

# Parallel programming for GPUs

*Modern computing for physics*

---

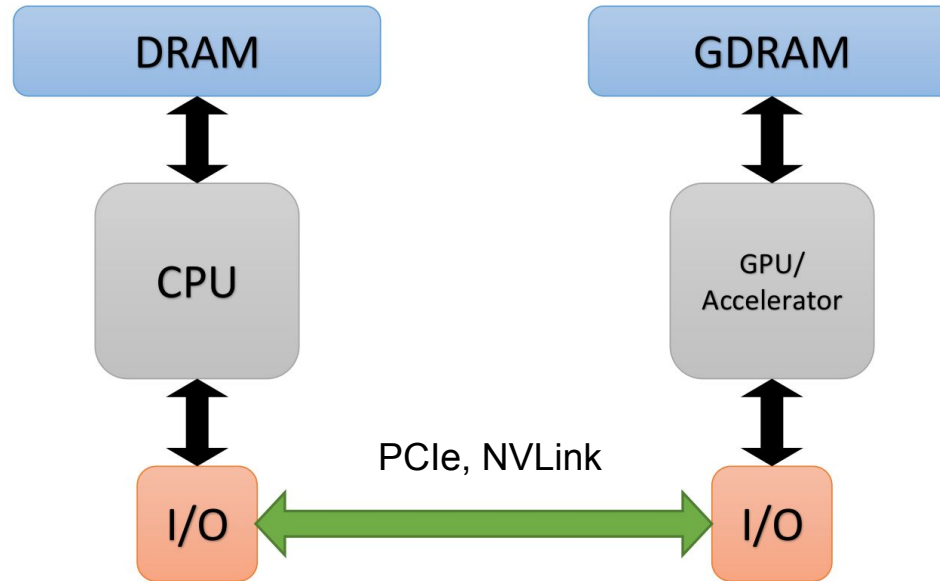
**J.Pazzini**  
*Padova University*

Physics of Data  
AA 2024-2025



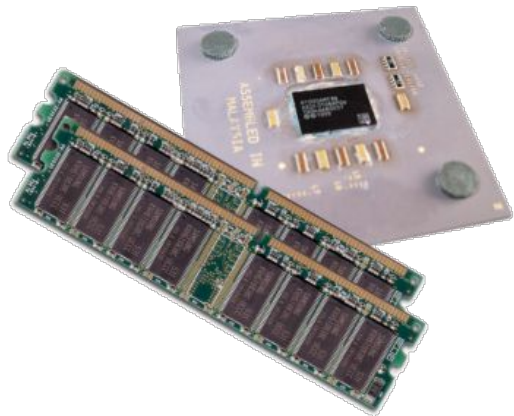
UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

# Heterogeneous computing



# Heterogeneous computing

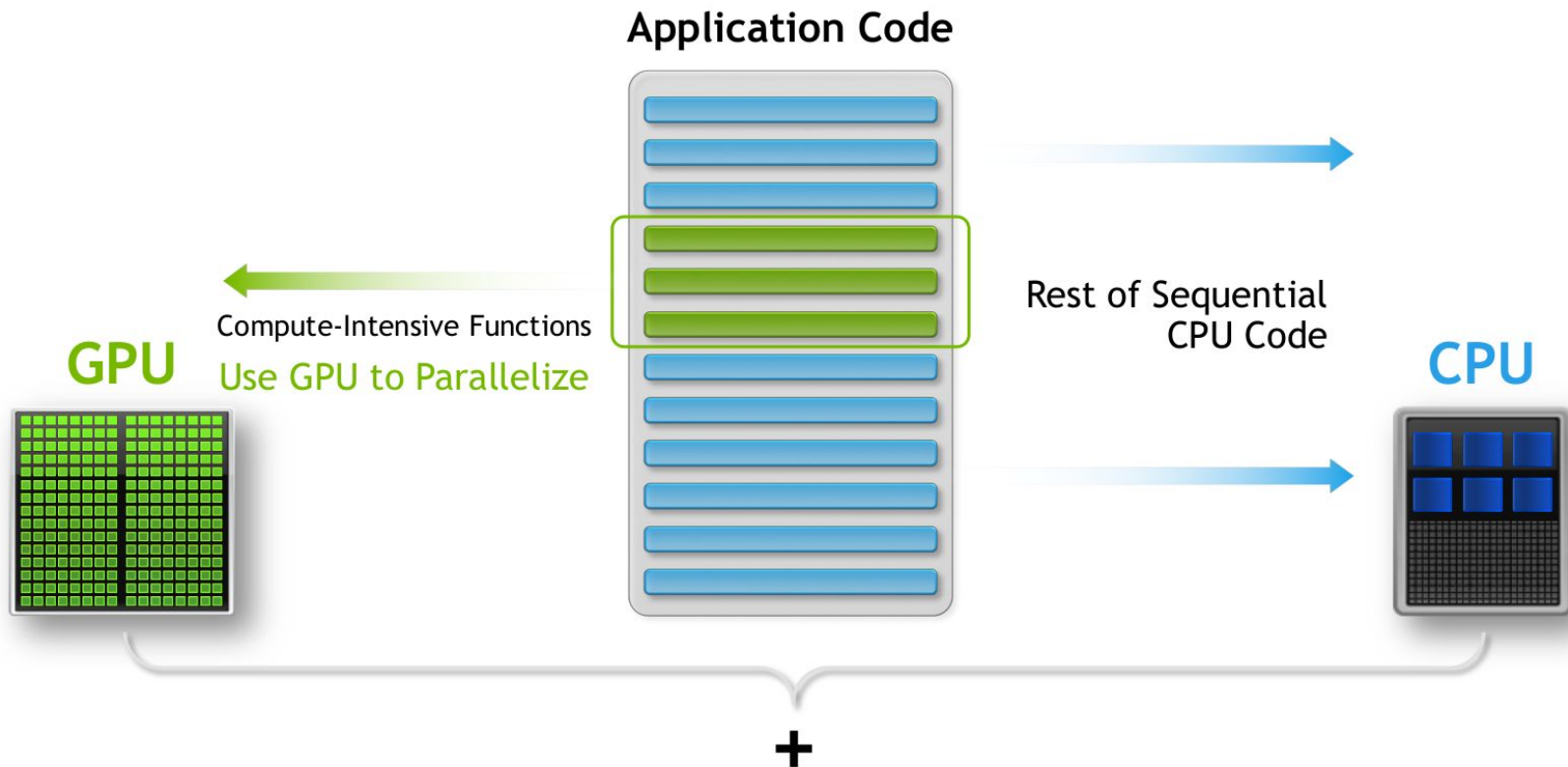
**HOST** → The CPU and its memory  
(host memory)



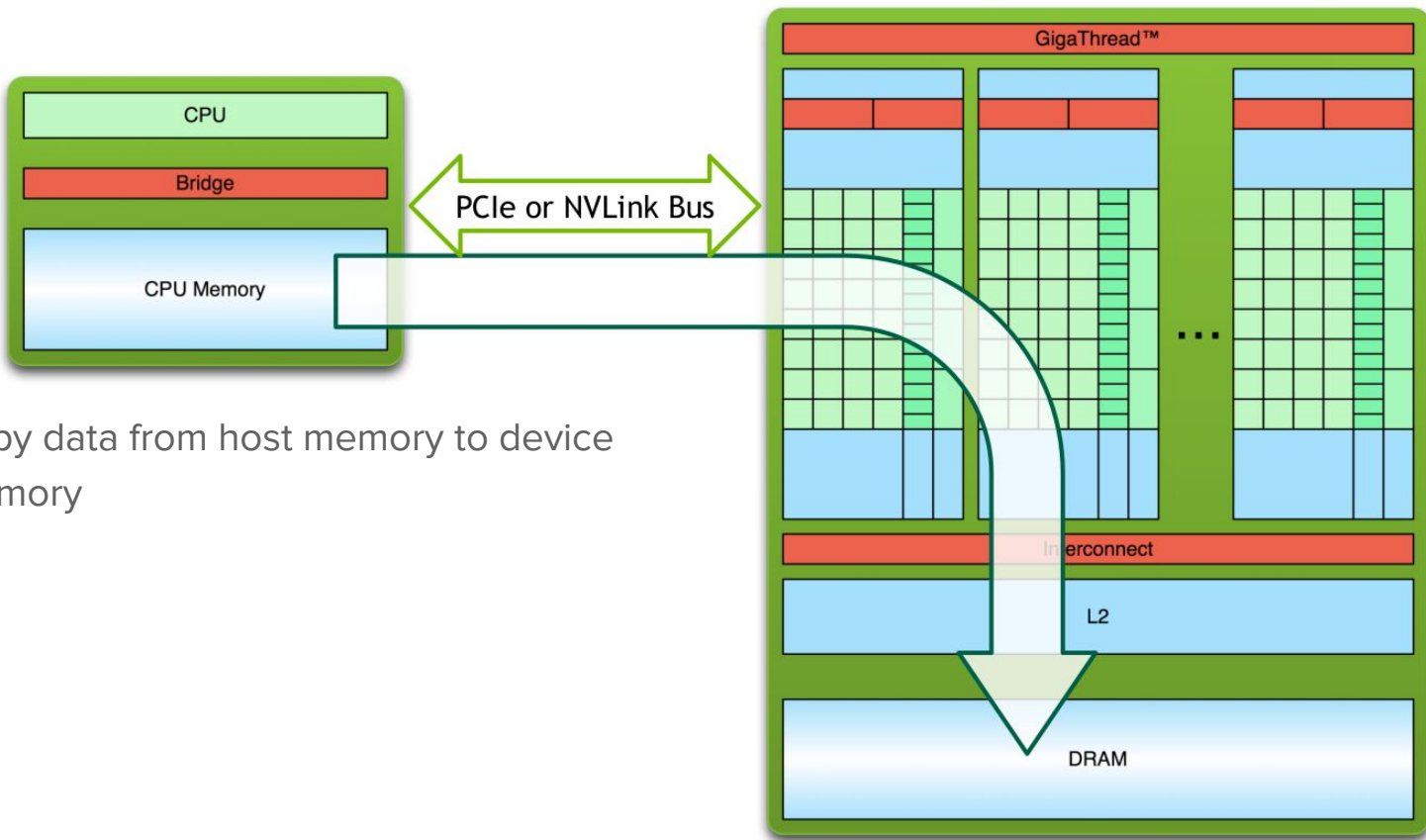
**DEVICE** → The GPU and its memory  
(device memory)



# The general idea

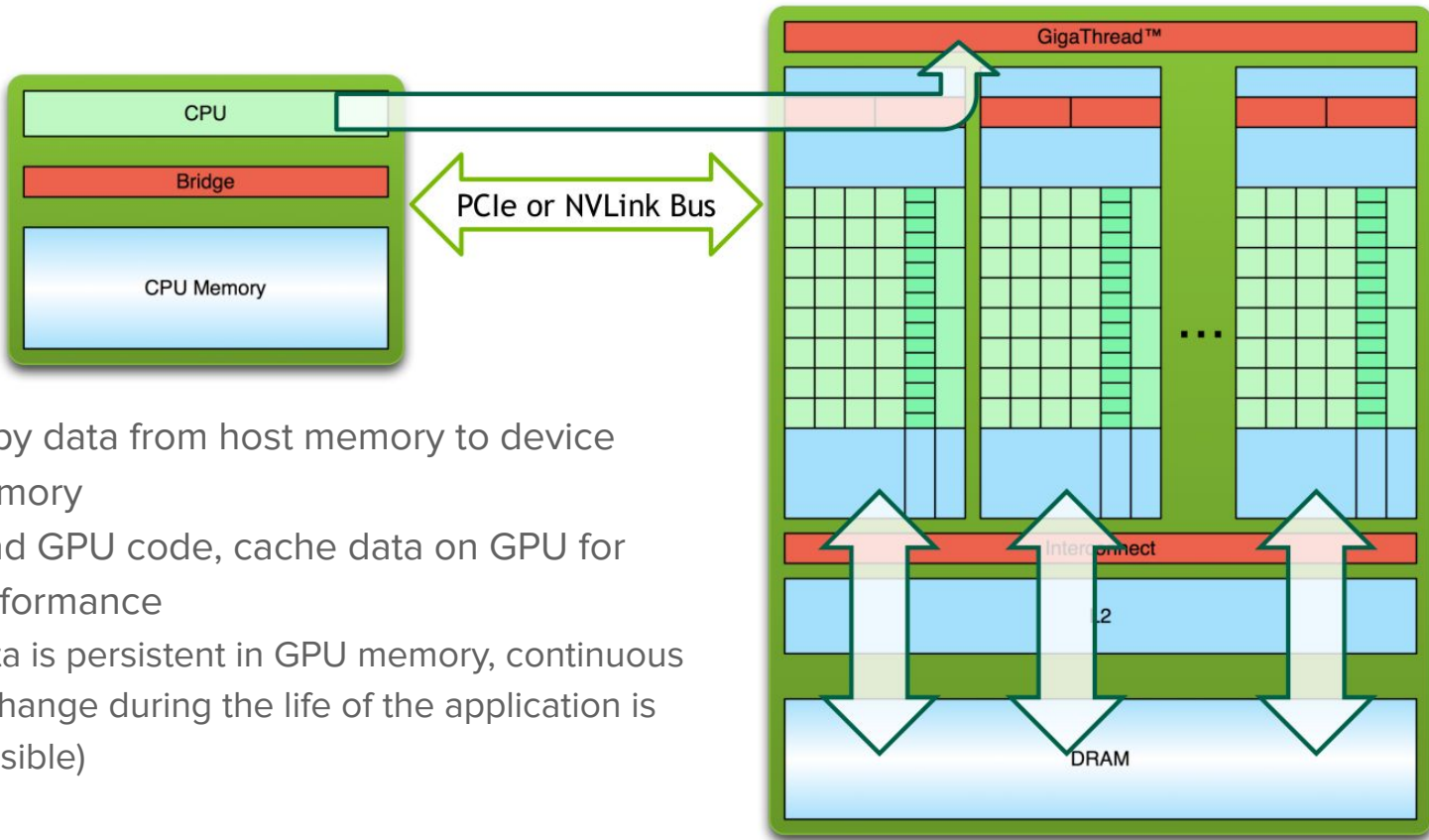


# The general idea



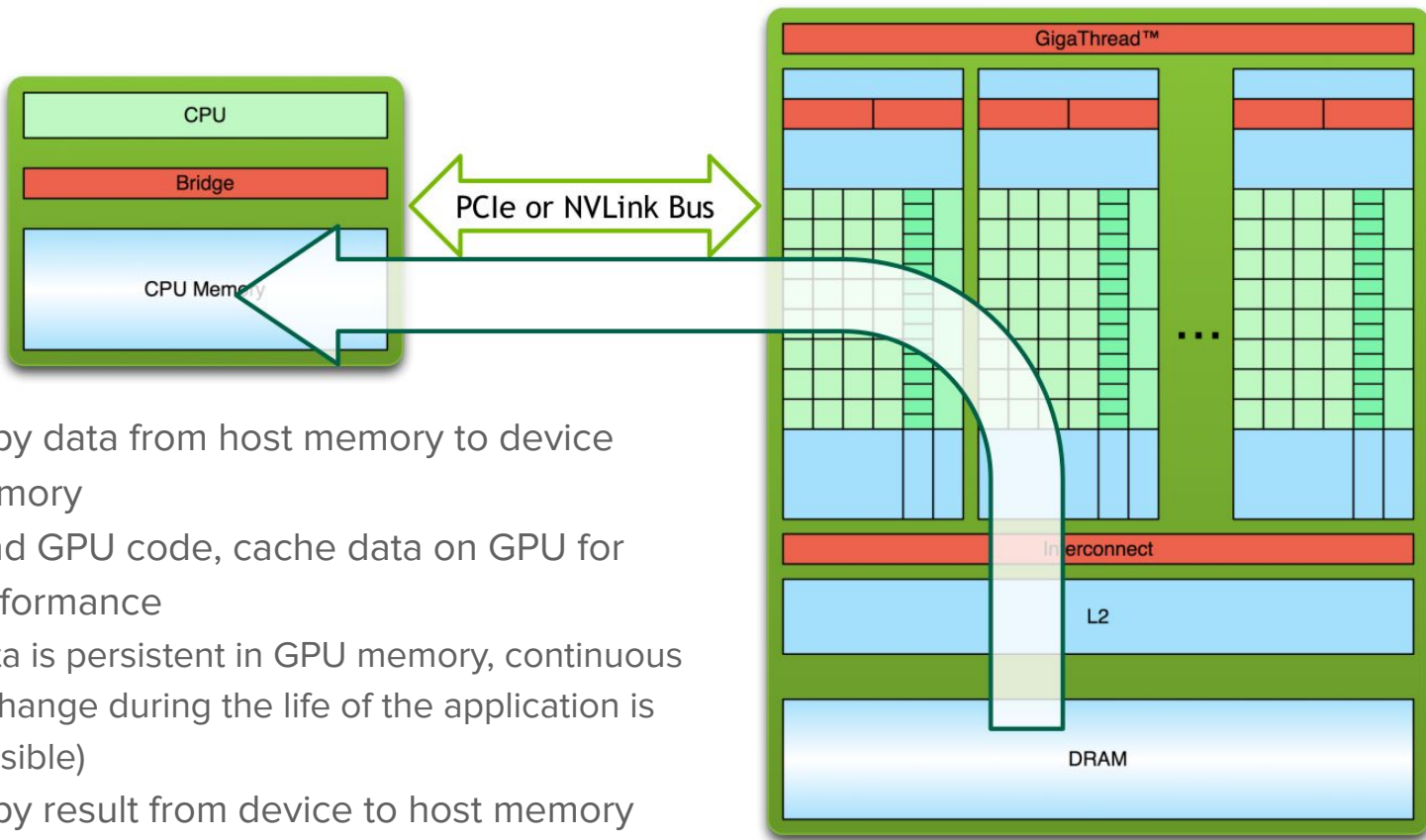
1. Copy data from host memory to device memory

# The general idea



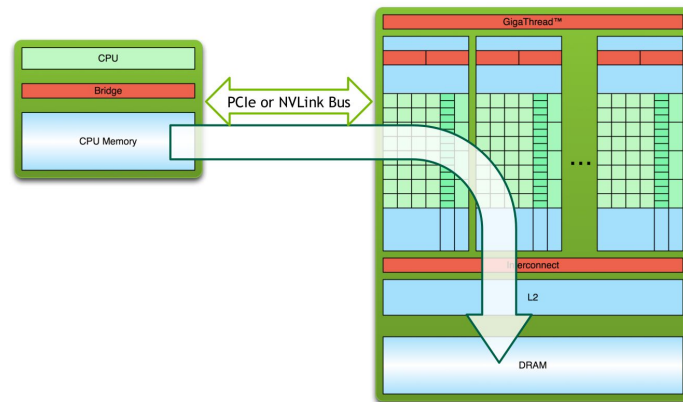
1. Copy data from host memory to device memory
2. Load GPU code, cache data on GPU for performance  
(data is persistent in GPU memory, continuous exchange during the life of the application is possible)

# The general idea



# The common pitfalls

- **Host and device memory are separate entities**
  - ⇒ Data that is in the host memory (“CPU memory”) is not visible/usable by the GPU
  - ⇒ During the life of the application on the GPU, intermediate stage results are still only on the memory of the GPU, and are not stored in the host memory

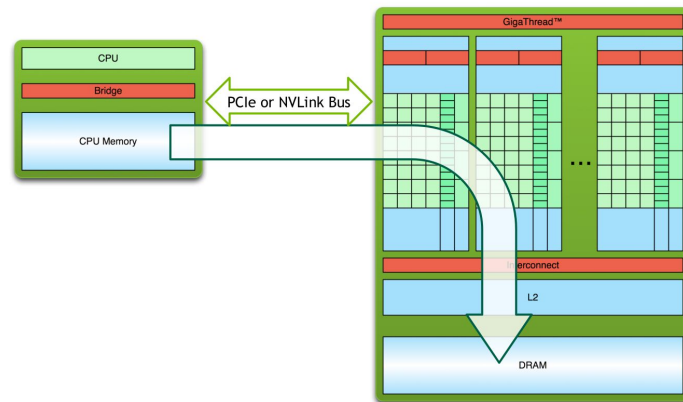




# The common pitfalls

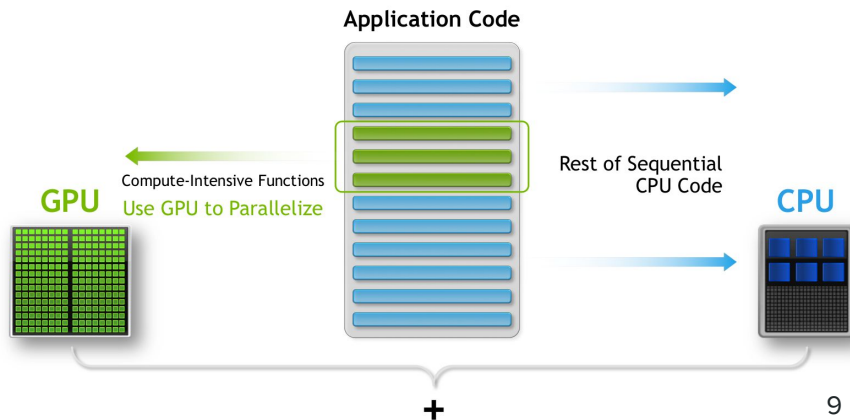
- **Host and device memory are separate entities**

- ⇒ Data that is in the host memory (“CPU memory”) is not visible/usable by the GPU
- ⇒ During the life of the application on the GPU, intermediate stage results are still only on the memory of the GPU, and are not stored in the host memory

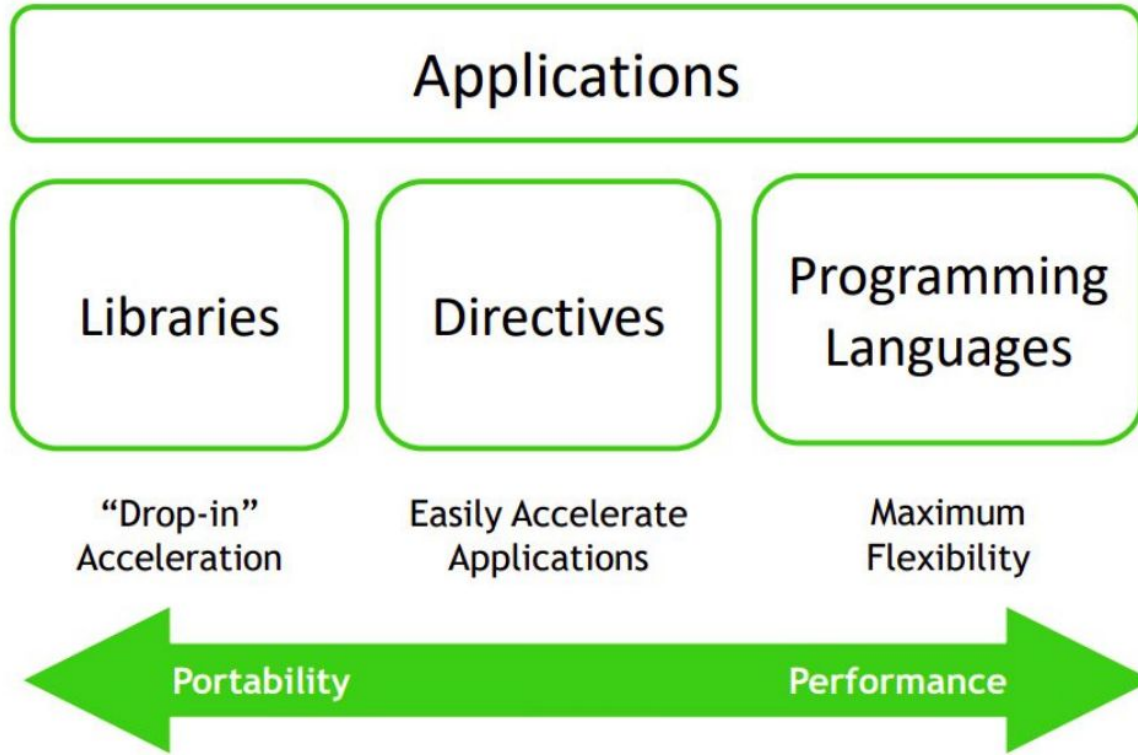


- **Parallel processing on the GPU is asynchronous**

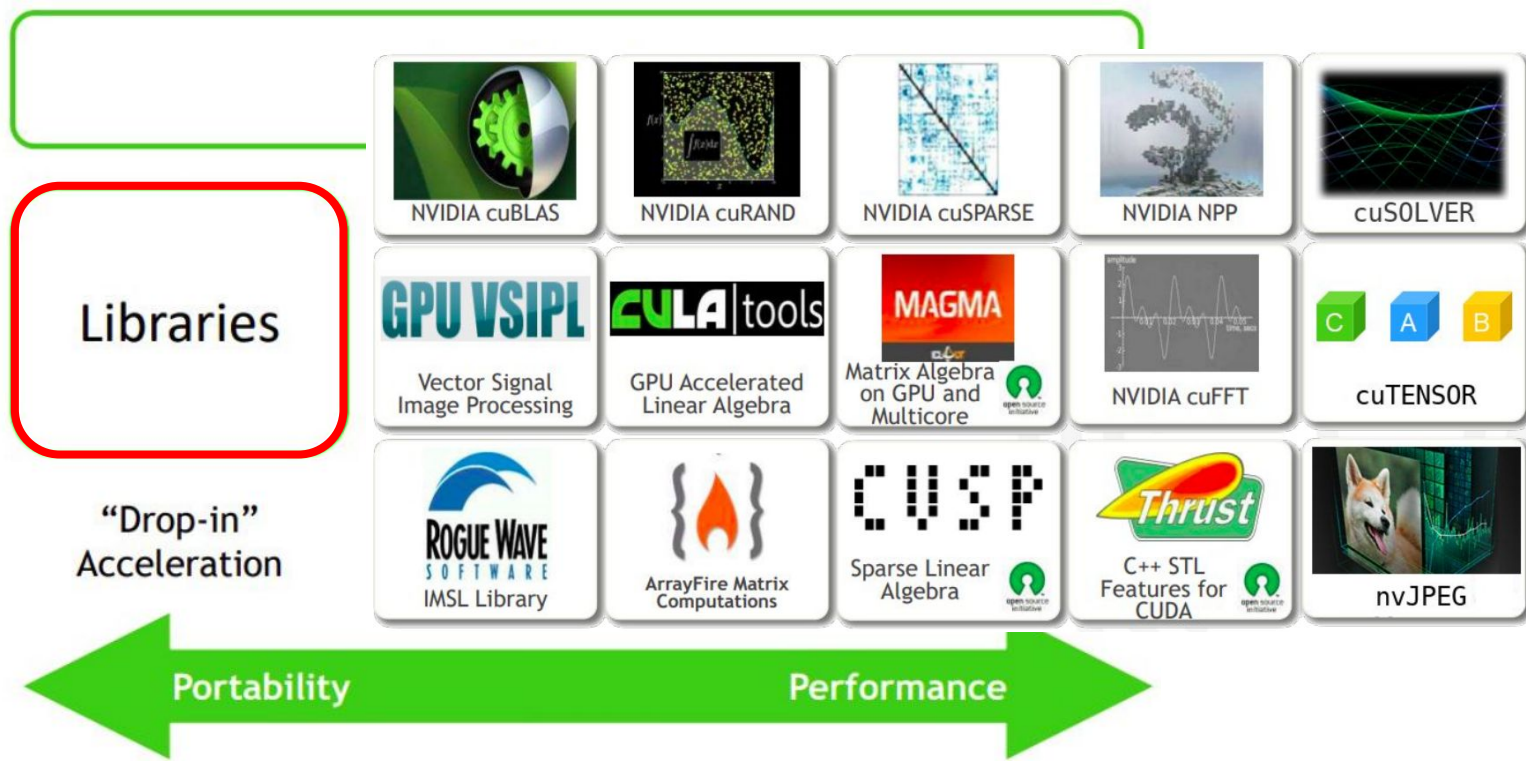
- ⇒ Processing will start but won't block the main application from continuing onward
- ⇒ If a result computed by the GPU is required before moving forward with the CPU computation, a device-to-host memory transfer is required



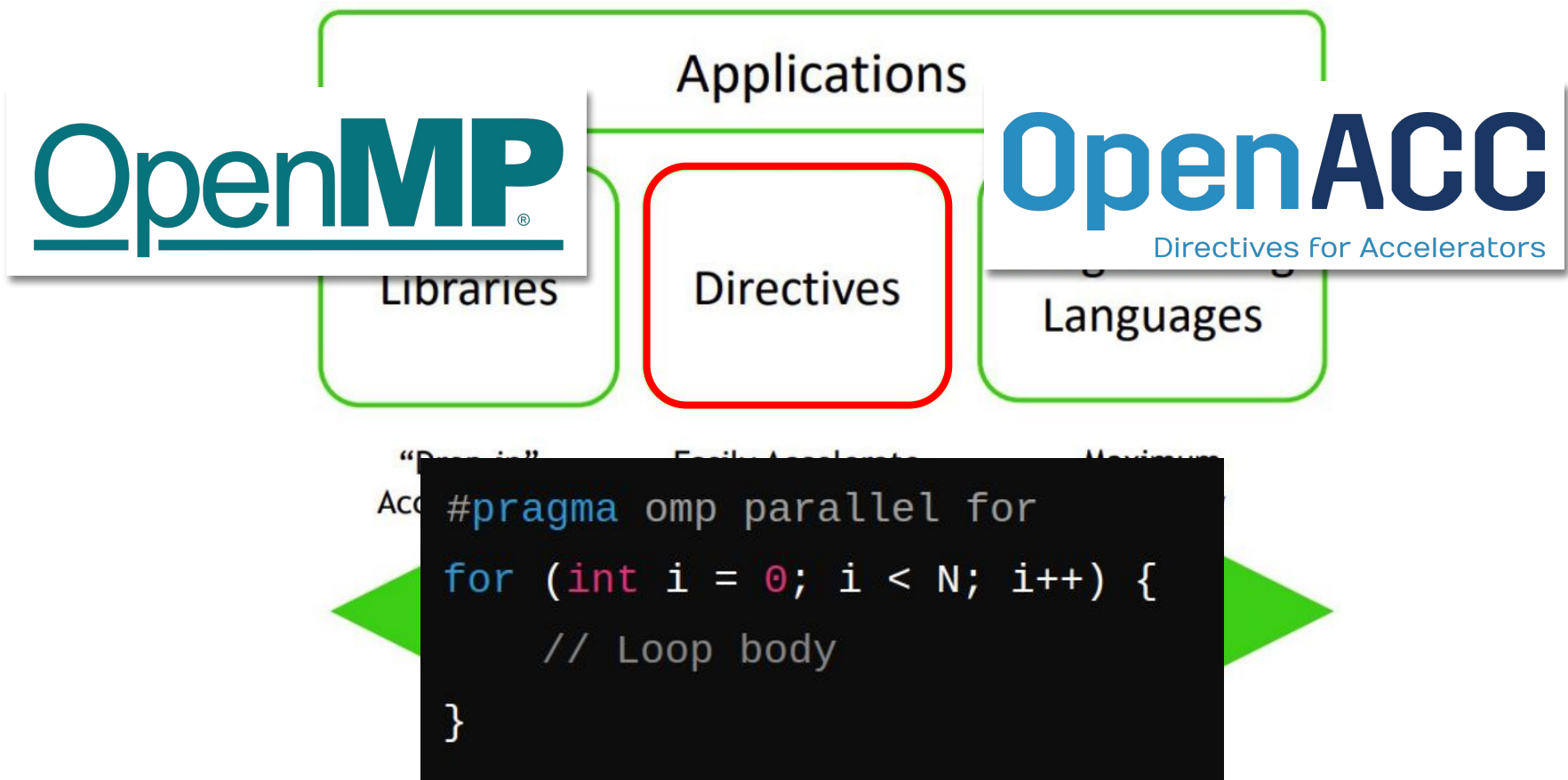
# The alternatives



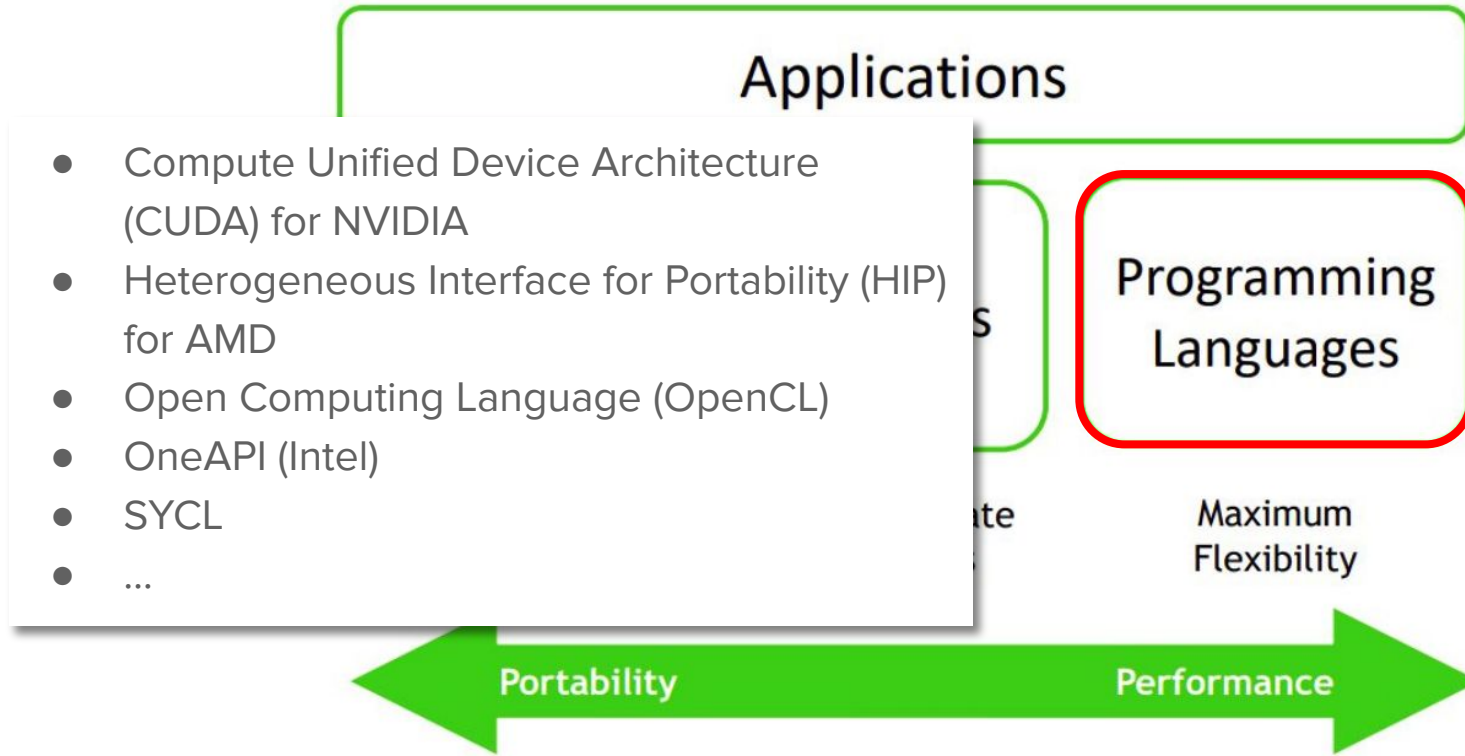
# The alternatives



# The alternatives



# The alternatives



# Once again, focus on the memory

- Data movement between CPU and GPU is often the bottleneck of many GPU applications
  - ⇒ The bus transfer can be quite slow with respect to the GPU throughput capacity!
  - ⇒ Sometimes data transfer can take more than the data initialization and computation on GPU
- This is the main reason we are going to tackle programming GPUs with a bottom-up approach
  - Learning first how to explicitly program a GPU, including data movements between host and device memories
  - Before getting to higher-level languages with more emphasis on portability

# Introduction to CUDA

# NVIDIA CUDA



## CUDA → Compute Unified Device Architecture

CUDA is a general purpose parallel computing platform and programming model (not directly a programming language per se) introduced by NVIDIA around 2006/7

It consists of:

- a hierarchical multi-threaded programming paradigm that allows to exploit the GPU hardware structure
- a set of extensions to higher level programming languages (C/C++ and Fortran) to design CPU–GPU applications, where the GPU is exploited as a coprocessor for heavy parallel task
- a new architecture instruction set called PTX (Parallel Thread eXecution) to match GPU typical hardware
- a developer toolkit to compile, debug, and profile programs
- a set of GPU accelerated libraries for common scientific algorithms



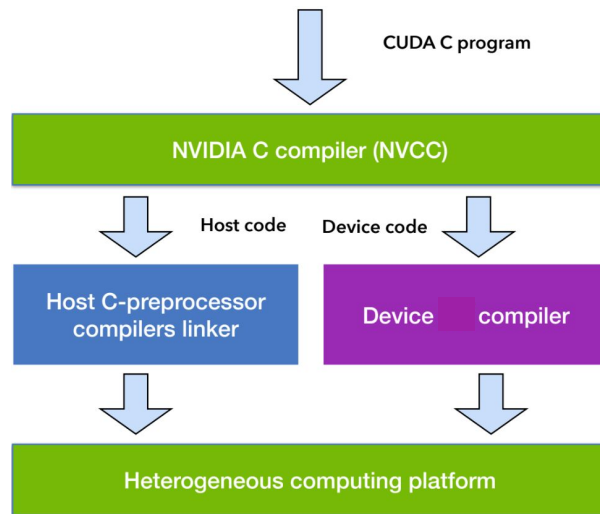
# NVIDIA CUDA-C/C++

CUDA-C/C++ is an extension of the standard ANSI C/C++ language with extensions to enable heterogeneous programming, and also APIs for:

- Control and query available GPU devices
- Manage GPU memory allocation and data transfers
- Manage and control independent work queues

The structure of a CUDA-C program reflects the coexistence of a **host (CPU)** and one or more **devices (GPUs)** in the computer

- ⇒ The **source file** can have a **mixture of both host and device code**
- ⇒ Any “plain” C program can be considered a CUDA-C program that only contains code to be run on the host



# NVIDIA CUDA Compute Capability

GPU HWs are differentiated based on their “compute capabilities”, describing the degree of implementation of features (→ the higher the more modern, i.e. better)

e.g.:

- The total number of concurrent functions executions on a GPU varies strongly with the device Compute Capability
- The “recent” Tensor Cores functionalities are available starting from Compute Capability 7.x

Volta architecture (e.g. V100)	7.0
Ampere architecture (e.g. A100)	8.0
GeForce RTX 4090	8.9
GeForce RTX 3080	8.6
Jetson Nano	5.3

# Before getting to the language... the strategy!

1. **Identify computational intensive, data-parallel, portions of your task/application**
  - Isolate them into functions
  - Identify involved data to be moved between host and device

# Before getting to the language... the strategy!

1. **Identify computational intensive, data-parallel, portions of your task/application**
  - Isolate them into functions
  - Identify involved data to be moved between host and device
2. **Translate identified function candidates into real *CUDA kernels***
  - Make the function such that every ***thread*** (kernel call) will act on independent data
  - Change code so to assign to each thread its own data

# Before getting to the language... the strategy!

## 1. Identify computational intensive, data-parallel, portions of your task/application

- Isolate them into functions
- Identify involved data to be moved between host and device

## 2. Translate identified function candidates into real *CUDA kernels*

- Make the function such that every **thread** (kernel call) will act on independent data
- Change code so to assign to each thread its own data

## 3. Modify code in order to manage memory and call **CUDA kernels**

- Allocate memory on the device
- Transfer needed data from host to device memory
- Include calls to CUDA kernels with execution configuration (e.g. number of threads)
- Transfer resulting data from device to host memory

# A refresher on C (++)

# Compiled

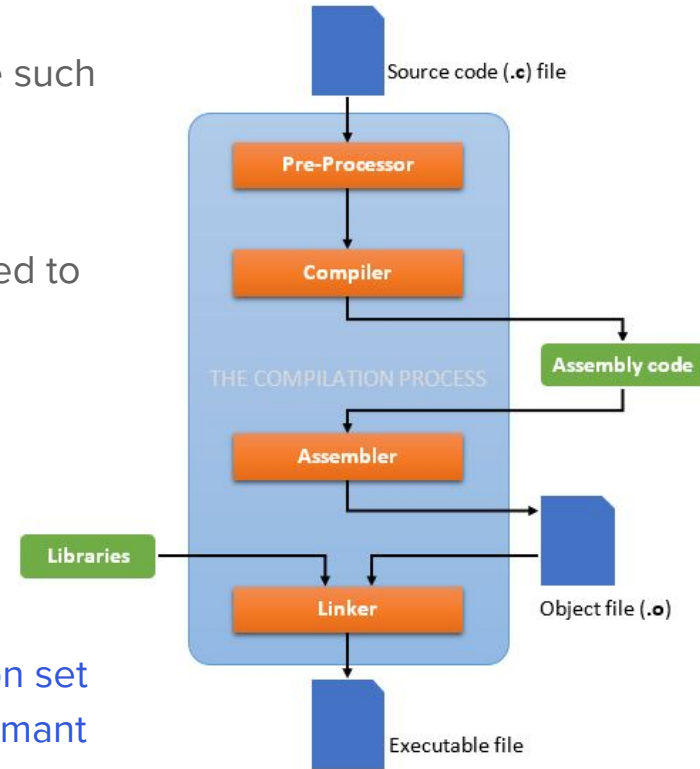
## C and C++ are compiled languages

This means that in order to run a C(++) program, one has to write such program –**the source file**–, and invoke another program –**the compiler**– to translate all our language directly to machine code

This will generated a new file –**the executable**– that can be called to run the machine code



- This makes the code optimized for the machine's instruction set
- The program is thus usually very efficient and highly performant
- But it makes the executable not portable across machines with different architectures



# Compiled

```
// Include the standard input/output library
#include <stdio.h>

// Main function, entry point of the program
int main(int argc, char **argv){

    // Print "Hello world" to the console
    printf("Hello world\n");

    // Return 0 to indicate successful execution
    return 0;
}
```

source code → test.c



invoking the compiler on the source code

```
$ gcc test.c
```

if everything goes right (no compilation errors)  
an executable is created

default executable → a.out

one can provide a custom name to the executable, via  
the -o option

```
$ gcc test.c -o my_exe
```



executing the machine code

```
$ ./my_exe
```



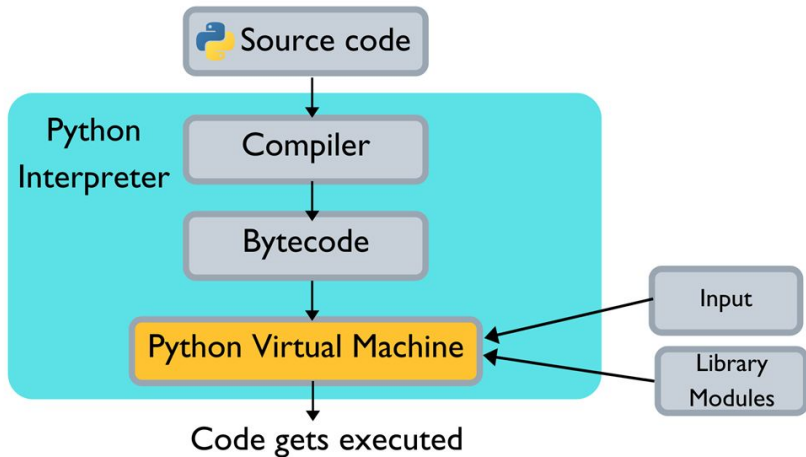
# (vs Interpreted)

## Python is an interpreted language

Does this mean that there is zero compilation involved? ... not exactly...

Python does convert the source code into **bytecode** by means of an *internal* compilation step by the interpreter

This code is not machine readable, and can be executed only by a dedicated Python Virtual Machine, where the code is effectively run



(does this sound any familiar with the JVM concept of Spark...?!)

BTW, the standard Python interpreter (CPython) is written in C

# Statically typed

C++ is a language where the type of the variables used must be declared explicitly, as it must be known at compile time

```
int a = 12;
float b = 3.14;
float c[4] = {0., 0.5, 0.25, 0.125};
```

Pythonic-style variable declarations won't work, and will result in an error during compilation

```
d = -9.;
```

```
test.c: In function 'main':
test.c:14:5: error: 'd' undeclared (first use in this function)
  14 |     d = -9;
      |     ^
test.c:14:5: note: each undeclared identifier is reported only once for each function it appears in
```

# Yet weakly typed

In C the type of the variable dictates the results of the operations

Operations such as addition, subtraction, multiplication, division and assignment will implicitly convert data different data types to the data another data type

```
int a = 12;  
int b = 15;  
float c;  
  
c = a/b;
```

- a. 0.8
- b. 0 (int)
- c. 0 (float)
- d. compilation error

```
float a = 12.;  
float b = 15.;  
int c;  
  
c = a/b;
```

- a. 0.8
- b. 0 (int)
- c. 0 (float)
- d. compilation error

# Yet weakly typed

In C the type of the variable dictates the results of the operations

Operations such as addition, subtraction, multiplication, division and assignment will implicitly convert data different data types to the data another data type

```
int a = 12;  
int b = 15;  
float c;  
  
c = a/b;
```

- a. 0.8
- b. 0 (int)
- c. 0 (float)**
- d. compilation error

```
float a = 12.;  
float b = 15.;  
int c;  
  
c = a/b;
```

- a. 0.8
- b. 0 (int)**
- c. 0 (float)
- d. compilation error

# Program structure

Any C(++) source code must have a `main()` **function** as its entry point

```
// Main function, entry point of the program
int main(){

    /*
        Do something
    */

    // Return 0 to indicate successful execution
    return 0;
}
```

# Program structure

Any C(++) source code must have a `main()` **function** as its entry point

```
// Main function, entry point of the program
int main(){

    /*
       Do something
    */

    // Return 0 to indicate successful execution
    return 0;
}
```

All main syntactic constructs in C/C++ must be enclosed by curly braces { }

- function
- loops
- if/else statements

# Program structure

Any C(++) source code must have a `main()` **function** as its entry point

```
// Main function, entry point of the program
int main() {

    /*
     * Do something
     */

    // Return 0 to indicate successful execution
    return 0;
}
```

Comments in C can be:

- single line → `//`
- multi line → `/* ... */`

# Program structure

Any C(++) source code must have a `main()` **function** as its entry point

```
// Main function, entry point of the program
int main() {

    /*
       Do something
    */

    // Return 0 to indicate successful execution
    return 0;
}
```

Functions, like variables, must have a type

- the type of the function denotes what the function will return
- a type must be assigned even if the function is not supposed to return a value
  - `void` type exists for this purpose



# Program structure

Any C(++) source code must have a `main()` **function** as its entry point

```
// Main function, entry point of the program
int main(){

    /*
       Do something
    */

    // Return 0 to indicate successful execution
    return 0;
}
```

All input parameters of a function (if any) will be listed between parentheses, e.g.:

```
main (int a_int, float a_float){ ... }
```

# Program structure

Any C(++) source code must have a `main()` **function** as its entry point

```
// Main function, entry point of the program
int main() {

    /*
        Do something
    */

    // Return 0 to indicate successful execution
    return 0;
}
```

When finishing to write a line of code, don't forget the semicolon!

# Variable types

Integer type

C type	stdint.h type	Bits	Sign	Range
char	uint8_t	8	Unsigned	0 .. 255
signed char	int8_t	8	Signed	-128 .. 127
unsigned short	uint16_t	16	Unsigned	0 .. 65,535
short	int16_t	16	Signed	-32,768 .. 32,767
unsigned int	uint32_t	32	Unsigned	0 .. 4,294,967,295
int	int32_t	32	Signed	-2,147,483,648 .. 2,147,483,647
unsigned long long	uint64_t	64	Unsigned	0 .. 18,446,744,073,709,551,615
long long	int64_t	64	Signed	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807

Floating point type

C type	IEE754 Name	Bits	Range
float	Single Precision	32	-3.4E38 .. 3.4E38
double	Double Precision	64	-1.7E308 .. 1.7E308

plus...

`void`

# Variable declaration and initialization

Being C statically typed, any variable must be:

- First declared...

```
int a;  
int b;  
float c;
```

```
float x, y, z;
```

- ... then initialized

```
a = 12;  
b = 15;  
c = 3.14;
```

```
x = 0.;  
y = 0.;  
z = 0.;
```

# Variable declaration and initialization

Alternatively, C allows for in-place declaration at the time of initialization

```
int a = 12;  
int b = 15;  
float c = a/b;
```

```
float x=0., y=0., z=0.;
```

```
float x,y,z;  
x=y=z=0.;
```

Uninitialized variables do not contain “zero” or other default values...

You must assume they contain garbage!

→ Random values given by the state of the memory location that this variable represents

# Variable declaration and initialization

If a variable is not supposed to be modified during execution, it can be declared as constant

```
const int a = 10;
```

```
...
```

```
a = a + 20;
```

This will be caught by the compiler, and will result in an error

```
test_.c: In function 'main':
test_.c:13:6: error: assignment of read-only variable 'a'
  13 |     a = a + 20;
      |     ^
```

# Headers (libraries)

In order to exploit external libraries, a preprocessor directive must be added to the source code to include the header files (.h extension)

```
#include <stdio.h> // Include the standard IO library

// Main function, entry point of the program
int main() {

    ...
}
```

Common C headers are for instance:

- `stdio.h` → standard input/output functions
- `stdlib.h` → memory management
- `math.h` → mathematical functions
- `time.h` → time manipulation functions

# Printing to shell

The standard C function to print strings and variables to shell is `printf`, included in the `stdio` library

```
#include <stdio.h> // Include the standard IO library

// Main function, entry point of the program
int main() {

    float x=0.,y,z;

    printf("value of x = %f\n",x);
    printf("value of y = %f\n",y);
    printf("value of z = %f\n",z);

    // Return 0 to indicate successful execution
    return 0;
}
```

Strings are enclosed in double quotes " not single ' !



# Printing to shell

The standard C function to print strings and variables to shell is `printf`, included in the `stdio` library

```
#include <stdio.h> // Include the standard IO library

// Main function, entry point of the program
int main() {

    float x=0.,y,z;

    printf("value of x = %f\n",x);
    printf("value of y = %f\n",y);
    printf("value of z = %f\n",z);

    // Return 0 to indicate successful execution
    return 0;
}
```

To printout a variable, we must explicitly state its type in the string

`%d` → integer

`%f` → float

The variable(s) must then follow, after comma(s)

# Printing to shell

The standard C function to print strings and variables to shell is `printf`, included in the `stdio` library

```
#include <stdio.h> // Include the standard IO library

// Main function, entry point of the program
int main() {

    float x=0.,y,z;

    printf("value of x = %f\n",x);
    printf("value of y = %f\n",y);
    printf("value of z = %f\n",z);

    // Return 0 to indicate successful execution
    return 0;
}
```

To end a line, a newline character must be used

`\n`

# Printing to shell

All this is quite old school, as it's specific to C. In C++ things are a bit easier, using:

```
#include <iostream> // Include the C++ IO library

// Main function, entry point of the program
int main() {

    float x=0.,y,z;

    std::cout << "value of x = " << x << std::endl;
    std::cout << "value of y = " << y << std::endl;
    std::cout << "value of z = " << z << std::endl;

    // Return 0 to indicate successful execution
    return 0;
}
```

`std::cout` → to initiate the printout

`<<` → to append new strings/variables

`std::endl` → to end the current line

# Hello world

```
// Include the standard input/output library
#include <stdio.h>

// Main function, entry point of the program
int main(){

    // Print "Hello world" to the console
    printf("Hello world\n");

    // Return 0 to indicate successful execution
    return 0;
}
```

# Hello world

```
// Include the C++ IO library
#include <iostream>

// Main function, entry point of the program
int main(){

    // Print "Hello world" to the console
    std::cout << "Hello world" << std::endl;

    // Return 0 to indicate successful execution
    return 0;
}
```

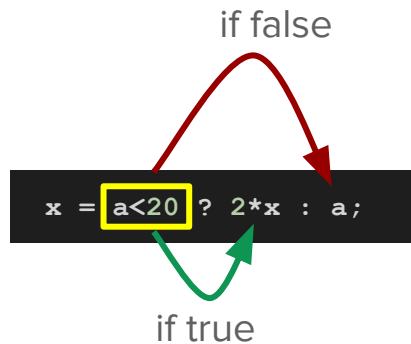
# Conditional statements

if/else statements are extremely similar to Python, with the usual caveats of remembering the curly brackets to separate the individual statements

```
if (a < 20) {  
    x = 2*a;  
}  
else {  
    x = a;  
}
```

Alternatively, a more compact (yet less readable) notation using the ternary operators ? :

if false



The diagram shows the ternary operator expression `x = a < 20 ? 2*x : a;` on a dark background. The condition `a < 20` is enclosed in a yellow rectangular box. A red curved arrow originates from the top of this box and points to the `a` at the end of the expression, labeled "if false". A green curved arrow originates from the bottom of the box and points to the `2*x` part of the expression, labeled "if true".

```
x = a < 20 ? 2*x : a;
```

if true

# Conditional statements

if/else statements are extremely similar to Python, with the usual caveats of remembering the curly brackets to separate the individual statements

```
if (a < 20) {  
    x = 2*a;  
}  
else if (a < 10) {  
    x = a;  
}  
else {  
    x = 0;  
}
```

Alternatively, a more compact (yet less readable) notation using the ternary operators ? :

```
x = a<20 ? 2*a : a<10 ? a : 0;
```

# Conditional statements

Pythonic versions of the logic operators such as `and`, `or`, `not` are not available in C

`and` → `&&`

`or` → `||`

`not` → `!`

```
if !( (a > 15) && (a < 20) ) {  
    x = 2*a;  
}
```

same  logic

```
if ( (a <= 15) || (a >= 20) ) {  
    x = 2*a;  
}
```



# For loops

For loops require three components:

1. The iterator initialization → e.g. `int i = 0`
2. The conditional expression → e.g. `i < 10`
3. The iterator increment logic → e.g. `++i`

```
float x = 0.;  
  
for(int i = 0 ; i < 10 ; ++i){  
  
    x += (float)i/2.;  
  
    printf("x(%d) = %f\n",i,x);  
}
```

# For loops

For loops require three components:

1. The iterator initialization → e.g. `int i = 0`
2. The conditional expression → e.g. `i < 10`
3. The iterator increment logic → e.g. `++i`

```
float x = 0.;  
  
for(int i = 0 ; i < 10 ; ++i){  
  
    x += (float)i/2.;  
  
    printf("x(%d) = %f\n",i,x);  
}
```

Casting an integer to a float

# For loops

```
float x = 0.;

for(int i = 0 ; i < 10 ; ++i){

    // if i is even - skip
    if (i%2 == 0)
        continue;

    // if i is larger than 5 - stop
    if (i>5)
        break;

    x += (float)i/2.;

    printf("x(%d) = %f\n",i,x);

}
```

As in Python, **continue** skips to the next iteration of the loop

And **break** exits the current scope, i.e. the code delimited by { }

# Functions

Functions work like main():

1. returned data type
2. name of the function
3. list of input parameters with their types

```
// Function computing x
float compute_x(float x_, int i_){

    x_ += (float)i_/2.;

    return x_;
}
```

```
int main(){

    float x = 0.;

    for(int i = 0 ; i < 10 ; ++i){

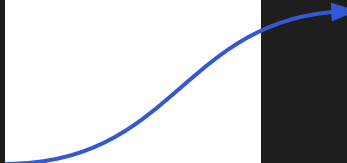
        ...

        x = compute_x(x, i);

        ...

    }

    return 0;
}
```



# Memory management

**C** is a **memory-managed** language, differently from **Python**, which is a **garbage-collected** one

This means that in **Python** unused variables will be marked to be deleted, and will be *eventually* unallocated automatically.

Users can at most mark data for deletion but the memory won't be freed up instantaneously, or won't be controlled by users' actions.

```
my_variable = 42
del my_variable
```

This reduces the risk of memory leaks (the continuously growing usage of memory), but can lead to less control over resource usage

# Memory management

**C** is a **memory-managed** language, differently from **Python**, which is a **garbage-collected** one

In **C**, memory is allocated and deallocated by the user.

C provides explicit control over memory, which can increase efficiency but also the risk of errors like memory leaks or segmentation faults (attempts to access non-existent memory locations).

C offers 2 memory allocation patterns:

- static memory allocation
- dynamic memory allocation

# Static memory allocation

n-dimensional arrays (vectors, matrices, ...) can be created via the static allocation approach

In C, a n-dimensional arrays is a **single-type** list of a **fixed number** of elements

```
// a list of 10 float elements
float x[10] = {0.,0.,0.,0.,0.,0.,0.,0.,0.,0.}; // initialized
float y[10];                                // un-initialized

for(int i = 0 ; i < 10 ; ++i){
    x[i] = (float)i;
    y[i] = x[i]*2.;
}
```

Statically allocated vectors are placed on the **stack** → the faster yet smaller and limited in size part of the memory

Data in a vector/matrix is allocated sequentially in memory, one element after the other

But are not flexible... their size is fixed at compile time and can't be changed during execution

# Static memory allocation

n-dimensional arrays (vectors, matrices, ...) can be created via the static allocation approach

In C, a n-dimensional arrays is a **single-type** list of a **fixed number** of elements

```
// a matrix of 10x8 int elements
int M[10][8];

for(int i = 0 ; i < 10 ; ++i){
    for(int j = 0 ; j < 8 ; ++j){
        M[i][j] = i*10 + j;
    }
}
```

Statically allocated vectors are placed on the **stack** → the faster yet smaller and limited in size part of the memory

Data in a vector/matrix is allocated sequentially in memory, one element after the other

But are not flexible... their size is fixed at compile time and can't be changed during execution



# Static memory allocation

n-dimensional arrays (vectors, matrices, ...) can be created via the static allocation approach

In C, a n-dimensional arrays is a **single-type** list of a **fixed number** of elements

It's also possible to not specify the number of elements in the variable declaration, but it must be initialized in the same line (so... it's basically the same)

```
// an array of 3 elements
int array[] = {1,2,3};
printf("2nd array element: %d\n",array[1]);

// a string --> an array of characters
char text[] = "a string!";
printf("my string: %s\n",text);
```

# Pointers

Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

# Pointers

Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

The \* symbol is used to declare a pointer

# Pointers

Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

The & symbol is used to get the address of a variable

# Pointers

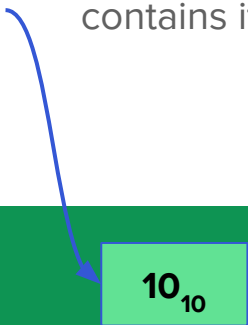
Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

**a** contains its “value”



**MEMORY SPACE**

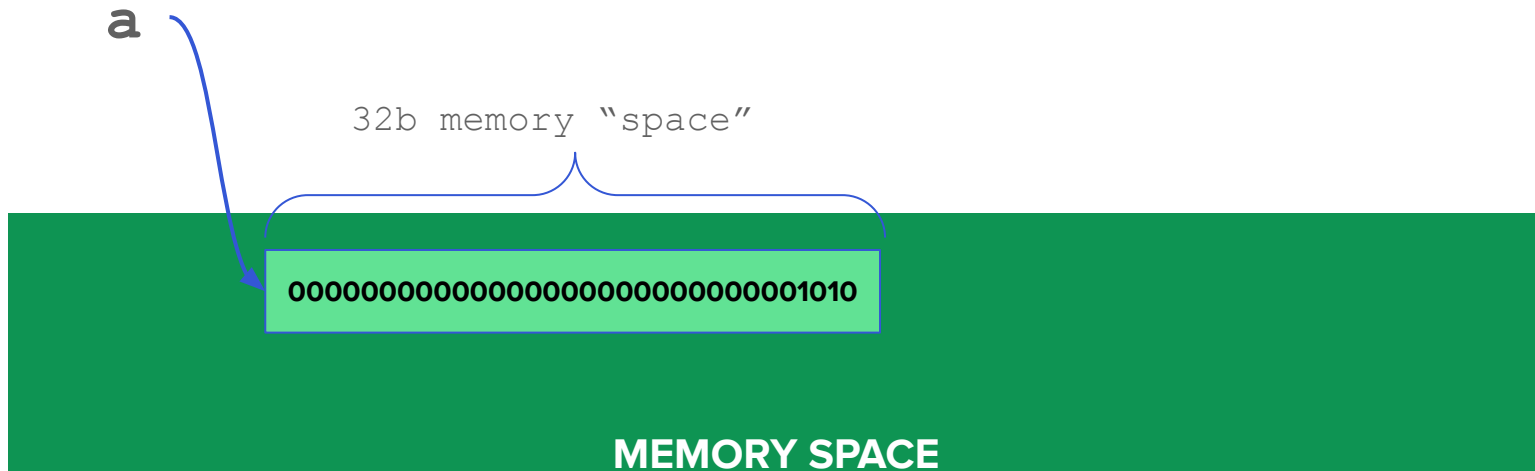
# Pointers

Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```



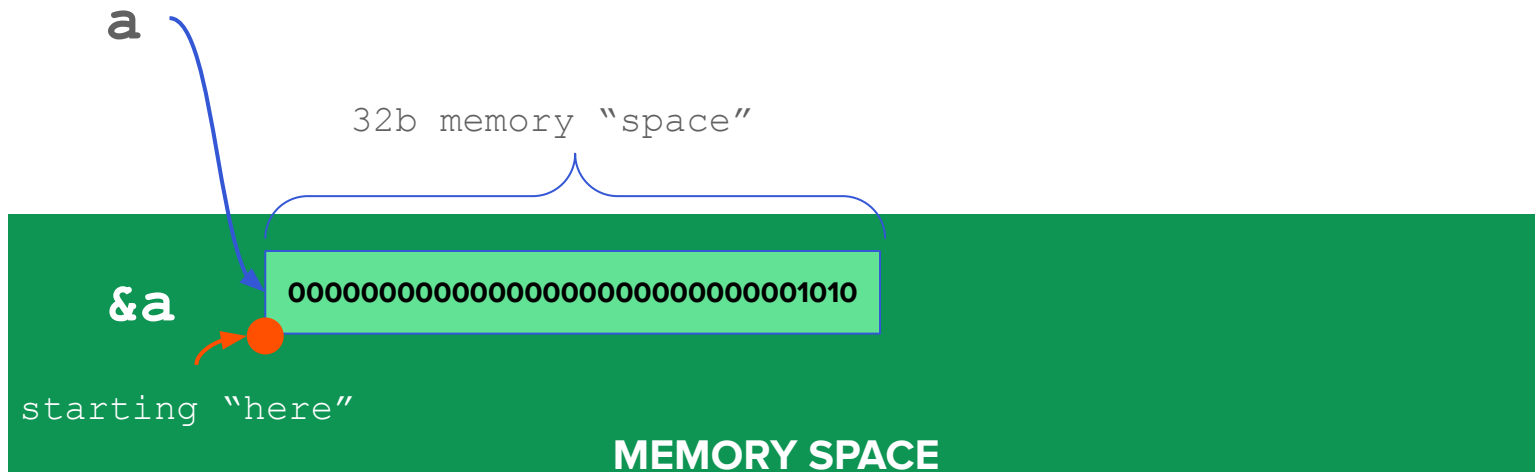
# Pointers

Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```



# Pointers

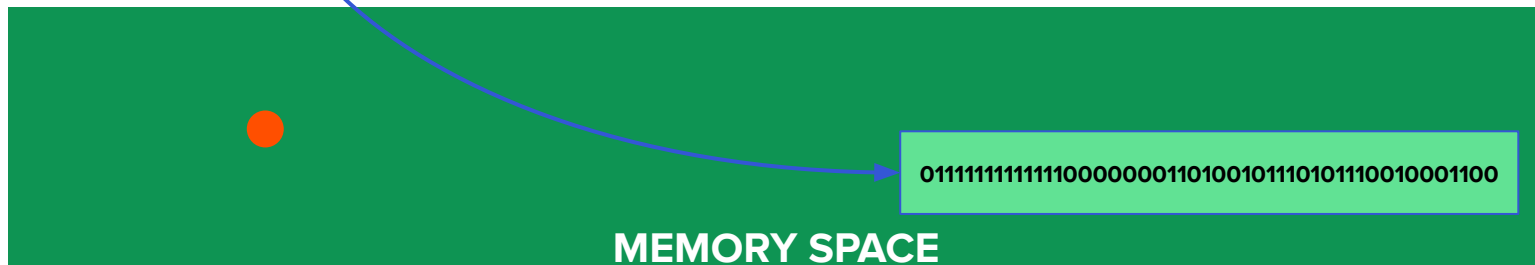
Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

**p** contains the “location” in memory where to find the “value” of a





# Pointers

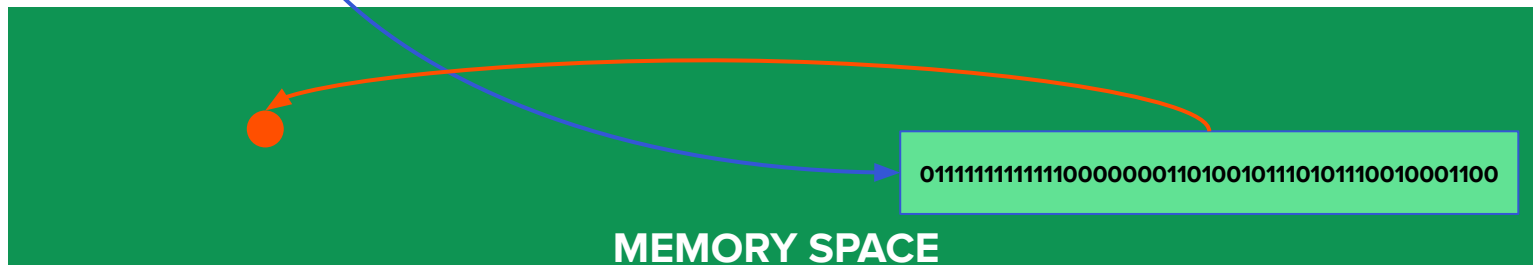
Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

**p** contains the “location” in memory where to find the “value” of a



# Pointers

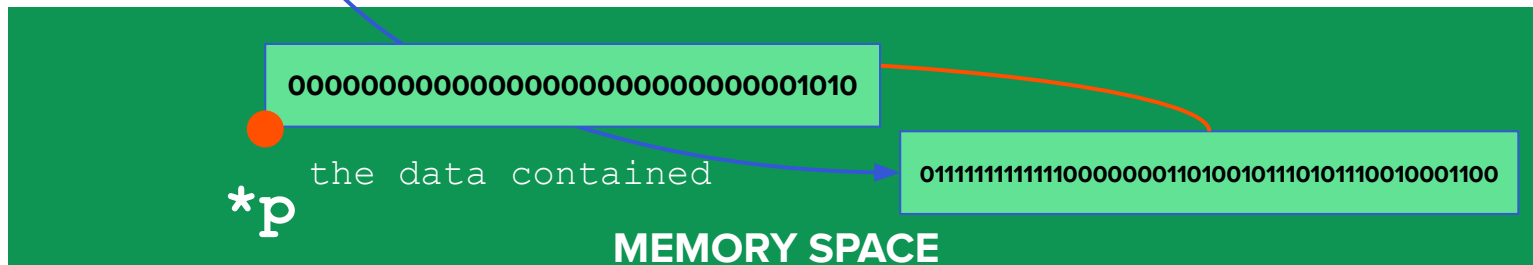
Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

**p** contains the “location” in memory where to find the “value” of a



# Pointers

Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

```
printf(" a = %d\n",a); // the value of a
printf("&a = %p\n",&a); // the address of a

printf(" p = %p\n",p); // the pointer to a
                        // (i.e. the address of a)
printf("*p = %d\n",*p); // the value pointed by the pointer
                        // (i.e. the value of a)
```

# Pointers

Memory can be accessed dynamically by means of **pointers**

A pointer is a variable whose content is not its value, but an address in the memory space

A pointer in fact stores the memory address of another variable

```
int a = 10;    // a holds the value 10
int *p = &a;   // p is a pointer that holds the address of 'a'
```

```
printf(" a = %d\n",a); // the value of a
printf("&a = %p\n",&a); // the address of a

printf(" p = %p\n",p); // the pointer to a
                        // (i.e. the address of a)
printf("*p = %d\n",*p); // the value pointed by the pointer
                        // (i.e. the value of a)
```

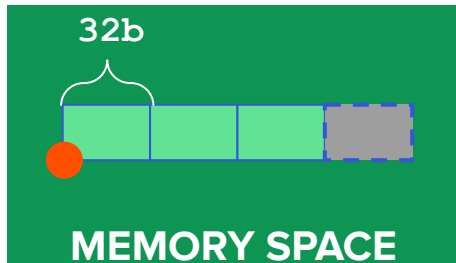
```
a = 10
&a = 0x7ffc06975c8c
p = 0x7ffc06975c8c
*p = 10
```

# Pointers - array duality

Given that a pointer is an address in memory, we have a good duality between arrays and pointers

We can get the address of the first element in an array, and then access all other elements by incrementing our pointer

```
int arr[3] = {1, 2, 3};  
int *p = arr;    // p points to the first element of arr  
  
printf("%d\n", *p);        // prints the first element  
printf("%d\n", *(p+1));    // prints the second element  
printf("%d\n", *(p+2));    // prints the third element  
printf("%d\n", *(p+3));    // prints the fourth element !?
```



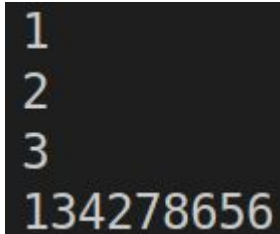
This won't protect us from accessing locations in memory that are un-allocated, or allocated for other stuff what we aren't supposed to mess up with

# Pointers - array duality

Given that a pointer is an address in memory, we have a good duality between arrays and pointers

We can get the address of the first element in an array, and then access all other elements by incrementing our pointer

```
int arr[3] = {1, 2, 3};  
int *p = arr;    // p points to the first element of arr  
  
printf("%d\n", *p);        // prints the first element  
printf("%d\n", *(p+1));    // prints the second element  
printf("%d\n", *(p+2));    // prints the third element  
printf("%d\n", *(p+3));    // prints the fourth element !?
```



1  
2  
3  
134278656

This won't protect us from accessing locations in memory that are un-allocated, or allocated for other stuff what we aren't supposed to mess up with

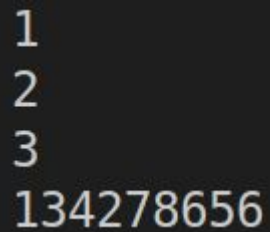
# Pointers - array duality

Given that a pointer is an address in memory, we have a good duality between arrays and pointers

We can get the address of the first element in an array, and then access all other elements by incrementing our pointer

```
int arr[3] = {1, 2, 3};
int *p = arr;    // p points to the first element of arr

printf("%d\n", *p);    // prints the first element
printf("%d\n", *(p+1)); // prints the second element
printf("%d\n", *(p+2)); // prints the third element
printf("%d\n", *(p+3)); // prints the fourth element !?
```



1  
2  
3  
134278656

$*p+1$

$\neq$

$*(p+1)$

Value hosted at the memory location  
pointed by  $p$ ... to which we add 1

Value hosted at the memory location  
pointed by  $(p + 1)$

# Dynamic memory allocation

In general, at runtime we can allocate and de-allocate memory by means of **malloc** and **free**

**malloc(numBytes)** performs dynamic memory allocation by allocating a chunk of memory of size numBytes

It returns a pointer to the address of the data in memory space

We can then use the pointer-array duality to “navigate” the memory space and access values

```
#include <stdlib.h> // Include the standard library for memory allocation

int main(){
    ...
    int *p = (int*) malloc(5 * sizeof(int)); // Allocate memory for 5 integers
    ...
}
```



# Dynamic memory allocation

In general, at runtime we can allocate and de-allocate memory by means of **malloc** and **free**

**malloc(numBytes)** performs dynamic memory allocation by allocating a chunk of memory of size numBytes

It returns a pointer to the address of the data in memory space

We can then use the pointer-array duality to “navigate” the memory space and access values

```
#include <stdlib.h> // Include the standard library for memory allocation

int main(){
    ...
    int *p = (int*) malloc(5 * sizeof(int)); // Allocate memory for 5 integers
    ...
}
```

the size (in Bytes) of the data type

# Dynamic memory allocation

In general, at runtime we can allocate and de-allocate memory by means of **malloc** and **free**

**malloc(numBytes)** performs dynamic memory allocation by allocating a chunk of memory of size numBytes

It returns a pointer to the address of the data in memory space

We can then use the pointer-array duality to “navigate” the memory space and access values

```
#include <stdlib.h> // Include the standard library for memory allocation

int main() {
    ...
    int *p = (int*) malloc(5 * sizeof(int)); // Allocate memory for 5 integers
    ...
}
```

allocated data will be of type  
“pointer to integer”

# Dynamic memory allocation

In general, at runtime we can allocate and de-allocate memory by means of **malloc** and **free**

**malloc(numBytes)** performs dynamic memory allocation by allocating a chunk of memory of size numBytes

It returns a pointer to the address of the data in memory space

We can then use the pointer-array duality to “navigate” the memory space and access values

```
#include <stdlib.h> // Include the standard library for memory allocation

int main(){
    ...
    int n_ints = 5;
    int *p = (int*) malloc(n_ints * sizeof(int)); // Allocate memory for 5 integers
    ...
}
```

Similar to *statically* allocating an array of 5 integers... but dynamic (during execution and not at compile time)!

```
int arr[5];
```

# Dynamic memory allocation

In general, at runtime we can allocate and de-allocate memory by means of **malloc** and **free**

**malloc(numBytes)** performs dynamic memory allocation by allocating a chunk of memory of size numBytes

It returns a pointer to the address of the data in memory space

We can then use the pointer-array duality to “navigate” the memory space and access values

```
// Fill the allocated memory
for (int i = 0; i < 5; i++) {
    p[i] = i + 1;
}

printf(" p      = %p\n", p);      // the address of the first element
printf("*p     = %d\n", *(p+1)); // the value of the second element
printf(" p[3]  = %d\n", p[3]);    // the value of the fourth element
```

```
p      = 0x59fb65fcd2a0
*p     = 2
p[3]  = 4
```

# Dynamic memory allocation

In general, at runtime we can allocate and de-allocate memory by means of **malloc** and **free**

**free(pointer)** frees the dynamically allocated memory to prevent memory leaks

This happens “now” during execution (unlike in garbage-collected languages).

```
free(p);    // Free the allocated memory

printf(" p    = %p\n", p);    // the address of the first element
printf("*p    = %d\n", *(p+1)); // the value of the second element
printf(" p[3] = %d\n", p[3]);  // the value of the fourth element
```

```
p      = 0x59fb65fcd2a0
*p     = 5
p[3]   = -382696961
```

# Dynamic memory allocation

If necessary, it's possible to manipulate memory to copy a block of memory (entire data structures for examples) from one location to another in the memory space

The **memcpy** (**\*to**, **\*from**, **numBytes**) function copies the specified number of bytes from one memory location to the other memory location regardless of the type of data stored

```
int arr[3] = {1, 2, 3};
int *p1 = arr; // p1 points to the first element of arr
int *p2;       // another pointer

printf("addr p1: %p\n", p1); // the location of p1
printf("addr p2: %p\n", p2); // the location of p2

printf(" p1[0]: %d\n", *p1); // the value pointed by p1
printf(" p2[0]: %d\n", *p2); // the value pointed by p2
```

```
addr p1: 0x7ffec311e63c
addr p2: 0x7ffec311eb39
p1[0]: 1
p2[0]: 1597388920
```

Interestingly enough, we have to include the string manipulation library to use memcpy

```
#include <string.h> // Include the string library for memcpy
```

# Dynamic memory allocation

If necessary, it's possible to manipulate memory to copy a block of memory (entire data structures for examples) from one location to another in the memory space

The **memcpy** (**\*to**, **\*from**, **numBytes**) function copies the specified number of bytes from one memory location to the other memory location regardless of the type of data stored


```
int arr[3] = {1, 2, 3};
int *p1 = arr; // p1 points to the first element of arr
int *p2;       // another pointer

printf("addr p1: %p\n", p1); // the location of p1
printf("addr p2: %p\n", p2); // the location of p2

printf(" p1[0]: %d\n", *p1); // the value pointed by p1
printf(" p2[0]: %d\n", *p2); // the value pointed by p2

memcpy(p2, p1, sizeof(p1));

printf(" p1[0]: %d\n", *p1); // the value pointed by p1
printf(" p2[0]: %d\n", *p2); // the value pointed by p2
```



```
addr p1: 0x7ffec311e63c
addr p2: 0x7ffec311eb39
p1[0]: 1
p2[0]: 1597388920
p1[0]: 1
p2[0]: 1
```

# Preprocessor directives

All statements including the pound symbol `#` are preprocessor **directives**

Directives are instructions that are processed by the preprocessor before the actual compilation of the program begins

These instructions are used to include external files, define constants, and manage conditional compilation

Preprocessor directives start with a `#` symbol and don't require a semicolon at the end

- **`#include`** → to include headers
- **`#define`** → to define macros or **constants**
- **`#pragma`** → (compiler-specific) to define special instructions to the compiler (e.g. parallelize loops with OpenMC/OpenACC)
- ...

```
#define PI 3 // Define a constant value
```