High Performance Programming, Lecture 3

More programming in C

Follow up on Lab 1

Check your quota on unix systems using `quota -v`:

```
vranx> quota -v
Disk quotas for anakr123 (uid 340973):
Filesystem     usage  quota  limit   .....
/home/anakr123
               24889 1048576 1048576   .....
vranx>
```

Find largest files and directories:

```
du -a | sort -n
```

C building process

4 steps: preprocessing, compilation, assembly, linking.

Look at C_build_process.pdf (see under Extra files in Student Portal) for more information.

Object files (.o) are "almost executables". Contain: the machine code + metadata about the addresses of all used variables and functions (also called symbols).

Undefined reference error:
That means that this symbol is referenced or used in this object file, but its value was not defined here.
Undefined symbols cannot be used unless a **definition** is linked in via a library or object file!

Follow up on Lab 1

Program's address space

Running program's view of memory in the system:

```
---------------------------------------
|                 stack               |
|-------------------------------------|
|                                     |
|                                     |
|                                     |
|-------------------------------------|
|                 heap                |
|-------------------------------------|
|            static data segment:     |
|      read-only data                 |
|      read-write initialized variables   |
|      read-write uninitialized variables |
---------------------------------------
```

Follow up on Lab 1

Program's address space

The stack is a LIFO (Last-In-First-Out) data structure. Memory allocation/deallocation is done automatically.

If memory is allocated on the heap, it exists as long as the developer wants.

Image: https://stackoverflow.com

Follow up on Lab 1

Program's address space (Try it!)

Example program: test stack and heap

```
stack at 0x7fff0df1b697, heap at 0x1e12260
stack at 0x7fff0df1b667, heap at 0x1e12690
stack at 0x7fff0df1b637, heap at 0x1e126b0
stack at 0x7fff0df1b607, heap at 0x1e126d0
stack at 0x7fff0df1b5d7, heap at 0x1e126f0
```

```c
#include <stdlib.h>
#include <stdio.h>
void f(int depth) {
    if (depth <= 0) return;
    char c;
    char *ptr = malloc(1);
    printf("stack at %p, heap at %p\n", &c, ptr);
    f(depth-1);}
int main() {
    f(5);
    return 0;}
```

Note the difference!

```
char *s1 = "Hello";
```

Put the literal string "Hello" in the read-only memory and let s1 point to it. Any writing operation on this memory illegal!

```
char s2[] = "Hello";
```

Put the literal string "Hello" in the read-only memory and copy it to newly allocated memory on the stack (s2 is an array of 6 characters: 'H', 'e', 'l', 'l', 'o', and '\0').
Now `s[0] = 'J';` is legal.

Follow up on Lab 1

Compare

```
void assign_string(char
    *g)
{
    g = "Good!";
}
int main()
{
  char *s = "Hello";
  assign_string(s);
  printf("%s\n", s); //
    ‘‘Hello’’
  return 0;
}
```

```
void assign_string(char
    **g)
{
    *g = "Good!;
}
int main()
{
  char *s = "Hello";
  assign_string(&s);
  printf("%s\n", s); //
    ‘‘Good!’’
  return 0;
}
```

Follow up on Lab 1

Consider the following code:

```
int * p = (int *) malloc (sizeof(int));
*p = 0; // !!!!!!
for(int i = 0; i < N; i++)  *p += i;
printf("%d\n", *p);
```

(Pseudo)random numbers

`rand` returns a pseudo-random number in the range of 0 to RAND_MAX.

`srand` sets seed for the pseudo-random number generator used by rand().

`time` returns the time since the Epoch (00:00:00 UTC, January 1, 1970) in seconds.

```
    srand(time(NULL));   // Initialization (called
   once).
    int r = rand();      // Returns a pseudo-random
   integer between 0 and RAND_MAX.
  }
```

Follow up on Lab 1

Structure datatype. Compare:

```
struct node {
  int val;
  struct node * next;
};
typedef struct node
 node_t;

node_t A;
```

```
typedef struct node {
  int val;
  struct node * next;
} node_t;

node_t A;
```

Followup after Lab 1

Small example programs available

For those who are new to the C language, a few example programs are available in the Student Portal:

Documents ⟶ Example programs

- `input_args.c`: **argc and argv** in main() function
- `pointers.c`: playing around with **pointers**
- `file_io.tar.gz`: **reading and writing files** using fopen(), fread(), fwrite(), etc. Reading and writing text files.
- `struct_example.c`: simple example of using a **struct** datatype
- `func_with_ptr_arg.c`: simple examples of using **pointer arguments** to functions
- `read_input_data_into_array.c`: reading input characters into an array
- and some more

If you missed Lab 1

Look at the document CONTENT $\longrightarrow$ "Getting started with linux" in the Student Portal.

Labs are mandatory.
The rule is:
if you miss a lab you must submit a small report for that lab instead, no later than **3 working days after the lab**.

Submit using Student portal under ASSIGNMENTS $\longrightarrow$ " Extra reports" or send me by email (first option is preferable)

Follow up on Lab 1

Any questions?

Check that they get marked as completed in the Progress function in the Student Portal.

Assignment 1

IMPORTANT NOTES:

- no binary files such as object files or executable files should be included
- makefile should have the name “Makefile”, use the checking script! It should finish with “Congratulations” message.
- specify dependencies correctly, if file is changed it should be recompiled
- read carefully “Preparing your submission” part!

Submit using Student portal. Deadline is this Friday (February 1)!

Assignment 2

Assignment 2 is available in Student Portal. Submit your solution using Student portal. Deadline is next Friday (February 8)!

For Assignments 2-6 you are expected to **work in pairs**, "groups" with two students/group, using the "group division" function in the Student Portal. When you are in a group, you can access the files for Assignment 2.

If you **do not have a teammate, send email** to Anastasia about it, then I can pair you up with someone.

Today:

- Linked list (very briefly)
- Multidimensional arrays
- Pointers to functions
- Timing and complexity (depends on time left)

Next time:
Optimization – Part I (how to do less work)

Linked list

Linked list is a simple example of a dynamic structure which uses pointers.

Advantages over arrays:

- Items can be added or removed from the middle of the list;
- No need to define an initial size.

Disadvantages:

- No random access;
- Uses pointers $\Longrightarrow$ complications of code;
- Larger overhead due to dynamic allocation.

Linked list

```
 -----------------            ----------------
|        |        |          \ |        |        |
|  DATA  |  NEXT  |--------------|  DATA  |  NEXT |
|        |        |          / |        |        |
 -----------------            ----------------
```

Define a linked list node:

```c
struct node {
int val;
struct node * next;
};

typedef struct node node_t;
```

Linked list

Add item to the end of the list:

```
void push(node_t * head, int val) {
  node_t * current = head;
  while (current->next != NULL) {
          current = current->next;
  }

  current->next = (node_t *) malloc(sizeof(node_t));
  current->next->val = val;
  current->next->next = NULL;
}
```

Linked list

Look at the tutorial page:
`https://www.learn-c.org/en/Linked_lists`.

You will need to create your own linked list in Assignment 2.

Multidimensional arrays

General form: `type name[size1][size2]...[sizeN];`

Arrays are always laid out *contiguously* in memory.

2D array initialization:

```
int arr[2][5] = {
{0, 1, 2, 3} ,
{5, 6, 7, 8, 9}
};
int value= a[1][3]; // value = 8
```

Multidimensional arrays

General form: `type name[size1][size2]...[sizeN];`
Arrays are always laid out *contiguously* in memory.
2D array initialization:

```
int arr[2][5] = {
{0, 1, 2, 3} ,
{5, 6, 7, 8, 9}
};
int value= a[1][3]; // value = 8
```

`arr[2][5]` is a matrix with 2 rows and 5 columns (array of arrays)

In memory it is stored in row-major order: after `arr[0][3]` comes `arr[0][4]` and after `arr[0][4]` comes `arr[1][0]`.

0	1	2	3	0	5	6	7	8	9

Multidimensional arrays

2D array initialization:

```
int arr[2][5] = {
        {0, 1, 2, 3} ,
        {5, 6, 7, 8, 9}
};
//or
int arr[2][5] = { 0, 1, 2, 3, 0, 5, 6, 7, 8, 9};
// or skipping the first dimension (not the second!)
int arr[][5] = {
{0, 1, 2, 3} ,
{5, 6, 7, 8, 9}
};
```

0	1	2	3	0	5	6	7	8	9

Dynamic memory allocation

Allocate memory for a matrix 3×5 dynamically.

```
int **arr = (int **)malloc(3 * sizeof(int*));
for (i = 0; i < 3; i++)
  arr[i] = (int *)malloc(5 * sizeof(int));
```

Dynamic memory allocation

Allocate memory for a matrix 3×5 dynamically.

```
int **arr = (int **)malloc(3 * sizeof(int*));
for (i = 0; i < 3; i++)
  arr[i] = (int *)malloc(5 * sizeof(int));
```

How to free the allocated memory?

Dynamic memory allocation

Allocate memory for a matrix 3×5 dynamically.

```
int **arr = (int **)malloc(3 * sizeof(int*));
for (i = 0; i < 3; i++)
  arr[i] = (int *)malloc(5 * sizeof(int));
```

How to free the allocated memory?

```
for (int i = 0; i < 3; i++)
  free(arr[i]);
free(arr);
```

Dynamic and static memory allocation of a matrix

Compare:

```
int arr[2][3]; // 2D array, size is known at the
    compile time
int* arr[3];    // array of pointers
int **arr;  // pointer to pointer
```

Example:

```
int main(int argc, char *argv[]) { /* ... */ }
// or
int main(int argc, char **argv) { /* ... */ }
```

Pointers to functions

Declaration of a function:

```
function_return_type function_name(function argument
    list)
```

Declaration of a pointer to a function:

```
function_return_type (*pointer_name)(function
    argument list)
```

For example (note parentheses () here):

```
void  (*ptrfun)()
void  (*ptrfun)(double, char)
double  (*ptrfun)(double, char *)
int*  (*ptrfun)(int*)
```

Check out this “dictionary” for C/English: `https://cdecl.org/`.

Pointers to functions

```
#include <stdio.h>
#include <stdlib.h>
double add(double a, double b)
{return a+b;}

void print_output(double a, double b, double (*funptr)(
    double, double))
{printf("Value is %lf \n", (*funptr)(a, b));}

int main(int argc, char const *argv[])
{
 int n = atoi(argv[1]);
 double (*funptr)(double, double); // declare pointer to
    function
 funptr = &add; // assign address of a function
 print_output(3, 4, funptr);
 return 0;
}
```

qsort

`qsort` function (from stdlib.h) for sorting arrays:

```
void qsort (void* base, size_t num, size_t size, int
    (*compar)(const void*,const void*));
```

For types that can be compared using regular relational operators, a general compar function may look like:

```
int compareMyType (const void * a, const void * b)
{
  if ( *(MyType*)a <  *(MyType*)b ) return -1;
  if ( *(MyType*)a == *(MyType*)b ) return 0;
  if ( *(MyType*)a >  *(MyType*)b ) return 1;
}
```

Note: for strings we should use `strcmp`!

qsort (Test it!)

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

int CmpDouble(const void * p1, const void *p2)
{
  double a = *(double *)p1;
  double b = *(double *)p2;
  if(a > b) return -1;
  if(a < b) return 1;
  return 0;
}

int main() {
    double arrDouble[] = {9.3, -2.3, 1.2, -0.4, 2, 9.2, 1, 2, 0};
    int arrDoubleLen = sizeof(arrDouble) / sizeof(double);
    qsort (arrDouble, arrDoubleLen, sizeof(double), CmpDouble);
    for (int i=0; i<arrDoubleLen; ++i)
        printf("%d: %lf\n", i, arrDouble[i]);
return 0;
}
```

Another example (later in the course)

Function for creating a new POSIX thread:

```
#include <pthread.h>

int
pthread_create (pthread_t *thread_id,
                const pthread_attr_t *attributes,
                void *(*thread_function)(void *),
                void *arguments);
```

`thread_function` is the function the new thread is executing.

Time measuring

Bash command `time`.

C commands:

- `gettimeofday()`
- `clock_gettime()` on Solaris or Linux, or `clock_get_time()` on Mac.

Time measuring

 = user time

 = system time

 = some other user's time

 = real (wall clock) time

***Usually the word "time" refers to user time*.**

 **cumulative user time**

The time command

Type in your terminal:

```
$ time ./executable
```

You will get something like this:

```
$ time ./executable
real        0m0.143s
user        0m0.001s
sys         0m0.010s
$
```

The `time` command gives three timing measurements:

- **real**: "wall-clock time"
- **user**: time spent in user code, i.e. what you've written
- **sys**: time spent in system calls, e.g. malloc, printf, etc

The time command

Unreliable if runtime is very short.
$\longrightarrow$ *Make sure to run your code long enough!*

user + sys will tell you how much actual CPU time your process used. Can potentially exceed the wall clock time in multi-threaded programs.

Time measuring

```
int gettimeofday(struct timeval *tv, struct timezone
    *tz);
```

gives the number of seconds and microseconds since the Epoch (1970-01-01 00:00:00 (UTC))

```
struct timeval {
 time_t      tv_sec;   /* seconds */
 suseconds_t tv_usec;  /* microseconds */
};
struct timezone {
 int tz_minuteswest; /* minutes west of Greenwich */
 int tz_dsttime;     /* type of DST correction */
};
```

Time measuring gettimeofday

In your code:

Do not forget to `#include <sys/time.h>`
(You may need to link with `-lrt`)

```
struct timeval t0, t1;
gettimeofday(&t0, 0);
/* your code */
gettimeofday(&t1, 0);

long elapsed_time_usec = (t1.tv_sec-t0.tv_sec)*1e6 + t1.
    tv_usec-t0.tv_usec;
double elapsed_time_sec= (t1.tv_sec-t0.tv_sec) + (t1.
    tv_usec-t0.tv_usec)/1e6;

printf("%ld microsec, %lf sec\n", elapsed_time_usec,
    elapsed_time_sec);
```

Time measuring

```
int clock_gettime(clockid_t clk_id, struct timespec *
    tp);
```

gives the number of seconds and **nanoseconds** since the Epoch (1970-01-01 00:00:00 (UTC))

```
clk_id is CLOCK_REALTIME // we will use this one

struct timespec {
  time_t   tv_sec;         /* seconds */
  long     tv_nsec;        /* nanoseconds */
};
```

Time measuring clock_gettime

In your code:

Do not forget to `#include <time.h>`
(You may need to link with `-lrt`)

```
struct timespec t0, t1;
clock_gettime(CLOCK_REALTIME, &t0);
/* your code */
clock_gettime(CLOCK_REALTIME, &t1);

long elapsed_time_nsec = (t1.tv_sec-t0.tv_sec)*1e9 + t1.
    tv_nsec-t0.tv_nsec;
double elapsed_time_sec = (t1.tv_sec-t0.tv_sec) + (t1.
    tv_nsec-t0.tv_nsec)/1e9;

printf("%ld nano sec, %lf sec\n", elapsed_time_nsec,
    elapsed_time_sec);
```

Guidelines for measuring execution time

- Use a high resolution timer, such as clock_gettime()
- Check the amount of time your program spends in the OS using the time command
- Do not trust too short timings
- Measure several runs to calibrate variability (3 or more)
- Pick a shortest or an average time as a representative
- Outliers can be important!

Example

Example program in the Student Portal:

Documents $\longrightarrow$ Example programs $\longrightarrow$ `timings.c`: measuring **timings** for different parts of a program using gettimeofday()

Complexity

The same problem often can be solved by different algorithms.

Which algorithm to choose?

Computational complexity: how many resources we need in order to solve some problem?

We want to compare algorithms, not computers!

Complexity

Space complexity - memory needed for an algorithm to solve a given problem. We are measuring total allocated memory in some units.

Time complexity - time needed for an algorithm to solve a given problem. Time is measured in some units, for example seconds or minutes, it can be number of cycles.

Complexity

Examples:

- $n! = 1 * 2 * 3 \ldots (n-1)n$: $n-1$ multiplications.
- nested loops: $n \times m$ function calls

```
for(int i = 0; i < n; ++i)
  for(int j = 0; j < m; ++j){ f(); }
```

- matrix multiplication: $2n^2$ storage, $n^2(2n-1) = 2n^3 - n^2$ operations.

The time required to solve each of these problems will depend on a computer and on an implementation. With increasing the problem size n the time will in general increase.

We consider just a **dominant part** of the instruction count. Matrix multiplication requires $\approx 2n^3$ operations.

Complexity

Matrix multiplication of matrices of size n on a computer X takes $30n^3$ microseconds:
$n = 100$, it needs 30 seconds
$n = 200$, it needs 240 seconds - **8 time more!**

On another computer Y multiplication takes $0.3n^3$ microseconds:
$n = 100$, it needs 0.3 seconds
$n = 200$, it needs 2,4 seconds - **8 time more!**

We want to compare algorithms, not computers!

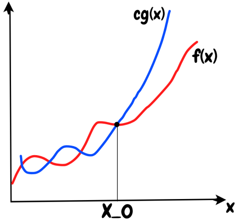
Implementations on similar computer architecture may give different timings up to a constant.

Matrix multiplication requires cn^3 operations, where $c = const$.

Complexity

Introduce function f which gives a feeling about the amount of work required for a given problem size, f is monotonously growing.

Let f and g are functions from $S \subset \mathbb{R}$ to $\mathbb{R}$.
f is not growing faster than g if
$\exists\ x_0 \in S$ and $c > 0$ such that
$\forall\ x > x_0,\ |f(x)| < c|g(x)|$.



We denote such relation as $\mathbf{f} \in \mathbf{O}(\mathbf{g})$ (when $x \to \infty$) — it says that the algorithm has an **order g** complexity.

Complexity

Examples:

$$-2x^3 + 4x^2 + x = O(x^3)$$

$$n/2 = O(n)$$

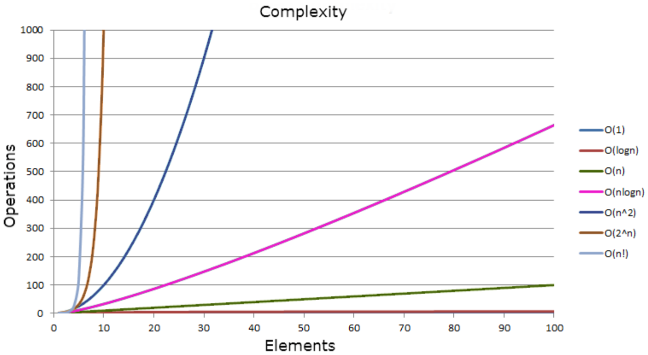
$$\log n + n - 2 = O(n)$$

Complexity

Common time complexities:

- $\mathcal{O}(1)$, i.e. constant $\longleftarrow$ access element in an array
- $\mathcal{O}(\log n)$, i.e. logarithmic $\longleftarrow$ search element in a binary search tree
- polynomial:
  $\mathcal{O}(n)$ linear $\longleftarrow$ compute $n!$
  $\mathcal{O}(n \log n)$ $\longleftarrow$ quick sort
  $\mathcal{O}(n^2)$ $\longleftarrow$ matrix addition
  $\mathcal{O}(n^3)$ $\longleftarrow$ matrix multiplication
- $\mathcal{O}(2^{poly(n)})$, i.e. exponential $\longleftarrow$ Fibonacci numbers $\mathcal{O}(2^n)$
- $\mathcal{O}(n!)$, i.e. factorial $\longleftarrow$ find all permutations of a string

Complexity


Complexity
Operations
Elements
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

n-body problem

Example problem: evolution of the galaxy (N stars) in a selected space region.

Simple approach: each star acts with some force to other N-1 stars. Then the total number of force calculations is $\mathcal{O}(N^2)$. If the number of particles is increased by factor ten the number of calculations increases by a factor of 100...

Use approximation! A tree based approximation scheme reduces the computational complexity of the problem from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$ (The Barnes-Hut algorithm).

The basic idea: if the group is sufficiently far away, we can approximate its gravitational effects by using its center of mass.

That's all