Sparse Topics

Luca Tornatore - I.N.A.F.

"Foundation of HPC - basic" course

DATA SCIENCE &
SCIENTIFIC COMPUTING

2022-2023 @ Università di Trieste

Outline

Sparse and different topics.
Either concepts and notions that were preparatory for the course or on-the-spot in-depth details that were aftermath of Q&A

Table of contents

Expressing performance

Expressing performance

In these slides we introduce a metrics to estimate the performance of a code in exploiting some of the CPU's resources.

Specifically, we will focus on *(i)* how "fast" a code is and *(ii)* how well a code exploits the instruction-level parallelism capability of a CPU.

The metrics we will introduce are best-suited for those code sections that performs loops, which indeed are a large and significant fraction of scientific codes in general.

The "performance" intended differently, for instance as energy-to-solution or memory imprint, needs some different metrics and in some cases some dedicated measures.

Expressing performance

As you know, a CPU does not work “continuosly”. At the opposite, its activity is regulated by an internal clock whose pace is of the order of billions ticks per seconds.
The typical CPU frequency is between 2 and 4 GHz, meaning that the time taken by a “clock cycle” is of the order of 0.5 - 0.35 *ns*.
We will refer to this time span as “a cycle”.

Moreover, on the purpose if energy saving the CPUs have throttling capabilities, meaning that the clock frequency is not fixed but may be adapted to the worload. The larger the workload, the higher the clock and vice-versa.

Expressing performance

Due to the variability of the CPU's clock, measuring the "wall-clock time-to-solution" is not always the best, or at least the only, metrics to be collected for the purpose of evaluating how some code snippet actually behave.

Quantifying the number of clock cycles spent on a code section is it may be even more informative than knowing how many seconds it recquired to execute.

Focusing on code sections that repeat a block of instructions over an array[*], which are very common and often represent the most computationally intense hotspots, that easily translates in a *cycles-per-element* (CPE) metrics.

Expressing performance

In fact, we would be interested in knowing how much efficient our code is *per-element* rather than *per-iteration* since our implementation may be able to process more elements per iteration.

Conversely, in the case we wanted to estimate how well we are exploiting the super-scalar capability of the CPU, assessing how many i*nstructions-per-cycle* (IPC) are executed would be the adecquate metrics to look at.

We will see in the next lectures how to collect these sophisticated metrics in practice. As for now, the focus is on clarifying how the performance must be expressed and measured.

Expressing performance

Still, measuring the "execution time" of a code is a fundamental metrics that we should gather, at least for a first assessment.

Basically, you have access to 3 different types of "time".

1. "wall-clock" time
   Basically the same time you can get from the wall-clock in this room.
   It is a measure of the "absolute time".
   In POSIX systems, it is the amount of the number of seconds elapsed since the start of the Unix epoch at 1 January 1970 00:00:00 UT.

2. "system-time"
   The amount of time that the whole system spent executing your code.
   It may include I/O, system calls, etc.

3. "process user-time"
   The amount of time spent by CPU executing your code's instructions, strictly speaking.

Expressing per

Still, measuring the "execution time" of a code is a fundamental metrics that we should gather, at least for a first assessment.

Basically, you have access

1. "wall-clock" time
   Basically the same tim
   It is a measure of the
   In POSIX systems, it i
   since the start of the
2. "system-time"
   The amount of time t
   It may include I/O, sy
3. "process user-time"
   The amount of time s
   strictly speaking.

POSIX systems

POSIX is an ensemble of IEEE standards meant to define a standard environment and a standard API for applications, as well as the applications' expected behaviour.

It enlarges the C API, for instance, the CLI utilities, the shell language and many things.

All ⋆NiX systems are POSIX systems. Other compliant systems are

- AIX *(IBM)*
- OSX *(Apple)*
- HP-UX *(HP)*
- Solaris *(Oracle)*

Expressing performance

You can measure all the quoted times :

- *Outside* your code
  → you measure the whole code execution
  you ask the OS to measure the time your code took to execute time:
  - using the `time` command (see `man time`)
  - using `perf` profiler
  - .. discover other ways on your system

- *Inside* your code
  → you can measure separate code's section
  you access system functions to access system's counter

- ☐ What time do we need ?
  Real, User, System, …
- ☐ What precision do we need ?
  1s, 1ms, 1us, 1ns
- ☐ What wrap-around time do we need ?
- ☐ Do we need a *monotonic* clock ?
- ☐ Do we need a *portable* function call ?

Expressing performance

Baseline: you call the correct system function right before and after the code snippet you're interested in, and calculate the difference (yes, you're including the time function's overhead).

- **`gettimeofday(..)`** returns the wall-clock time with μs precision
  Data are given in a `timeval` structure:

```
struct timeval {time_t       tv_sec;       // seconds
                useconds_t   tv_usec; };   // microseconds
```

- **`clock_t clock()`** returns the **user-time + system-time** with μs precision.
  Results must be divided by `CLOCKS_PER_SEC`

- **`int clock_gettime(clockid_t clk_id, struct timespec ..)`**

  `CLOCK_REAL_TIME` system-wide realtime clock;
  `CLOCK_MONOTONIC` monotonic time
  `CLOCK_PROCCES_CPUTIME` High-resolution **per-process** timing
  `CLOCK_THREAD_CPUTIME_ID` high-precision **per-thread** timing
  Resolution is **1 ns**

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

Expressing performance

Baseline: you call the correct system function right before and after the code snippet you're interested in, and calculate the difference (yes, you're including the time function's overhead).

- **`int getrusage(int who, struct rusage *usage)`**

  `RUSAGE_SELF` process + all threads
  `RUSAGE_CHILDREN` all the children hierarchy
  `RUSAGE_THREAD` calling thread

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

Expressing performance

A possibility on a POSIX system is:

```
#define CPU_TIME (clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &ts ),\
                              (double)ts.tv_sec +        \
                              (double)ts.tv_nsec * 1e-9)

....

Tstart = CPU_TIME ;

// your code segment here

Time = CPU_TIME - Tstart;
```

Accumulate data on performance

Independently of what metrics you are accumulating, dealing with only 1 measure is not a good estimate: computer systems are really complicate ones and lots of things are going on continuosly, above all on modern architectures, and you may observe significant variations in your metrics from a run to another run.

As such, you must procede "*statistically*", i.e. by accumulating several measure and modelling the measure and system overhead.

For instance, acquiring the cycles' number or the time requires itself a number of cycles; a loop as an amount of inherent overhead. And so on.

Quite often, averaging over a "sufficient" number of runs and subtracting the known overhead is sufficient to get an good-enough estimate.

Accumulate data on performance

examples in pseudo-language:

```
double t_overhead;
timing_overhead = get_time();
for ( int i = 0; i < LOTS_OF_ITER; i++ )
   double this_time = get_time();
t_overhead = (get_time() - t_overhead) /
                LOTS_OF_ITER;
```

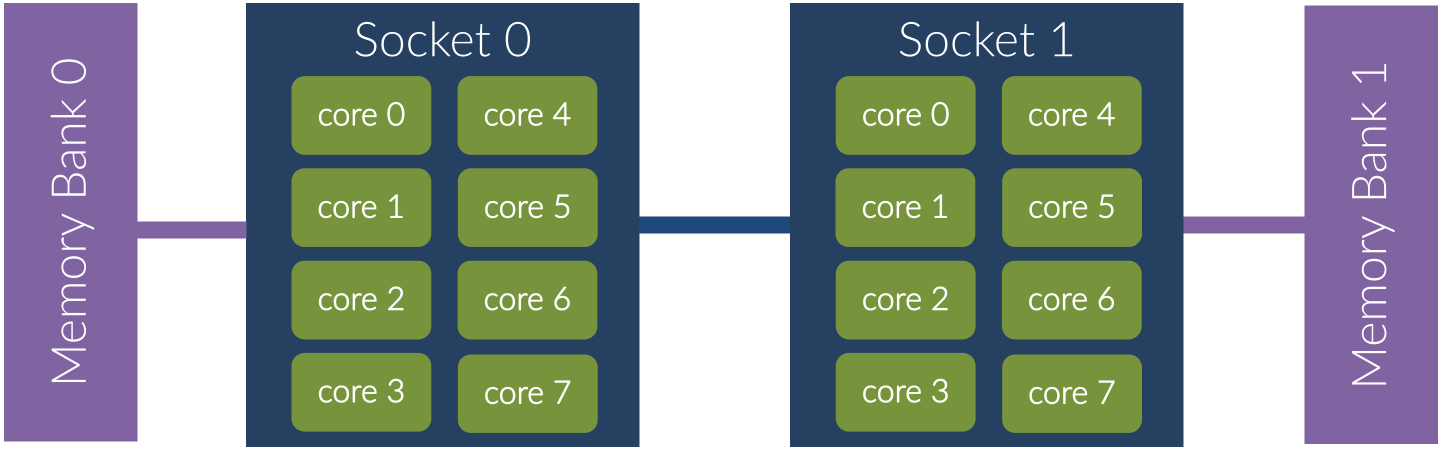
```
double timing = get_time();
  ...
  block of code you want to characterize
  ...
timing = get_time() - timing - t_overhead;
```

```
double timing = 0;
double stddev;
for( int i = 0; i < MANY_ITER; i++ )
  {
     ...
     double time0 = get_time();
     block_to_be_measured
     double time1 = get_time();
     double elapsed = time1-time0;
     timing += elapsed - t_overhead;
     stddev += elapsed * elapsed;
     ...
}

timing = timing/MANY_ITER;
stddev = sqrt( stddev/MANY_ITER - timing*timing);
```

Accumulate data on performance

Inside a socket there are many cores, from 4-8 in commodity CPUs found on laptops or desktop computer up to ~64 in high-end CPUs for servers and computational nodes



The O.S. can *migrate* you code’s threads from one core to another and also from one socket to another. Whether the data are also migrated depends on the adopted policy.

Accumulate data on performance

Then, running a program without *binding* it to a specific core and a specific memory banck may result in a non-optimal behviour.

So, when you are interested in profiling a code, you must be sure that it will not be migrated.
You can ask that to the O.S. by using

```
taskset
numactl
```

just have a look to the related man pages for all the details

Accumulate data on performance

`numactl -H`	exposes the topology of the node
`numactl --cpunodebin=`*n* `numactl -C` *n*	bind the execution to core *n*
`numactl --membind=`*n* `numactl -m` *n*	bind the memory to memory bank associated with core *n*
`numactl -C +`*list*	bind the execution to the *relative* cores listed in *list*. *example:* `numactl -C +0,2,4 prog.x` *will execute* `prog.x` *on the cores 0,2 and 4 of the cpuset given to the job.*

THE
C
PROGRAMMING
LANGUAGE

Some sparse topic on C

Outline

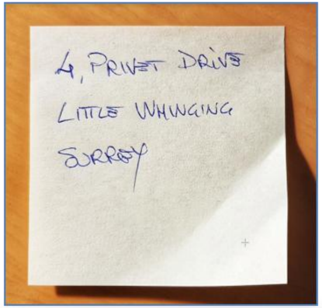
1. Pointers

Pointers: link to reality

As first, let's start with a very simple concept.
I guess you have a special physical place, however you love to imagine it, that you call "home".

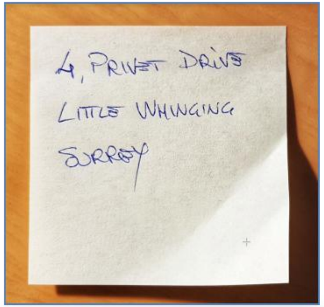
Let's suppose you are boring normal people, and that your place has an *address:*

4, Privet Drive, Little Whinging, Surrey



You appreciate the fact that this address needs some memory storage to be kept; in my case, a simple sticky note.

Pointers : link to reality

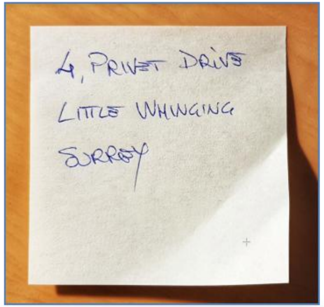


4, Privet Drive, Little Whinging, Surrey

You appreciate the fact that this address needs some memory storage to be kept; in my case, as we said, a simple stick note.

You also appreciate that there is a clear difference between the string written on the note and your actual home
(try to inhabit my stick note if don't sense that difference).

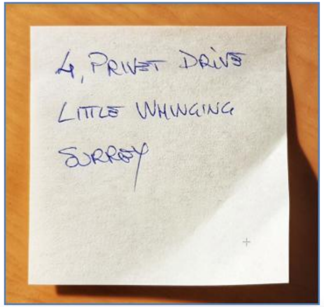
Pointers : link to reality



So, my note *points* to your home, and occupy some well-defined physical space for the purpose (the sticky note sheet), but it is *not* your home whose physical occupancy does not depend on my note (it's hard to know from the note whether it's a castle or a roulotte).

Conversely, your home is somewhere else, in a well-defined place that is reachable - let us say addressed - by using my note.

Pointers : reality contents changes, ptr doesn't



If you change something in your house, the address on my note is still valid and can be useful to reach you.

Nobody noticed that you renewed your bathroom.

Pointers : change to link to a different point

12, Grimmaud Place, London

If you move, and your home has now a different address, I need to know it to get there and invite myself for dinner.
To save your new address, I can still use the same sticky note sheet, i.e. the same physical storage of the same size.

The physical location has changed, but I can still use the same sticky note to reach you.

Pointers

All in all, then:

- a *pointer* is a variable, i.e. a memory location of fixed size (8B in 64bits systems) which contains an *address*, specifically a memory address and not your place's one.
That address is the *starting point* of a memory area.
So, a pointer can point to an integer (4B), a double (8B) an array of 10G items. Whatever stays in memory has some location where it is stored and that location can be pointed to by a pointer variable.

- ***de-referenceing a pointer*** means to get to the pointer variable, i.e. at the memory location that the variable occupies, to read those bytes acquiring the address and then to get to that memory address

Thinking about memory

The "**memory**" is nothing else but a long **1D string of bytes**.

You can uniquely identify every byte in your memory by its distance from the "byte 0".

That distance is every byte's "**address**".

Thinking about memory

In most languages there are basic types with a well defined size, i.e. a length in bytes:

`char`	`1 byte`
`short integer`	`2 bytes`
`(long) integer`	`4 bytes`
`long long integer`	`8 bytes`
`floating-point single precision`	`4 bytes`
`floating-point double precision`	`8 bytes`
`floating-point ext. precision`	`10 bytes`

Please refer to the paper "What every computer scientist should know about floating-poin" that you find among the materials. It is ***very important*** *that you understand sharply the IEEE floating-point representation.*

Thinking about memory

In most languages there are basic types with a well defined size, i.e. a length in bytes

single-byte:
8 bits [0-255]
printable chars

4 bytes
32 bits

```
integer: 0 -> 2^32-1
u int  :-2^31 -> 2^31-1
fp: -3.40282347E+38 -> -1.17549435E-38 , 0 ,
    1.17549435E-38 -> 3.40282347E+38
```

8 bytes
64 bits

```
integer: 0 -> 2^64-1
u int  :-2^63 -> 2^63-1
fp: -1.797693134862315E+308 -> -2.225073858507201E-308,
0, 2.225073858507201E-308 -> -1.797693134862315E+308
```

Pointers in practice

- a *pointer* is declared as

```
type *ptr_variable_name;
```

examples:

`char *c;`	points to a char
`double *d;`	points to a double
`struct who_knows *w;`	points to a struct `who_knows`

you assign a value to a pointer variable by assignment:

c = 0x123456; c = &my_preferred_letter;

you read the address it points to by de-referencing:

*c is actually the content of the byte pointed by c, not the c's value

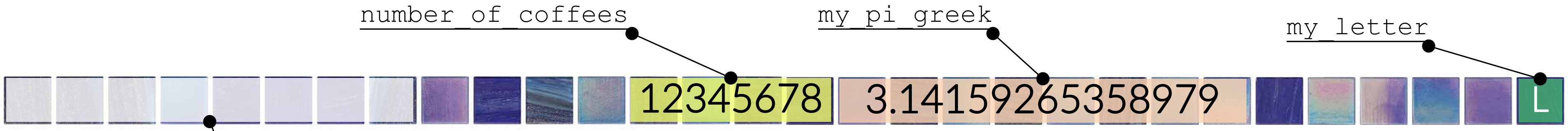
note that &c is the c's own address.

Pointers in practice

We are using this 8-bytes memory region to store the value of our pointer **`void ptr`**

We have populated our memory using 1-byte, 4-bytes and 8-bytes variable and we want to use our pointer to address them

Pointers in practice

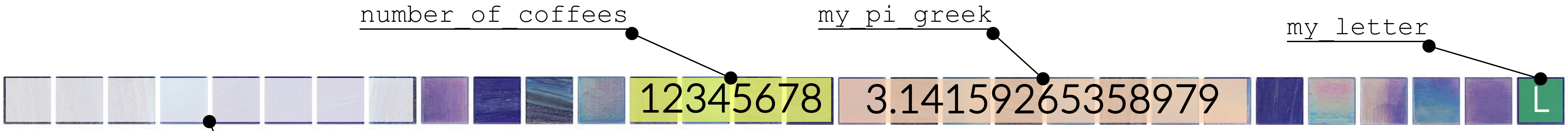


We are using this 8-bytes memory region to store the value of our pointer **`void ptr`**

1) let's point to our integer variable `number_of_coffee`, whose **value** is 12345678
Its **address** is 12, because its first byte is positioned at the 12th position

```
ptr = (void*)&number_of_coffes
 ptr eq 12
*ptr eq 12345678
```

Pointers in practice

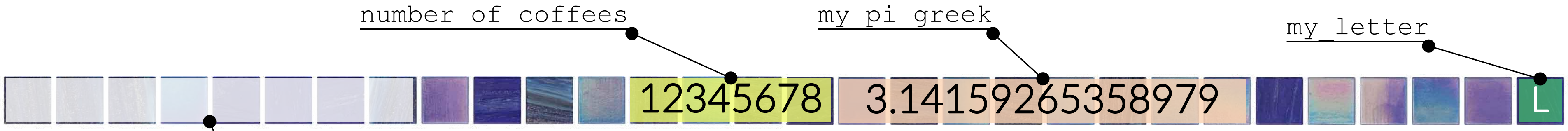


We are using this 8-bytes memory region to store the value of our pointer **`void ptr`**

2) let's point to our double-precision fp variable `my_pi_greek`, whose **value** is pi greek.
Its **address** is 16, because its first byte is positioned at the 16th position

```
ptr = (void*)&my_pi_greek
 ptr eq 16
*ptr eq 3.14159265358979
```

Pointers in practice



We are using this 8-bytes memory region to store the value of our pointer **void ptr**

3) let's point to our single-byte character `my_letter`, whose **value** is 'L'
Its **address** is 29, because its byte is positioned at the 29th position

```
ptr = (void*)&my_letter
 ptr eq 29
*ptr eq 'L'
```

Pointers in practice

Why have I used the **void** type for my pointer while few slides before I said that the pointers are declared as “pointer to a given variable type” ?

```
char   *ptr_to_my_letter;
double *ptr_to_my_pi_greek;
int    *ptr_to_number_of_coffees;
```

There is no “material” difference between

```
char   *ptr_to_my_letter;
double *ptr_to_my_pi_greek;
```

Both of them occupy 8 bytes and their content is a memory address.
The void declaration allows you to use the pointe “type-neutrally”.

So, why to declare a typed pointer?

Because then we have automatic pointer arithmetics.

Pointer arithmetics

Pointer arithmetics is useful when you are not pointing to a single item but to a series of equally-sized algorithms.
Basically, on what we call and **array** (however, the pointer and the array concepts overlap only partially).

Let's have an array of n elements like

```
type array[n];
```

each element has size `sizeof(type)` (i.e. 1, 2, 4, 8, 10 - let us consider on ly basic types) and will have an address:

```
&array[i] = &array[0] + i*sizeof(type)
&array[i] - &array[j] = (i-j)*sizeof(type)
```

Pointer arithmetics

Let's have an array of n elements like

```
type array[n];
```

If we also declare a pointer

```
type *ptr1 = &array[0];
type *ptr2 = &array[n];
```

then:

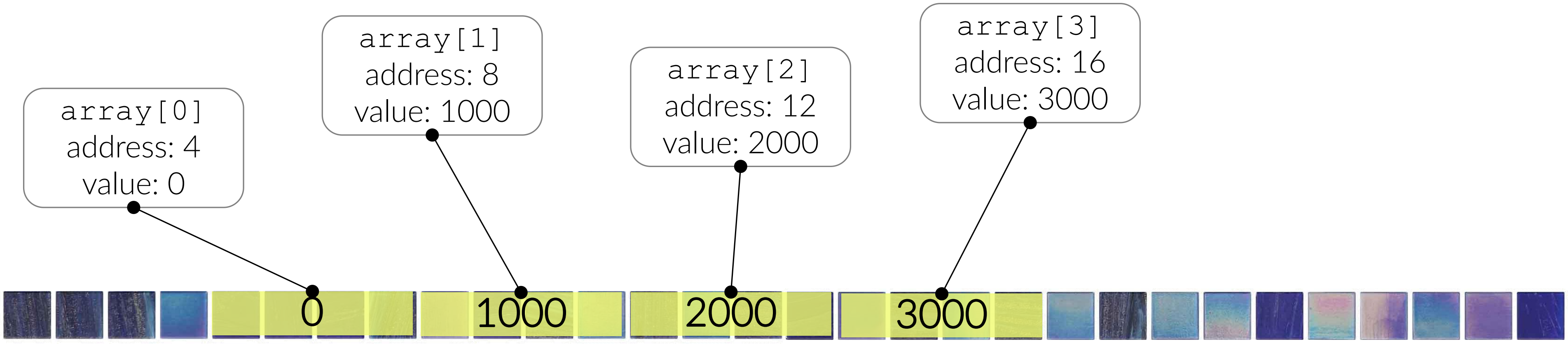
```
ptr1++ eq &array[1];   // ptr1+1 is NOT the address of
                          array[0] plus 1 byte !
ptr1 + i eq &array[i];

ptr2 - ptr1 eq n
```

Pointer arithmetics

let's make an example:

```
int array[4];     // this declares an array of 4 integers,
                     each 4bytes long. The total size is then 16 bytes
```



Allocating memory

Actually, pointers are the way in C you address dynamically-allocated array:

```
double *array = malloc( sizeof(double) * N );
```

you have allocated room for `N` double entries and the location at the beginning of that memory region is stored in `array`.
Hence, you can access the *ith* element both by `*(array + i)` or by `array[i]`.

Since array is typed to double *, the pointer arithmetics comes automatically:

Allocating memory

```
double *array = malloc( sizeof(double) * N );
```

Since `array` is typed to `double *` the pointer arithmetics comes automatically:

`*array` gives you back the double value at the position 0

`*(array+i)` gives you the double at the position `i`:

- `array+i` is interpreted by the compiler as "the address of the *i*th double after the one pointed by the variable array"
  i.e. `array+i` becomes the address array + `i*sizeof(double)`
- `*(array +i)` de-reference that double, so that you can either read or write it

Allocating multi-dimensional arrays

How can you allocate dynamically a multi-dimensional array?

```
double array[ n ][ m ];  // an array of n rows and m column
```

There are basically three ways for that

Allocating multi-dimensional arrays

[1] separate allocation for each row

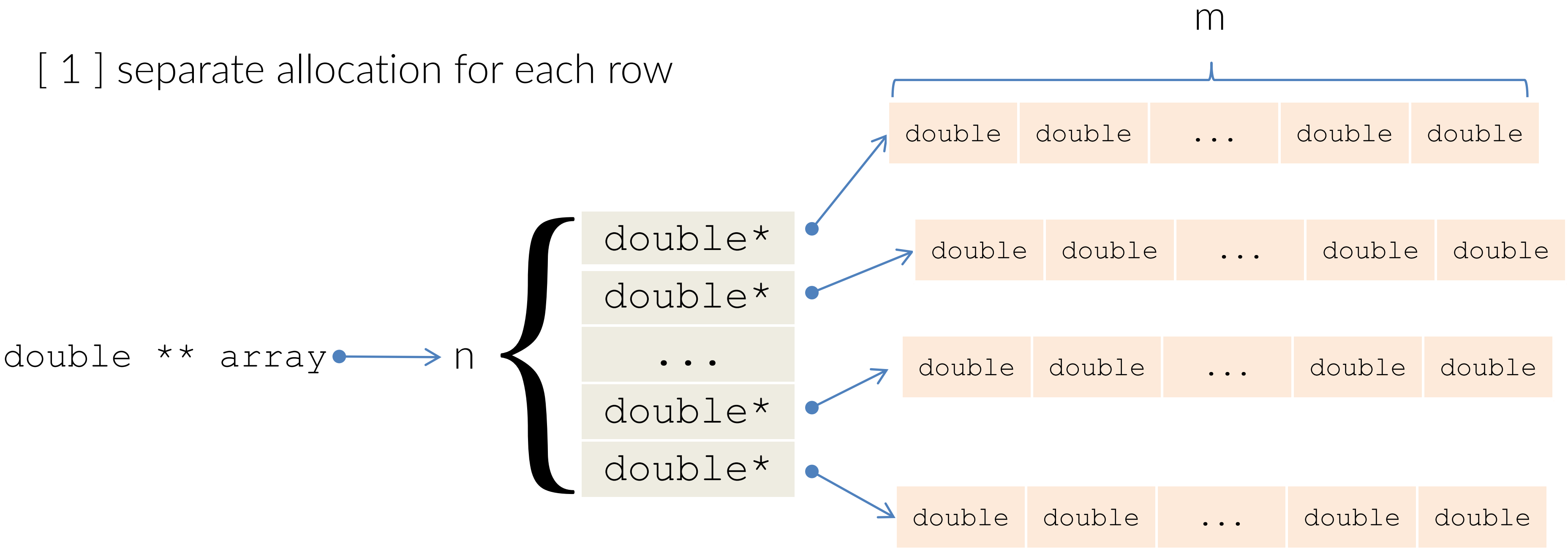
```
// define array as a pointer to double**
double **array;

// allocate n pointers to double*
array = (double**)malloc( n * sizeof(double*) );

// for each pointer, allocate enough memory to retain m doubles
for ( int i = 0; i < n; i++ )
   array[i] = (double*) malloc( m* sizeof(double) );
```

Allocating multi-dimensional arrays

[1] separate allocation for each row



Allocating multi-dimensional arrays

[2] unique allocation + displacement

```
// define array as a pointer to double**
double **array;

// allocate n pointers to double*
array = (double**)malloc( n * sizeof(double*) );

// perform a unique allocation
array[0] = (double*)malloc( n*m * sizeof(double) );

// assign all the pointers by pointer arithmetics
for ( int i = 1; i < n; i++ )
   array[i] = array[i-1] + m;
```

Allocating multi-dimensional arrays

[3] unique 1d allocation, use pointer arithmetics to address [row,col] pairs

```
// define array as a pointer to double*
double *array;

// allocate all the data you need
array = (double*)malloc( n * m * sizeof(double) );

// refer to [i,j] by pointer arithmetics
*(array + i*m + j) ... ;
```

Allocating multi-dimensional arrays

Can you generalize to 3d? 4d? …

Functions returning pointers

What if a function allocates some memory? How can you return that to the caller?

```
int * foo( ... )
 {
     ...;
     int * ptr = (int*)malloc( .. );

     ...;
     return ptr;
}
```

```
void foo( ...; int **ptr; ... )
 {
     ...;
     *ptr = (int*)malloc( .. );

     ...;
     return;
}
```

Pointers to functions

A pointer can point to anything that has an address.
A function *has* an address: it is a well-defined ensemble if instructions, and hence its address is the memory addrress of its first instruction.

```
int (*func)();       // that is a pointer, named func
                     // to a function that returns an int

int (*func)( int, double); // to a function that returns
                           // an int and requires 2 args
```

Pointers zoo

what this strange creature is ?

```
char **monster;
```

Pointers zoo

what this strange creature is ?

```
char **monster;
```

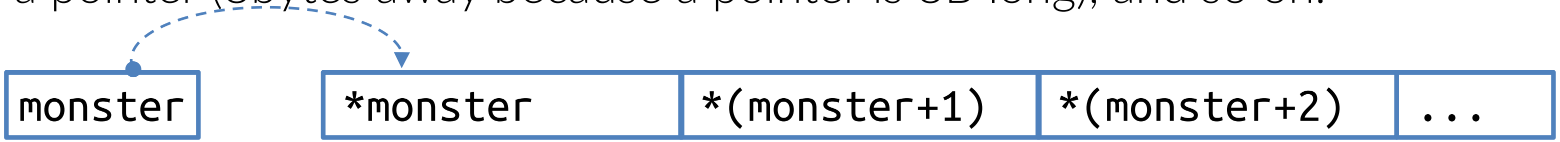
Let's reason by steps, from right to left (as you should read a C declaration)

1) `monster` → we declare a variable
2) `*monster` → it is a pointer
3) `**monster` → it points to a pointer
   (remind, a pointer is just a variable, and as such it can be pointed to)
4) `char **monster` → the pointed pointer points to a `char`

so `*monster` is a pointer which points to a `char`.
Good to know. But what does it mean?

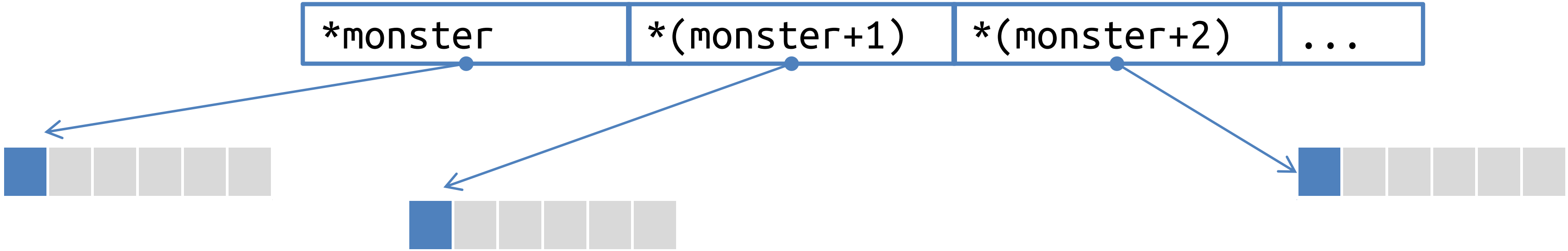
Pointers zoo

Then, it happens that since **`monster`** points to a pointer, also **`monster+1`** points to a pointer (8bytes away because a pointer is 8B long), and so on:



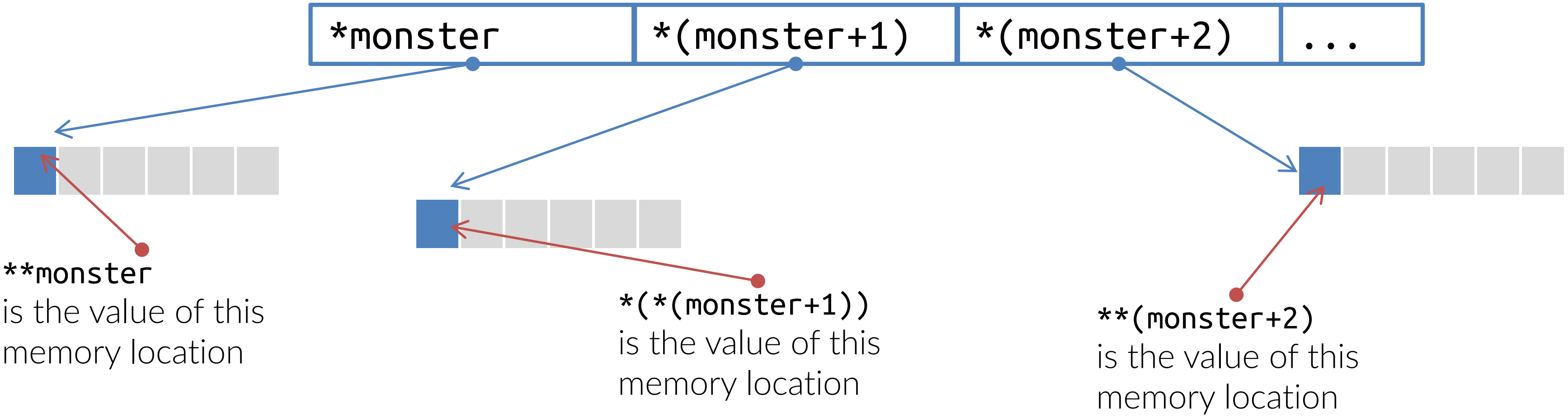
note that *monster+n is *very* different than *(monster+n)

`*(monster+n)` are all interpretable (formally they are since we are referencing them through **`monster`**) as pointers to **`char`**:



Pointers zoo

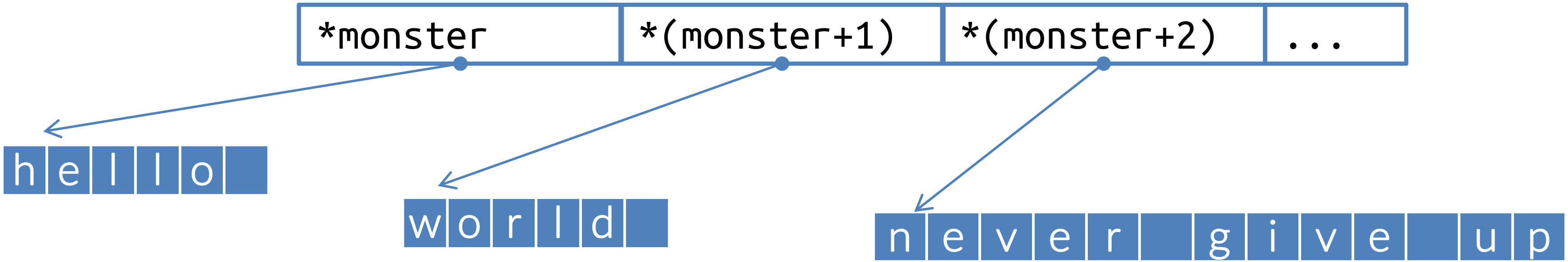
`**(monster+n)` are the value at the byte pointed by `*(monster+n)`



`*(*(monster+n)+j)` is the *jth* byte after `*(monster+n)` which actually starts to look like a string..

Pointers zoo

`**(monster+n)` are the value at the byte pointed by `*(monster+n)`



then

`**monster` = ‘h’, `*((*monster)+1)` =’e’, `*((*monster)+2)` =’l’, …

`**(monster+1)` = ‘w’, `*(*(monster+1)+1)` =’o’, `*(*(monster+1)+2)`=’r’, …

or, in other words,

`*monster` = “hello”, `*(monster+1)` = “world”, `*(monster+2)` = “never give up”

Pointers zoo

That is actually how you access the command-line's arguments, for instance:

```
int main (int argc, char **argv)
  {
     ... let's see the worked example, arguments.c
  }
```