: malloc (in /usr/libexec/valgrind/
    vgppreload_memcheck-amd64-linux.so)
120 ==5801==    by 0x1091B5: main (memLeak.c:7)
121 ==5801==
122 ==5801== LEAK SUMMARY:
123 ==5801==    definitely lost: 12 bytes in 1 blocks
124 ==5801==    indirectly lost: 0 bytes in 0 blocks
125 ==5801==    possibly lost: 0 bytes in 0 blocks
126 ==5801==    still reachable: 0 bytes in 0 blocks
127 ==5801==    suppressed: 0 bytes in 0 blocks
128 ==5801==
129 ==5801== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

```
memLeak.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p=NULL;
7     p=(int*)malloc(3*sizeof(int));
8     p[0]=1;
9     p[1]=2;
10    int c=0;
11    p=&c;
12    printf("Hello!");
13    return 0;
14 }
15
```

# Pointers – common errors

- Dangling pointer
- Use Valgrind to detect dangling pointer

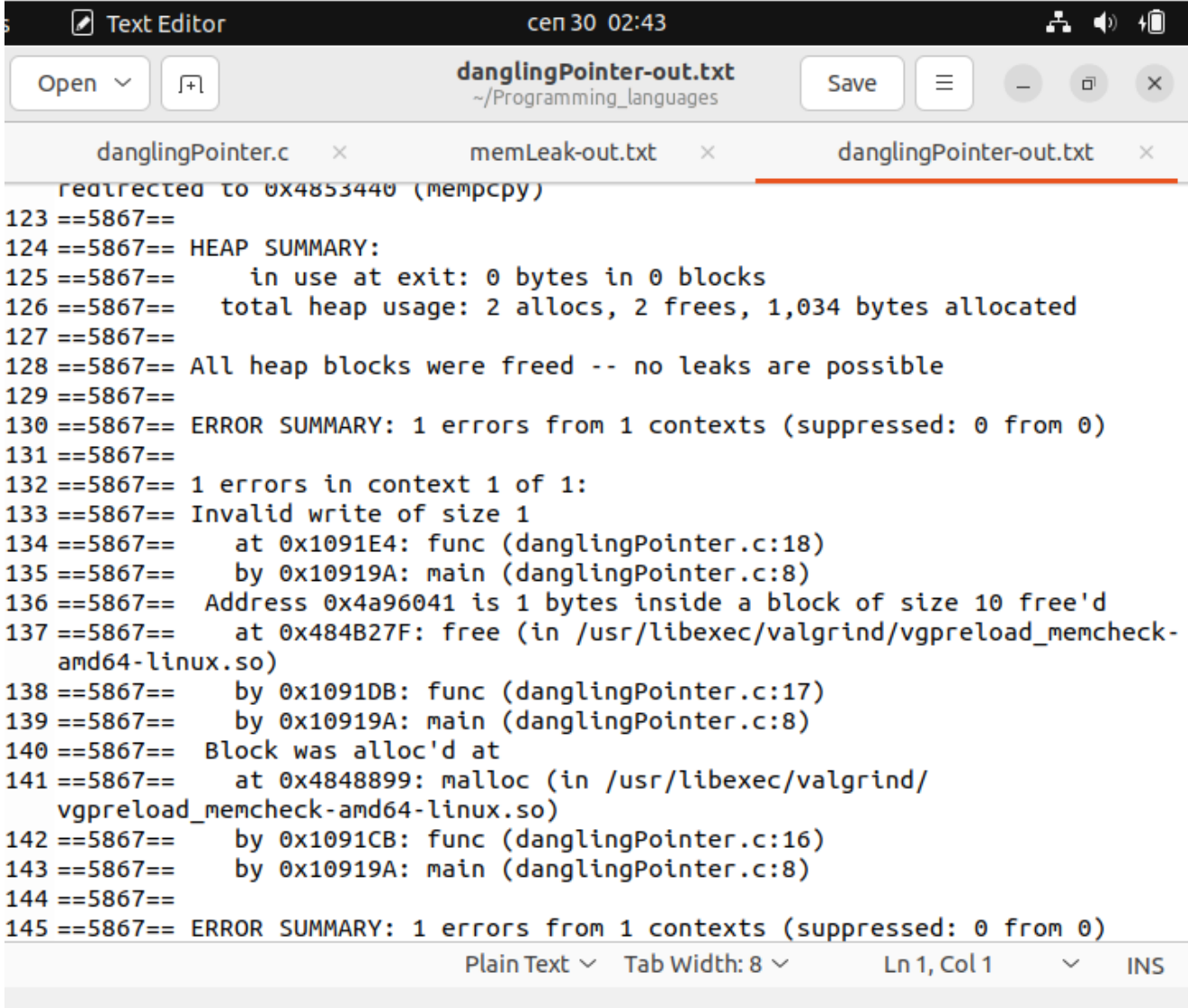
```
void func()
{
    char *p = malloc(10);
    free(p);
    //p = NULL;
}
```

```
int *func(void)
{
    int num = 1234;
    /* ... */
    return &num;
}
```



```
student@ProgrammingLanguages: ~/Programming_langua...
student@ProgrammingLanguages:~/Programming_languages$ gcc -g -o danglingPointer danglingPointer.c
student@ProgrammingLanguages:~/Programming_languages$ ./danglingPointer
HelloHello from main!student@ProgrammingLanguages:~/Programming_languages$
```

# Pointers – common errors



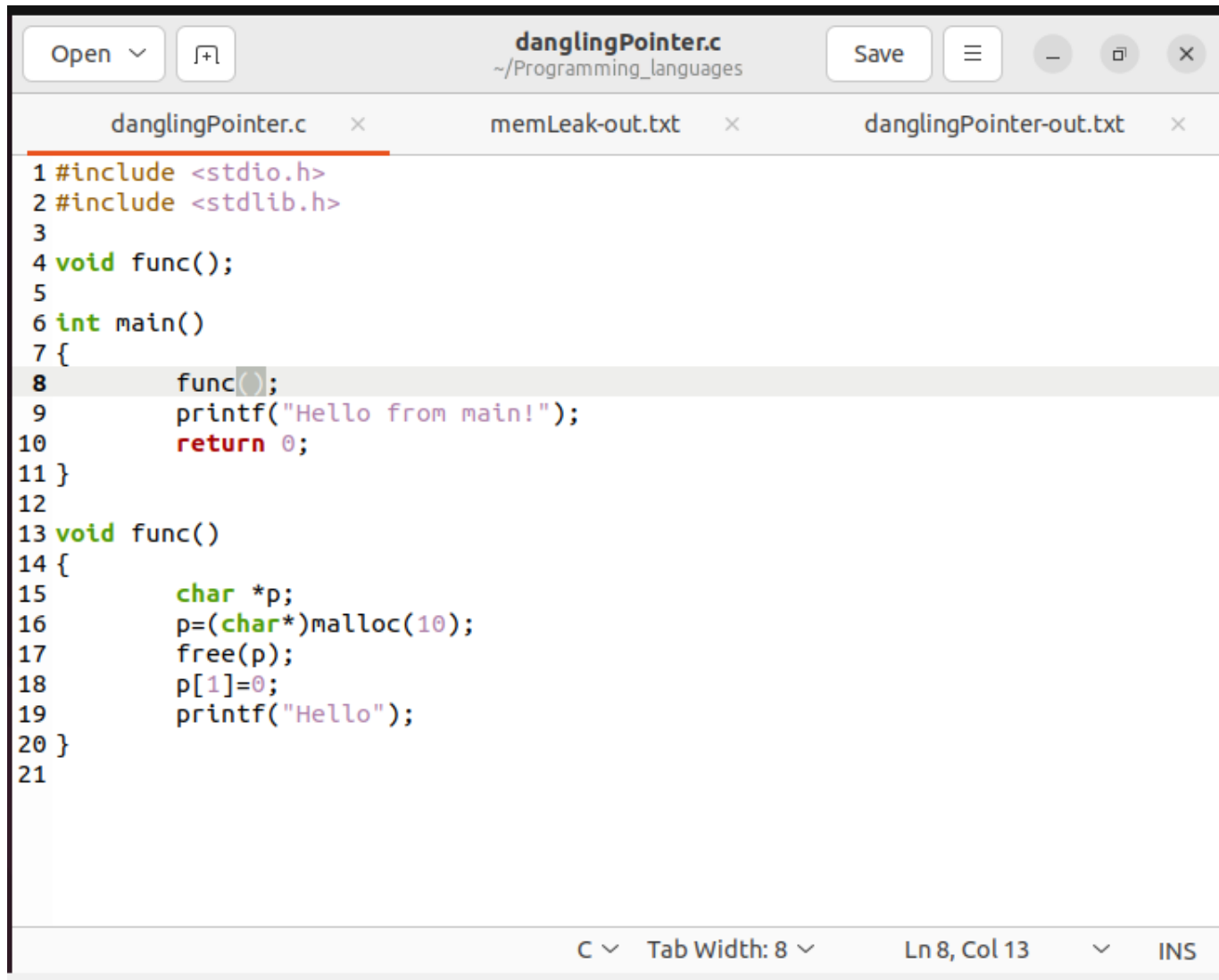
The screenshot shows a text editor window titled "danglingPointer-out.txt" with the path "~/Programming\_languages". The editor has three tabs: "danglingPointer.c", "memLeak-out.txt", and "danglingPointer-out.txt". The content of the active tab is a Valgrind output report. The report starts with "redirected to 0x4853440 (mempcpy)". It then shows a "HEAP SUMMARY:" indicating 2 allocations, 2 frees, and 1,034 bytes allocated. It states "All heap blocks were freed -- no leaks are possible". However, it also shows an "ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)". The error is an "Invalid write of size 1" at address 0x1091E4, caused by a write in "danglingPointer.c:18" from "main (danglingPointer.c:8)". The report further details that the address 0x4a96041 is 1 byte inside a block of size 10 that was freed at 0x484B27F. It also shows the block was allocated at 0x4848899 via malloc. The error summary is repeated at the end of the output.

```
redirected to 0x4853440 (mempcpy)
123 ==5867==
124 ==5867== HEAP SUMMARY:
125 ==5867==     in use at exit: 0 bytes in 0 blocks
126 ==5867==   total heap usage: 2 allocs, 2 frees, 1,034 bytes allocated
127 ==5867==
128 ==5867== All heap blocks were freed -- no leaks are possible
129 ==5867==
130 ==5867== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
131 ==5867==
132 ==5867== 1 errors in context 1 of 1:
133 ==5867== Invalid write of size 1
134 ==5867==    at 0x1091E4: func (danglingPointer.c:18)
135 ==5867==    by 0x10919A: main (danglingPointer.c:8)
136 ==5867== Address 0x4a96041 is 1 bytes inside a block of size 10 free'd
137 ==5867==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-
    amd64-linux.so)
138 ==5867==    by 0x1091DB: func (danglingPointer.c:17)
139 ==5867==    by 0x10919A: main (danglingPointer.c:8)
140 ==5867== Block was alloc'd at
141 ==5867==    at 0x4848899: malloc (in /usr/libexec/valgrind/
    vgpreload_memcheck-amd64-linux.so)
142 ==5867==    by 0x1091CB: func (danglingPointer.c:16)
143 ==5867==    by 0x10919A: main (danglingPointer.c:8)
144 ==5867==
145 ==5867== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Plain Text ▾ Tab Width: 8 ▾ Ln 1, Col 1 ▾ INS



# Pointers – common errors



The screenshot shows a code editor window titled "danglingPointer.c" with the path "~/Programming\_languages". The editor has three tabs: "danglingPointer.c", "memLeak-out.txt", and "danglingPointer-out.txt". The code in "danglingPointer.c" is as follows:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void func();
5
6 int main()
7 {
8     func();
9     printf("Hello from main!");
10    return 0;
11 }
12
13 void func()
14 {
15     char *p;
16     p=(char*)malloc(10);
17     free(p);
18     p[1]=0;
19     printf("Hello");
20 }
21
```

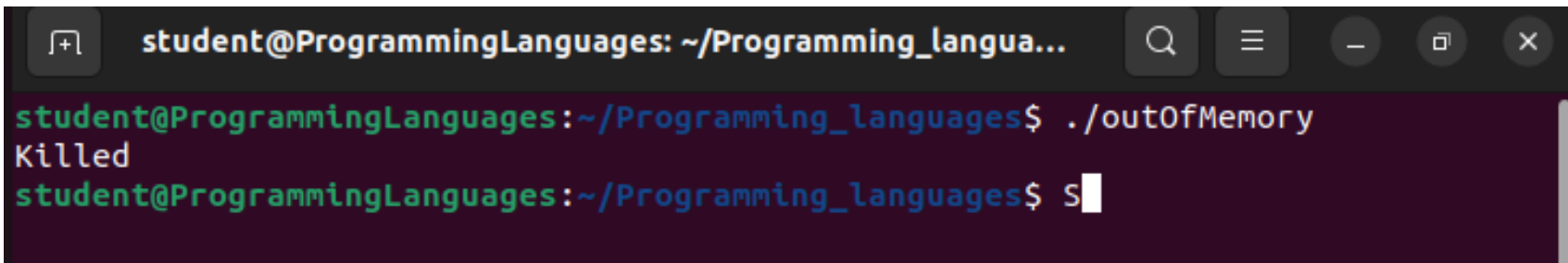
The cursor is positioned at line 8, column 13. The status bar at the bottom indicates "C", "Tab Width: 8", "Ln 8, Col 13", and "INS".

# Out of memory

- Out of memory

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

int main()
{
    int *p;
    for (int i=0;i<100000000;i++)
        p = (int*)malloc(1000000000 * sizeof(int));
    //do important stuff
    free(p);
    return 0;
}
```

A terminal window with a dark background. The title bar shows the user 'student' and the directory '~/Programming\_languages'. The command './outOfMemory' has been executed, resulting in the output 'Killed'. The prompt is now 'student@ProgrammingLanguages:~/Programming\_languages\$' followed by a space and the letter 'S'.

```
student@ProgrammingLanguages: ~/Programming_languages$ ./outOfMemory
Killed
student@ProgrammingLanguages:~/Programming_languages$ S
```

# Out of memory

```
6238          0 memcheck-amd64-
[nt cen 30 02:53:00 2022] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),c
puset=/,mems_allowed=0,global_oom,task_memcg=/user.slice/user-1000.slice/user@1
000.service/app.slice/app-org.gnome.Terminal.slice/vte-spawn-332747f8-ece5-4af6
-9c8e-490a679225a3.scope,task=memcheck-amd64-,pid=5996,uid=1000
[nt cen 30 02:53:00 2022] Out of memory: Killed process 5996 (memcheck-amd64-)
total-vm:3742864kB, anon-rss:1631396kB, file-rss:0kB, shmem-rss:0kB, UID:1000 p
gtables:7320kB oom_score_adj:0
student@ProgrammingLanguages:~/Programming_languages$
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void func();
5
6 int main()
7 {
8     int *p;
9
10    for(int i=1;i<1000000000;i++)
11        p=(int*)malloc(i*sizeof(int));
12    s|
13
14    free(p);
15    return 0;
16 }
17
18
```

# Questions

