High Performance Programming

Programming in C – part 2

Anastasia Kruchinina

Uppsala University, Sweden

April 18, 2017

Summary of Part 1 - what do we know

- syntax of basic constructions of C programming language

```
for, while, do...while; switch
```

- what is a pointer and how to use it

```
int var = 2;
int *p = &var;
*p = 3; // now var equals 3
```

- how to operate with strings and arrays

```
char filename[] = "data.txt";
double arr[3] = {4.5, 6.4, 2.4};
```

- how to allocate memory dynamically during runtime

```
int *arr = (int *) malloc (5*sizeof(int));
free(arr);
```

Q/A

What is the output of the following program?

```
#include<stdio.h>

void f(int *y)
{  int x = *y;
    x = 3;}
int main()
{
 int x = 5;
 f(&x);
 printf("%d", x);
 return 0;
}
```

Output: 5

Q/A

Choose the correct statement that is a combination of these two statements:

Statement 1: `char *p;`

Statement 2: `p = (char*) malloc(100);`

A - `char p = *malloc(100);`

B - `char *p = (char*)malloc(100);`

C - `char *p = (char) malloc(100);`

D - None of the above

Answer: B

Q/A

What does the "arr" indicate?

```
char* arr[30];
```

Answer: arr is an array of 30 character pointers

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

Test

"Arrays are always laid out *contiguously* in memory."

For those who don't trust me, try this:

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

for(int i = 0; i < 10; ++i)
    printf("%d ", *(&arr[0][0] + i));
```

Output: 0 1 2 3 0 5 6 7 8 9

Static vs dynamic memory

Static memory allocation: memory is allocated by the compiler.
Dynamic memory allocation: memory is allocated at the time of run time on *heap*.
You would use DMA if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.

Static allocation:

```
int arr[10];
```

Dynamic allocation

```
int *ptr;
ptr=(int *)malloc(sizeof(int)*10);
```

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));
```

Note: explicit cast from `void` $*$ before `malloc` call can be omitted in C, but in C++ it will result in error

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));
```

Note: explicit cast from `void` $*$ before `malloc` call can be omitted in C, but in C++ it will result in error

Deallocate memory:

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

Extra info: How does `free` knows how much memory to free? It looks at the extra information saved by malloc.

Dynamic memory allocation of a matrix

Compare:

```
int arr[2][3]; // size is known at the compile time
int **arr;  // allocate memory using malloc or calloc
int* arr[3];
```

Dynamic memory allocation of a matrix

Compare:

```
void f(int **p){}
void g(int *p[]){}
void h(int p[2][3]){}

int main(){
int **a;
allocate_mem(&a); // allocate memory for a
f(a);  // OK!
g(a);  // OK!
// h(a); // NOT OK

int b[2][3];
// f(b); // NOT OK
// g(b); // NOT OK
h(b);  // OK!

return 0;
}
```

Q/A

Which function declaration can be used in the main function?

```
(a) void f(int **b);
(b) void f(int b[3][]);
(c) void f(int b[][5]);
```

```
int main()
{
        int a[3][5] = {{1, 2, 3},
                      {4, 5 ,6},
                      {6, 7, 8, 9, 10}};
        f(a);
        return 0;

}
```

Answer: (c)

Pointers to functions

Declaration of a pointer to a function:

```
<function return type>(*<Pointer_name>)(function
    argument list)
```

For example

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

```c
#include <stdio.h>
#include <stdlib.h>
double add(double a, double b)
{return a+b;}
double sub(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
 if(n%2) funptr = &add; // assign address of a function
 else funptr = &sub;
 print_output(3, 4, funptr);
 return 0;
}
```

Note: you can omit symbol & in `&add` and `&sub`, and * in `(*funptr)(a, b)`

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

Time measuring

Type in your terminal:

```
$ time ./executable
```

You will get something like this:

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

real refers to actual elapsed time.
user refers to the CPU time used *by this process* spent in user-mode code (calls from your C code).
sys refers to the CPU time used *by this process* spent in kernel-mode code (ex. I/O, memory allocation).

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 timings unless they are at least 100 times longer than the CPU time resolution
- Measure several runs to gauge variability (3 or more)
- Pick the shortest time as a representative or an average
- Outliers can be important!

Complexity

The same problem often can be solved by different algorithms.

Which algorithm to choose?
The best, of course.

But by which criterion?

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

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. 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 complexity 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 growing function.
Consider asymptotic behavior of algorithms.

> *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** time complexity.

Complexity

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** time complexity.

Examples:
$-2x^3 + 4x^2 + x = O(x^3)$
$n/2 = O(n)$
$\log n + n - 2 = O(n)$

Complexity

Common time complexities:

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

Q/A

For a given problem size n two algorithms require $2n^7$ and $n(n^6 + \log n)$ instructions respectively. How do their complexities relate?

Answer: both are $\mathcal{O}(n^7)$

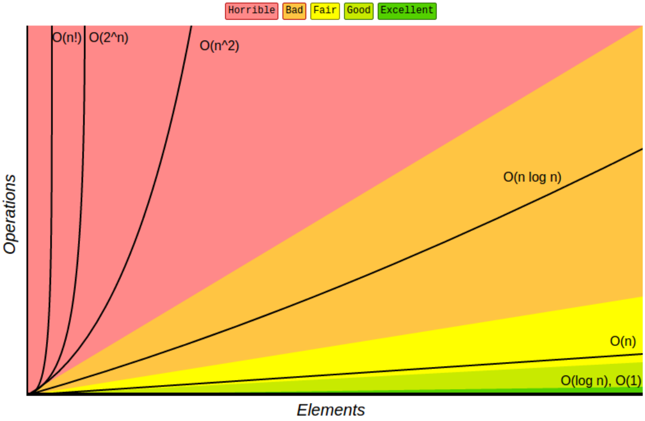
Q/A

What is the time complexity of the function f?

```
void f(int n)
{
int i, j;
for (i=0; i<log(n); i++)
  for (j=0; j<n; j++)
      printf("hello");
}
```

Answer: $\mathcal{O}(n \log n)$

Complexity



Picture: http://bigocheatsheet.com/

Computing x^n

Task: compute $y = x^n$ where x is a real number of $n > 0$ is an integer.

Solution:

```
double pow_n(double x, int n){
        double y = 1;
        for(int i = 0; i < n; ++i)
          y *= x;
}
```

Space complexity? Time complexity?

Computing x^n

Task: compute $y = x^n$ where x is a real number of $n > 0$ is an integer.

Solution:

```
double pow_n(double x, int n){
        double y = 1;
        for(int i = 0; i < n; ++i)
          y *= x;
}
```

Space complexity? Time complexity?
O(1) **O(n)**

Can we do better?

Computing x^n

Task: compute $y = x^n$ where x is a real number of $n > 0$ is an integer.

Second solution:

```
double pow_n(double x, int n){
        if(n == 0) return 1;
        double temp = pow_n(x, n/2);
        double y = temp*temp;
        if(n%2) y *= x;
}
```

Computing x^n

Task: compute $y = x^n$ where x is a real number of $n > 0$ is an integer.

Second solution:

```
double pow_n(double x, int n){
        if(n == 0) return 1;
        double temp = pow_n(x, n/2);
        double y = temp*temp;
        if(n%2) y *= x;
}
```

Space complexity? $\mathbf{O}(\log \mathbf{n})$ (we save data on the stack for each recursive call, $\mathbf{O}(\mathbf{1})$ can be achieved using iterative algorithm)
Time complexity? $\mathbf{O}(\log \mathbf{n})$

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)

(Space complexity ignores the space used by the input to the algorithm)

Picture: http://bigocheatsheet.com/

Summary

What do you know after the lecture:

- how to use multidimensional arrays
- how to measure time in C
- what is time and space complexity

Lab 3 is available on the Studentportalen.
Assignment 2 is available on the Studentportalen.

Extra example

Compare:

```
void allocate_mem(int** arr, int n, int m)
{
arr = (int**)malloc(n*sizeof(int*));
for(int i=0; i<n; i++)
arr[i] = (int*)malloc(m*sizeof(int));
}
```

```
void allocate_mem(int*** arr, int n, int m)
{
*arr = (int**)malloc(n*sizeof(int*));
for(int i=0; i<n; i++)
(*arr)[i] = (int*)malloc(m*sizeof(int));
}
```