Pointers, memory allocation and calling a function by address
Summary

1 - Pointers
2 - Dynamic Allocation
3 - Array with many dimensions
4 - Calling by address
Memory
Memory

0x0000 ...
...
...
...
...
...
...
...
...
0x7F00

System
Memory

System

Program
Memory

System
Program
System

int a;
int a;
int a;
a = 0x0902
int a;
short b;
a en 0x0902
Memory

System
Program
System

int a;
short b;
a en 0x0902
```
int a;
short b;
```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0900</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0A00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0B00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A variable is stored in a memory area when it is declared.
A variable is stored in a memory area when it is declared

```c
int a;
```
A variable is stored in a memory area when it is declared.

```c
int a; 0x0902 pointer onto (int) a
```
A variable is stored in a memory area when it is declared.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800</td>
<td></td>
</tr>
<tr>
<td>0x0900</td>
<td></td>
</tr>
<tr>
<td>0x0A00</td>
<td></td>
</tr>
<tr>
<td>0x0B00</td>
<td></td>
</tr>
<tr>
<td>0x0C00</td>
<td></td>
</tr>
</tbody>
</table>

```c
int a; 0x0902 pointer onto (int) a
short b;
```
A variable is stored in a memory area when it is declared.

```c
int a; 0x0902 pointer onto (int) a
short b; 0x0A0D pointer onto (short) b
```
A variable is stored in a memory area when it is declared & reference operator (address)

```c
int a; 0x0902 pointer onto (int) a
short b; 0x0A0D pointer onto (short) b
```
A variable is stored in a memory area when it is declared.

& reference operator (address)

int a; 0x0902 pointer onto (int) a
short b; 0x0A0D pointer onto (short) b
A variable is stored in a memory area when it is declared

& reference operator (address)

int a; 0x0902 pointer onto (int) a
short b; 0x0A0D pointer onto (short) b
Pointers = Addresses

int a, b;
long long c;

Addresses

&a

&b

&c
Pointers = Addresses

\& address opérateur (its reference)
Pointers = Addresses

& address operator (its reference)

&a returned the memory address of the variable a

int a, b;
l long long c;
Value Operation

&b

pa

...
Value Operation

`pa` pointer onto an integer

```
int *pa;
```

```
&b
```

```
*pa
```

```
b
```

```
c
```
Value Operation

\[ \text{**pa** pointer onto an integer int } \ast \text{pa,}
\]

* operation of dereference (value)
Value Operation

pa  pointer onto an integer int *pa;
*    operation of dereference (value)
*pa  represent the memory at address pa
Value Operation

`pa` pointer onto an integer `int *pa;`

* operation of dereference (value)

`*pa` represent the memory at address `pa`
Pointers of different types

int *pa;
⇒ pa pointer onto an int

float *pb;
⇒ pb pointer onto a float

char *pc;
⇒ pc pointer onto a char
Some examples

<table>
<thead>
<tr>
<th>Address</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
<th>Grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0800</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0900</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0A00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0B00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some examples

```
int a, b, *pi;
```
int a, b, *pi;

Somes examples
### Some Examples

```c
int a, b, *pi;
pi = &a;
```
int a, b, *pi;
pi = &a;
int a, b, *pi;
pi = &a;

a = 10; b = 30;
int a, b, *pi;
pi = &a;
a = 10; b = 30;
int a, b, *pi;
pi = &a;
a = 10; b = 30;
*pi = b;
int a, b, *pi;
pi = &a;
a = 10; b = 30;
*pi = b;
int a, b, *pi;
float c, *pf;
pi = &a;
a = 10; b = 30;
*pi = b;
Some examples

```
int a, b, *pi;
float c, *pf;
pi = &a;
a = 10; b = 30;
*pi = b;
```
int a, b, *pi;
pi = &a;

a = 10; b = 30;
*pi = b;

float c, *pf;

pf = &c;
### Some examples

```
int a, b, *pi;
pi = &a;
a = 10; b = 30;
*pi = b;

float c, *pf;
pf = &c;
```
int a, b, *pi;
pi = &a;
a = 10; b = 30;
*pi = b;

float c, *pf;
pf = &c;
c = 5;
Somes examples

```c
int a, b, *pi;
p i = &a;
a = 10; b = 30;
*pi = b;
```

```c
float c, *pf;
pf = &c;
c = 5;
```
Some examples

```c
int a, b, *pi;
pi = &a;
a = 10; b = 30;
*pi = b;

float c, *pf;
pf = &c;
c = 5;
*pf = *pf / 2;
```
int a, b, *pi;
pi = &a;
a = 10; b = 30;
*pi = b;

float c, *pf;
pf = &c;
c = 5;
*pf = *pf / 2;
Size of a pointer

In general, pointers are typed:

- `int *pa;` pointer onto an `int`
- `float *pb;` pointer onto a `float`

But every pointer is a memory address, 32 bits (GCC Linux)

⇒ in general, every pointers are equivalent
Dynamic Allocation

```c
int *pa;
```
Dynamic Allocation

```c
int *pa;
```

⇒ declare a pointer `pa` onto an `int`
Dynamic Allocation

```c
int *pa;
```

⇒ declare a pointer `pa` onto an `int`

• **allocation** of a memory area to store an address
Dynamic Allocation

\[
\text{int } \ast \text{pa;}
\]

⇒ declare a pointer \text{pa} onto an \text{int}

- allocation of a memory area to store an address
- no initialization
Dynamic Allocation

```c
int *pa;
```

⇒ declare a pointer `pa` onto an `int`

- allocation of a memory area to store an address
- no initialization
- no corresponding integer
Dynamic Allocation

\[ \text{int *} \text{pa;} \]

\[ \Rightarrow \text{declare a pointer } \text{pa} \text{ onto an int} \]

- allocation of a memory area to store an address
- no initialization
- no corresponding integer

\[ \Rightarrow \text{we need to allocate a memory area to} \]
Dynamic Allocation

\textbf{int }\ast \textbf{pa;}

$\Rightarrow$ declare a pointer \textbf{pa} onto an \textbf{int}

- \textit{allocation} of a memory area to store an address
- no initialization
- no corresponding integer

$\Rightarrow$ we need to allocate a memory area to store this integer: allocation and initialization of the
Dynamic Allocation

int *pa;

⇒ declare a pointer pa onto an int

• allocation of a memory area to store an address
• no initialization
• no corresponding integer

⇒ we need to allocate a memory area to store this integer: allocation and initialization of the pointer
Dynamic Allocation: `malloc`

```c
int *pa;
```

⇒ declaration of a pointer `pa` onto an `int`
Dynamic Allocation: `malloc`

```c
int *pa;
```

⇒ declaration of a pointer `pa` onto an `int`
Dynamic Allocation: \texttt{malloc}

\begin{verbatim}
int *pa;
\end{verbatim}

⇒ declaration of a pointer \texttt{pa} onto an \texttt{int}

\begin{verbatim}
0xA00
0xB00
0xC00
\end{verbatim}
Dynamic Allocation: `malloc`

- Declaration of a pointer `pa` onto an `int`

```
int *pa;
```

```
⇒
```

```
int a;
pa = &a;
```

```
0x0A00 0x0B00 0x0C00
```

```
  
  
  pa
```

Dynamic Allocation: `malloc`

```
int *pa;

⇒ declaration of a pointer `pa` onto an `int`
```

```
int a;
pa = &a;
```

```
0x0A00
0xB00
0x0C00
```

```
0x0A06
```
Dynamic Allocation: `malloc`

```c
int *pa;
⇒ declaration of a pointer `pa` onto an `int`
```

```c
int a;
pa = &a;
```
Dynamic Allocation: `malloc`

Declaration of a pointer `pa` onto an `int`:

```c
int *pa;
```

Initialization:

```c
int a;
pa = &a;
```

Allocation in the heap:

```c
pa = malloc(sizeof(int)); in the heap
```
Dynamic Allocation: `malloc`

```c
int *pa;

⇒ declaration of a pointer `pa` onto an `int`

int a;
pa = &a;
pa = malloc(sizeof(int)); in the heap
```
int *pa;
pa = malloc(sizeof(int));
Usage of malloc

```c
int *pa;
pa = malloc(sizeof(int));
```

```c
void *malloc(int n)
```
Usage of `malloc`

```c
int *pa;
pa = malloc(sizeof(int));
```

**void *malloc(int n)**

- allocation of \( n \) bytes of memory, the address of the first int is returned
Usage of malloc

```c
int *pa;
pa = malloc(sizeof(int));
```

void *malloc(int n)

- allocation of \( n \) bytes of memory, the address of the first int is returned
- generic pointer, \( \text{void } * \)
Usage of `malloc`

```c
int *pa;
pa = malloc(sizeof(int));
```

`void *malloc(int n)`

- allocation of \( n \) bytes of memory, the address of the first int is returned
- generic pointer, `void *`

\( \Rightarrow \) it is automatically cast in the typed pointer (as soon as possible)
Size of an object sizeof

The size of a type is not standardized
⇒ to improve portability in C, use sizeof(<type>))

GCC Linux:
• sizeof(int) → 4
• sizeof(char) → 1
• ...

Dynamic Allocation: array

\[
\text{int } *\text{pa};
\]

⇒ declare a pointer \text{pa} onto an \text{int}
Dynamic Allocation: array

```c
int *pa;
⇒ declare a pointer pa onto an int
```
Dynamic Allocation: array

```c
int *pa;
```

⇒ declare a pointer `pa` onto an `int`
Dynamic Allocation: array

```c
int *pa;
⇒ declare a pointer `pa` onto an `int`

int a;
pa = &a;
```
Dynamic Allocation: array

int *pa;

⇒ declare a pointer pa onto an int

int a;
pa = &a;
Dynamic Allocation: array

\[ \text{int } *pa; \]
\[ \Rightarrow \text{declare a pointer } pa \text{ onto an int} \]

\[ \text{int } a; \]
\[ pa = \&a; \]
Dynamic Allocation: array

```c
int *pa;

⇒ declare a pointer pa onto an int
```

```c
int a;
pa = &a;
```

```c
pa = malloc(3*(sizeof(int)));
```

Dynamic Allocation: array

int *pa;
⇒ declare a pointer `pa` onto an `int`

int a;
`pa = &a;`

`pa = malloc(3*(sizeof(int)));`
Dynamic Arrays

```java
int t[3];
```
⇒ array of 3 int
Dynamic Arrays

```plaintext
int t[3];
⇒ array of 3 int
```

equivalent to

t

```plaintext
t[0]  t[1]  t[2]
```
Dynamic Arrays

```c
int t[3];
```

⇒ array of 3 int

equivalent to

```c
int *t = malloc(3*(sizeof(int)));
```
Dynamic arrays (suite)

```c
float *Tf;
```

⇒ declare a pointer `Tf` onto a float
float *Tf;

⇒ declare a pointer Tf onto a float
Dynamic arrays (suite)

float *Tf;

⇒ declare a pointer Tf onto a float

Tf = malloc(6*(sizeof(float)));
float *Tf;

⇒ declare a pointer Tf onto a float

Tf = malloc(6*(sizeof(float)));
Dynamic array (suite)
Dynamic array (suite)

```c
float *Tf;
```

<table>
<thead>
<tr>
<th>0x0A00</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0B00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0D00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0E00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0F00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic array (suite)

float *Tf;
Tf = malloc(6*(sizeof(float)));
Dynamic array (suite)

float *Tf;
Tf = malloc(6*(sizeof(float)));
Dynamic array (suite)

float *Tf;
Tf = malloc(6*(sizeof(float)));
Dynamic array (suite)

float *Tf;
Tf = malloc(6*(sizeof(float)));

Tf[0] = 10.5; Tf[1] = 2.1;
Tf[3] = 0; Tf[5] = 1.1;
Dynamic array (suite)
Dynamic array (suite)

\[
\text{\texttt{\textless type\texttt{\textgreater} *t = malloc(k* (sizeof(\texttt{\textless type\texttt{\textgreater}))));}}
\]
define an array of \( k \) objects of type \(<\texttt{\textless type}\texttt{\textgreater}\), where \( k \) can be a variable:
sizee non defined at compilation
\(<\texttt{\textless type}\texttt{\textgreater}\) can be an array (pointer):
array with multiple dimensions
Arrays and pointers

float *Tf;
Tf = malloc(6*(sizeof(float)));

*Tf = 10.5; *(Tf+1) = 2.1;
*(Tf+3) = 0; *(Tf+5) = 1.1;

*Tf = 10.5; *(Tf+1) = 2.1;
*(Tf+3) = 0; *(Tf+5) = 1.1;
Arrays and pointers

float *Tf;

Tf = malloc(6*(sizeof(float)));

*Tf = 10.5; *(Tf+1) = 2.1;
*(Tf+3) = 0;  *(Tf+5) = 1.1;

*Tf    = 10.5; *(Tf+1) = 2.1;
*(Tf+3) = 0;  *(Tf+5) = 1.1;
Array with many dimensions

```c
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(int));
```
Array with many dimensions

```c
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(int));
```

T is a pointer on a pointer onto an int
= array of int array
Array with many dimensions

```c
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(int));
```

T is a pointer on a pointer onto an int
= array of int array
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(int));

T is a pointer on a pointer onto an int = array of int array
Array with many dimensions

```c
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(int));
```

*T is a pointer on a pointer onto an int = array of int array*
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(int));

T is a pointer on a pointer onto an int = array of int array
Many dimensions (suite)
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(char));
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(char));
```c
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(char));
```
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(char));
Many dimensions (suite)

```c
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(char));
```
Many dimensions (suite)

```c
int i, m = 3, n = 5;
char **T;
T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)
    T[i] = malloc(n * sizeof(char));
```
Many dimensions (suite)

```c
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```
Many dimensions (suite)

```c
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```
Many dimensions (suite)

```c
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```

```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```
Many dimensions (suite)

```c
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));
```

```
T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));

T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
Many dimensions (suite)

int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));

T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));

T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));

T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));

T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));

T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++)  T[i] = malloc(n * sizeof(char));

T[0][0] = 'l'; T[0][1] = 'a'; T[0][2] = '\n';
T[1][0] = 'i'; T[1][1] = 'c'; T[1][2] = 'i';
T[2][0] = 'a'; T[2][1] = '\n';
Memory Liberation: free

Memory allocated with malloc will be free at the end of the program.

We can decide to free this memory for other usage (the memory of a computer is not infinite).

⇒ function free
Memory Liberation: \texttt{free}

Memory allocated with \texttt{malloc} will be free at the end of the program.

We can decide to free this memory for other usage (the memory of a computer is not infinite).

⇒ function \texttt{free}

\begin{verbatim}
int *pa = malloc((sizeof(int));
\end{verbatim}
Memory Liberation: `free`

Memory allocated with `malloc` will be free at the end of the program.

We can decide to free this memory for other usage (the memory of a computer is not infinite).

⇒ function `free`

```c
int *pa = malloc((sizeof(int)); /*allocation*/
```
Memory Liberation : \texttt{free}

Memory allocated with \texttt{malloc} will be free at the end of the program.

We can decide to free this memory for other usage (the memory of a computer is not infinite).

\[ \Rightarrow \text{function } \texttt{free} \]

\begin{verbatim}
int *pa = malloc((sizeof(int));
/*allocation*/
free(pa);            /*free*/
\end{verbatim}
Functions and pointers

float *Pt;

A function manipulates copies of the values of its arguments
⇒ cannot modify the value of its arguments

void function(int A, double *B)

• int A, A receives a copy of 1st arg.
• double *B, B receives a copy 2nd arg., a pointer on a double
Calling by address

<table>
<thead>
<tr>
<th>Address</th>
<th>Grid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA00</td>
<td></td>
</tr>
<tr>
<td>0xB00</td>
<td></td>
</tr>
<tr>
<td>0xC00</td>
<td></td>
</tr>
</tbody>
</table>
Calling by address

void function(int A, double *B) {...
function(a, &b);

0xA00
0xB00
0xC00
void function(int A, double *B) {...}
function(a, &b);
Calling by address

```c
void function(int A, double *B) {...}
function(a, &b);
A is a copy of `a`
B is a copy of `&b`, a pointer on `b`
```
void function(int A, double *B) {...}
function(a, &b);
A is a copy of a
B is a copy of &b, a pointer on b
⇒ modification of *B will modify
the object pointed by B, which is b
⇒ persistent modification of b
Example II

```c
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```
Example II

```c
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```

```c
b = 3;
incrementation(&b);
```
Example II

```c
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```

\[b = 3;\]
\[\text{incrementation}(&b);\]
Example II

```c
void incrementation(int *pa) {
  (*pa) = (*pa)+1;
}
```

```c
b = 3;
incrementation(&b);
```

A modification of *pa will modify b:

\[(\text{*pa}) = (\text{*pa}) + 1\]
Example II

```c
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```

```c
b = 3;  
incrementation(&b);  
A modification of *pa  
will modify b :  
(*pa) = (*pa) + 1
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0A00</td>
<td>b</td>
</tr>
<tr>
<td>0x0B00</td>
<td></td>
</tr>
<tr>
<td>0x0C00</td>
<td>pa</td>
</tr>
</tbody>
</table>

&b : 0x0A06

0xA06 4

0x0A00 b

0x0C00 pa

0x0A06
Example II

```c
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```

```c
b = 3;
incrementation(&b);
```

A modification of *pa will modify b:

```c
(*pa) = (*pa) + 1
```

at the end, b is ???

```
0x0A00
0x0B00
0x0C00

b : 0x0A06
pa : 0x01A06
```

&b : 0x0A06
Example II

void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}

b = 3;
incrementation(&b);

A modification of *pa will modify b:
(*pa) = (*pa) + 1

at the end, b is 4

b = 3; incrementation(&b);

0x0A00
0x0B00
0x0C00

b = 4
pa = 0xA06
&b = 0xA06
The array

The function manipulate only copies of its arguments

```c
void initialisation(int *tab, int n){...
initialisation(T,l);
⇒ the value of an array T is a pointer on the first cell (address of the first cell)
```

\( T \) is a memory address
#define N 10
int *initialisation() {
    int T[N];
    int i;
    for (i=0; i<N; i++) T[i] = 0;
    return T;
}
Permanent Allocation

#define N 10
int *initialisation() {
  int T[N];
  int i;
  for (i=0; i<N; i++) T[i] = 0;
  return T;
}

Array T is a local variable of the function, and it is free at the end of the function.
Permanent Allocation

```c
#define N 10
int *initialisation() {
    int T[N];
    int i;
    for (i=0; i<N; i++) T[i] = 0;
    return T;
}
```

Array T is a local variable of the function, and it is free at the end of the function

```c
int *initialisation(int n) {
    int i;
    int *T = malloc(n * sizeof(int));
    for (i=0; i<n; i++) T[i] = 0;
    return T;
}
```
Call by value → Call by address

Let a function `addition` that adds the second argument to the first.

```c
void addition(int a, int b)
{
    a = a + b;
}
```
Call by value → Call by address

Let a function addition that adds the second argument to the first

```c
void addition(int a, int b)
{
    a = a + b;
}
```

Calling by value: 1st argument unchanged
⇒ Calling by address of 1st argument
Call by value → Call by address

Let a function `addition` that adds the second argument to the first:

```c
void addition(int a, int b)
{
    a = a + b;
}
```

Calling by value: 1st argument unchanged
⇒ Calling by address of 1st argument:

```c
void addition(int *pa, int b)
{
    (*pa) = (*pa) + b;
}
```
Conclusion

Pointer

Dynamic Allocation

• allocation of memory for pointers
• variable size array
• array with n dimensions
• dynamic free