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

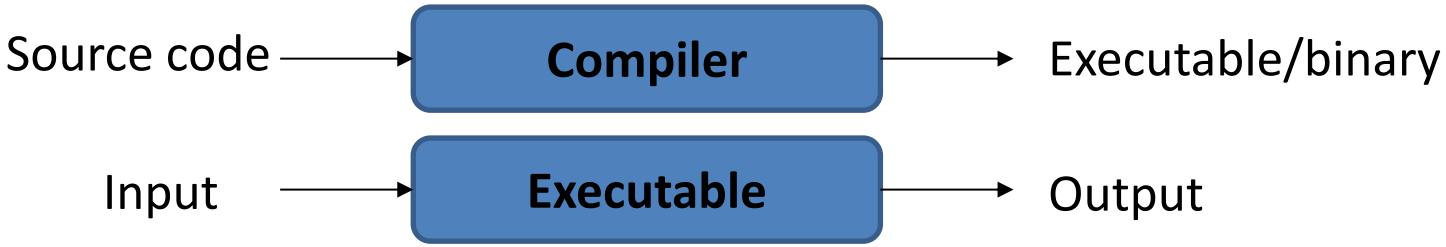
The C programming language

- 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)*.

```c
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;
}
```

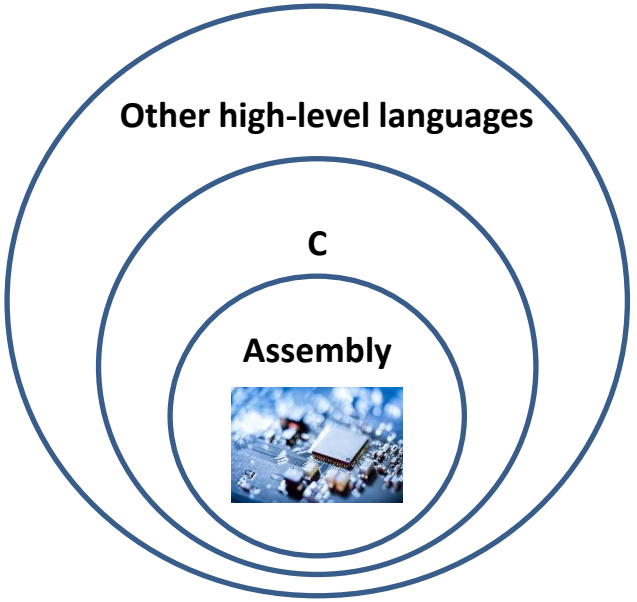
The C programming language

- 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.

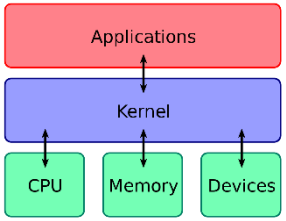


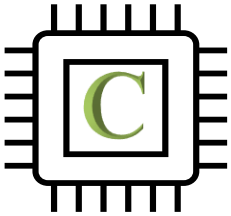
The C programming language

- 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

1 cheesecake – 60 minutes

1 bread – 10 minutes

10 apple pies – 10 x 30 minutes

10 cheesecakes – 10 x 60 minutes

10 breads – 10 x 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·logN)…

Big O

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

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

- 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 $[2\cdot N + 2\cdot(N\text{-}1)]\cdot N$ 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.

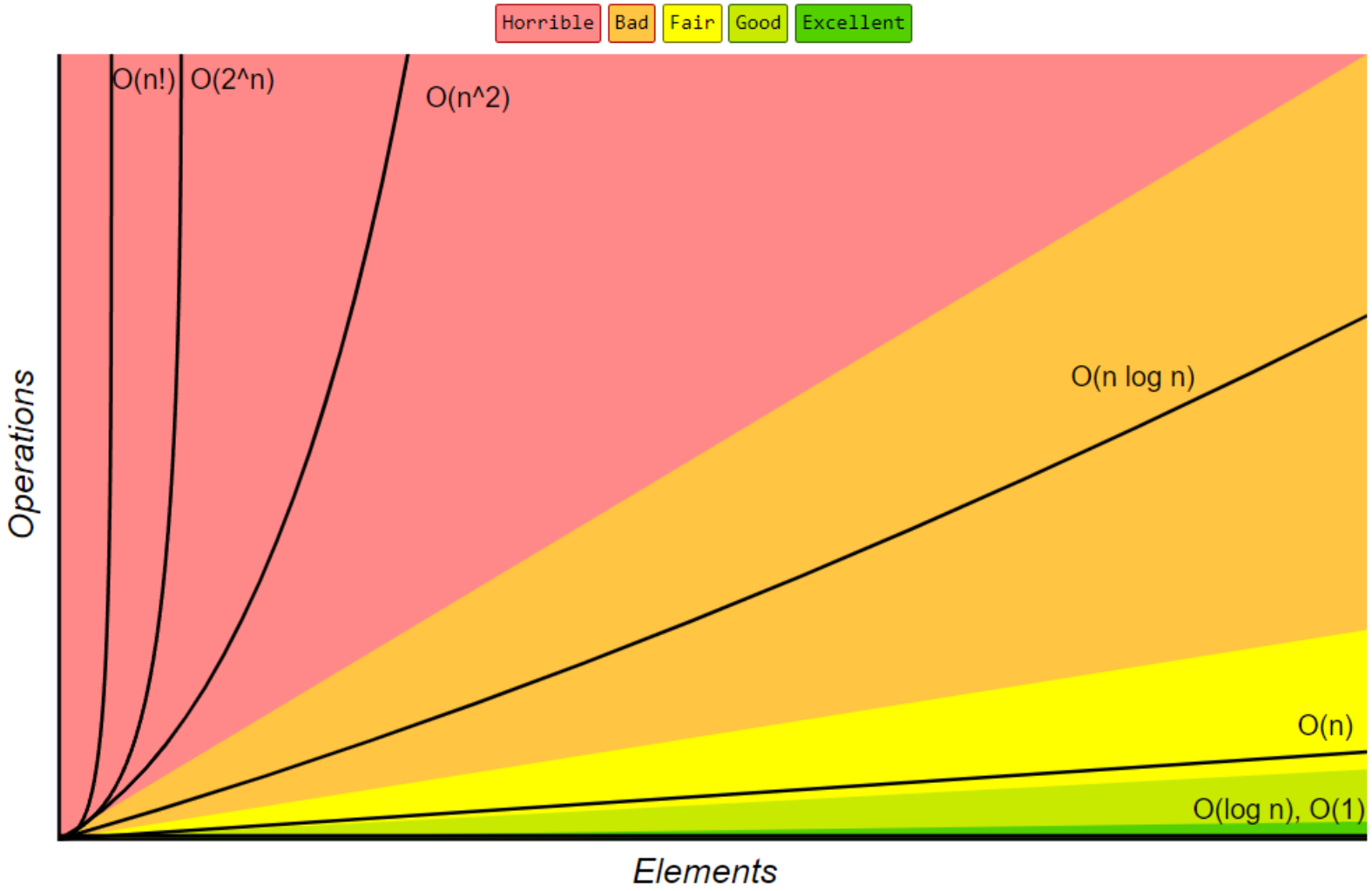
  O(5N) -> O(N)

  O(5+N) -> 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) -> 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  }
```

$N \cdot O(1)$ -> $O(N)$

```
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  }
```

$O(1) + N \cdot O(1)$ -> $O(N)$

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  }
```

$N \cdot N \cdot O(1)$ -> $O(N^2)$

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)

$N \cdot O(1)$ -> $O(N)$

$N \cdot N \cdot O(1)$ -> **$O(N^2)$**

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	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(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
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

https://www.bigocheatsheet.com/

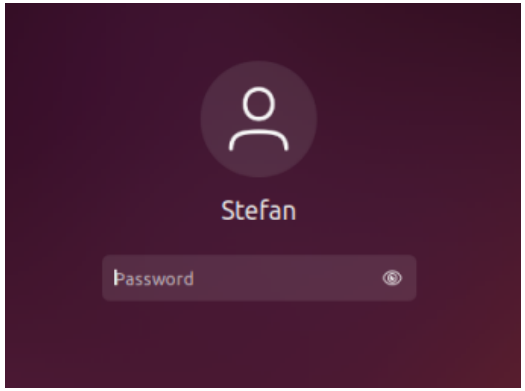
Big O – hands on

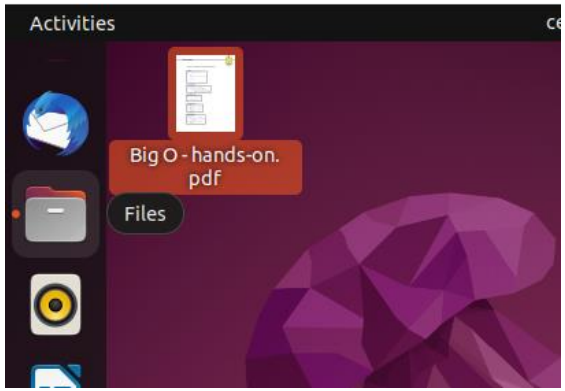
1. **Find the computational efficiency of the following code snippets, using big O.**
2. **For each code snippet, measure running time for different values of the size of the input data. Write your findings in the following table. Compare the results with the ones computed using big O.**

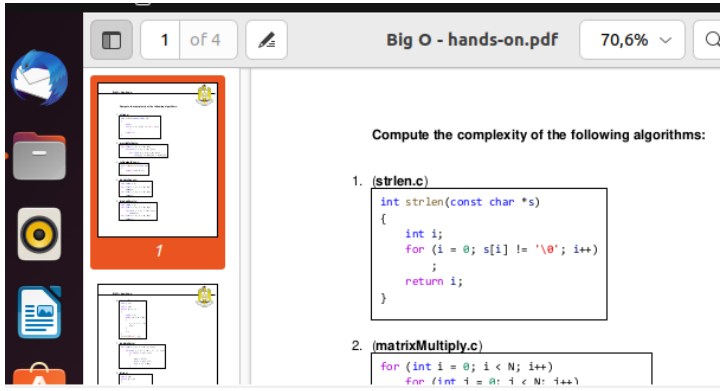
N	1	10	100	1000	10000	100000	1000000
Time							
Time –O3							
O(N)							

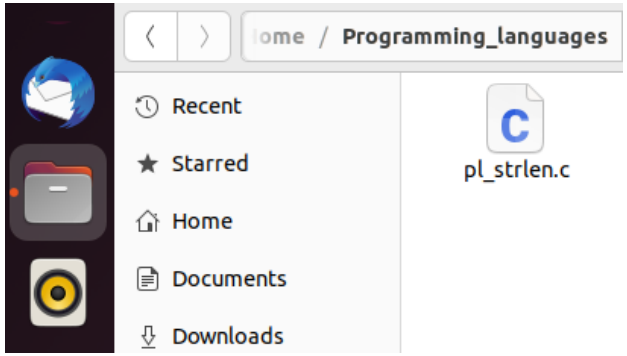
Use the VM provided by the teacher, to measure code efficiency.

Big O – hands on





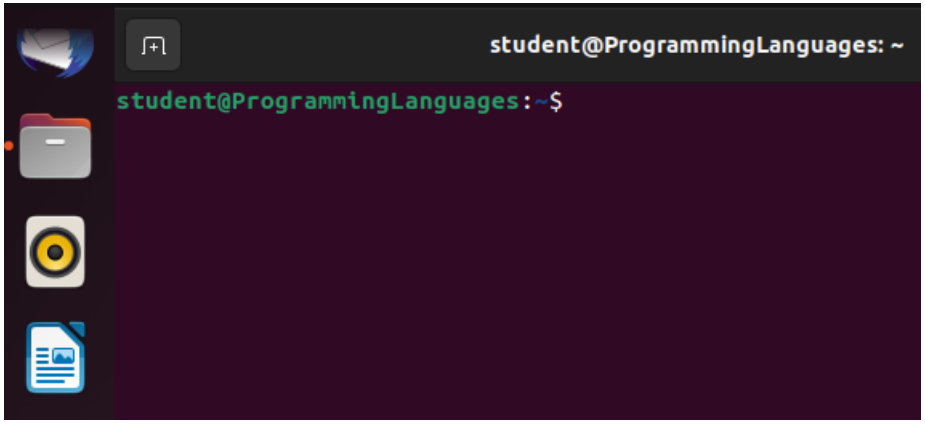




Code snippets from:

[courses/Enunt-laborator03.docx at master · cristianchilipirea/courses · GitHub]

Big O – hands on



CTRL + ATL + T

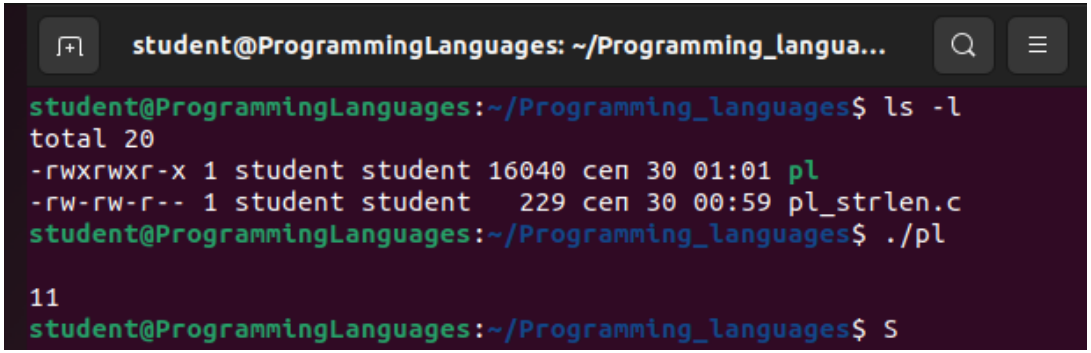
```
student@ProgrammingLanguages:~/Programming_languages$ gcc -O0 -o pl pl_strlen.c

student@ProgrammingLanguages:~/Programming_languages$
```

gcc –O0 –o pl1 pl_strlen.c

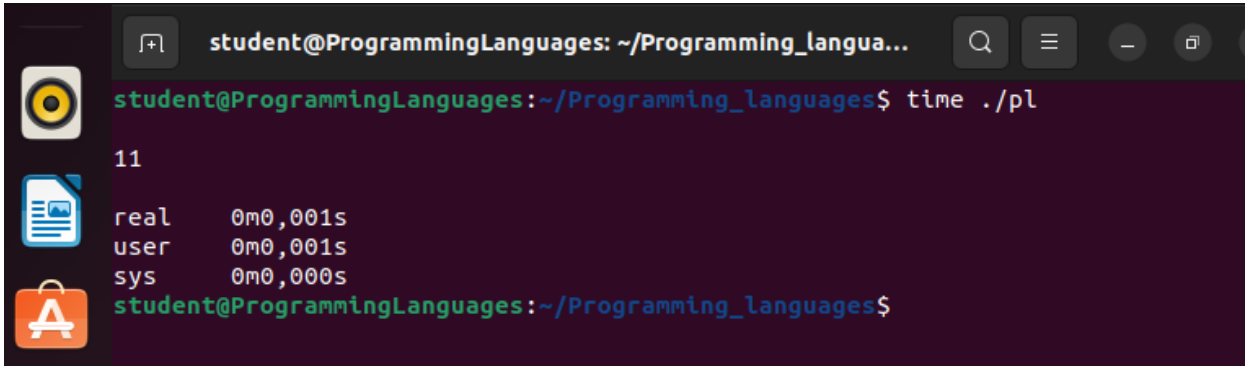
-O0 – no optimization

pl1 – executable file

pl_strlen.c – source code



Big O – hands on




```c
int main()
{
        char *string;
        int N=100000,length=0;
        double time_spent=0.0;

        /*Initialize array*/
        string=(char*)calloc(N,sizeof(char));
        for(int i=0;i<N-1;i++)
                string[i]='A';
        /*End*/

        clock_t begin= clock(); //start time
        length=myStrlen(string); //do important stuff
        clock_t end =clock();   //end time

        time_spent=(double)(end-begin)/CLOCKS_PER_SEC;
        printf("\nTime spent: %f\n",time_spent);

        free(string);

        return 0;
}
```

```
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 dot 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...]
```

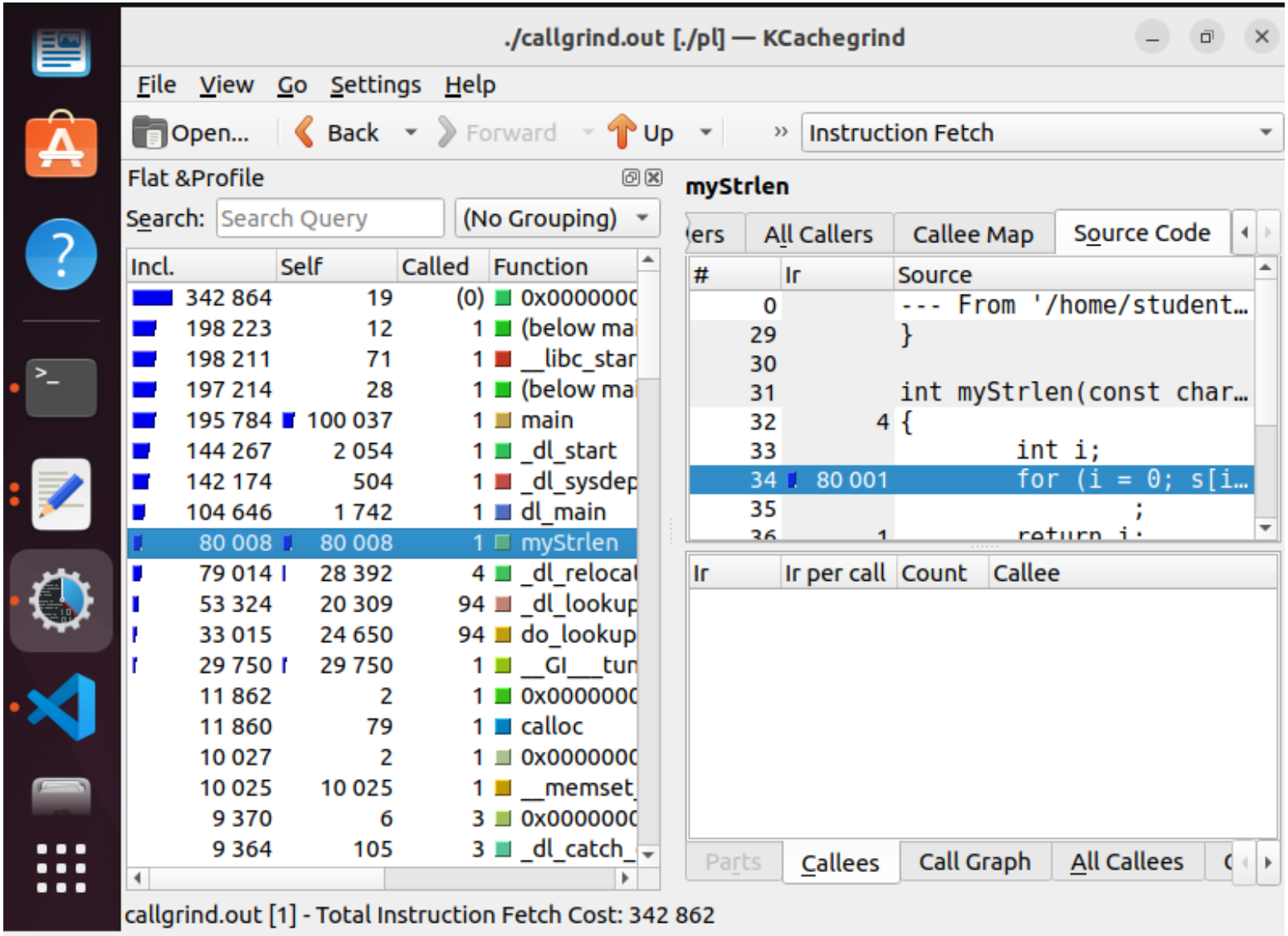
```
valgrind --tool=callgrind --dump-instr=yes --
simulate-cache=yes --collect-jumps=yes ./pl
```

```
kcachegrind
```

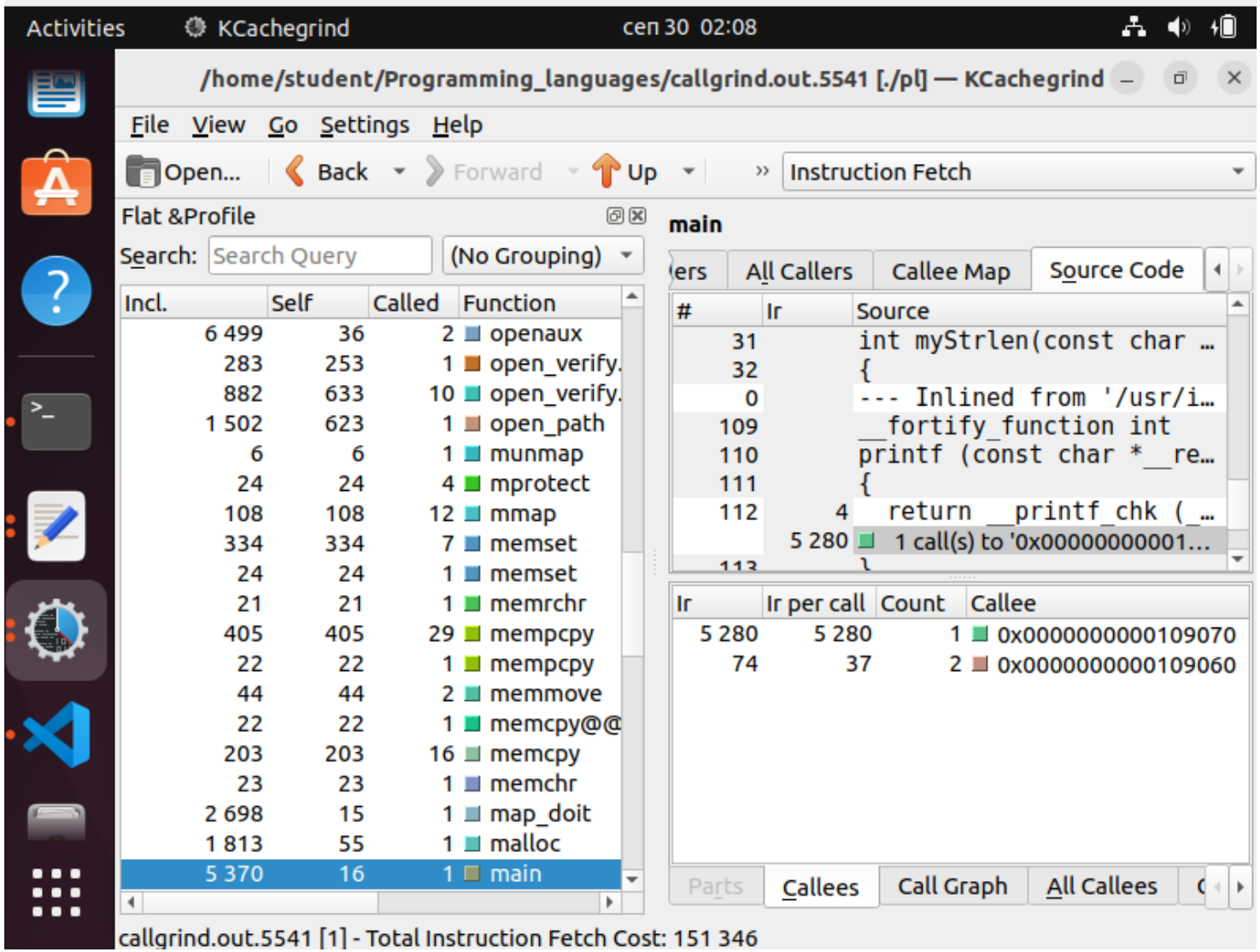
Code profiling. Valgrind

```
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%  )
```

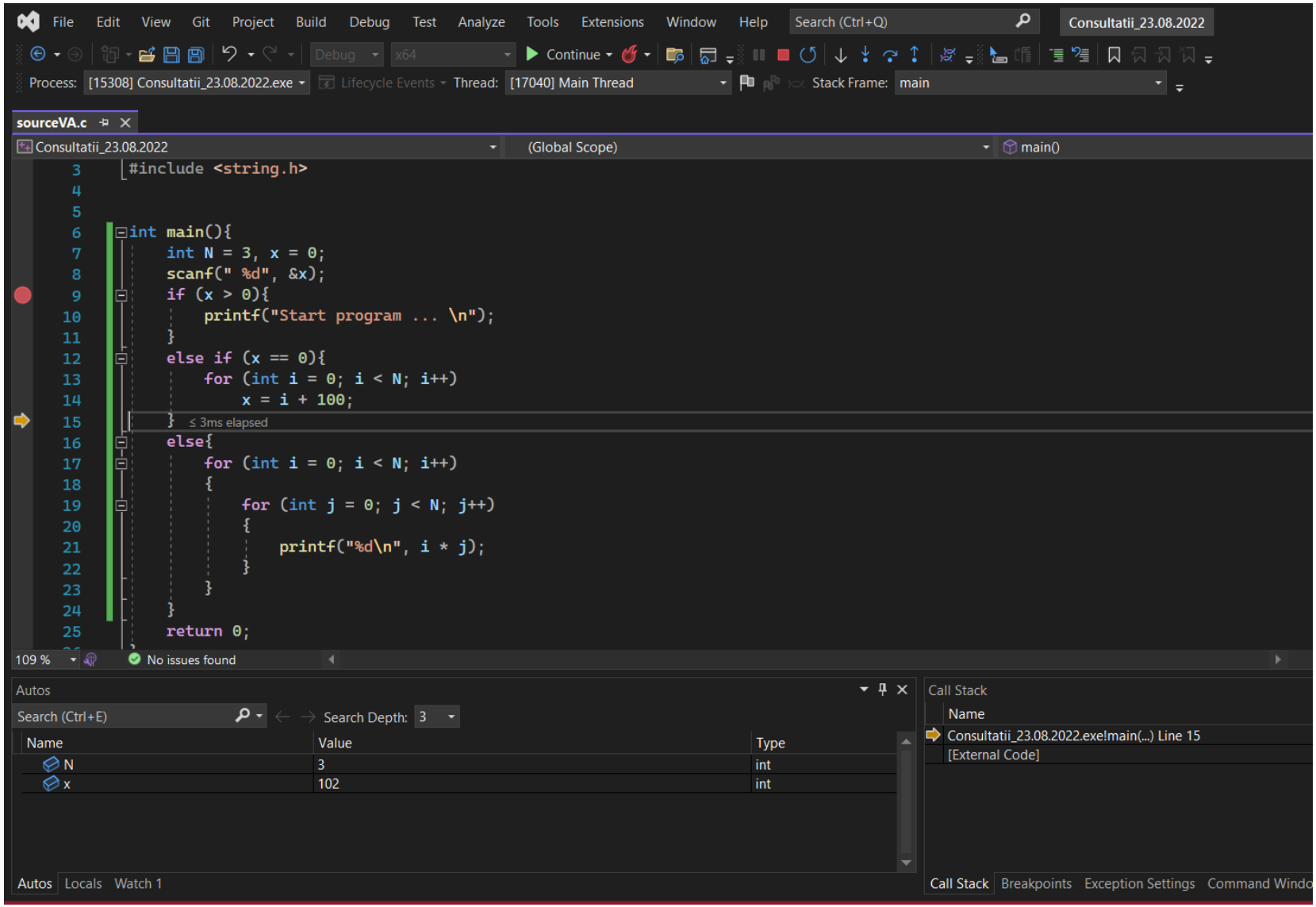
Code profiling. Kcachegrind



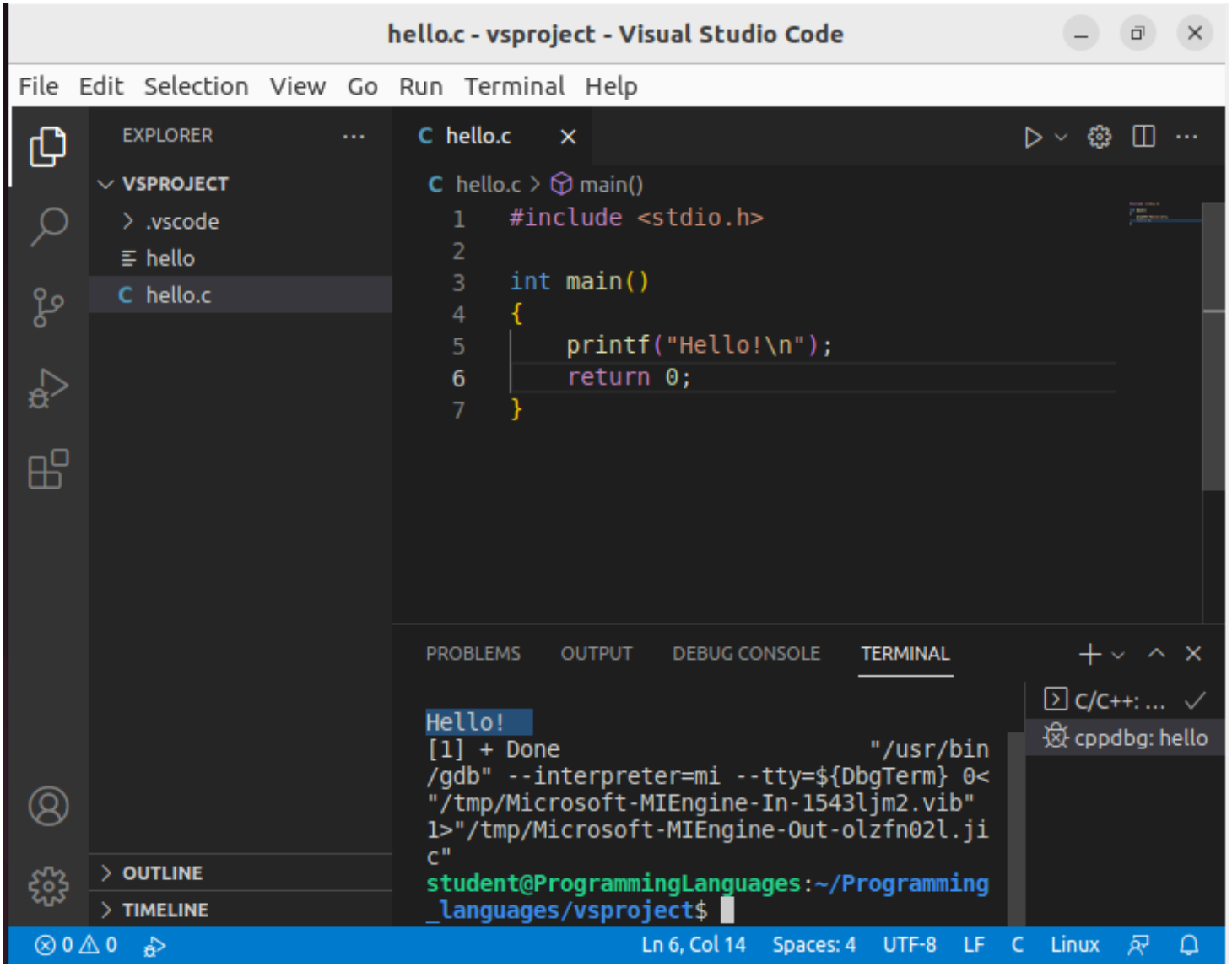
Code profiling. Kcachegrind



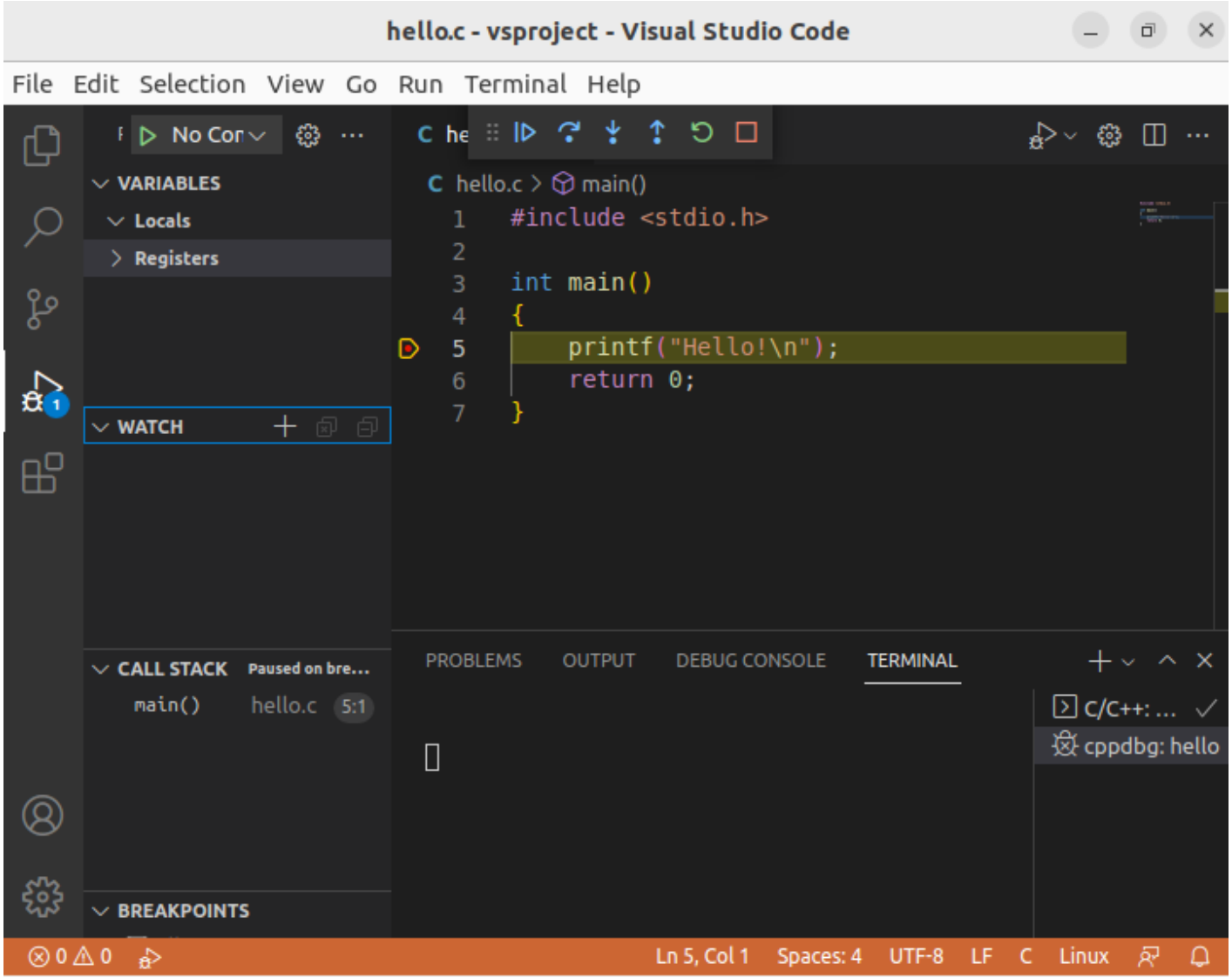
Count no. of operations using the MSVS debugger



Count no. of operations using VSCode



Count no. of operations using VSCode



FFT

- 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;
}

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

Exception Thrown

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

Copy Details

Exception Settings

Break when this exception type is thrown

Call Stack

```
student@ProgrammingLanguages: ~/Programming_langua...
student@ProgrammingLanguages:~/Programming_languages$ gcc -o badPointer badPoin
ter.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

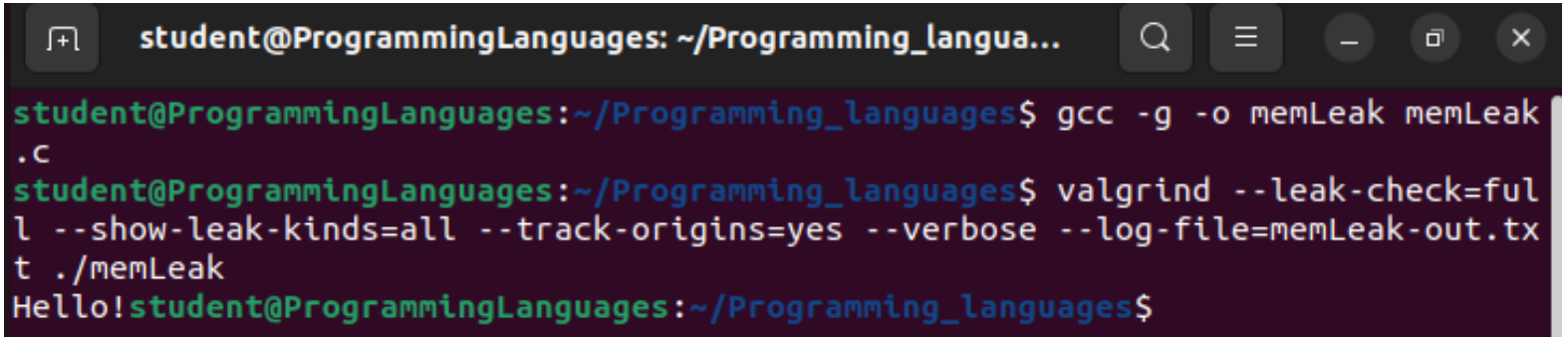
```c
#include <stdio.h>
#include <stdlib.h>

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$
```

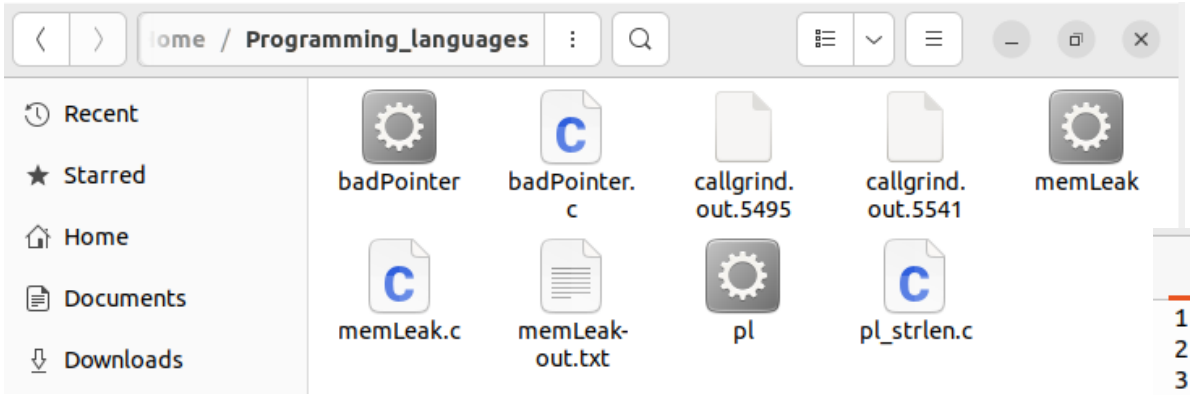
Pointers – common errors

```
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



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/
    vgpreload_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

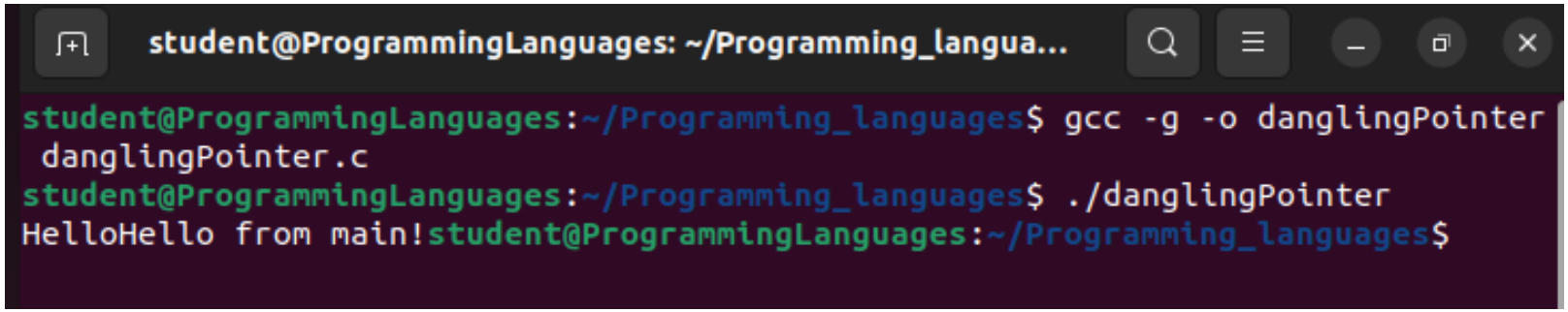
```c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6         int *p=NULL;
7         p=(int*)malloc(3*sizeof(int));
8         p[0]=1;
9         p[1]=2;
10        int c=0;
11        p=&c;
12        printf("Hello!");
13        return 0;
14 }
15
```

Pointers – common errors

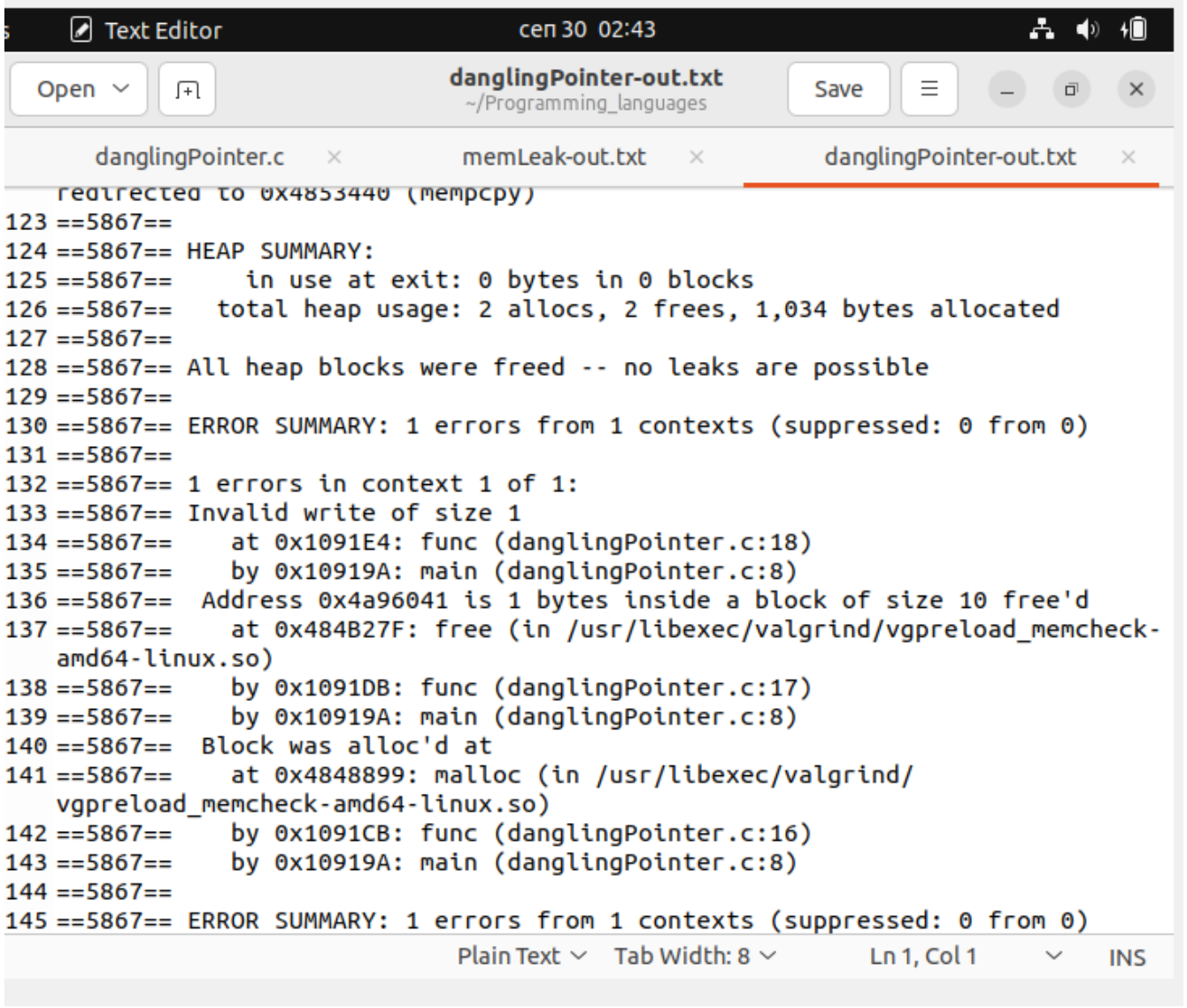
- Dangling pointer
- Use Valgrind to detect dangling pointer

```c
void func()
{
     char *p = malloc(10);

     free(p);
     //p = NULL;
}
```

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



Pointers – common errors



Pointers – common errors

Open | danglingPointer.c ~/Programming_languages | Save

danglingPointer.c | memLeak-out.txt | danglingPointer-out.txt

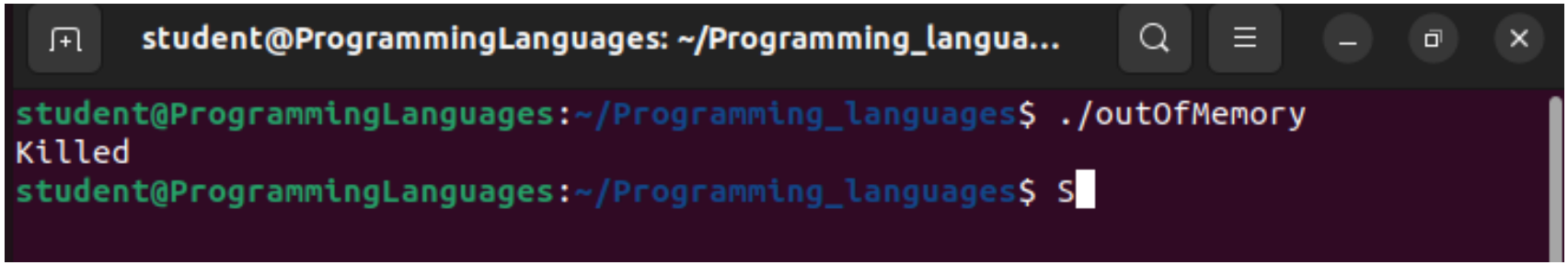
```c
#include <stdio.h>
#include <stdlib.h>

void func();

int main()
{
        func();
        printf("Hello from main!");
        return 0;
}

void func()
{
        char *p;
        p=(char*)malloc(10);
        free(p);
        p[1]=0;
        printf("Hello");
}

```

C | Tab Width: 8 | Ln 8, Col 13 | INS

Out of memory

- **Out of memory**

```c
#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;
}
```



Out of memory

```
6238             0 memcheck-amd64-
[пт сен 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
[пт сен 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$
```

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

void func();

int main()
{
        int *p;

        for(int i=1;i<100000000;i++)
                p=(int*)malloc(i*sizeof(int));
        S

        free(p);
        return 0;
}
```

Questions