Pointers, memory allocation and calling a function by address

Summary

1 - Pointers

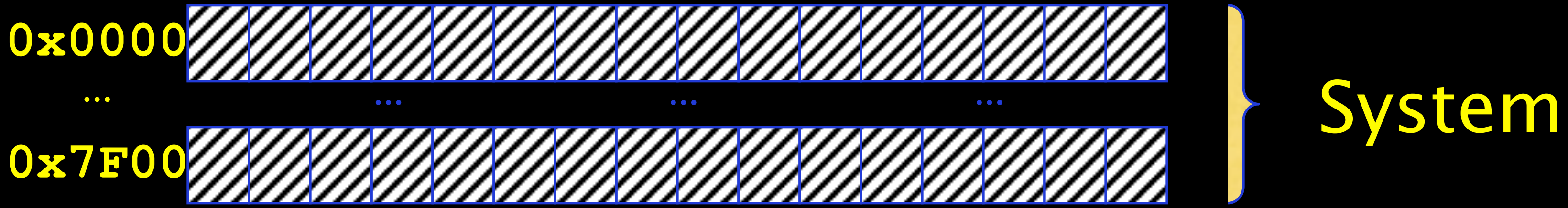
2 - Dynamic Allocation

3 - Array with many dimensions

4 - Calling by address

Memory

Memory



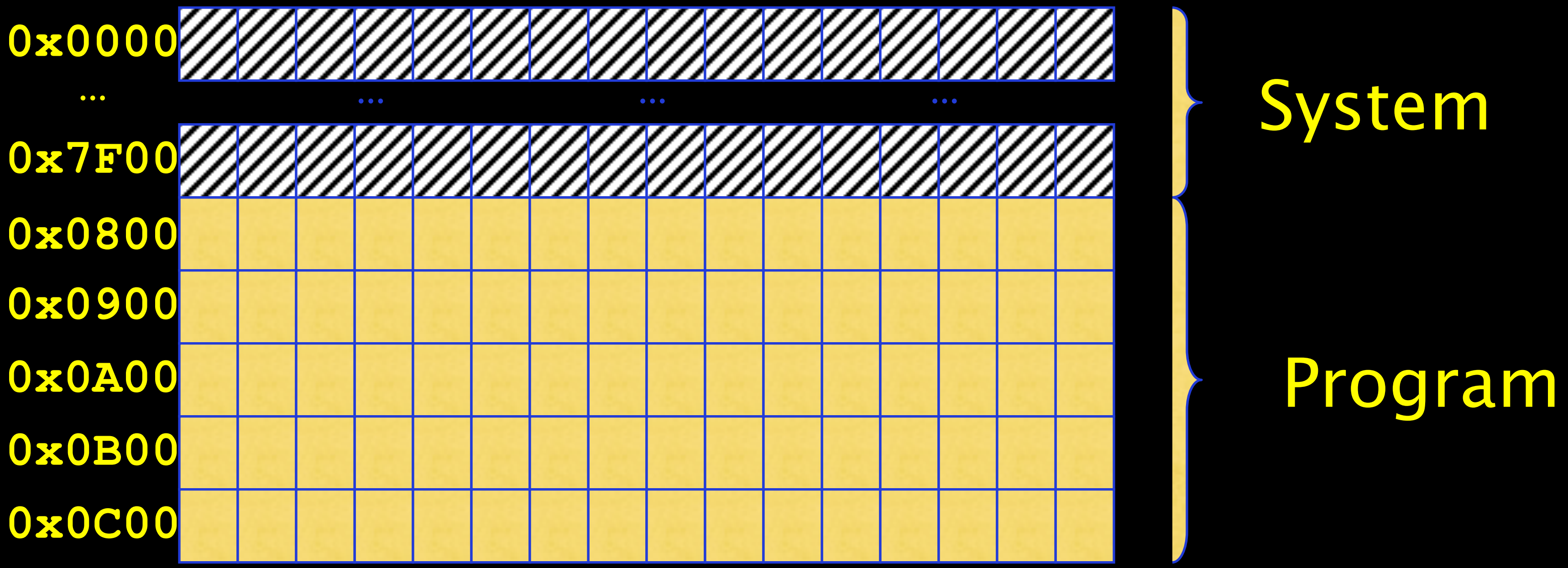
`0x0000`

…

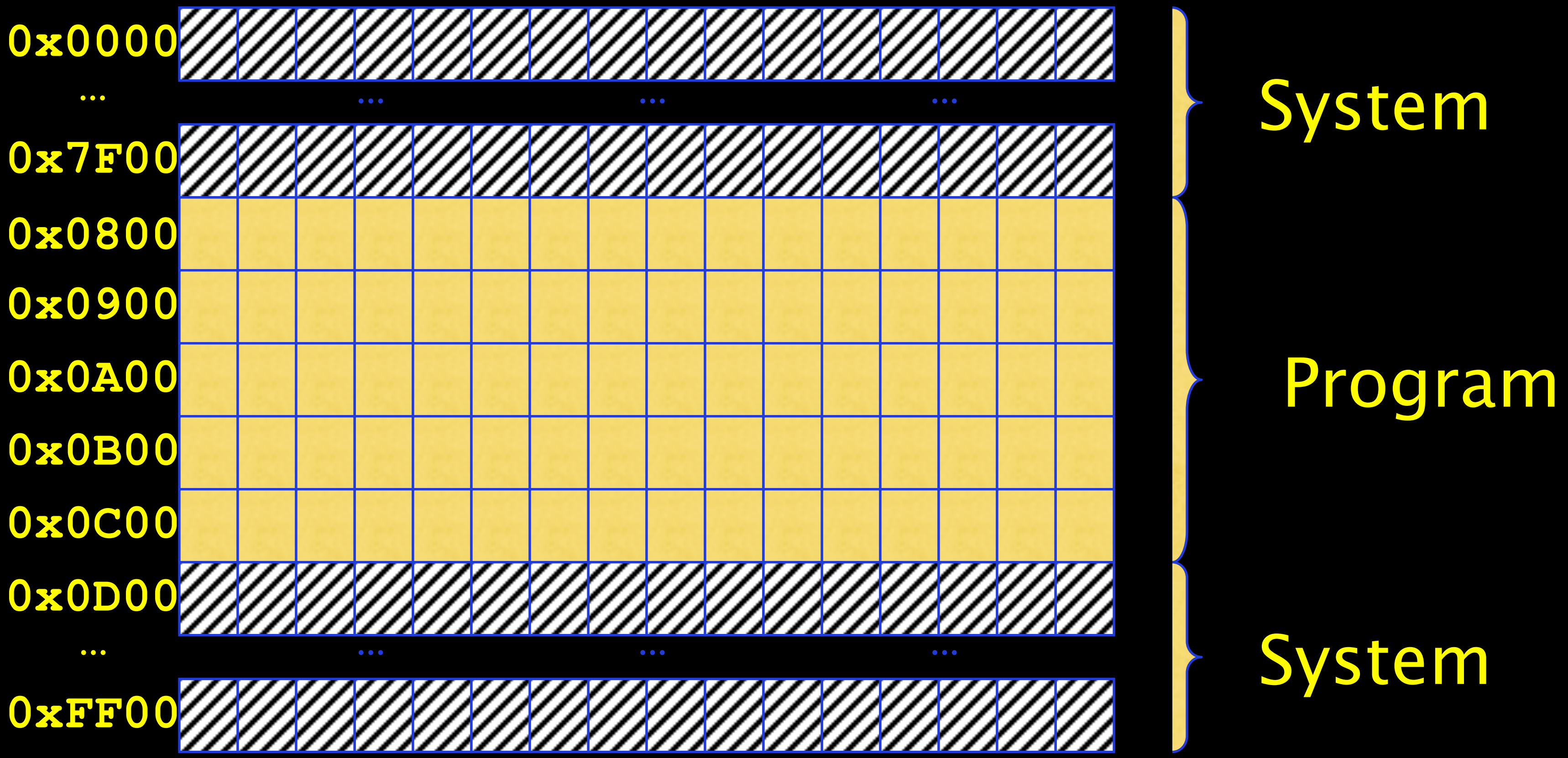
`0x7F00`

System

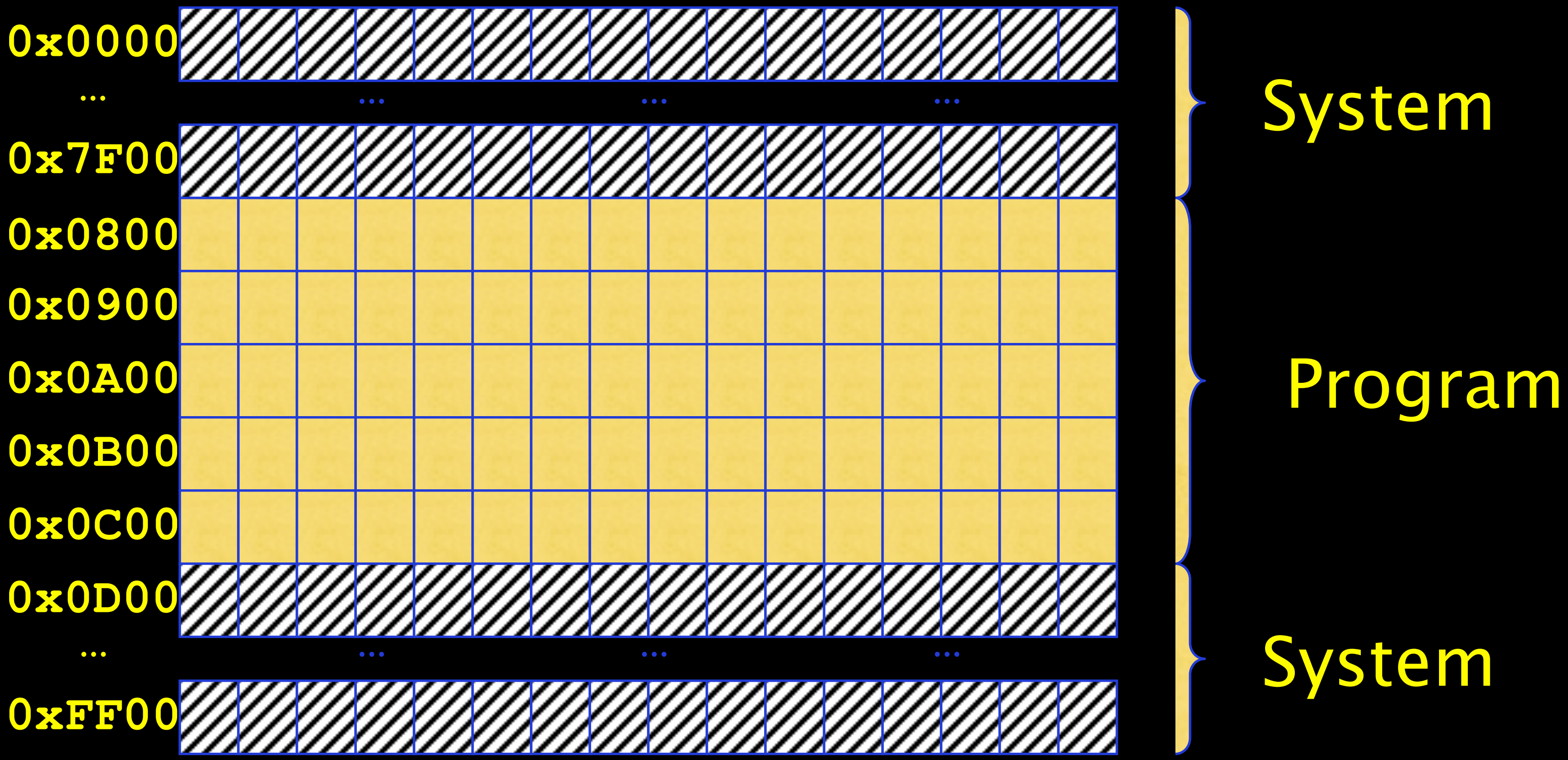
Memory


0x0000
…
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
…
…
…
System
Program

Memory


0x0000
…
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
…
0xFF00
System
Program
System

Memory



0x0000

…

0x7F00

0x0800

0x0900

0x0A00

0x0B00

0x0C00

0x0D00

…

0xFF00

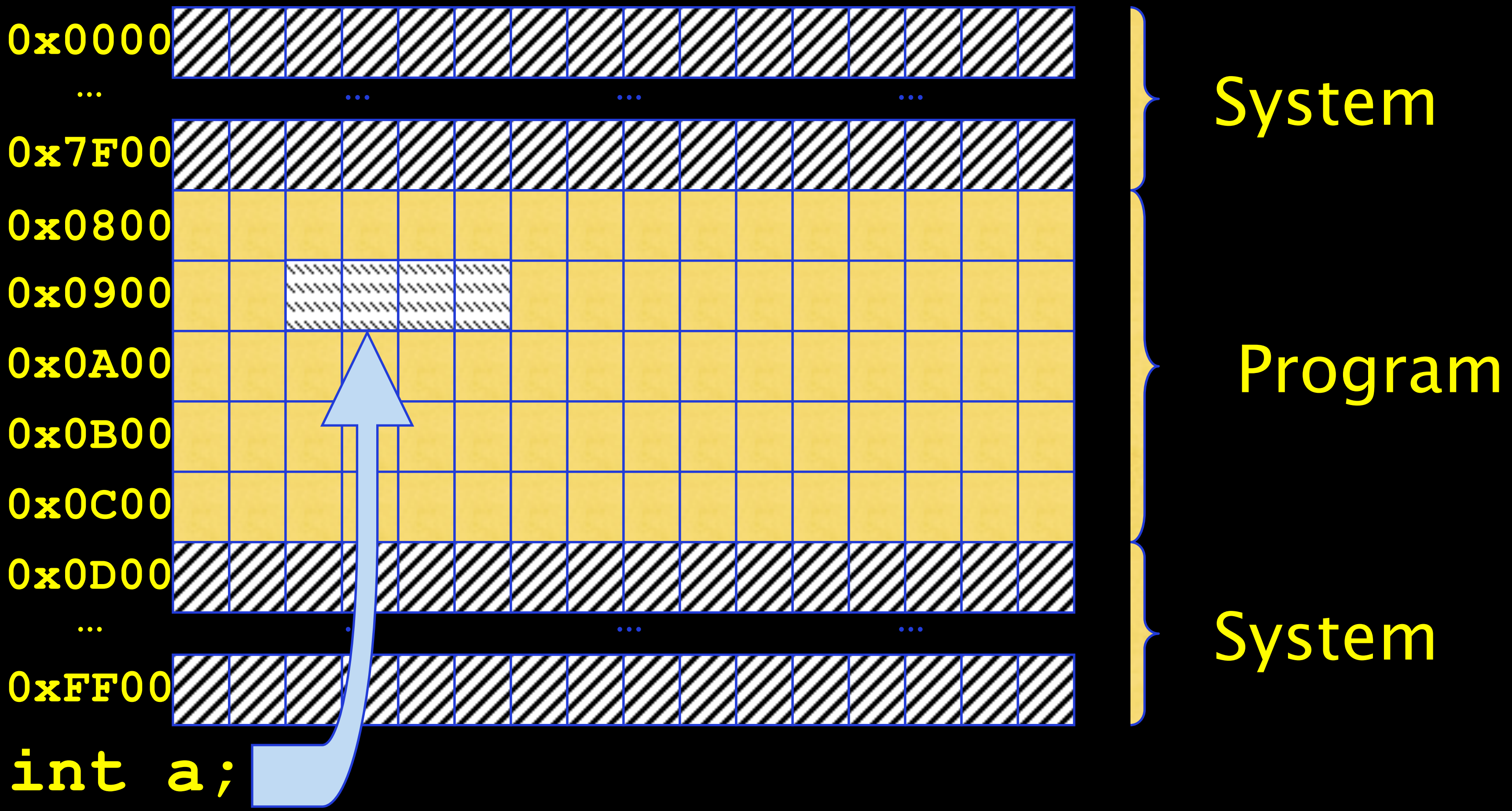
System

Program

System

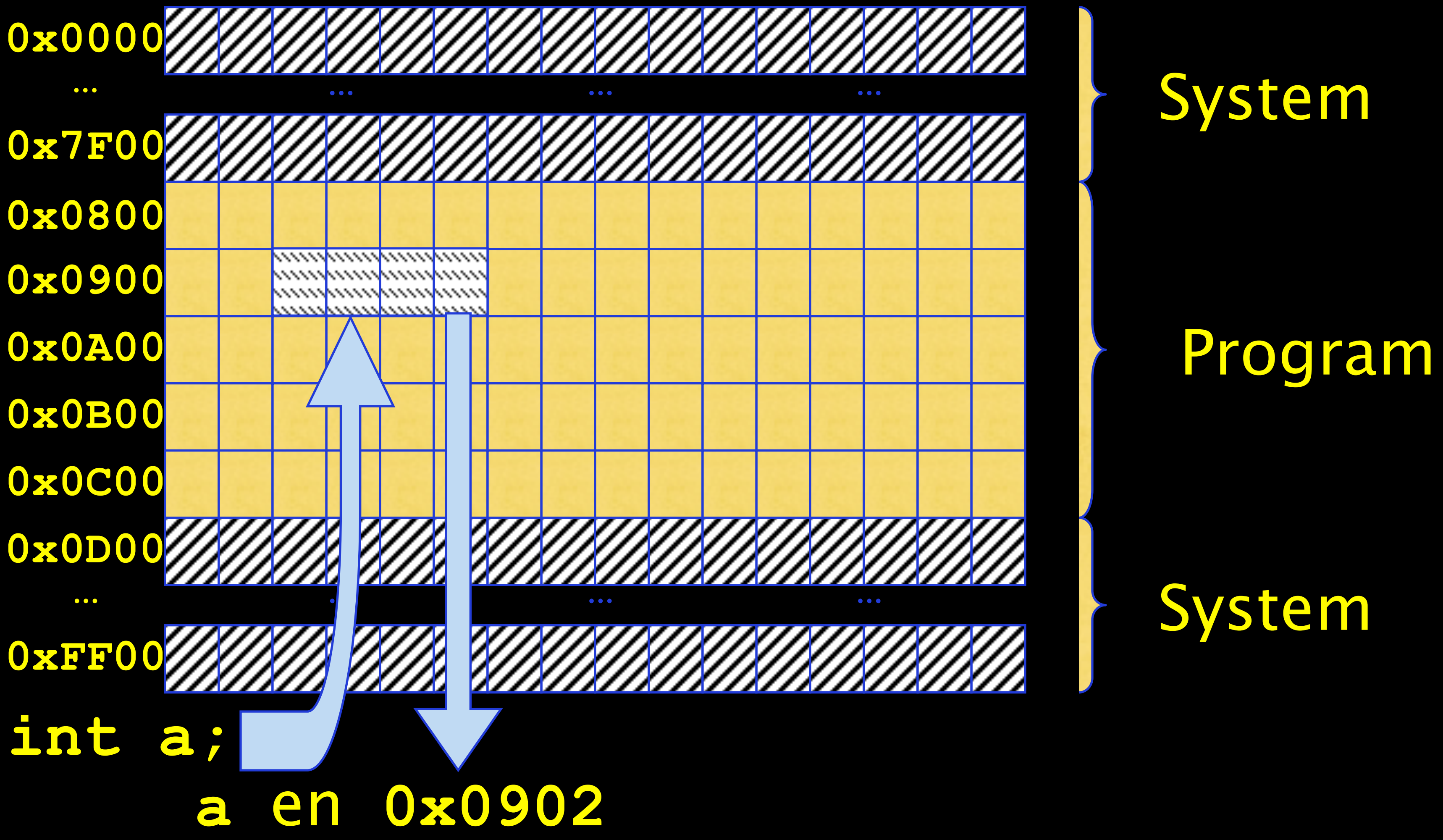
```
int a;
```

Memory

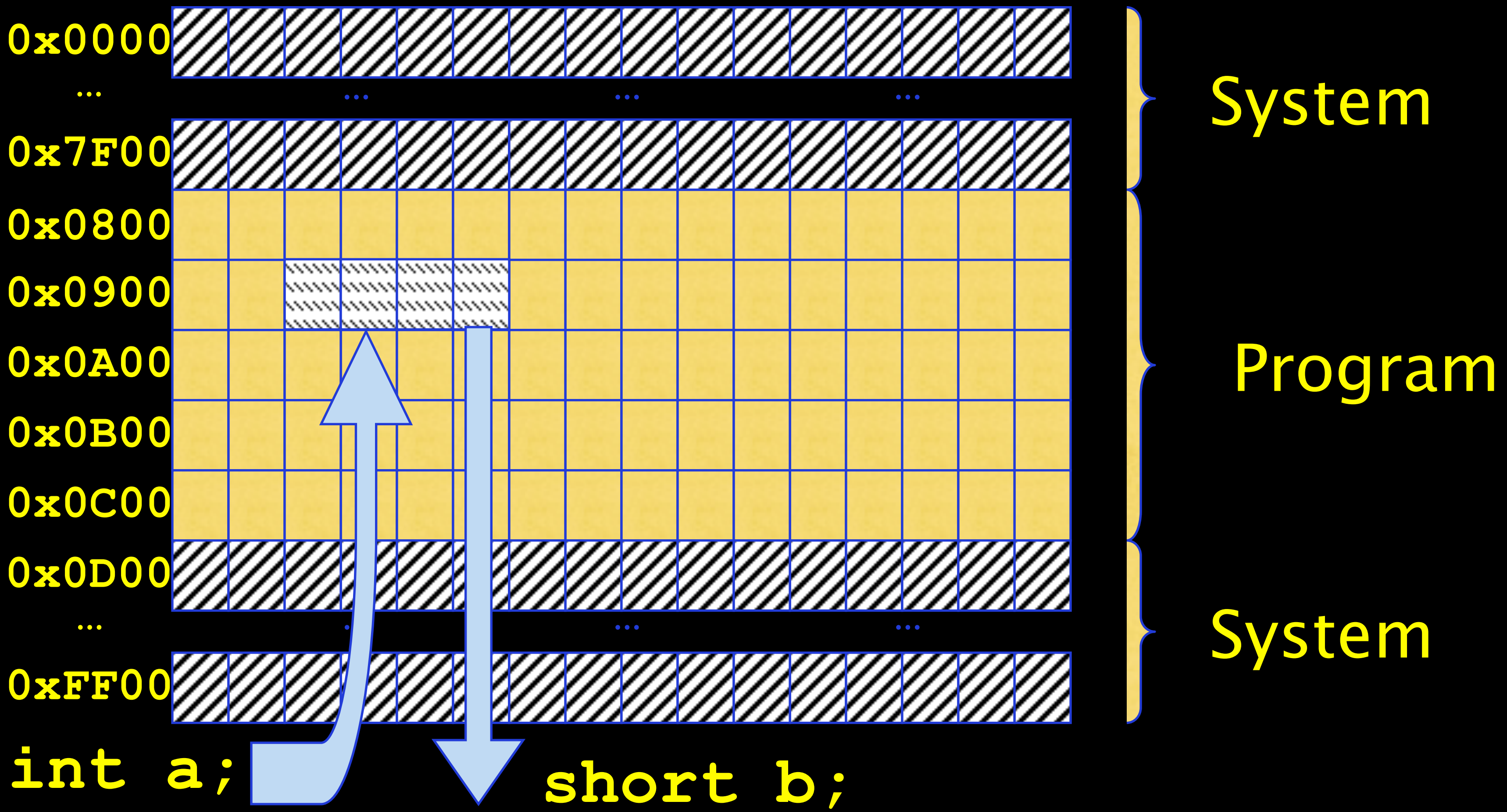

0x0000
…
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
…
0xFF00
…
…
…
…
…
…
System
Program
System

`int a;`

Memory


0x0000
…
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
…
0xFF00
System
Program
System
int a;
a en 0x0902

Memory



0x0000
…
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
…
0xFF00

System

Program

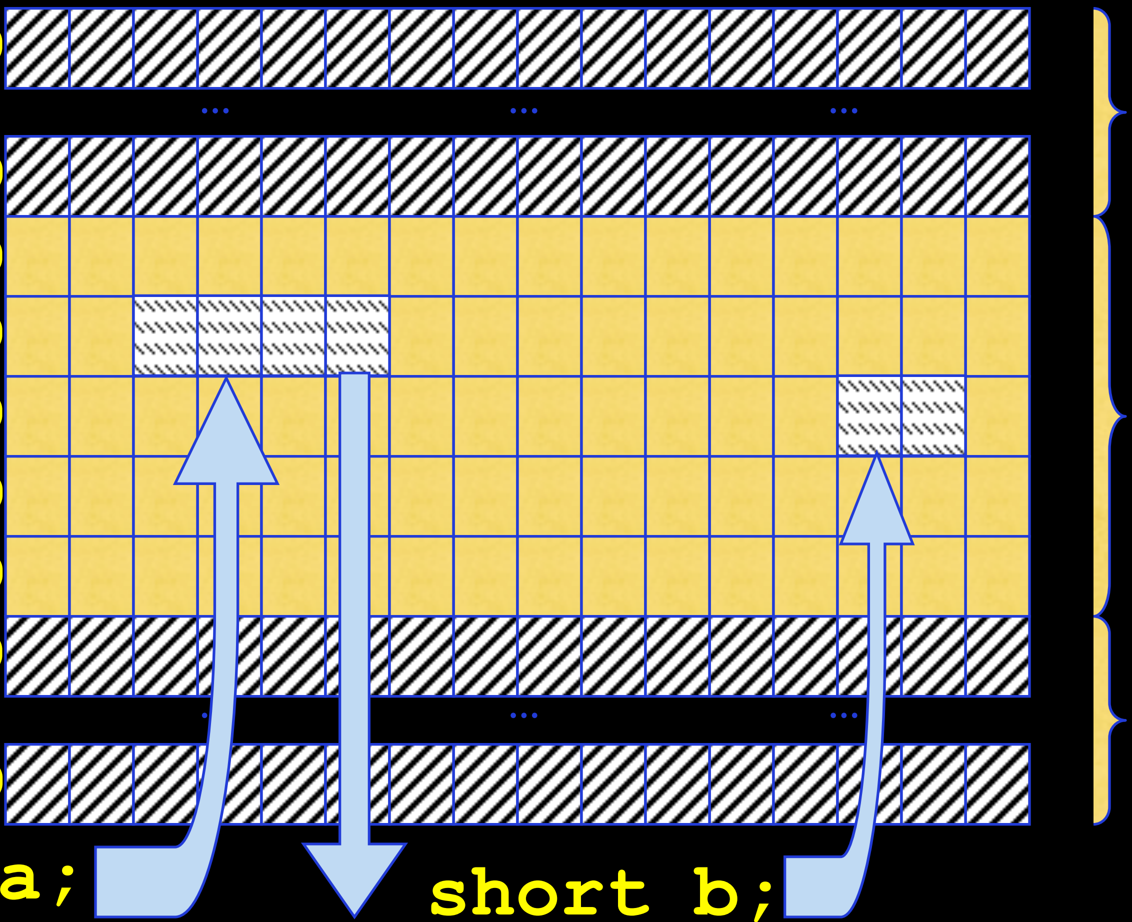
System

`int a;`

`short b;`

`a en 0x0902`

Memory



0x0000
…
0x7F00
0x0800
0x0900
0x0A00
0x0B00
0x0C00
0x0D00
…
0xFF00

System

Program

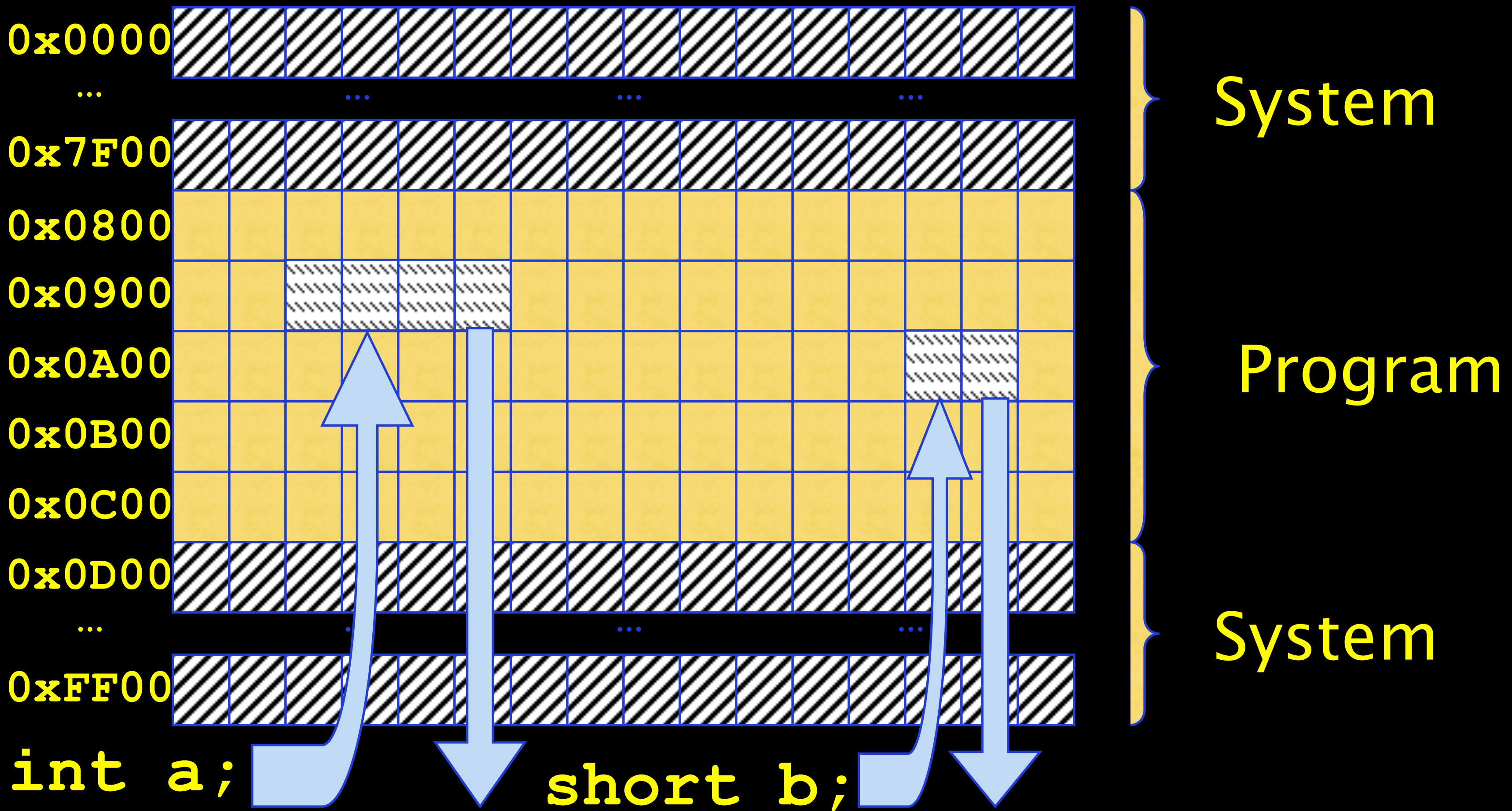
System

`int a;`

`short b;`

`a en 0x0902`

Memory

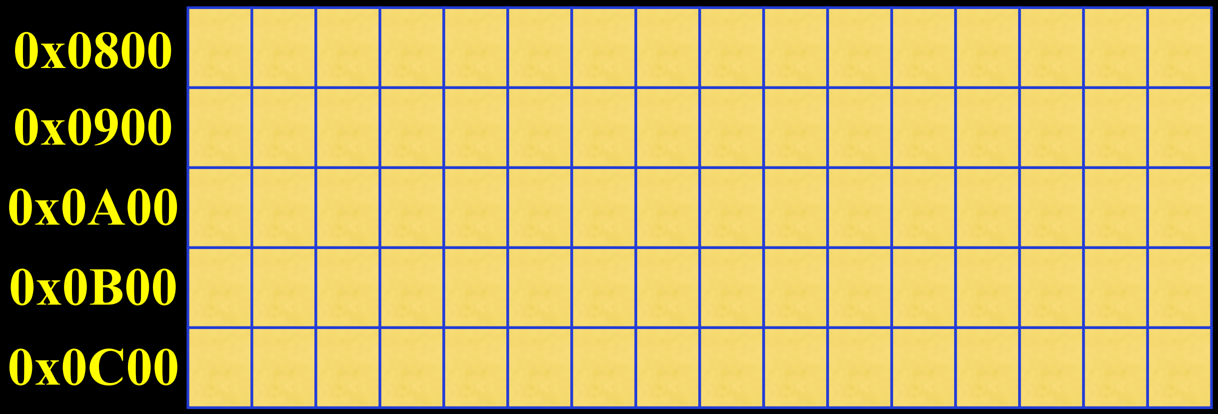


```
int a;
```

```
short b;
```

a en 0x0902 b en 0x0A0D

Pointer



0x0800

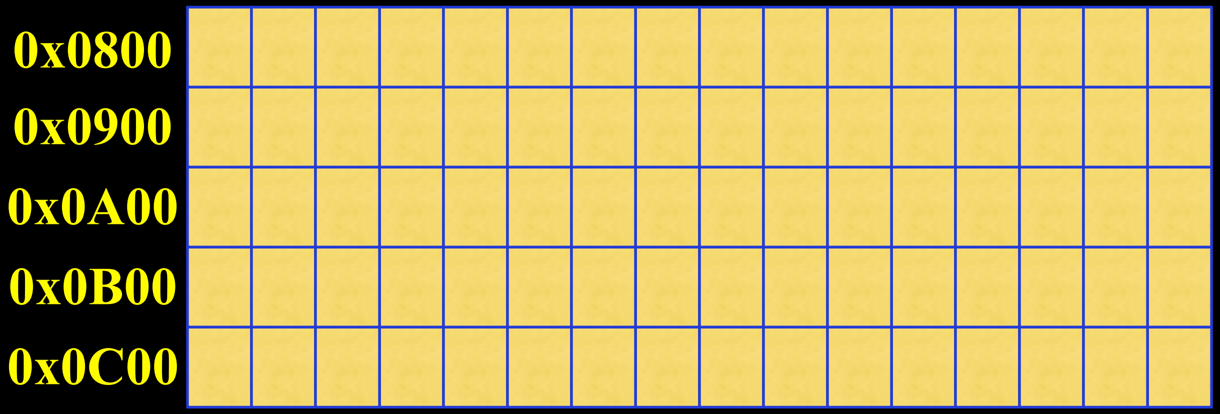
0x0900

0x0A00

0x0B00

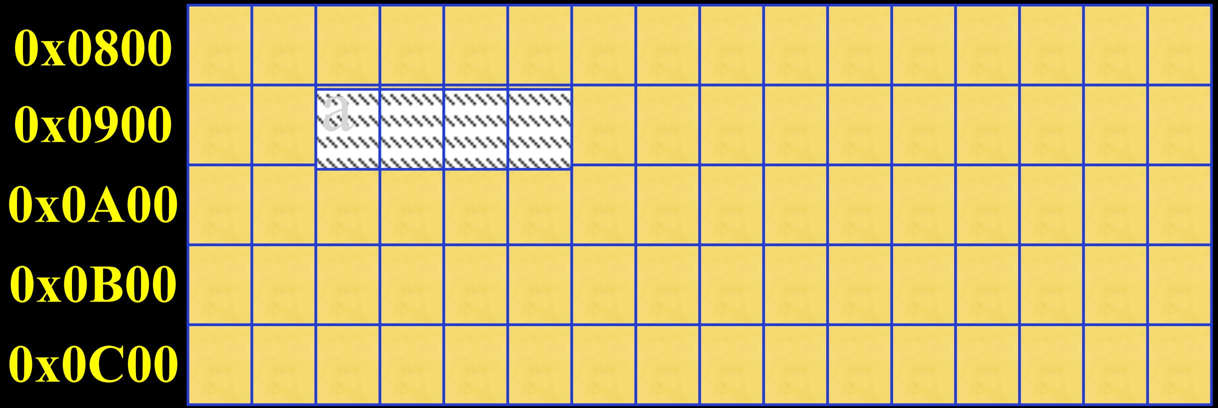
0x0C00

Pointer



A variable is stored in a memory area when it is declared

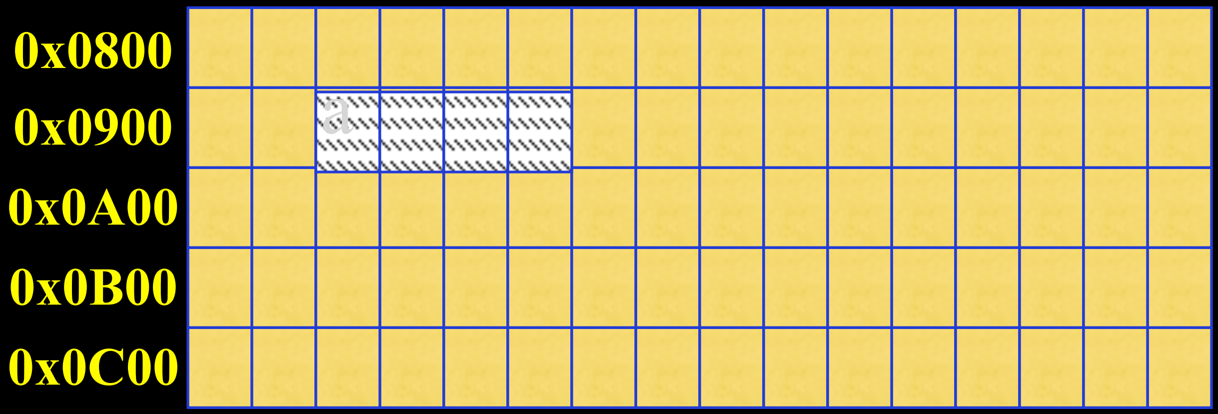
Pointer



A variable is stored in a memory area when it is declared

```
int a;
```

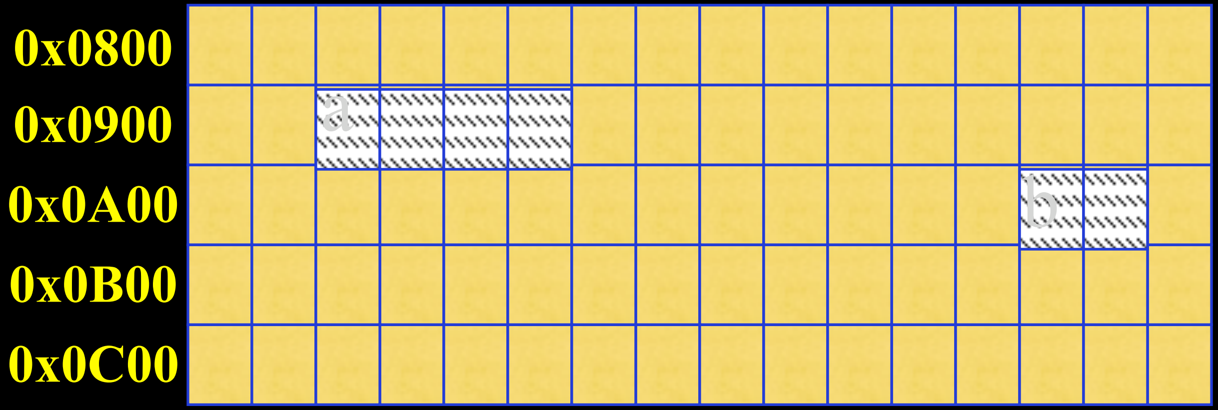
Pointer



A variable is stored in a memory area when it is declared

`int a;` 0x0902 pointer onto `(int) a`

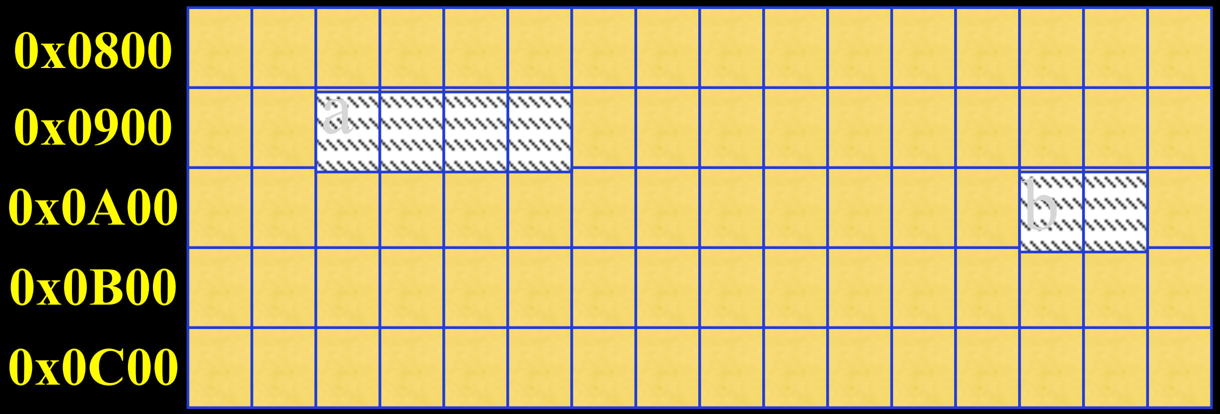
Pointer



A variable is stored in a memory area when it is declared

`int a;` 0x0902 pointer onto `(int) a`
`short b;`

Pointer

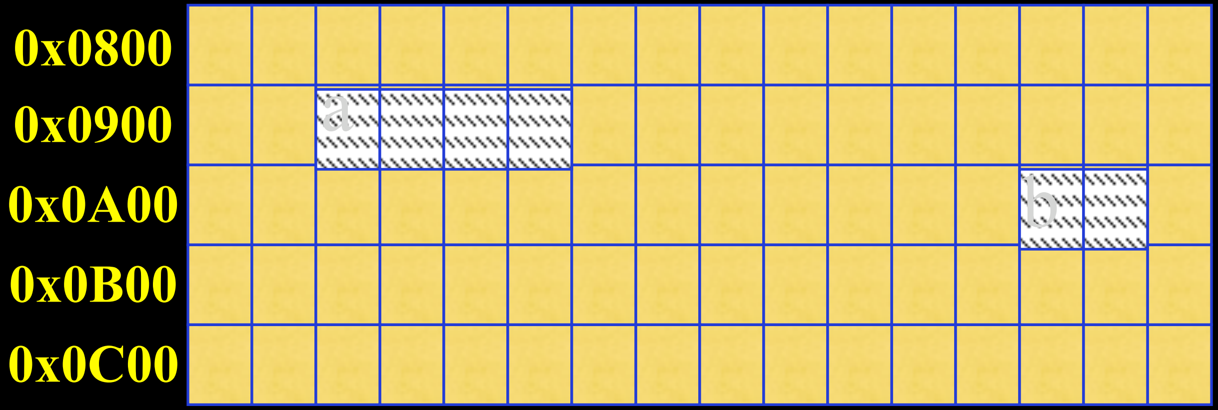


A variable is stored in a memory area when it is declared

`int a;` 0x0902 pointer onto `(int) a`

`short b;` 0x0A0D pointer onto `(short) b`

Pointer



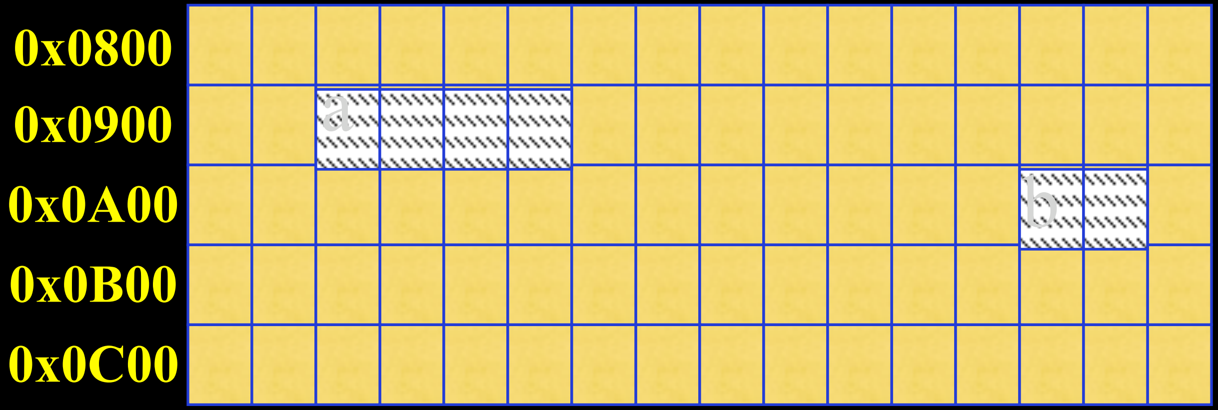
A variable is stored in a memory area when it is declared

& reference operateur (address)

`int a;` 0x0902 pointer onto `(int) a`

`short b;` 0x0A0D pointer onto `(short) b`

Pointer



`&a : 0x0902`

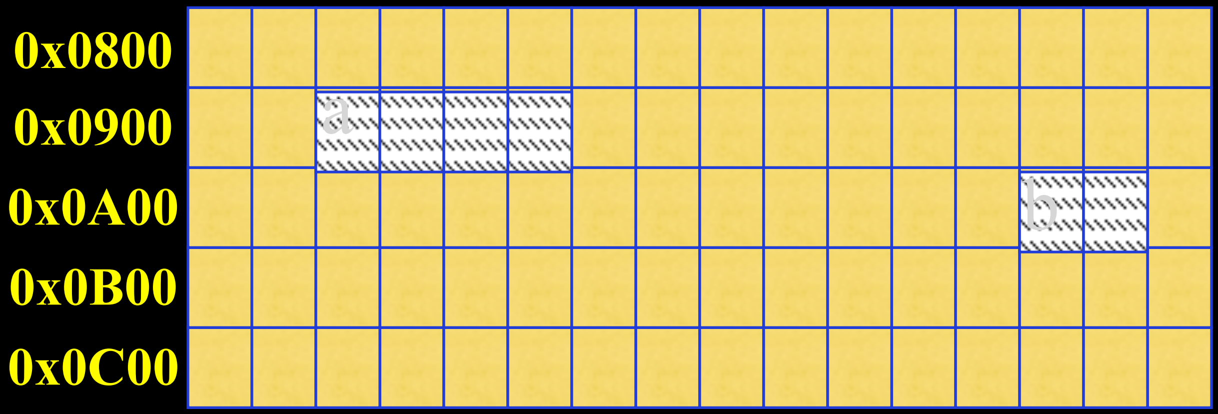
A variable is stored in a memory area when it is declared

& reference operateur (address)

`int a;` 0x0902 pointer onto `(int) a`

`short b;` 0x0A0D pointer onto `(short) b`

Pointer



`&a : 0x0902`

`&b : 0x0A0D`

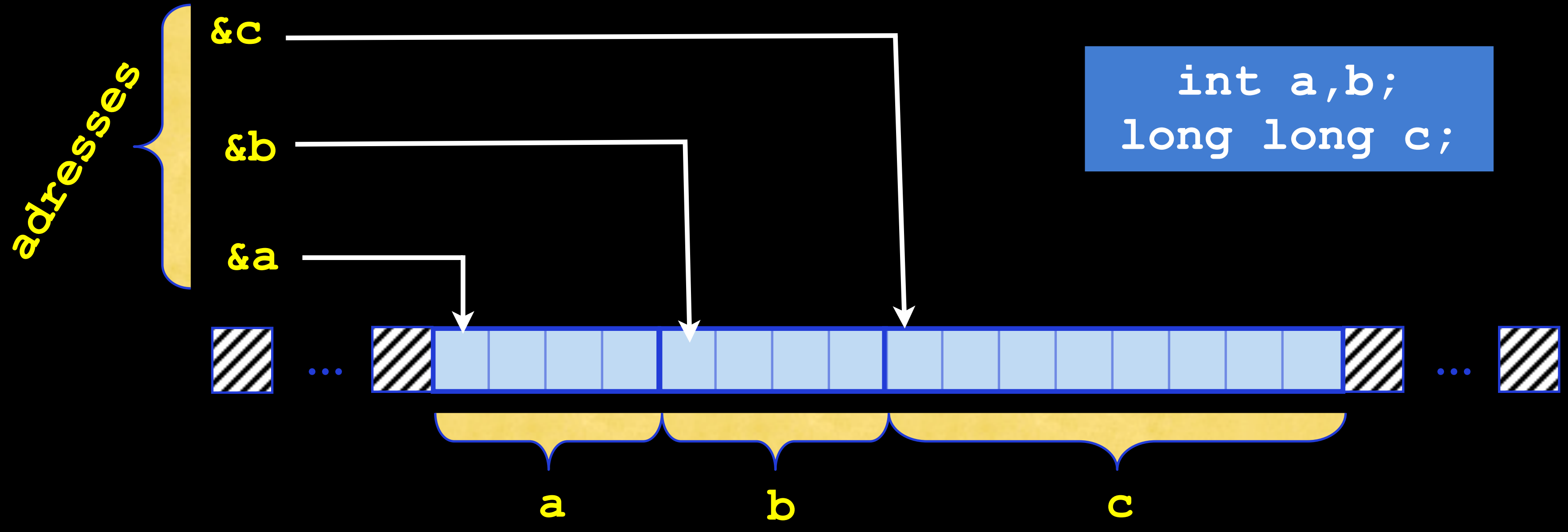
A variable is stored in a memory area when it is declared

& reference operateur (address)

`int a;` 0x0902 pointer onto `(int) a`

`short b;` 0x0A0D pointer onto `(short) b`

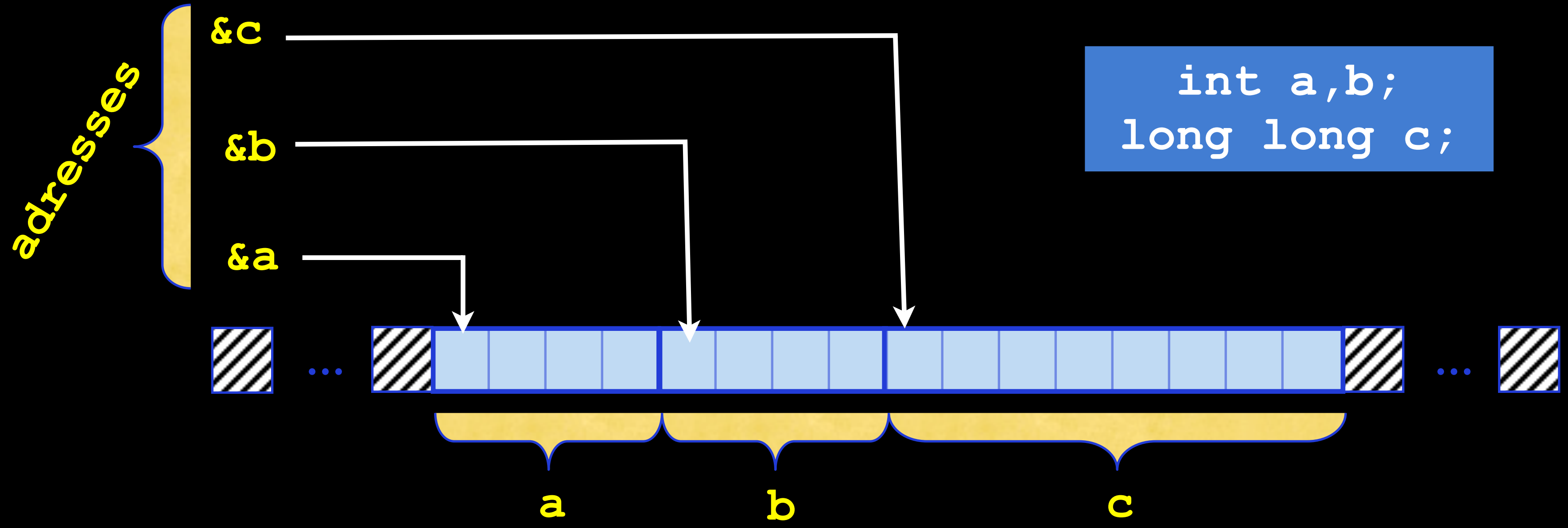
Pointers = Addresses



```
int a,b;
long long c;
```

Pointers = Addresses

& address operateur (its reference)

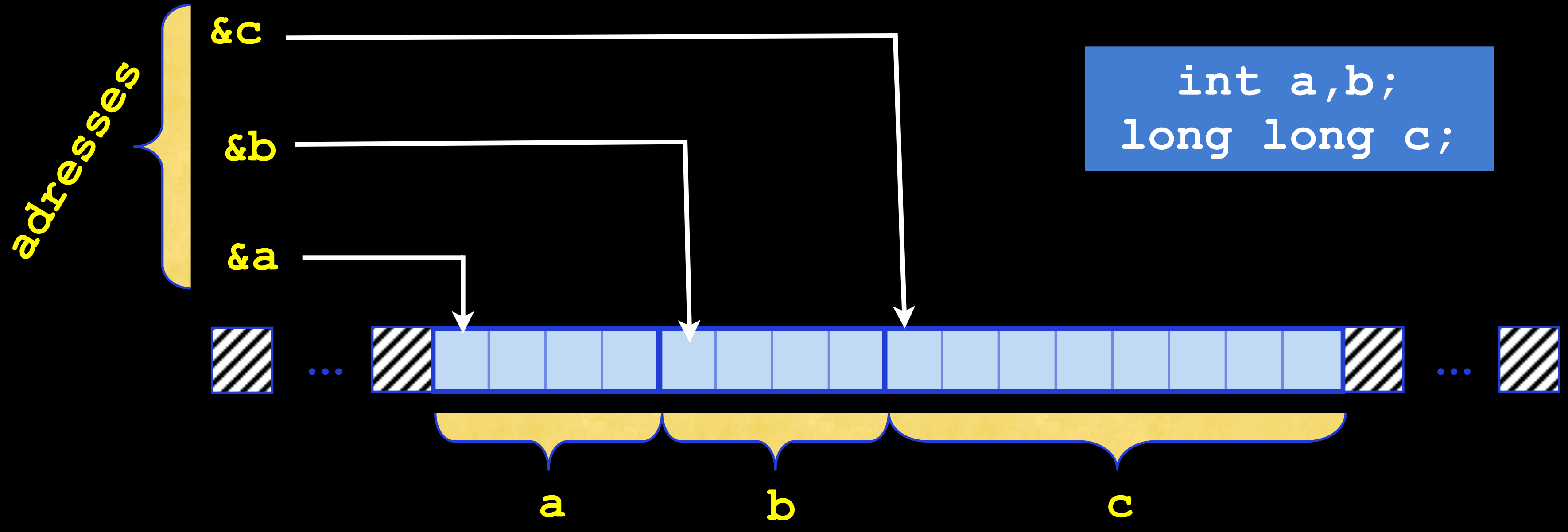


```
int a,b;
long long c;
```

Pointers = Addresses

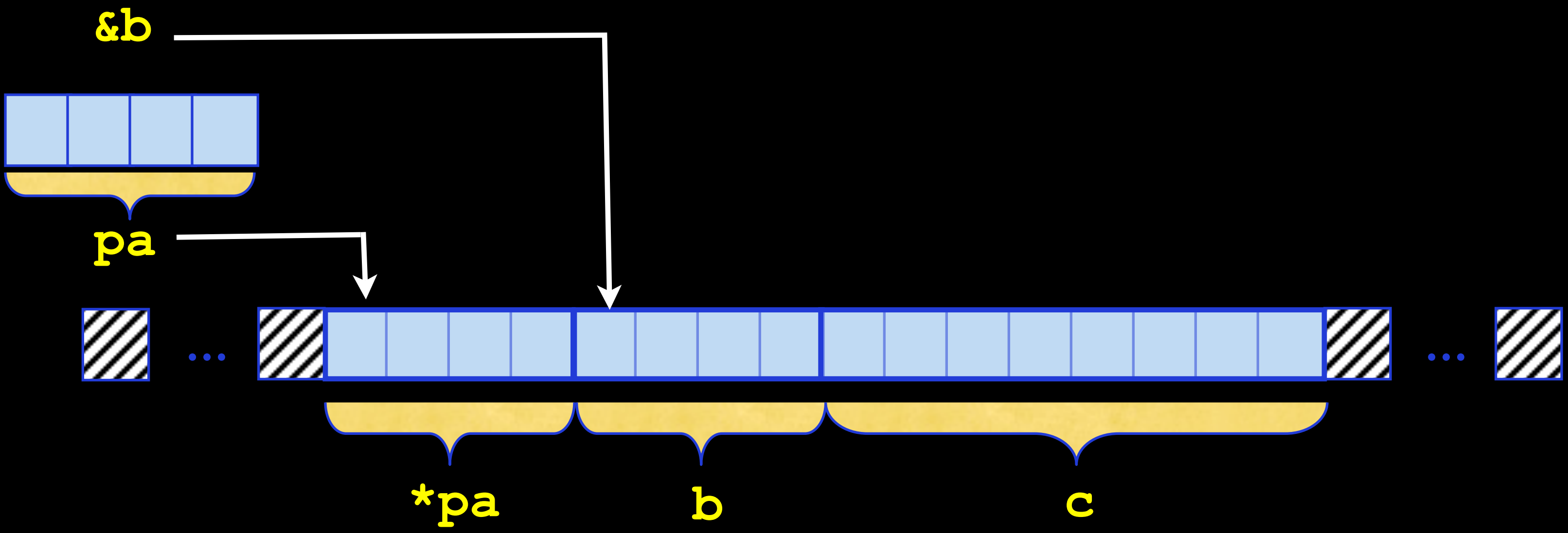
`&` address operateur (its reference)

`&a` returned the memory address of the variable `a`



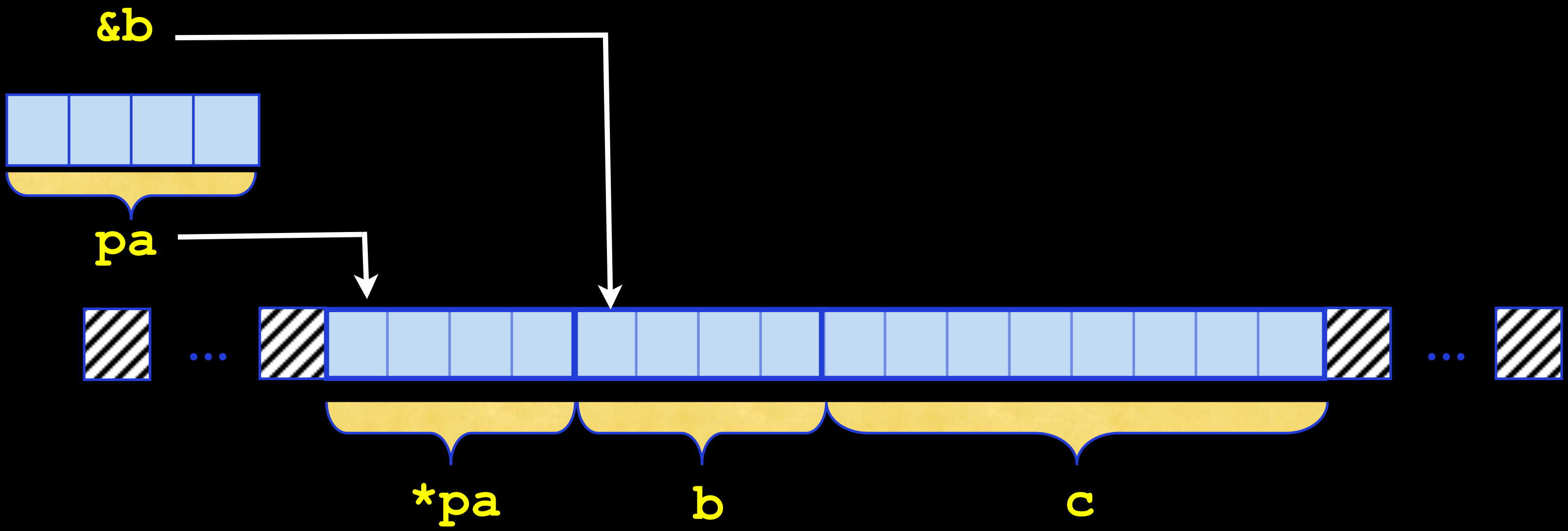
```
int a,b;
long long c;
```

Value Operation



Value Operation

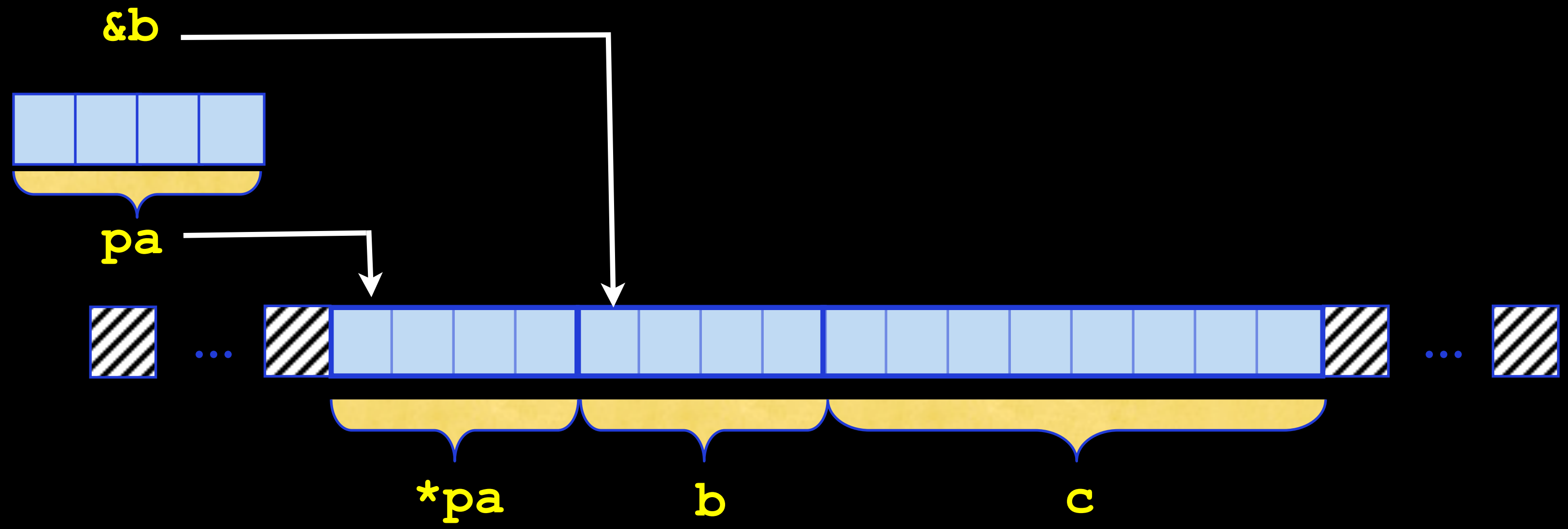
`pa` pointer onto an integer `int *pa;`



Value Operation

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

`*` operation of dereference (value)

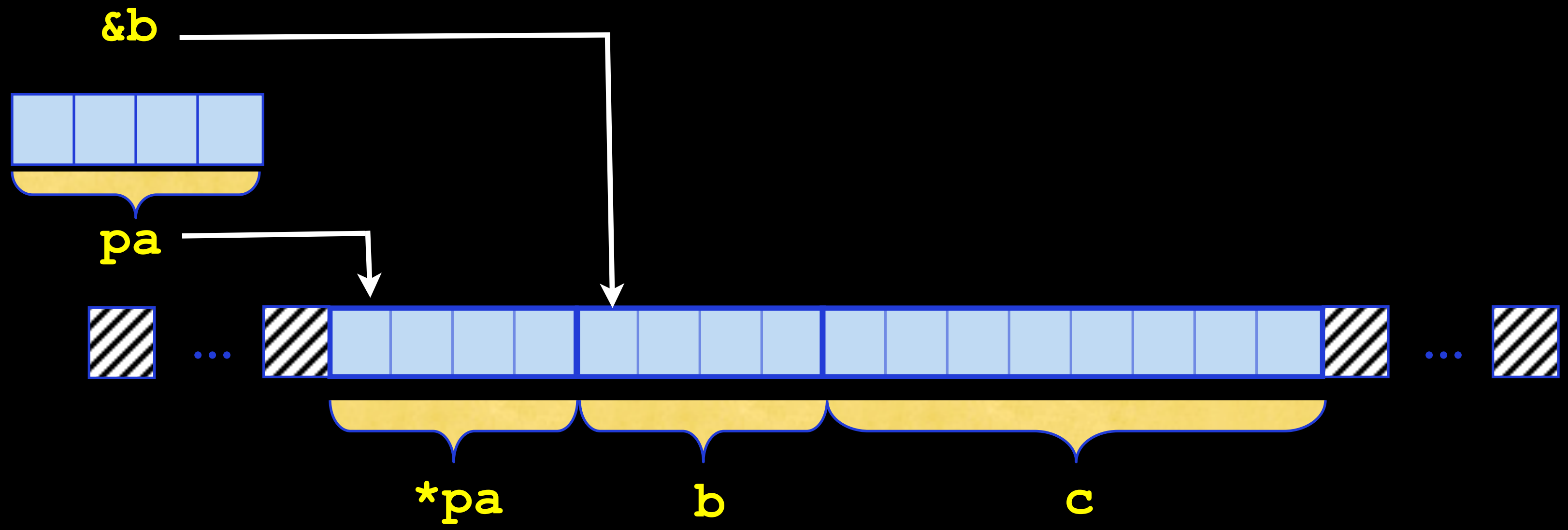


Value Operation

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

`*` operation of dereference (value)

`*pa` represent the memory at address **`pa`**

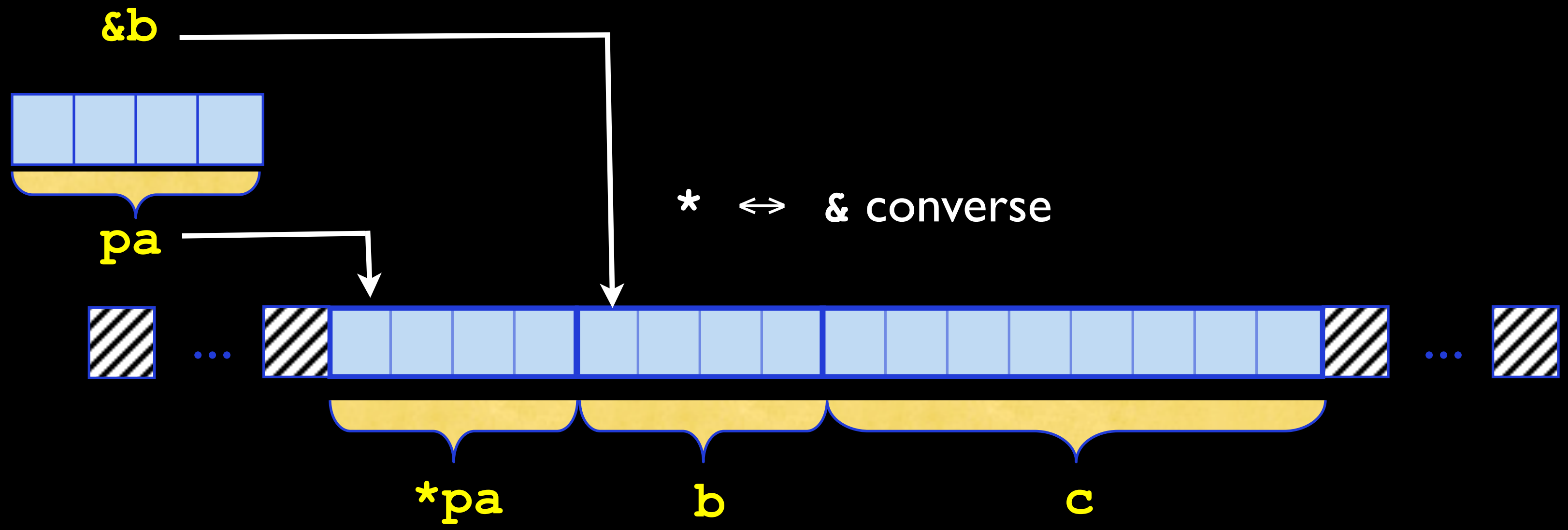


Value Operation

`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`**

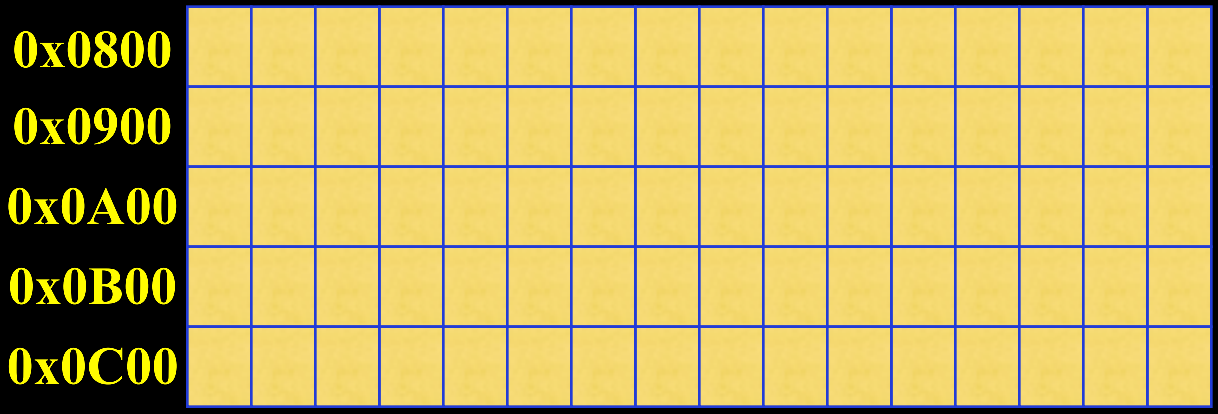
`float *pb;`

⇒ **`pb`** pointer onto a **`float`**

`char *pc;`

⇒ **`pc`** pointer onto a **`char`**

Somes examples



0x0800

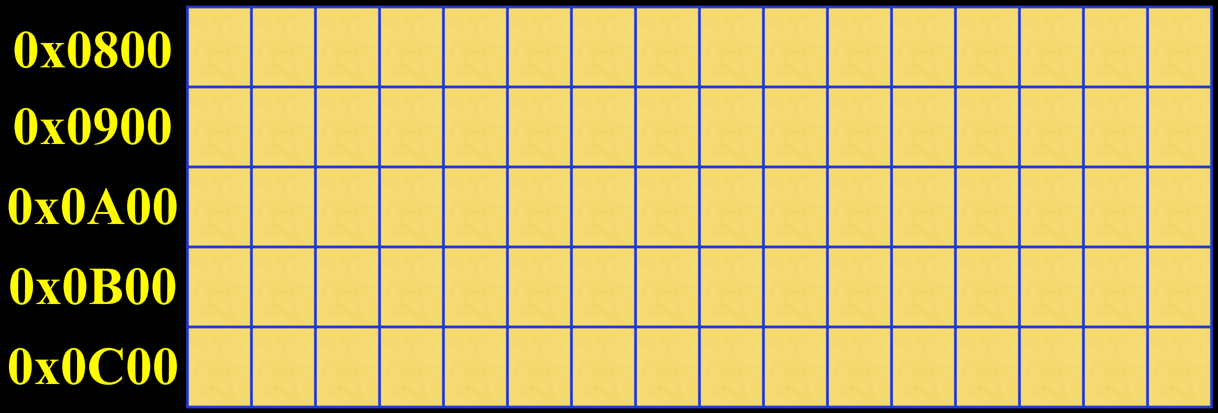
0x0900

0x0A00

0x0B00

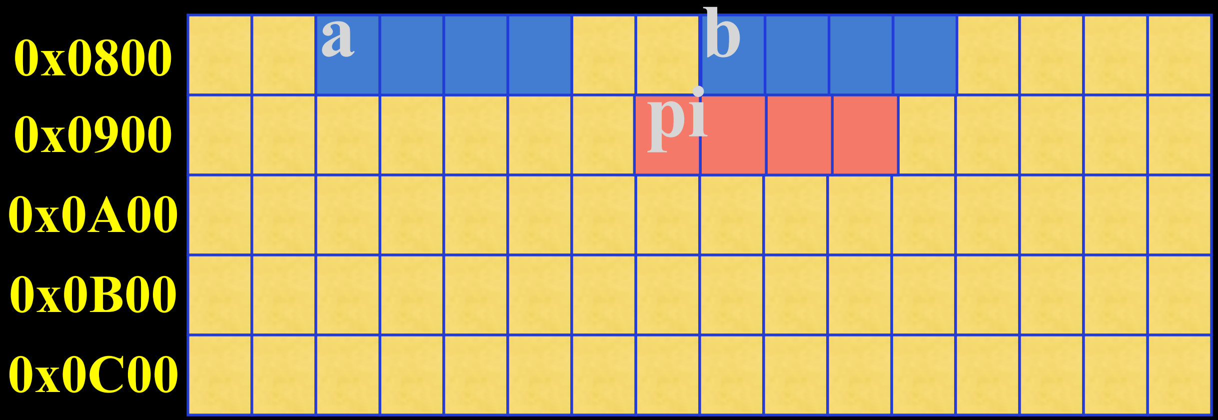
0x0C00

Somes examples



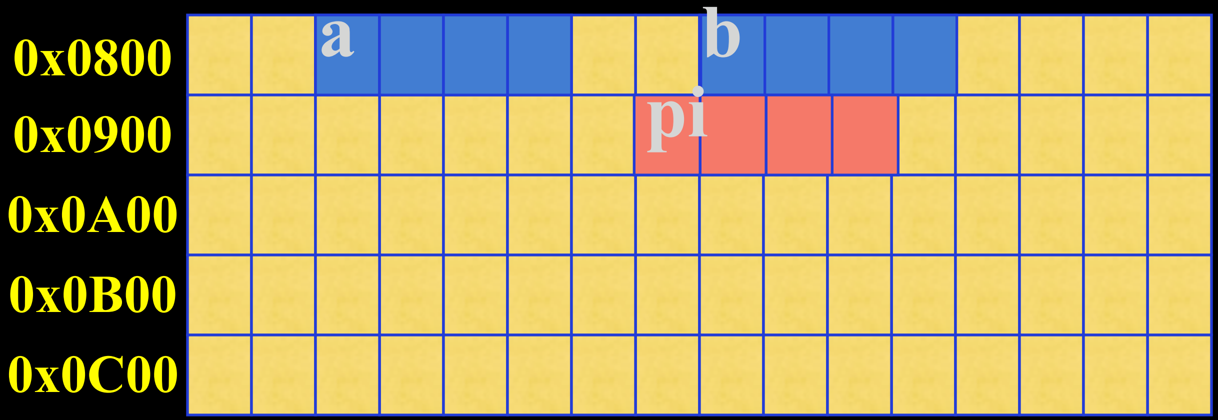
```
int a,b,*pi;
```

Somes examples



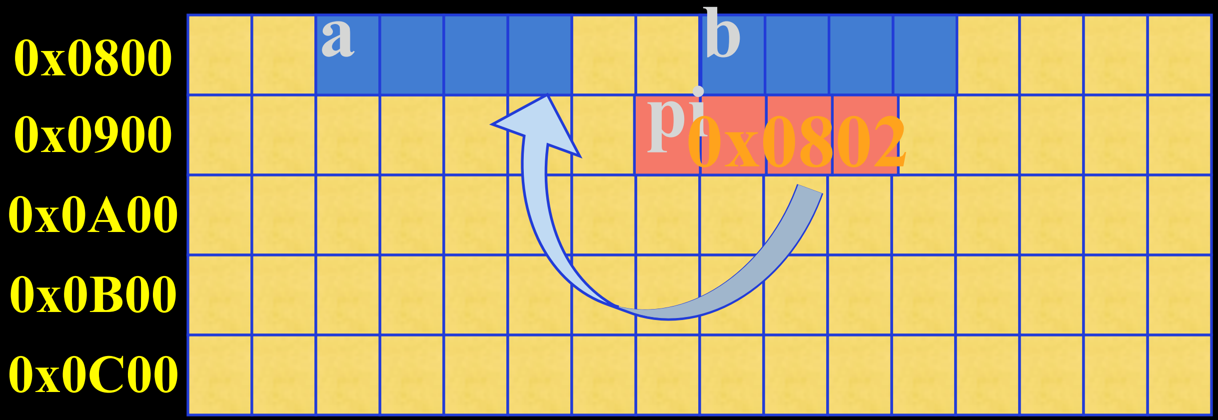
```
int a,b,*pi;
```

Somes examples



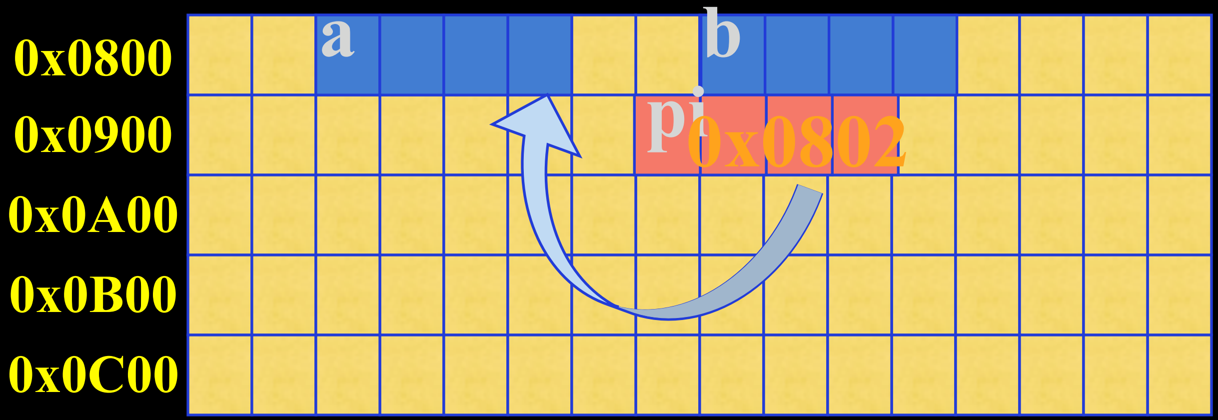
```
int a,b,*pi;
 pi = &a;
```

Somes examples



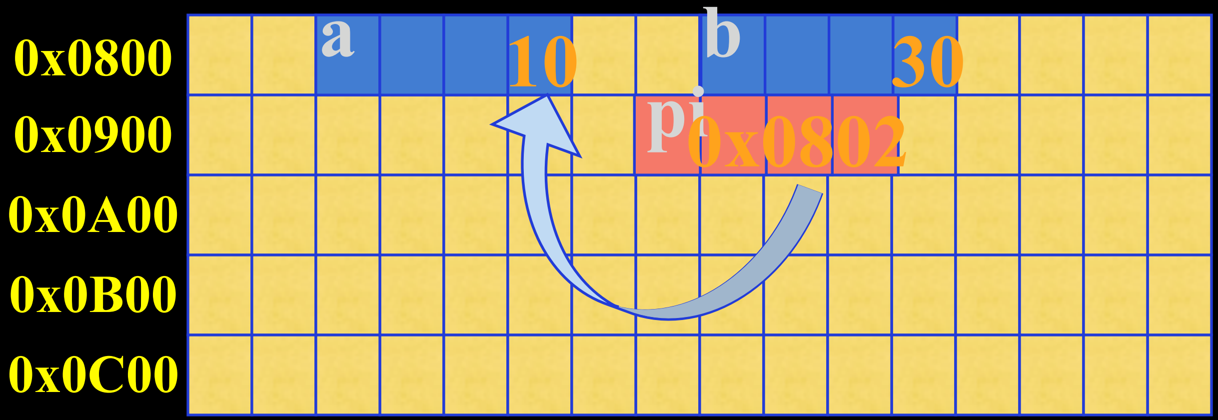
```
int a,b,*pi;
pi = &a;
```

Somes examples



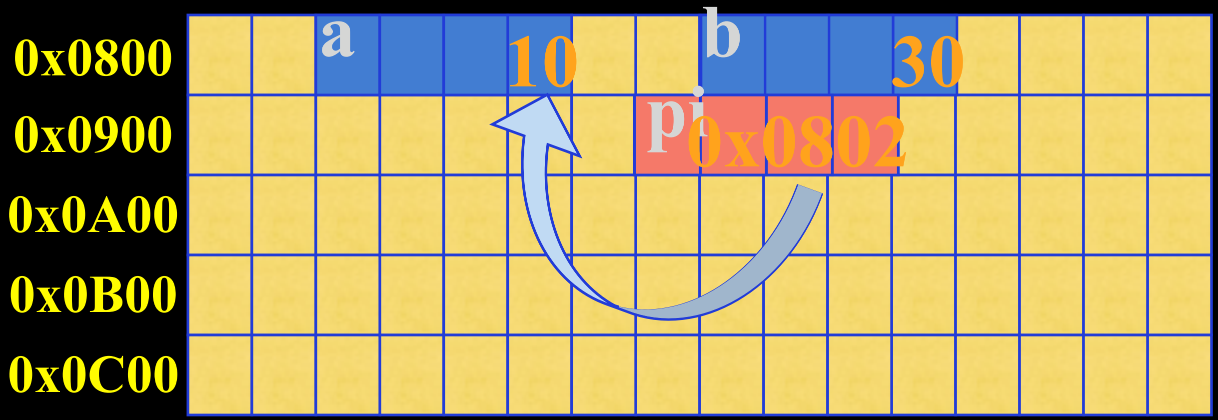
```
 int a,b,*pi;
 pi = &a;
a = 10; b = 30;
```

Somes examples



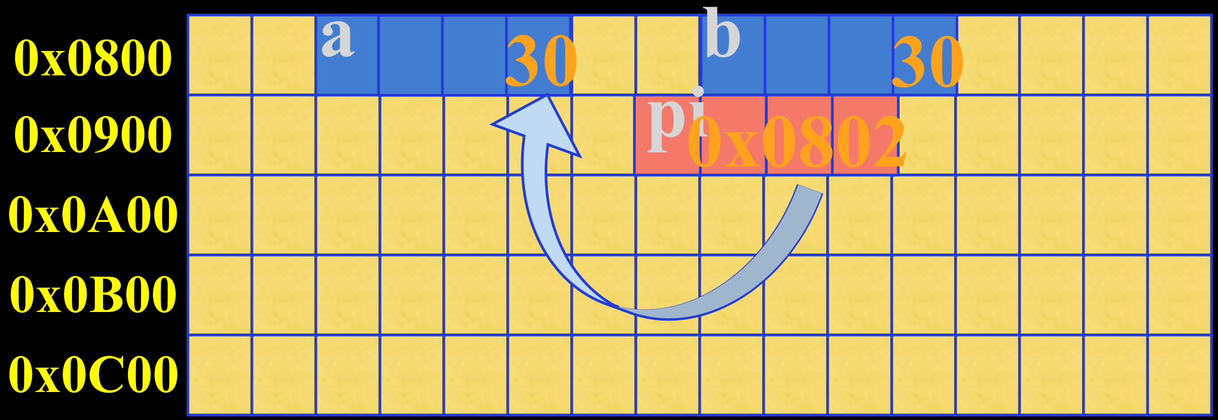
```
int a,b,*pi;
pi = &a;
a = 10; b = 30;
```

Somes examples



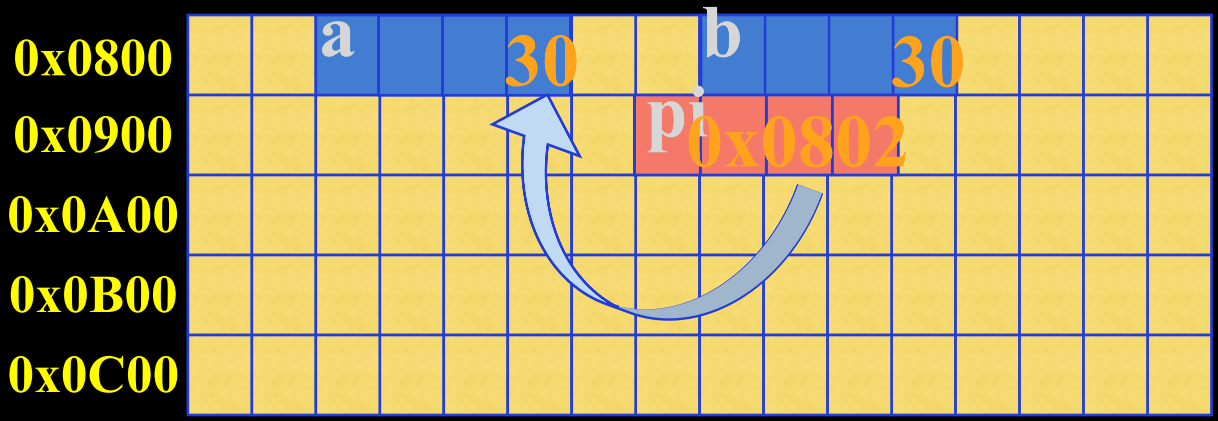
```
int a,b,*pi;
 pi = &a;
a = 10; b = 30;
*pi = b;
```

Somes examples



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

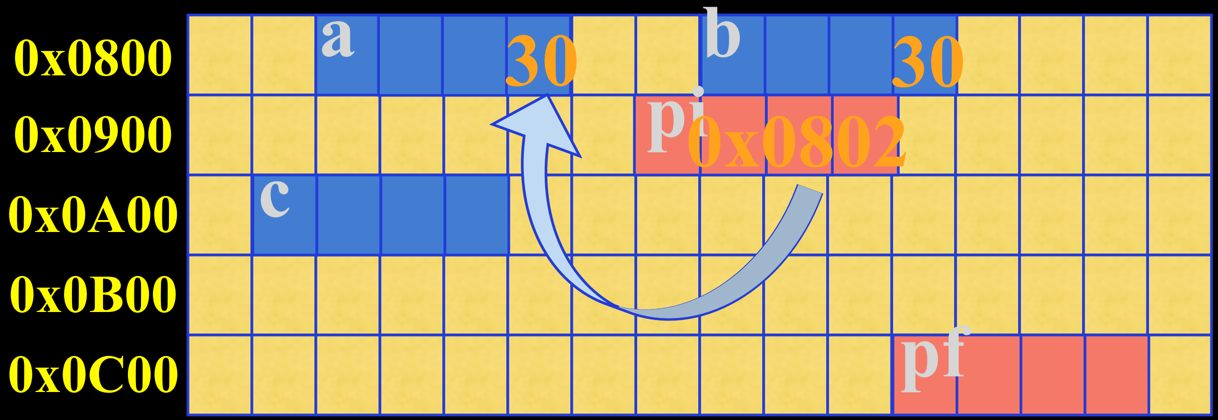
Somes examples



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

```
float c,*pf;
```

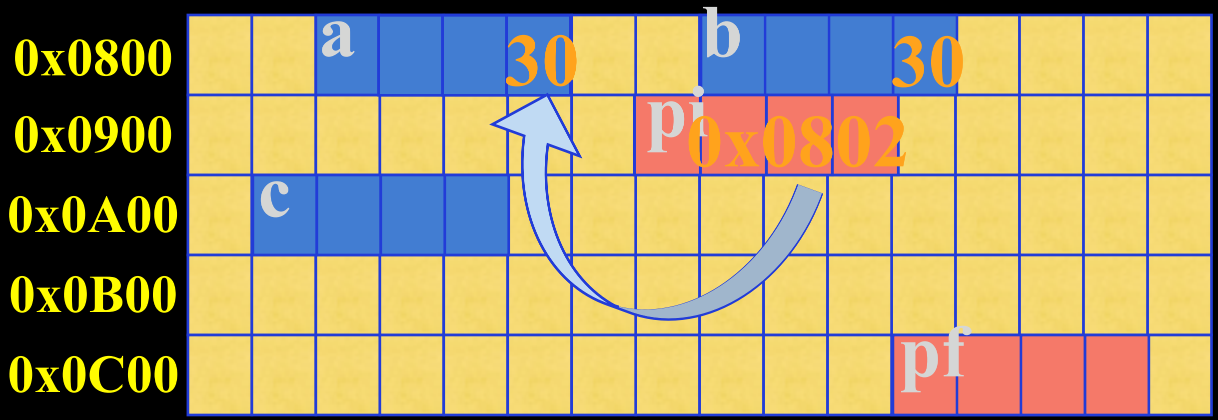
Somes examples



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

```
float c,*pf;
```

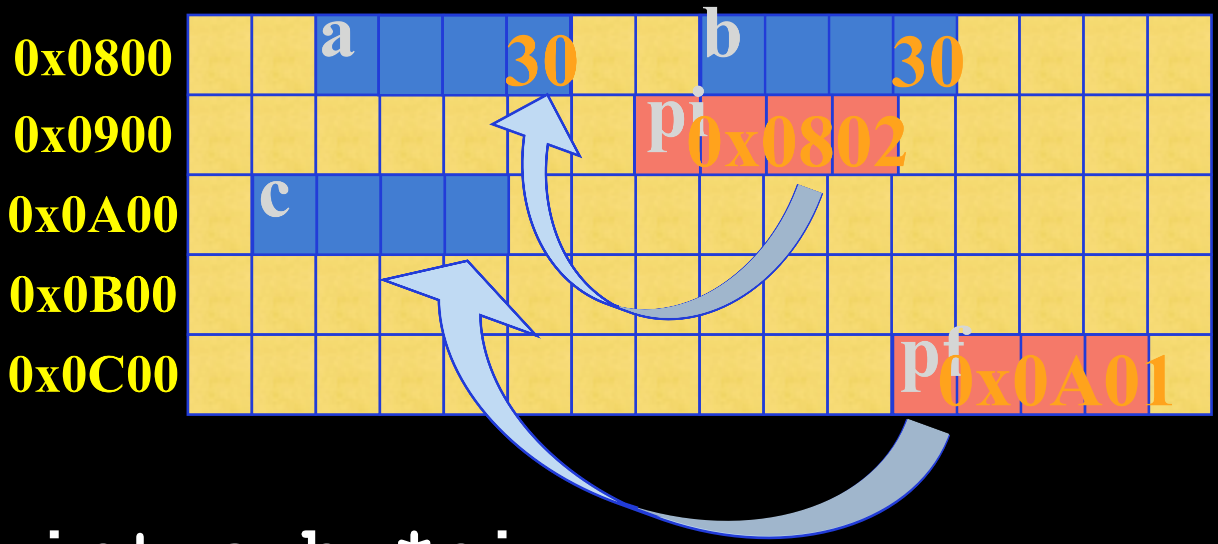
Somes examples



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

```
float c,*pf;
pf = &c;
```

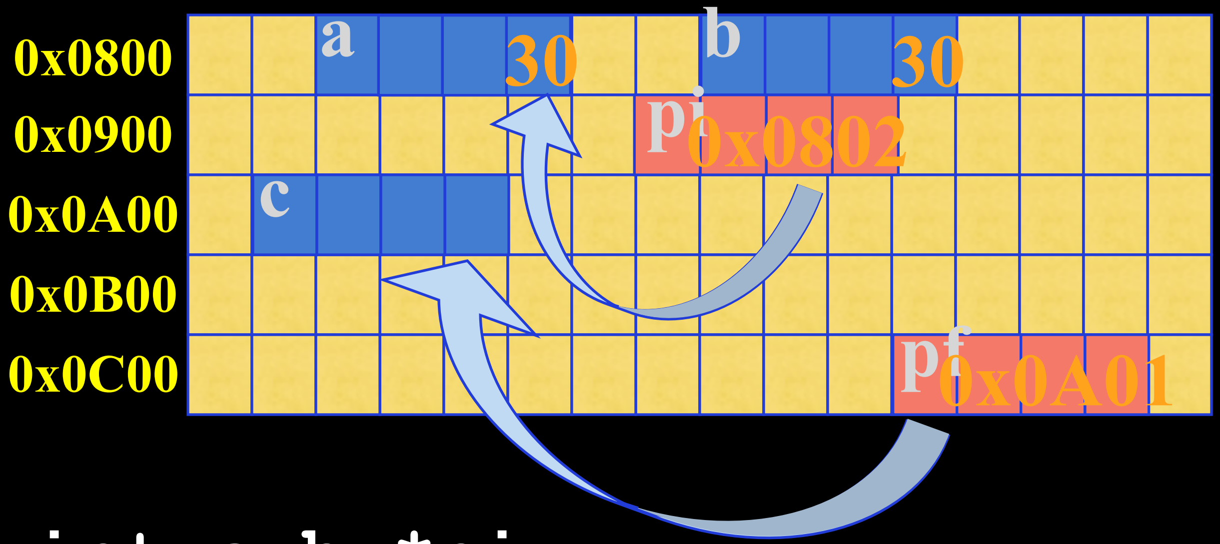
Somes examples



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

```
float c,*pf;
pf = &c;
```

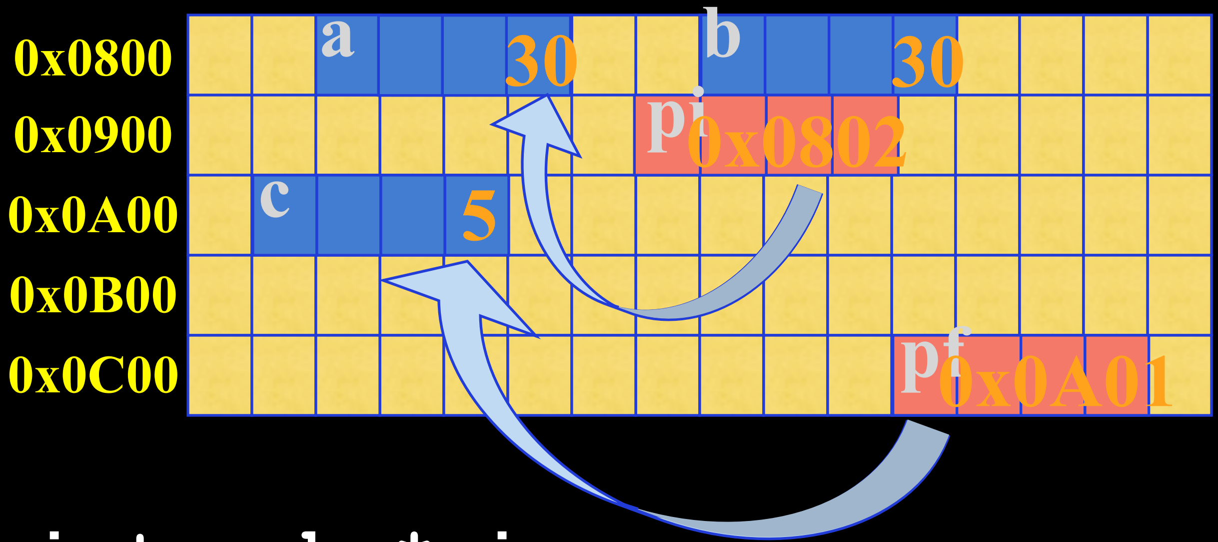
Somes examples



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

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

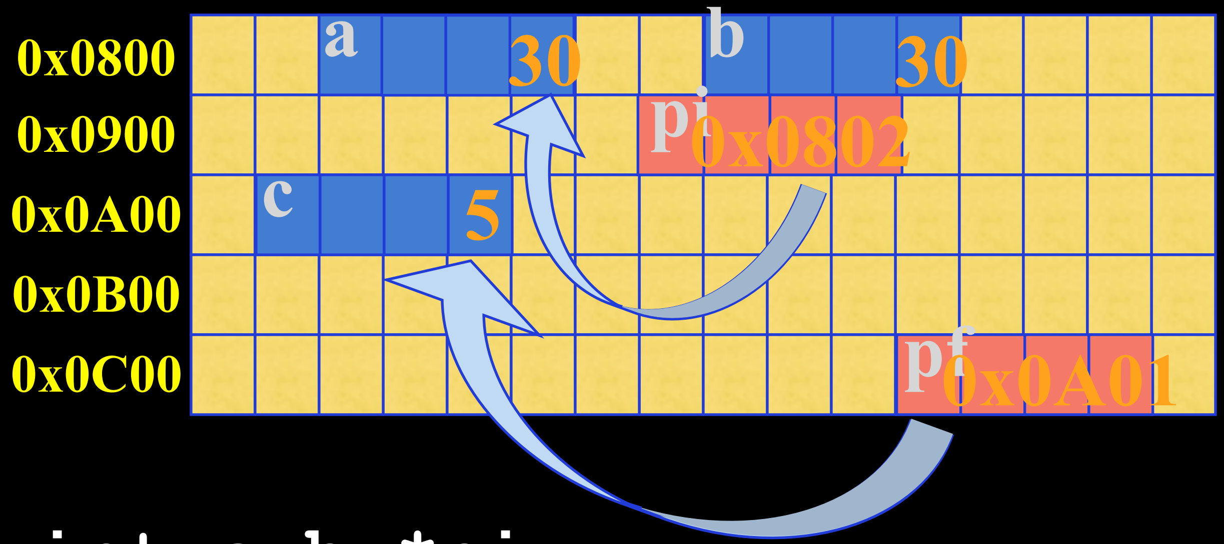
Somes examples



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

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

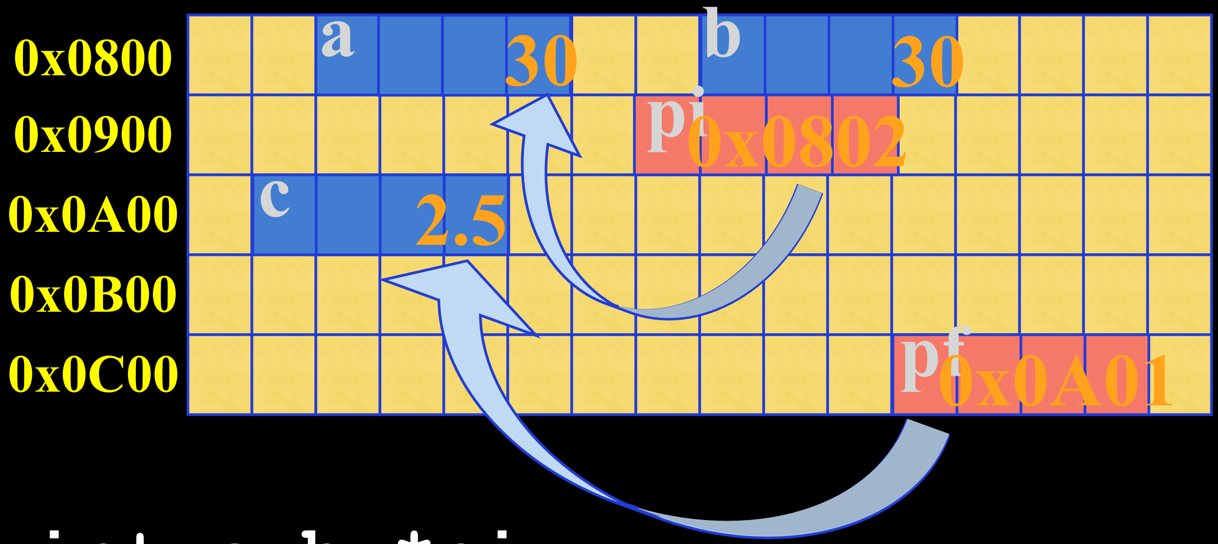
Somes examples



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

```
float c,*pf;
pf = &c;
 c = 5;
  *pf = *pf/2;
```

Somes examples



```
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 :

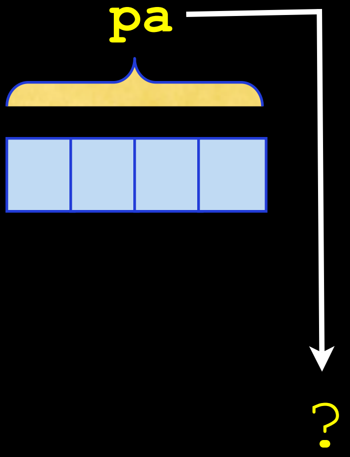
- **`int *pa;`** pointer onto an **`int`**
- **`float *pb;`** pointer onto a **`float`**

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

⇒ in general, every pointers are equivalent

Dynamic Allocation

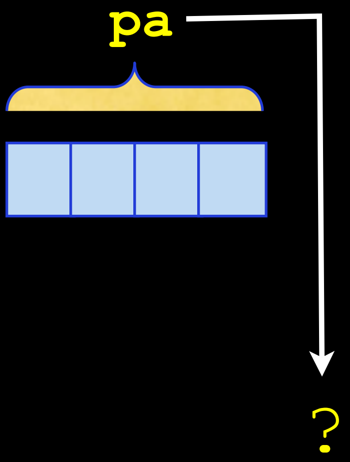
`int *pa;`



Dynamic Allocation

`int *pa;`

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

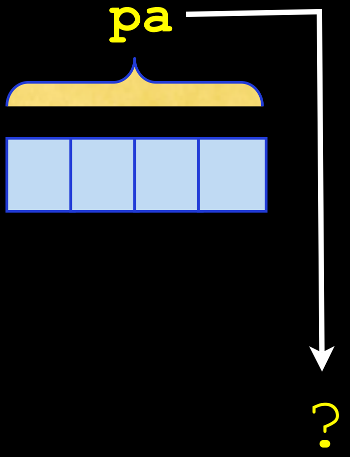


Dynamic Allocation

`int *pa;`

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

- **allocation** of a memory area to store an address

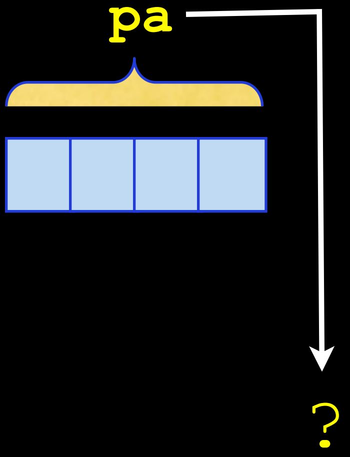


Dynamic Allocation

`int *pa;`

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

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

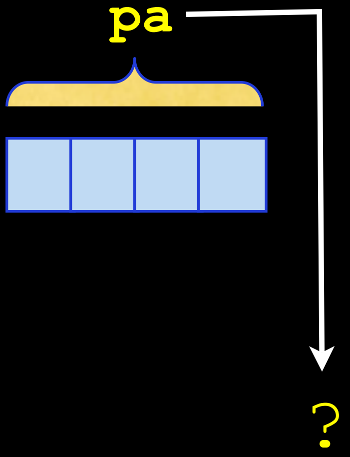


Dynamic Allocation

`int *pa;`

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

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



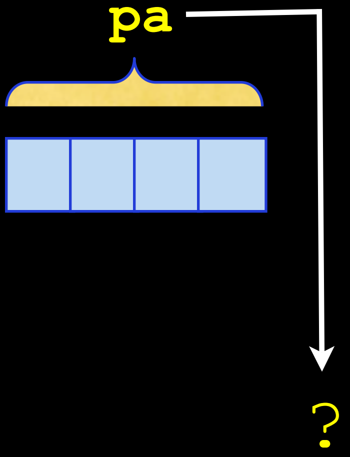
Dynamic Allocation

`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

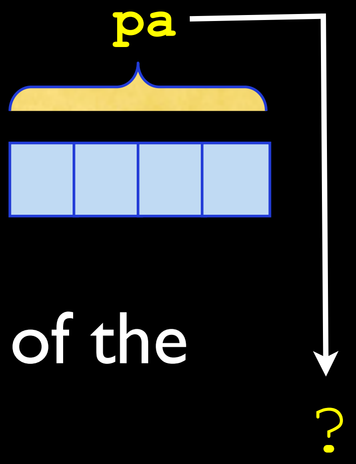


Dynamic Allocation

`int *pa;`

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

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



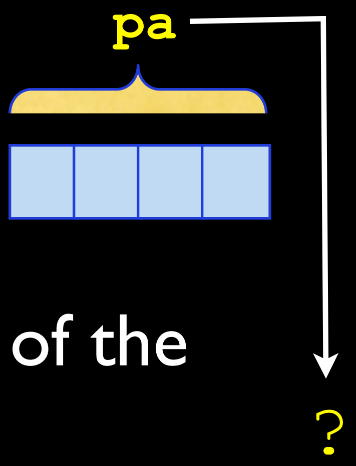
⇒ we need to allocate a memory area to store this integer : allocation and initialization of the

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

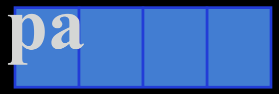
Dynamic Allocation : `malloc`

```
int *pa;
```

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

Dynamic Allocation : `malloc`

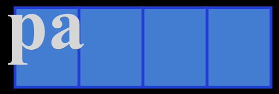
`int *pa;`



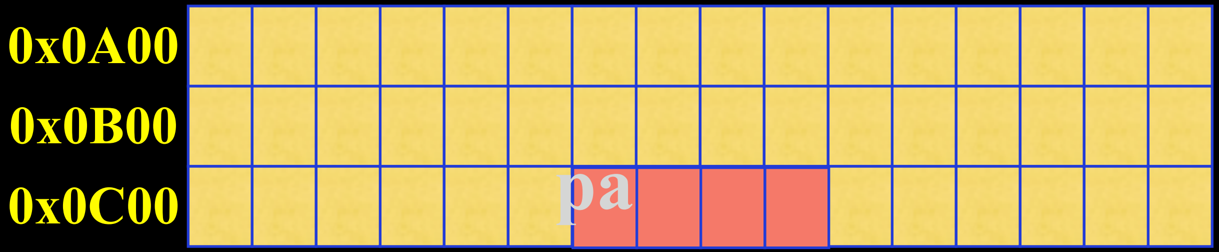
⇒ declaration of a pointer **`pa`** onto an **`int`**

Dynamic Allocation : `malloc`

`int *pa;`

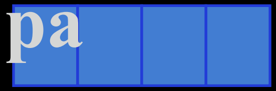


⇒ declaration of a pointer **`pa`** onto an **`int`**



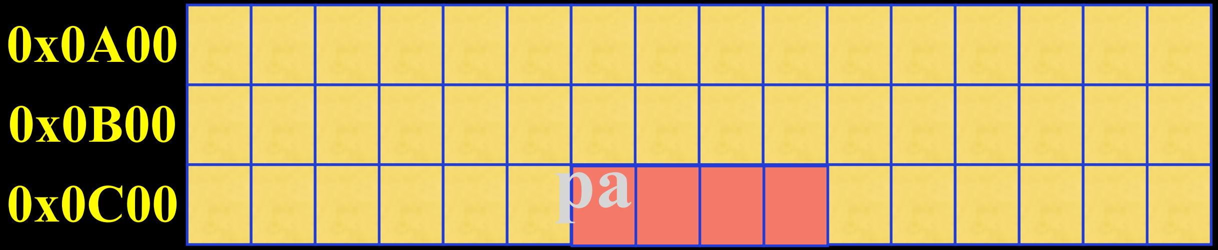
Dynamic Allocation : `malloc`

`int *pa;`



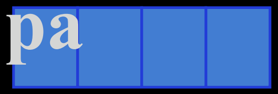
⇒ declaration of a pointer **`pa`** onto an **`int`**

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



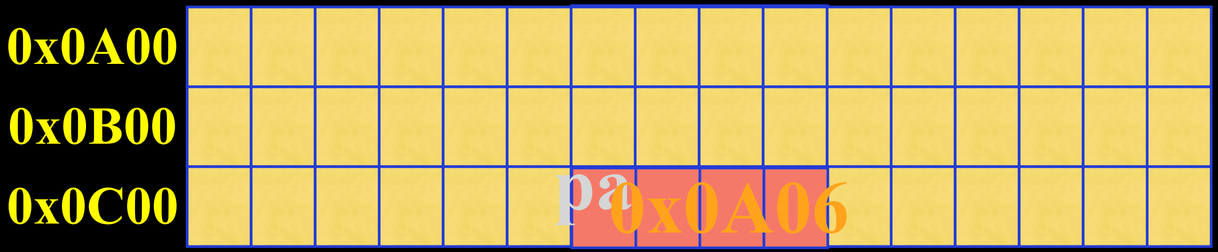
Dynamic Allocation : `malloc`

```
int *pa;
```



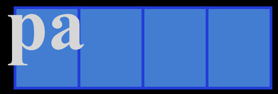
⇒ declaration of a pointer **`pa`** onto an **`int`**

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



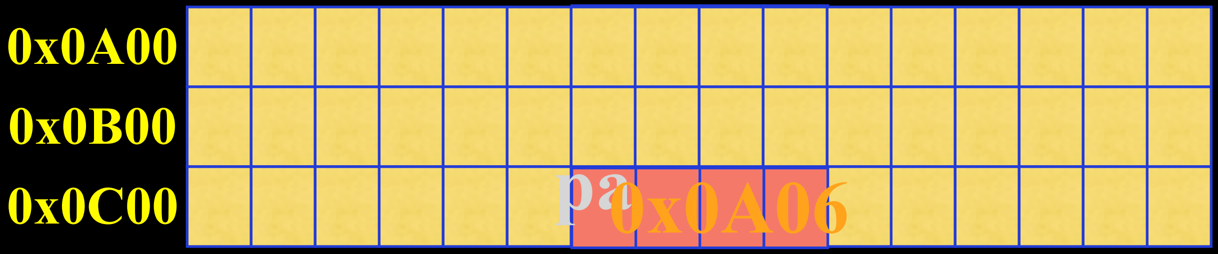
Dynamic Allocation : `malloc`

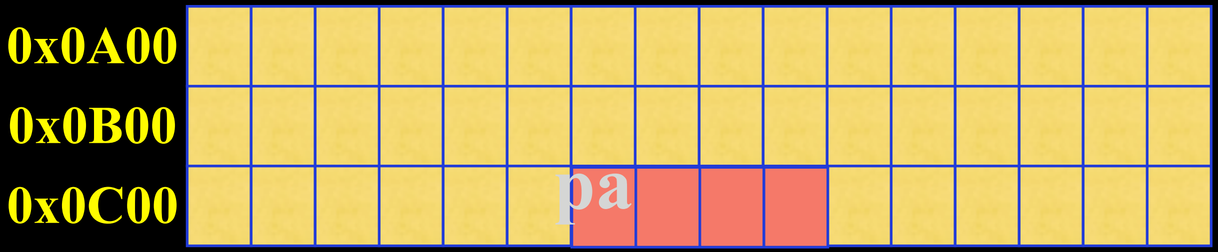
```
int *pa;
```



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

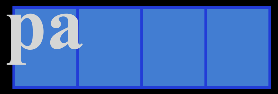
```
 int a;
pa = &a;
```





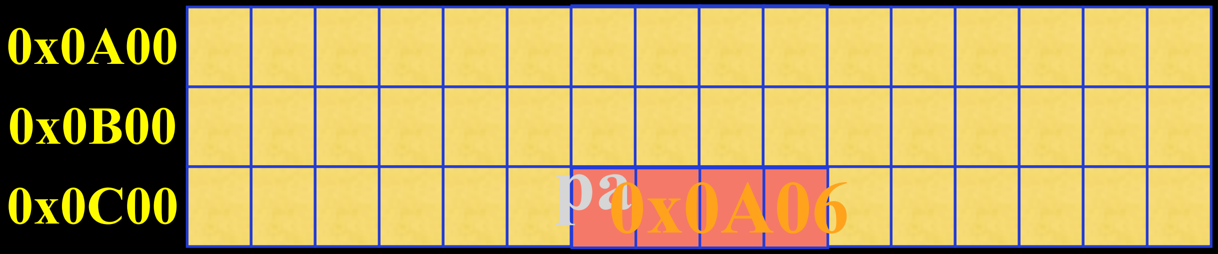
Dynamic Allocation : **`malloc`**

`int *pa;`

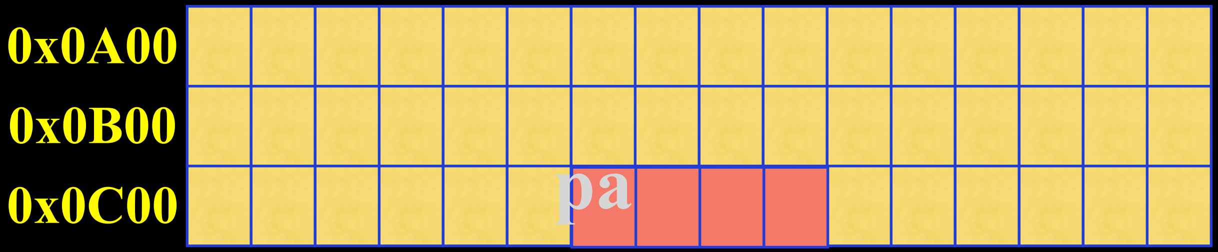


⇒ declaration of a pointer **`pa`** onto an **`int`**

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

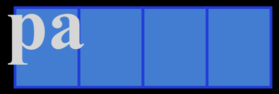


`pa = malloc(sizeof(int));` **in the heap**



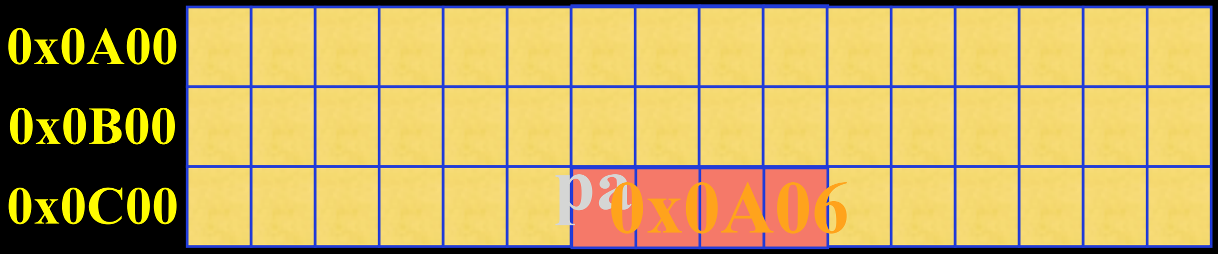
Dynamic Allocation : `malloc`

```
int *pa;
```

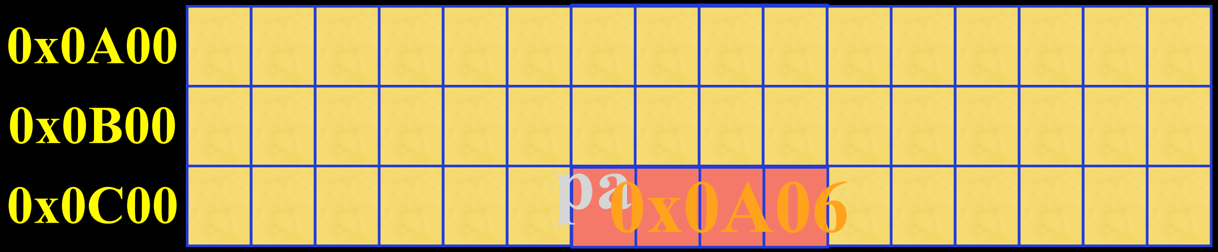


⇒ declaration of a pointer `pa` onto an `int`

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



`pa = malloc(sizeof(int));` **`in the heap`**



Usage of `malloc`

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

Usage of `malloc`

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

```
void *malloc(int n)
```

Usage of `malloc`

```
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`

```
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 *`

Usage of `malloc`

```
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 *`

⇒ 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
- …

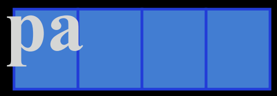
Dynamic Allocation: array

```
int *pa;
```

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

Dynamic Allocation: array

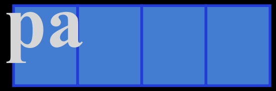
```
int *pa;
```



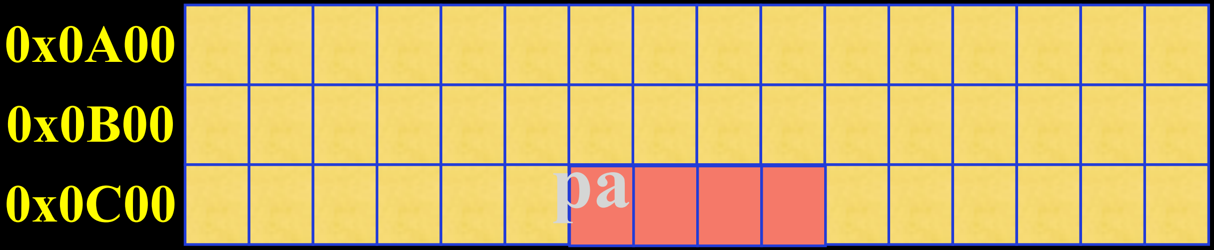
⇒ declare a pointer `pa` onto an `int`

Dynamic Allocation: array

`int *pa;`

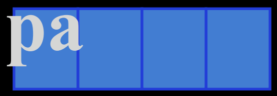


⇒ declare a pointer **`pa`** onto an **`int`**



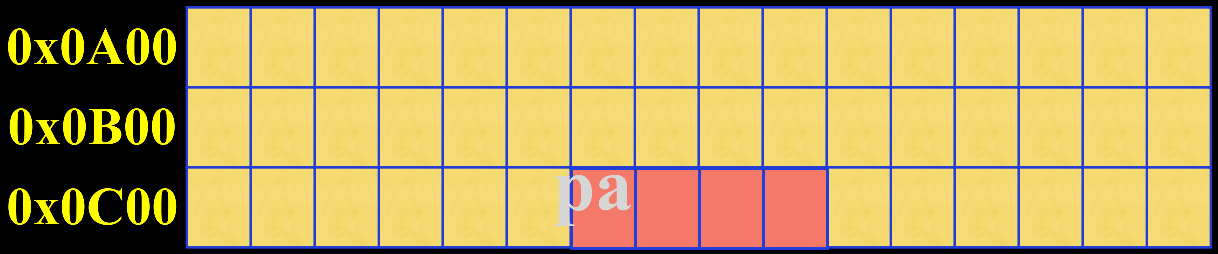
Dynamic Allocation: array

```
int *pa;
```



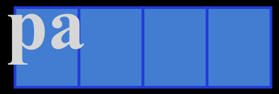
⇒ declare a pointer **`pa`** onto an **`int`**

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



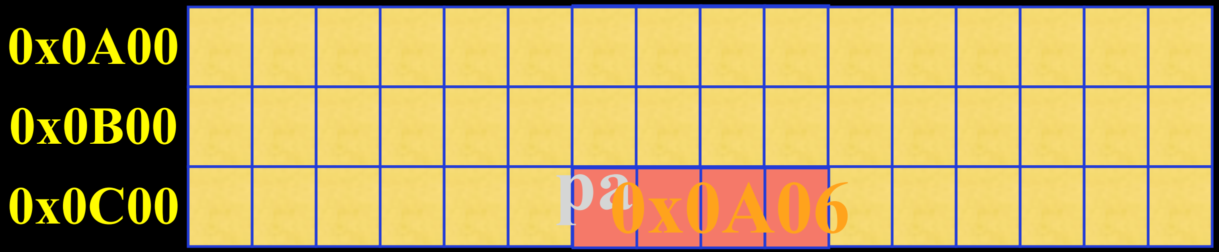
Dynamic Allocation: array

`int *pa;`



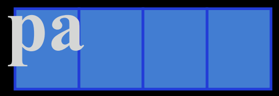
⇒ declare a pointer **`pa`** onto an **`int`**

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



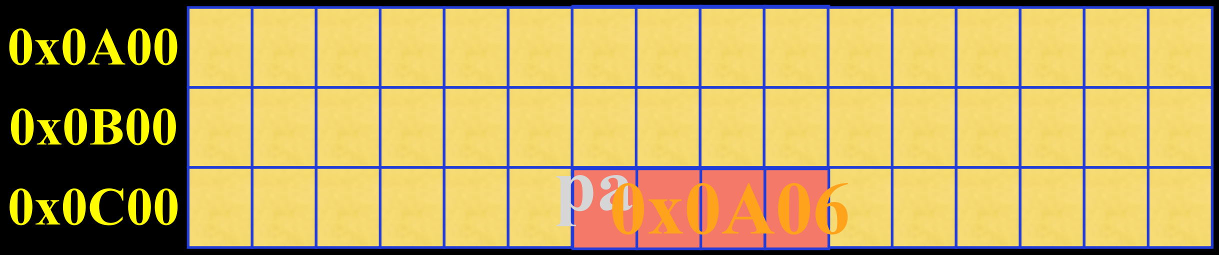
Dynamic Allocation: array

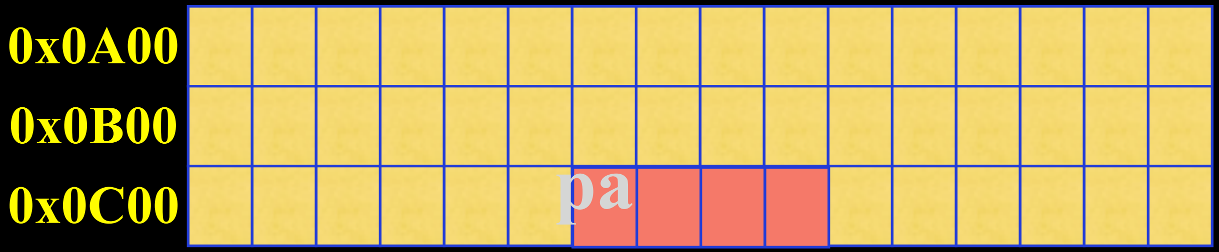
`int *pa;`



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

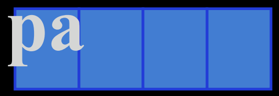
```
int a;
pa = &a;
```





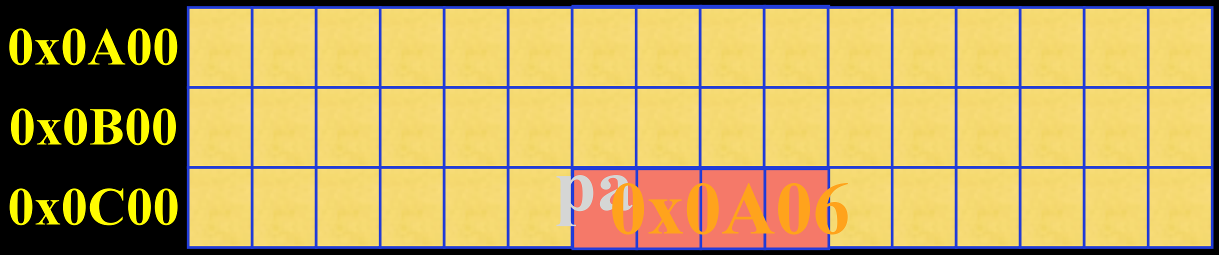
Dynamic Allocation: array

```
int *pa;
```

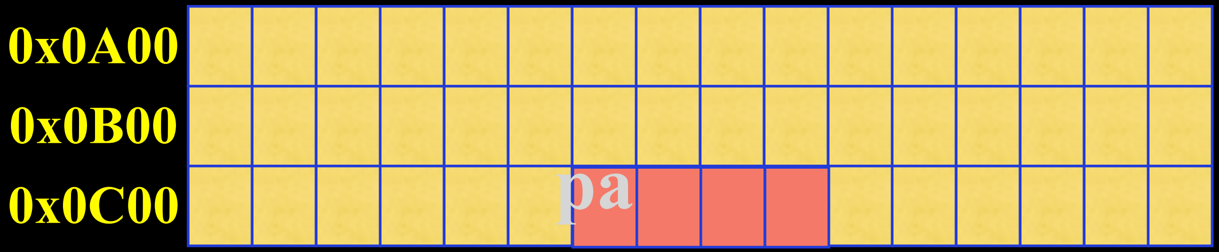


⇒ declare a pointer `pa` onto an `int`

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

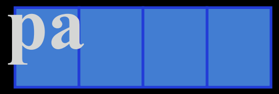


```
pa = malloc(3*(sizeof(int)));
```



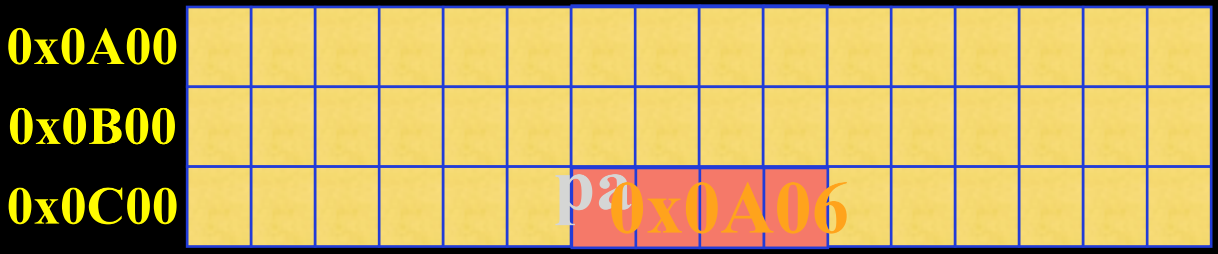
Dynamic Allocation: array

```
int *pa;
```

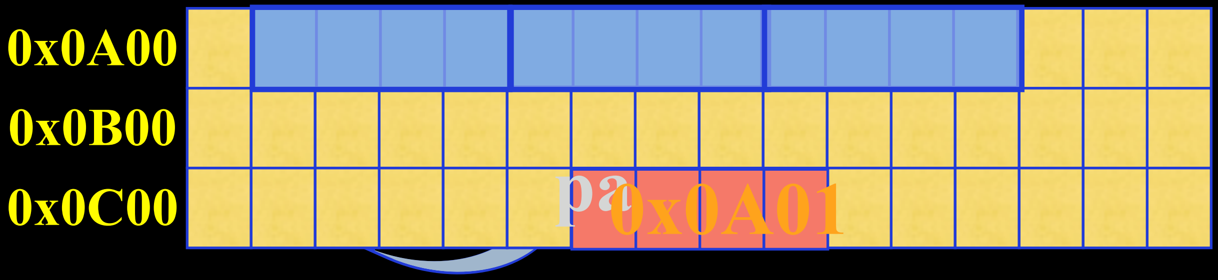


⇒ declare a pointer **`pa`** onto an **`int`**

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



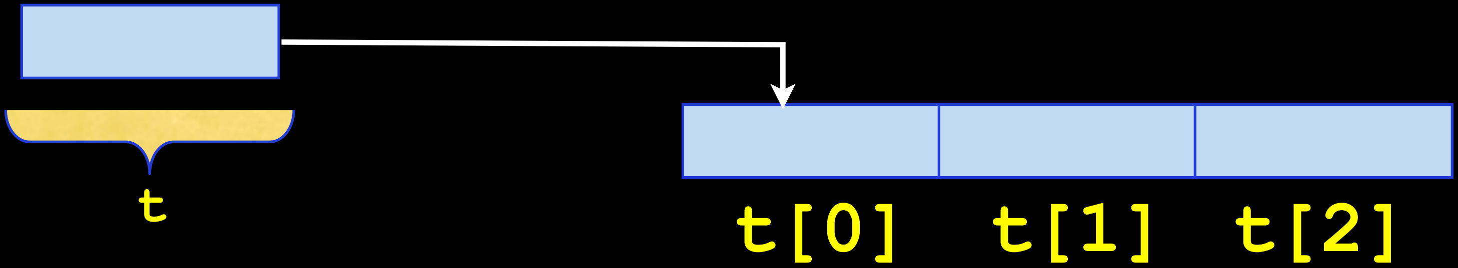
```
pa = malloc(3*(sizeof(int)));
```



Dynamic Arrays

`int t[3];`

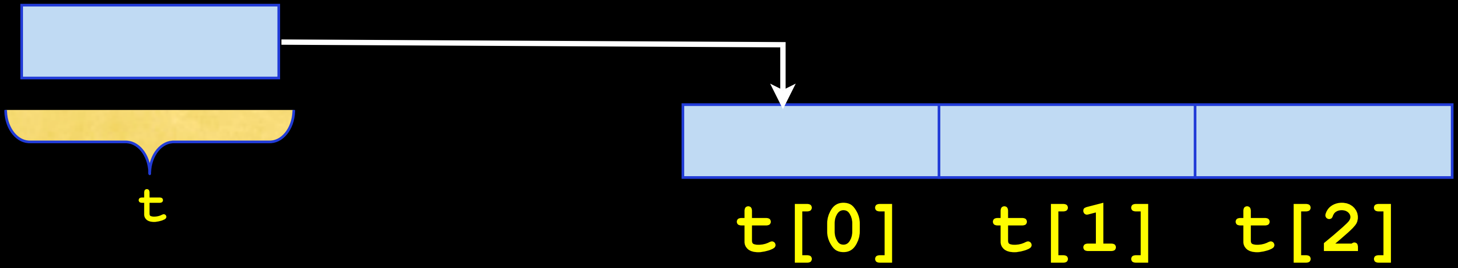
⇒ array of 3 int



Dynamic Arrays

`int t[3];`

⇒ array of 3 int

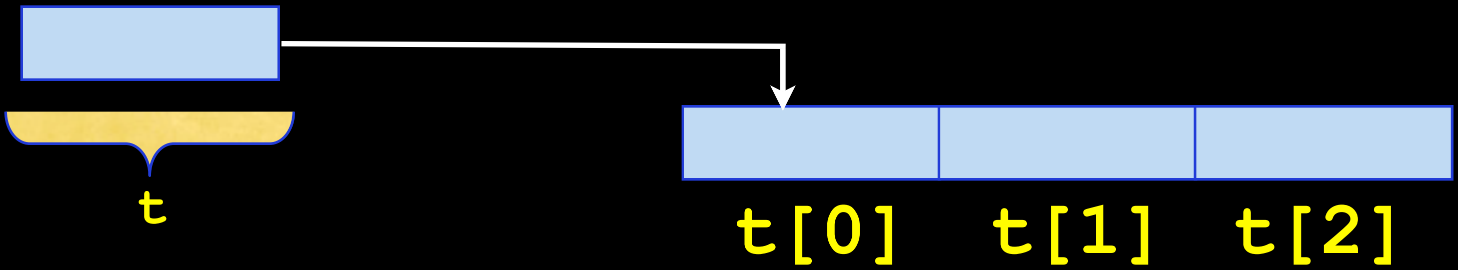


equivalent to

Dynamic Arrays

`int t[3];`

⇒ array of 3 int



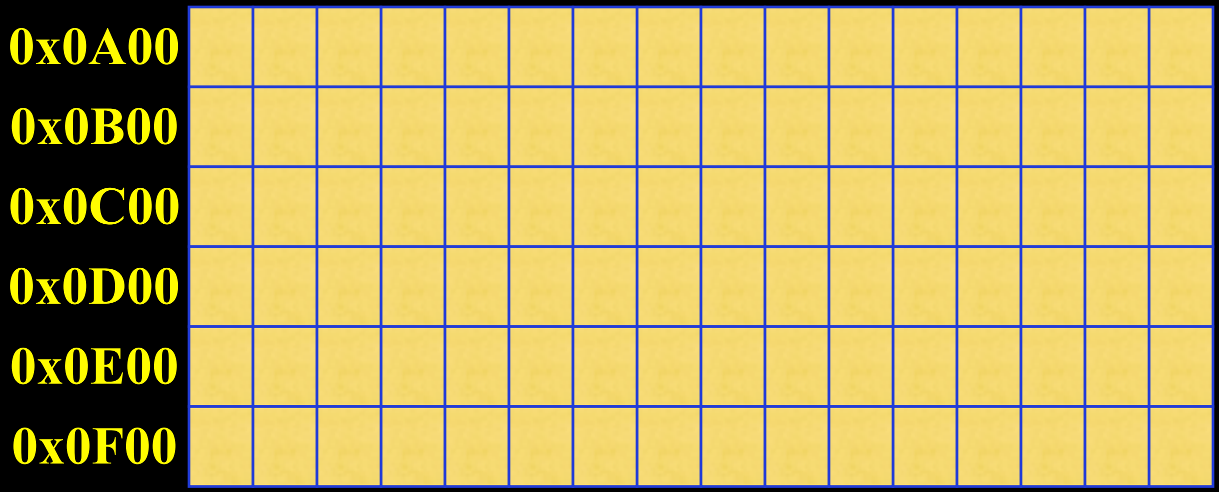
equivalent to

`int *t = malloc(3*(sizeof(int)));`

Dynamic arrays (suite)

`float *Tf;`

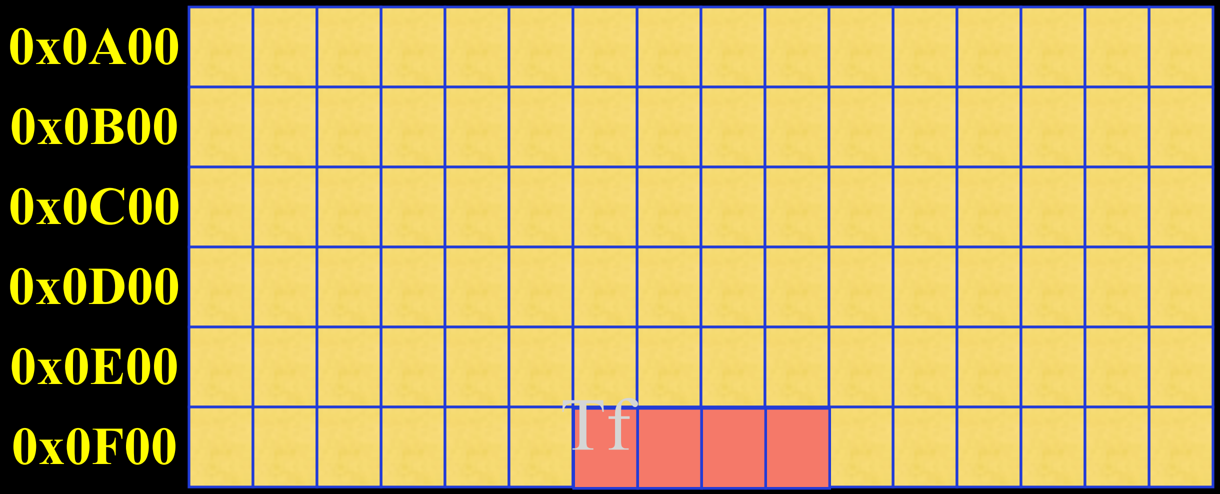
⇒ declare a pointer `Tf` onto a `float`



Dynamic arrays (suite)

`float *Tf;`

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

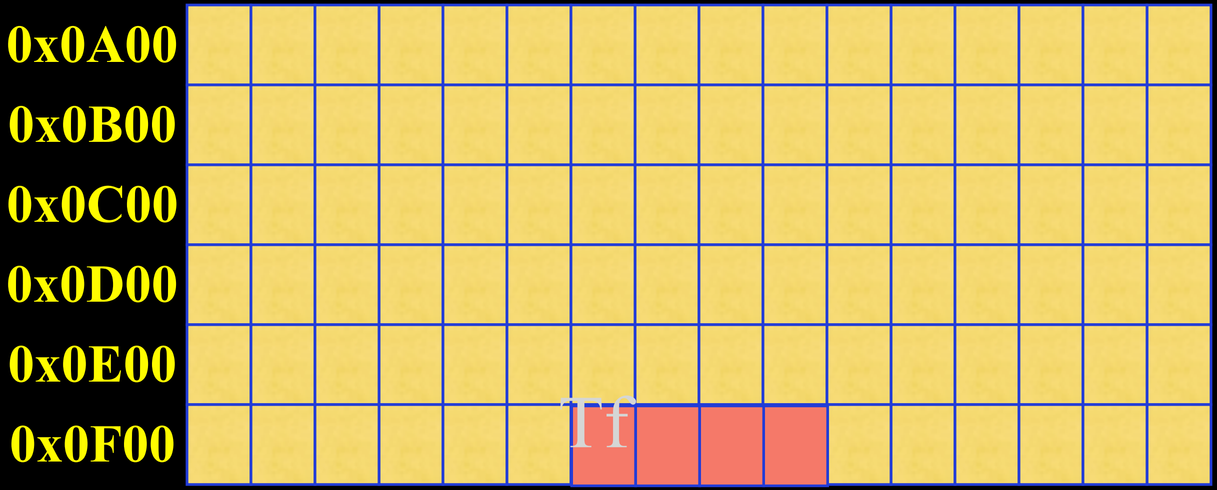


Dynamic arrays (suite)

`float *Tf;`

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

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

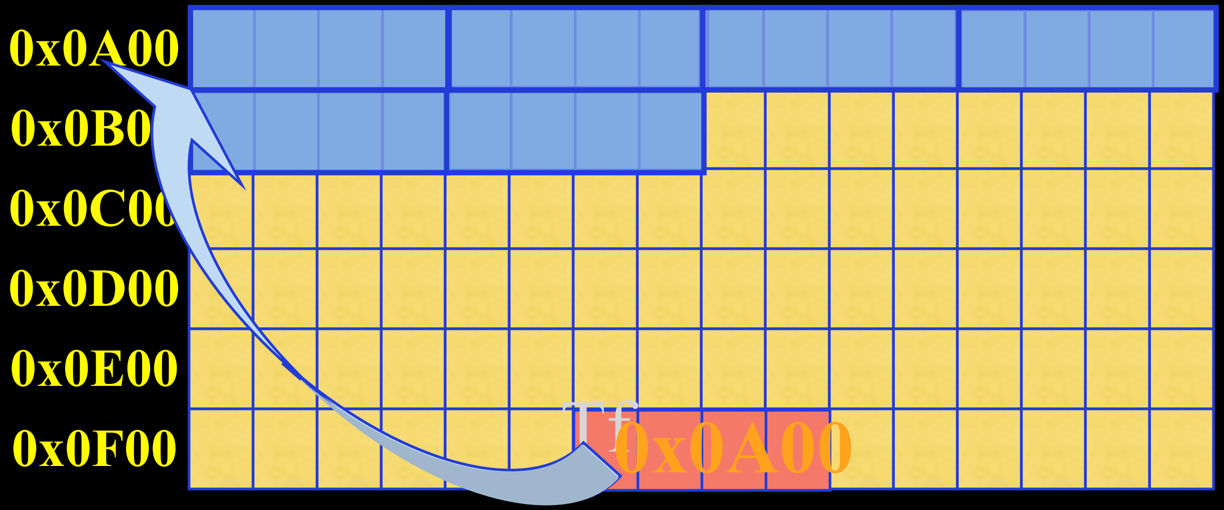


Dynamic arrays (suite)

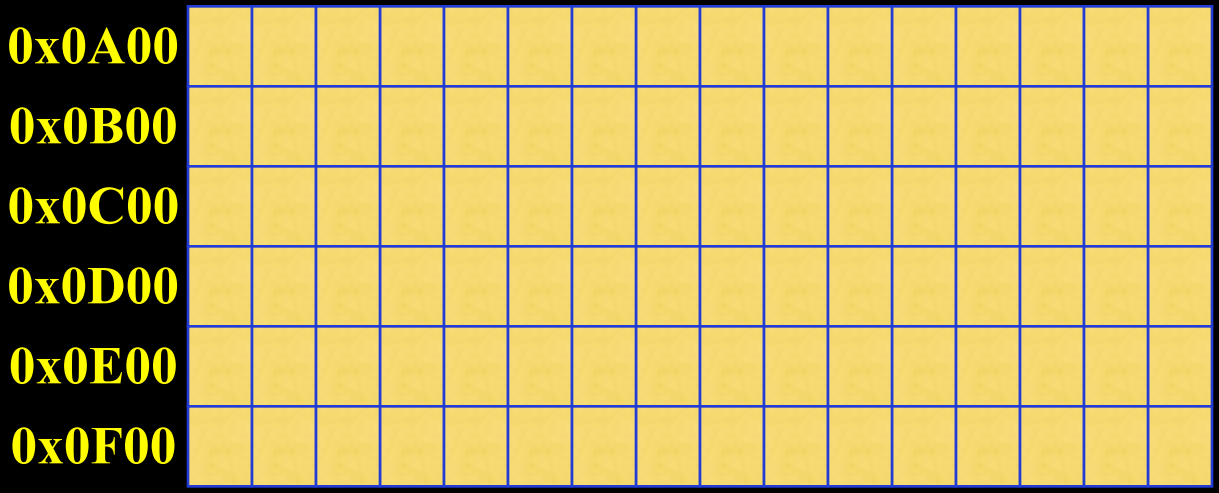
```
float *Tf;
```

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

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

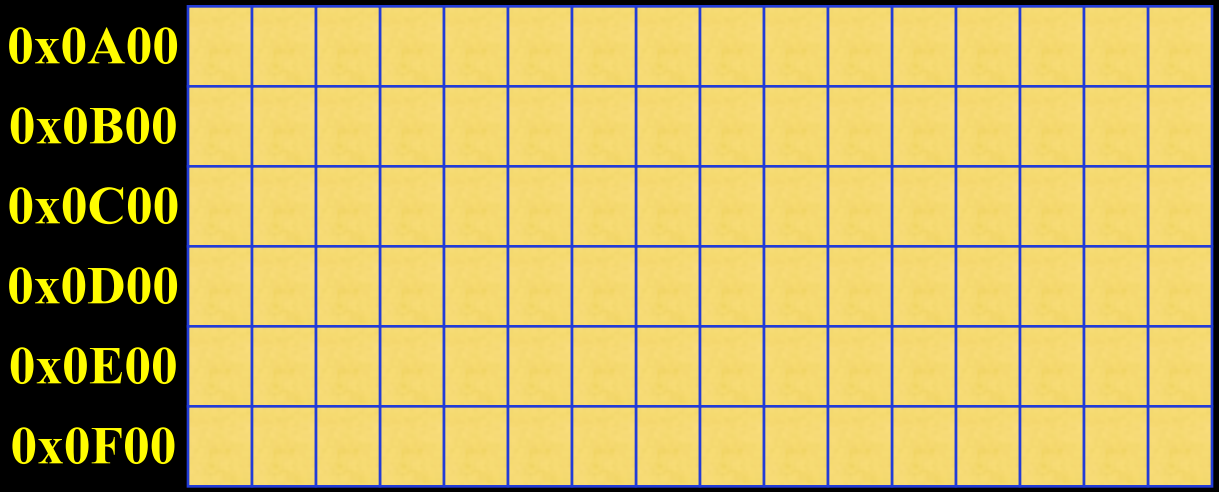


Dynamic array (suite)



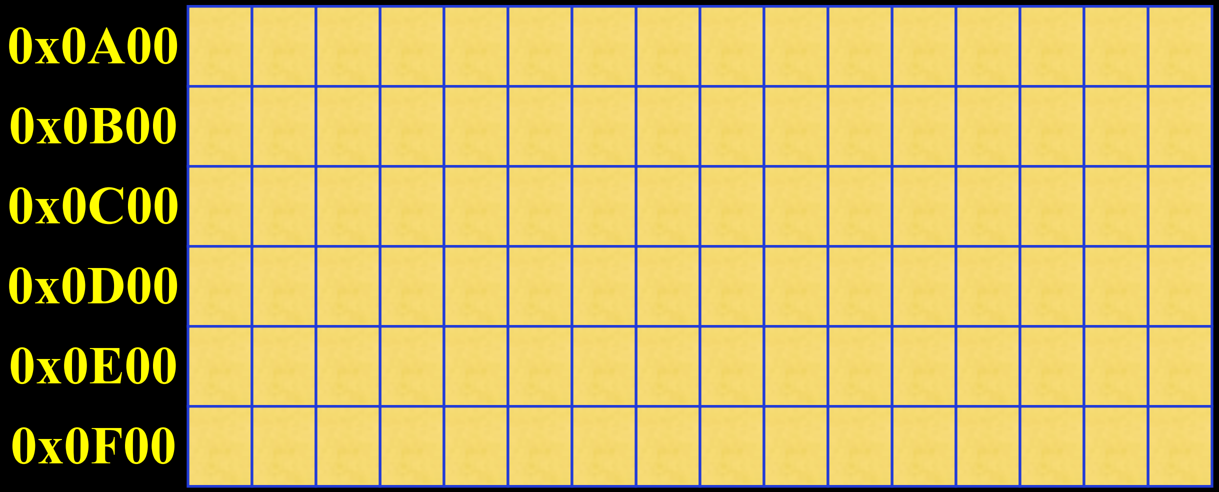
Dynamic array (suite)

```
float *Tf;
```



Dynamic array (suite)

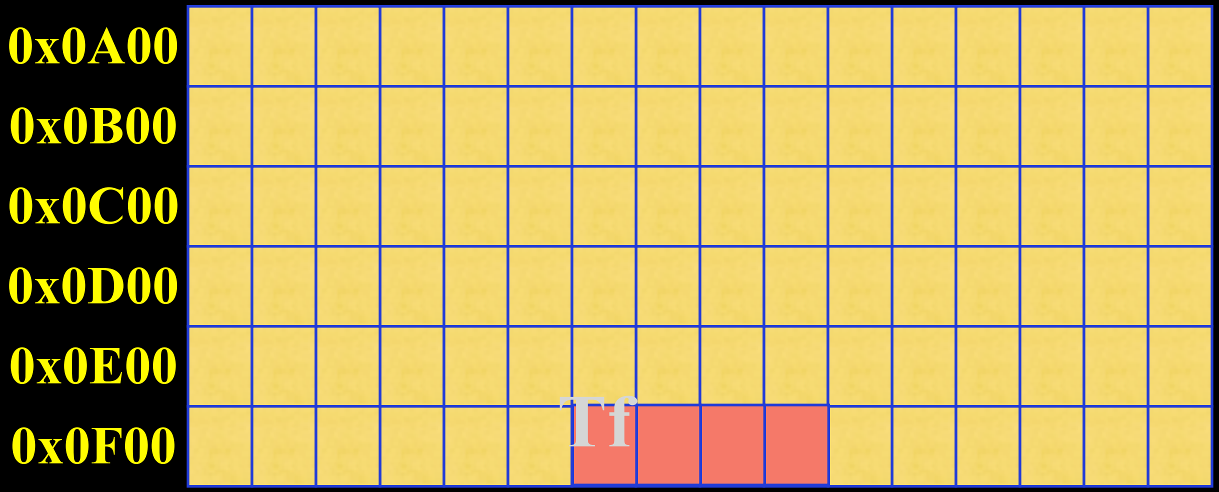
```
float *Tf;
Tf = malloc(6*(sizeof(float)));
```



Dynamic array (suite)

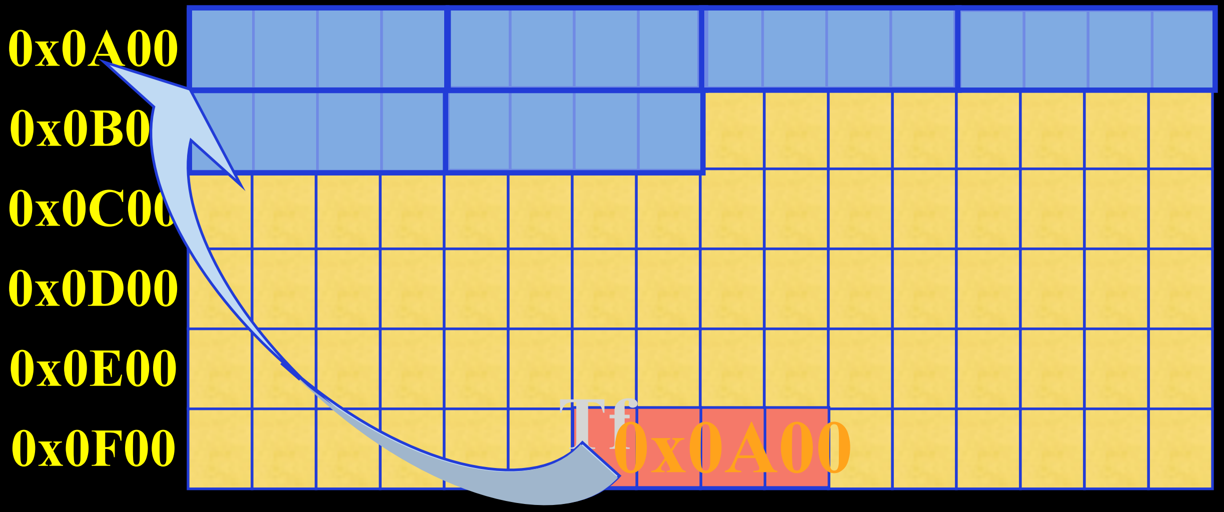
```
float *Tf;

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



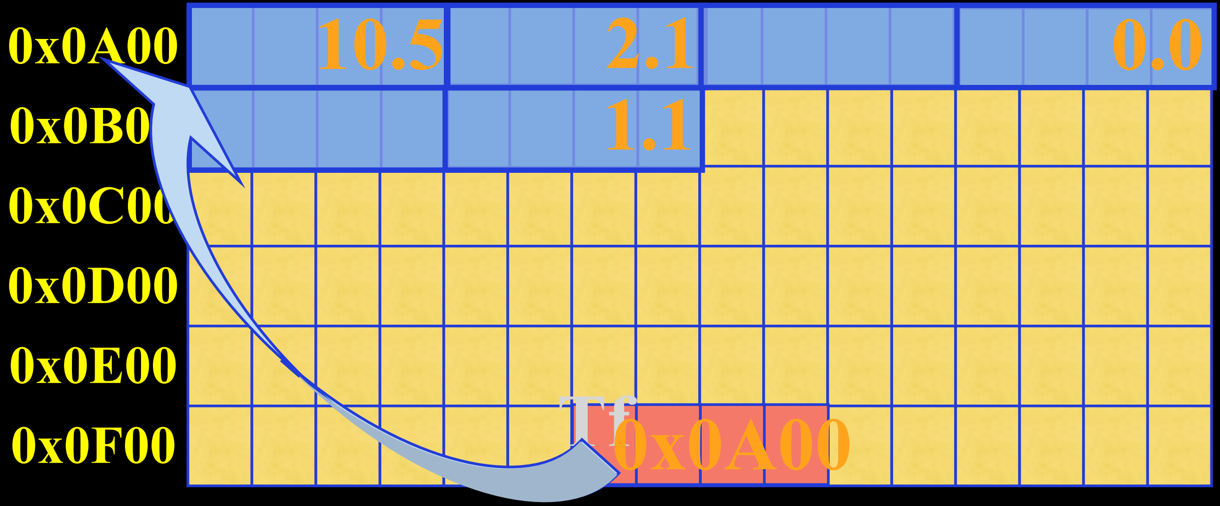
Dynamic array (suite)

```
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)

`<type> *t = malloc(k*(sizeof(<type>)));`

define an array of **`k`** objects of type **`<type>`**,
where **`k`** can be a variable :

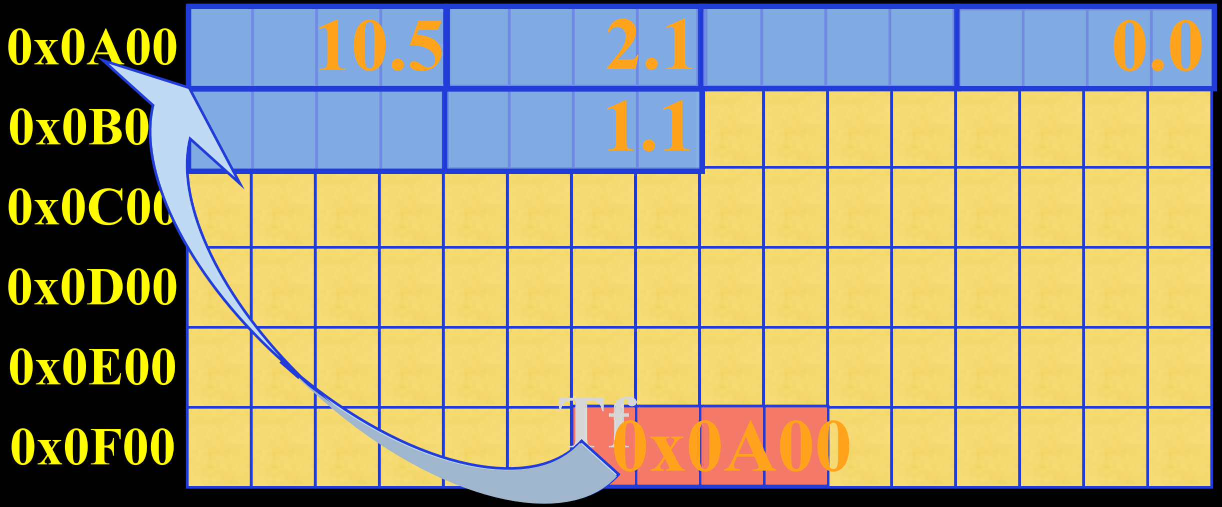
sizee non defined at compilation

`<type>` *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;
```

Arrays and pointers

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

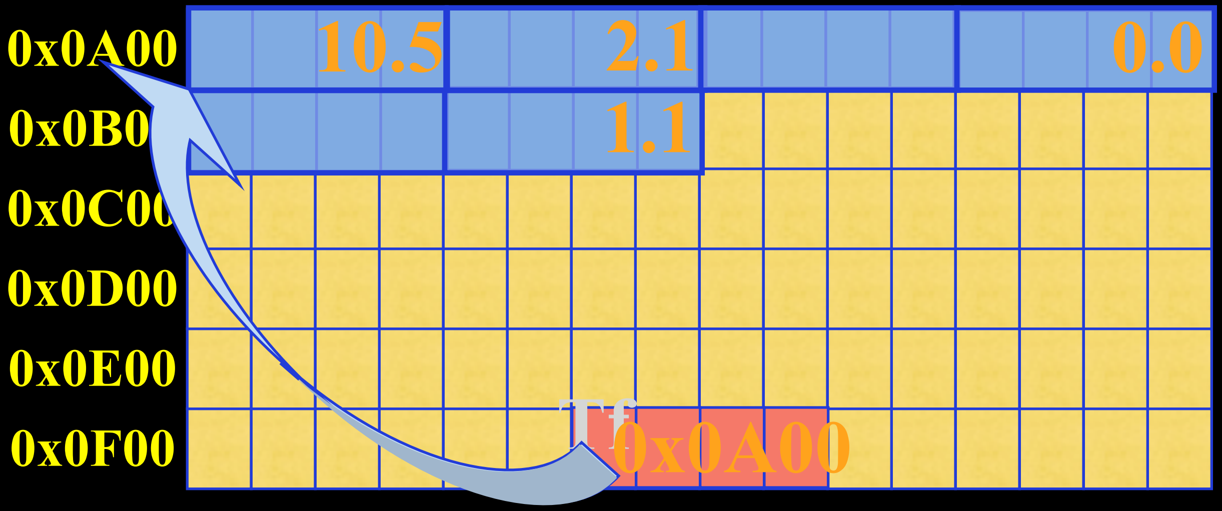


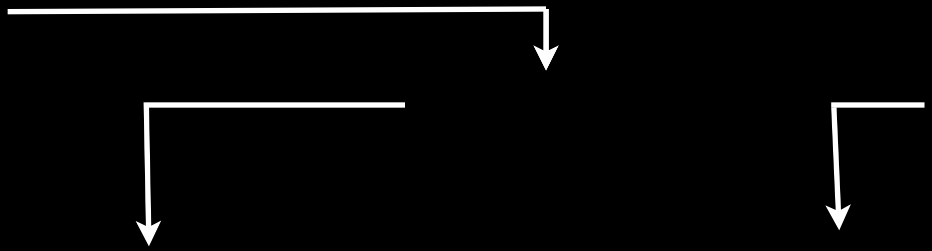
Tableau de `float`

```
Tf  : 0x0A00
Tf+1: 0x0A04
Tf+3: 0x0A0C
Tf+5: 0x0B04
```

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

Array with many dimensions

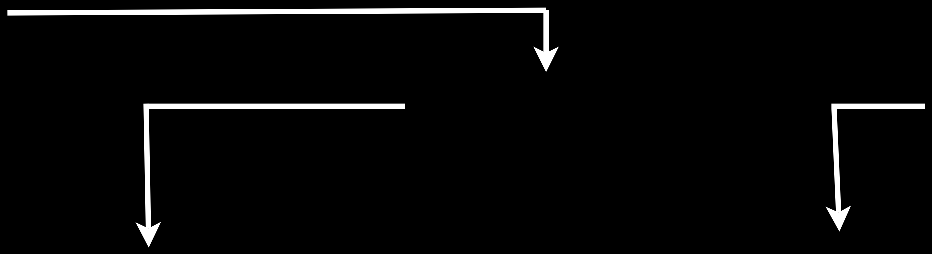
```
int i, m = 10, n = 15;
int **T;
T = malloc(m * sizeof(int *));
for (i=0; i<m; i++)
   T[i] = malloc(n * sizeof(int));
```



Array with many dimensions

```
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

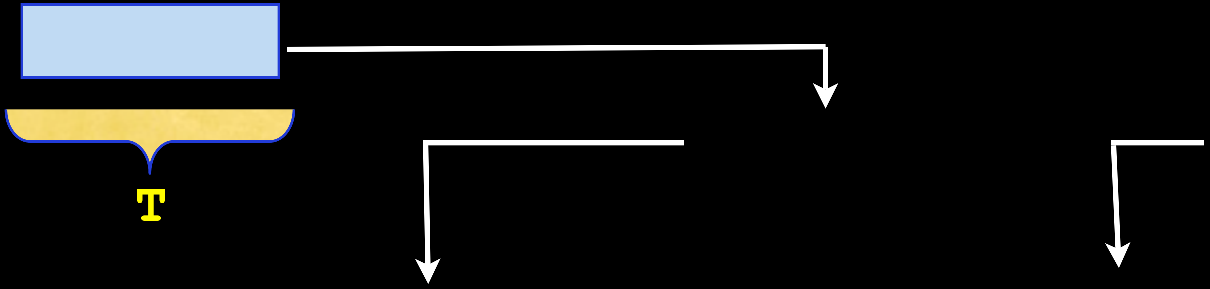


Array with many dimensions

```
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

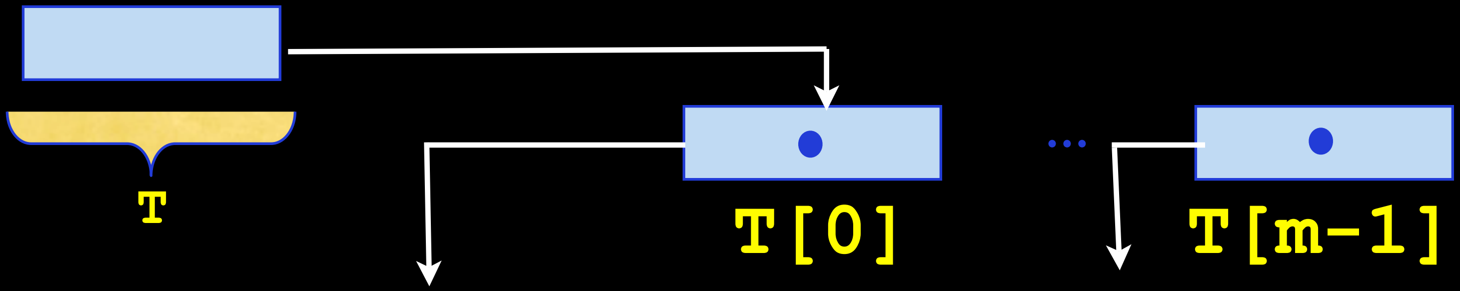


Array with many dimensions

```
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

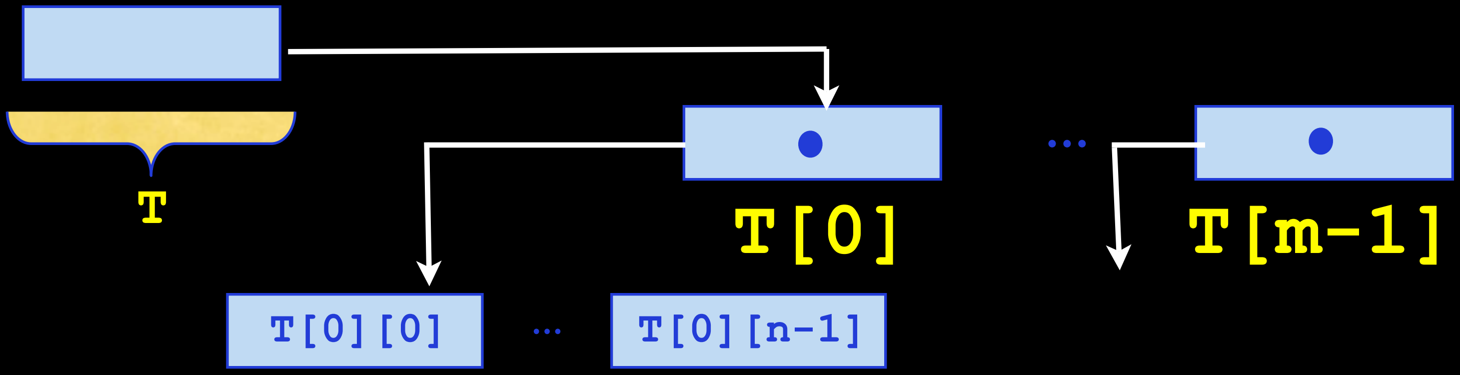
= array of int array



Array with many dimensions

```
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));
```

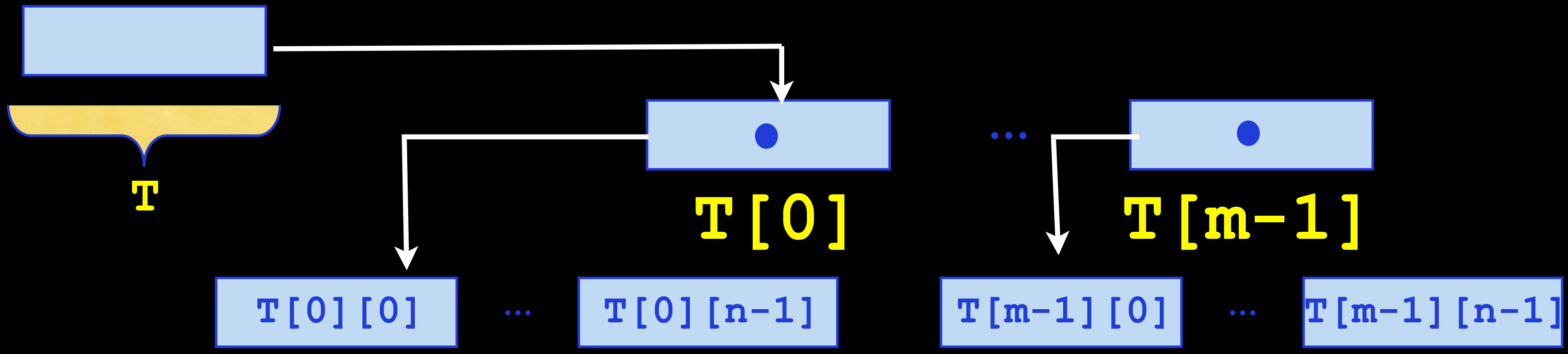
`T` is a pointer on a pointer onto an int
= array of int array



Array with many dimensions

```
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

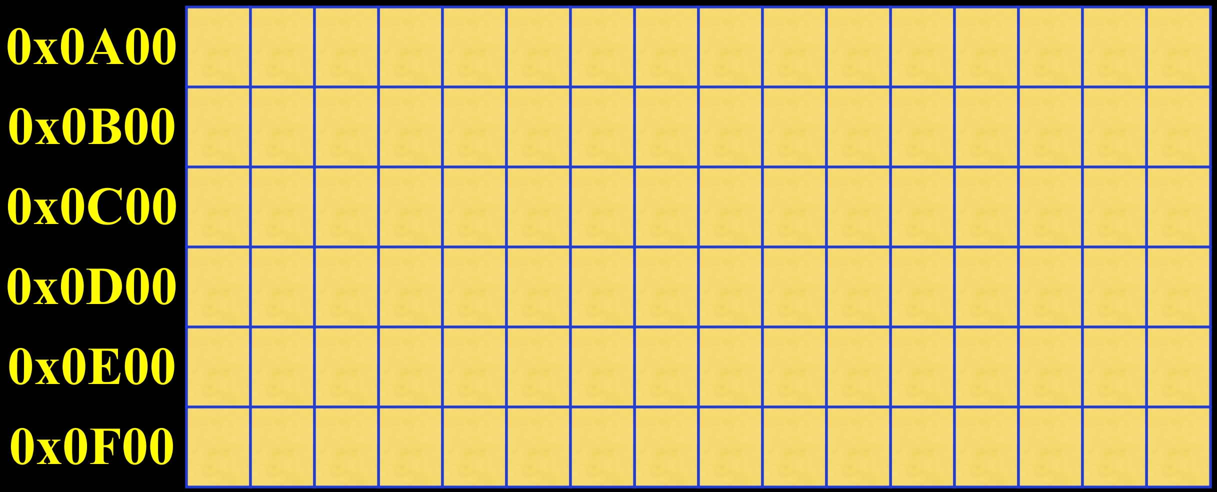


T

T[0] … T[m-1]

T[0][0] … T[0][n-1] T[m-1][0] … T[m-1][n-1]

Many dimensions (suite)



0x0A00

0x0B00

0x0C00

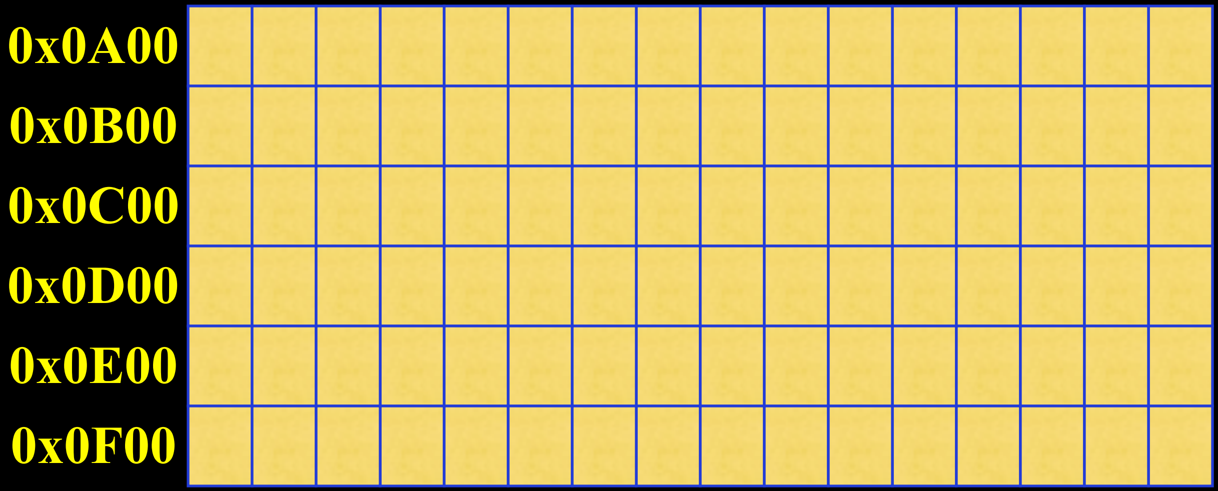
0x0D00

0x0E00

0x0F00

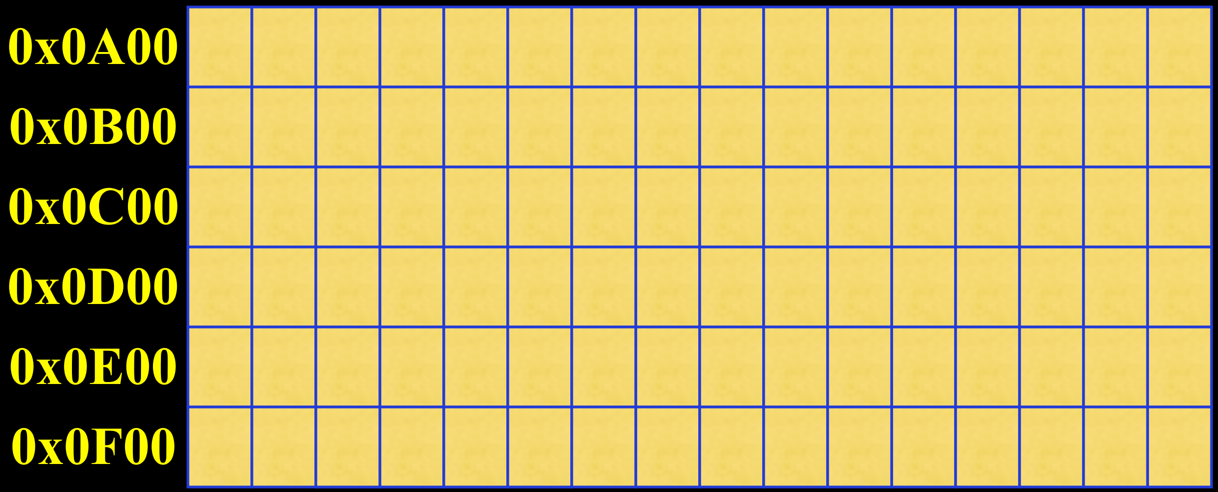
Many dimensions (suite)

```
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));
```



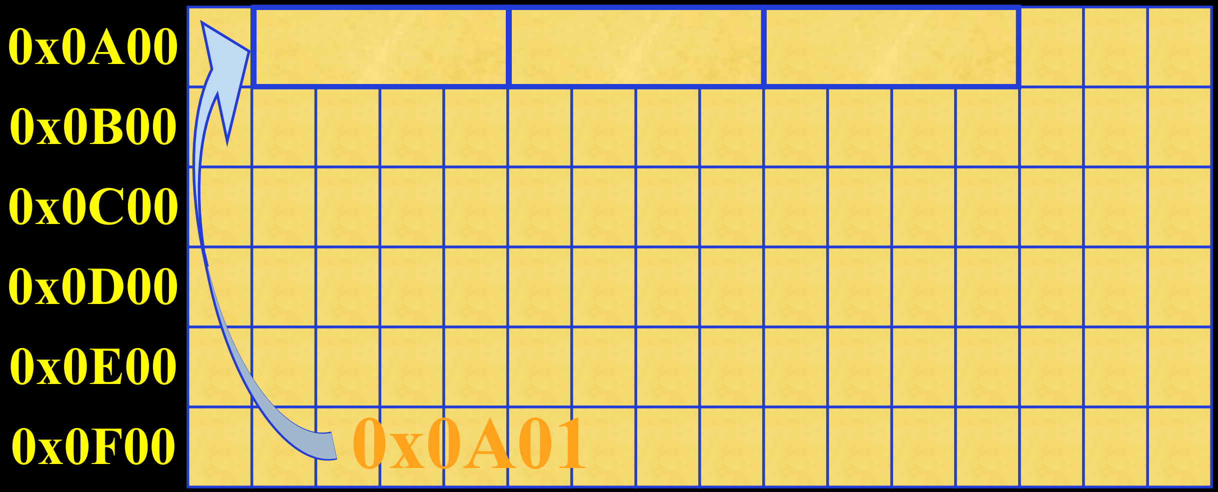
Many dimensions (suite)

```
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));
```



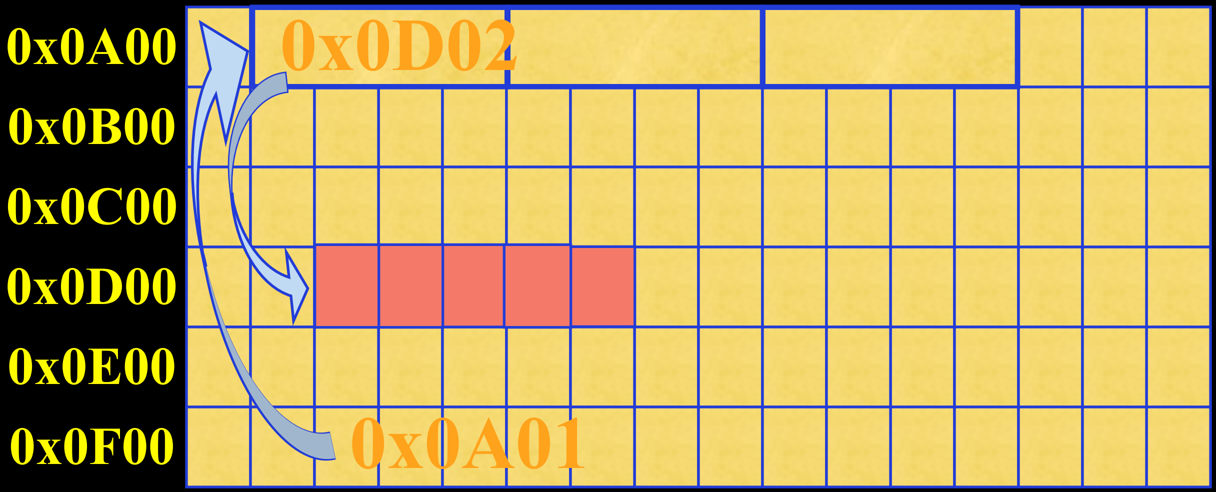
Many dimensions (suite)

```
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));
```



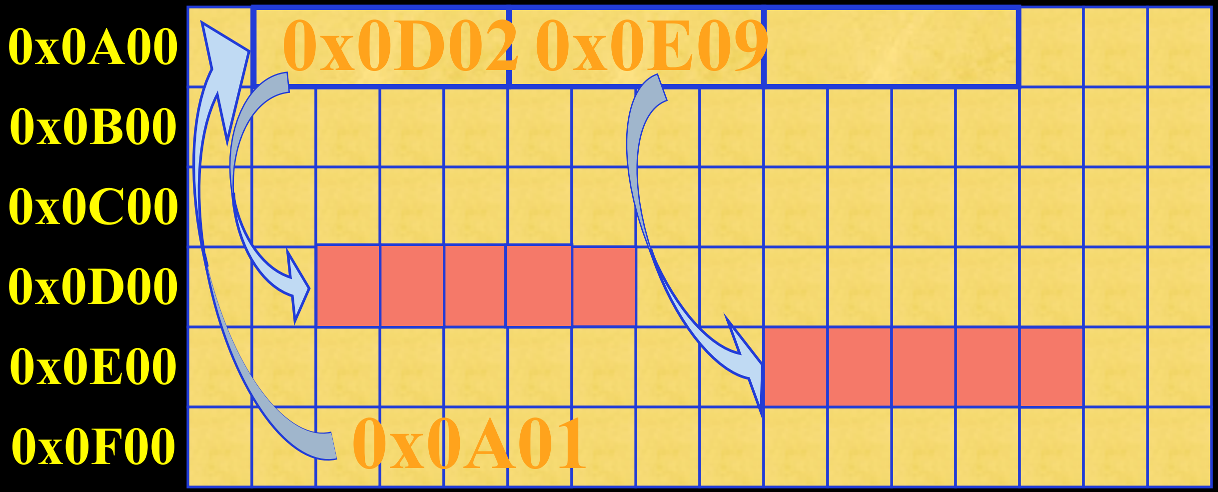
Many dimensions (suite)

```
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));
```



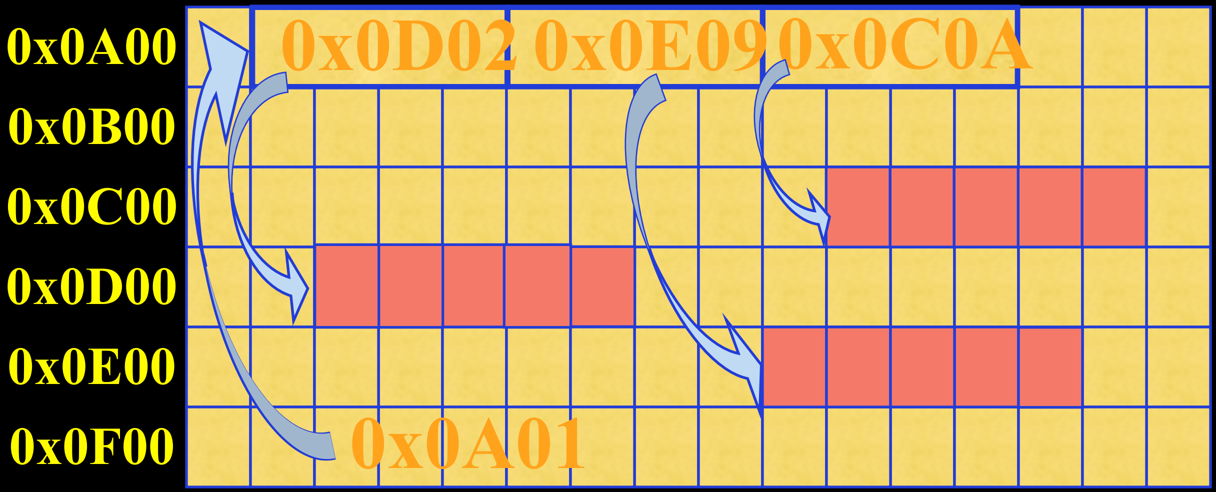
Many dimensions (suite)

```
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));
```



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));
```

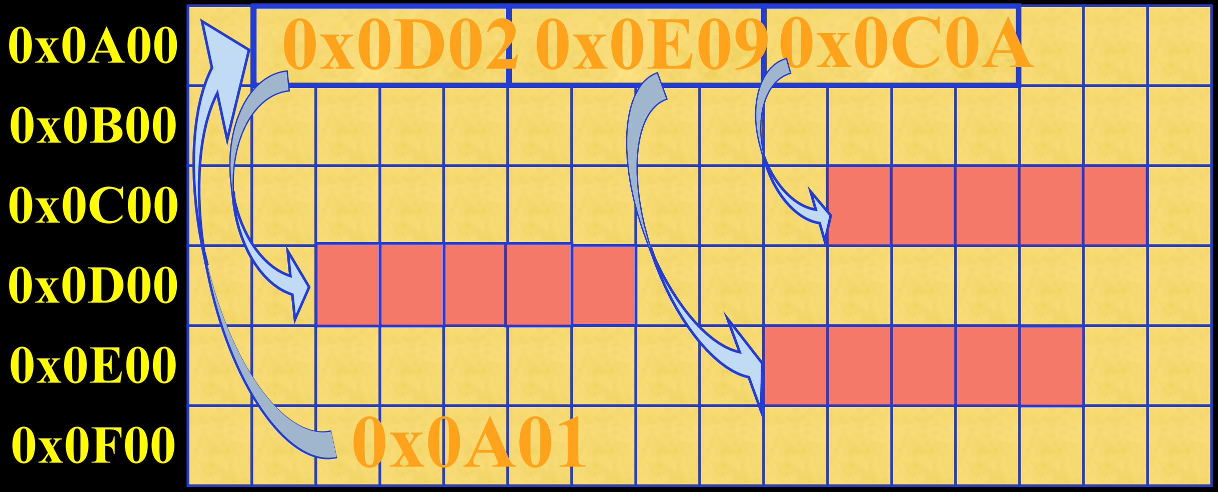


Many dimensions (suite)

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

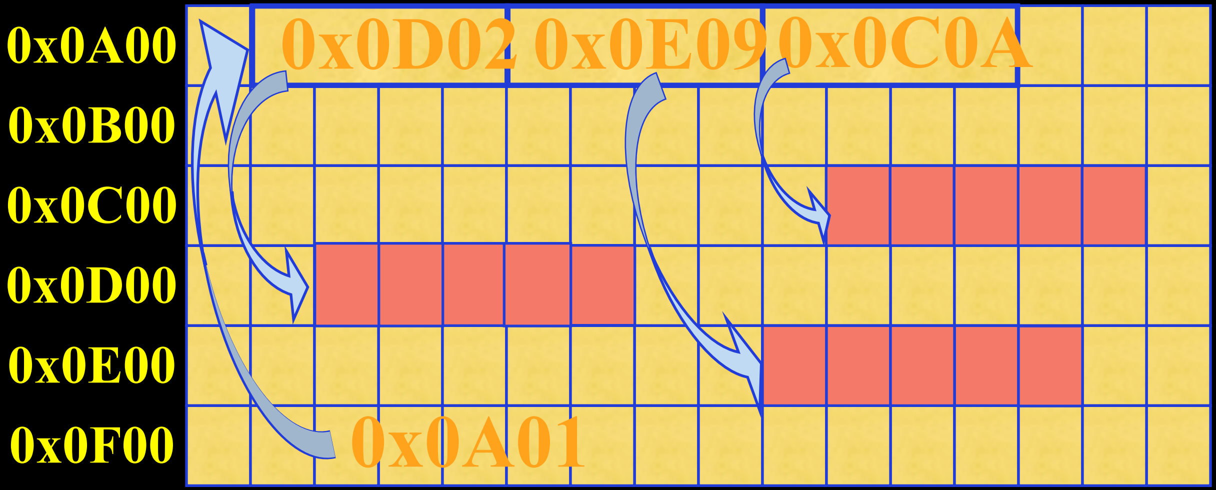
Many dimensions (suite)

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



Many dimensions (suite)

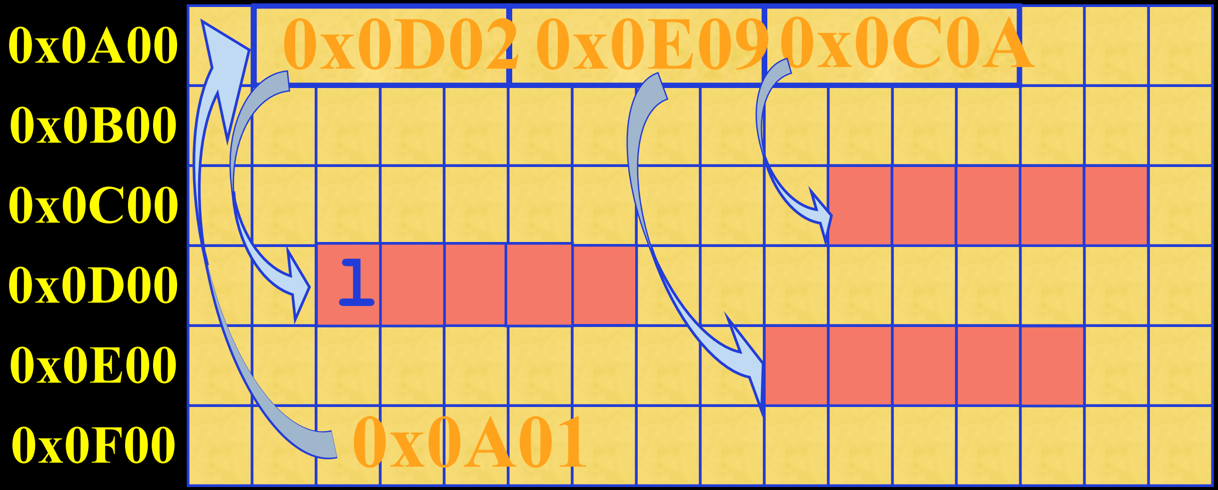
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
T[0][0] = '1'; 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)

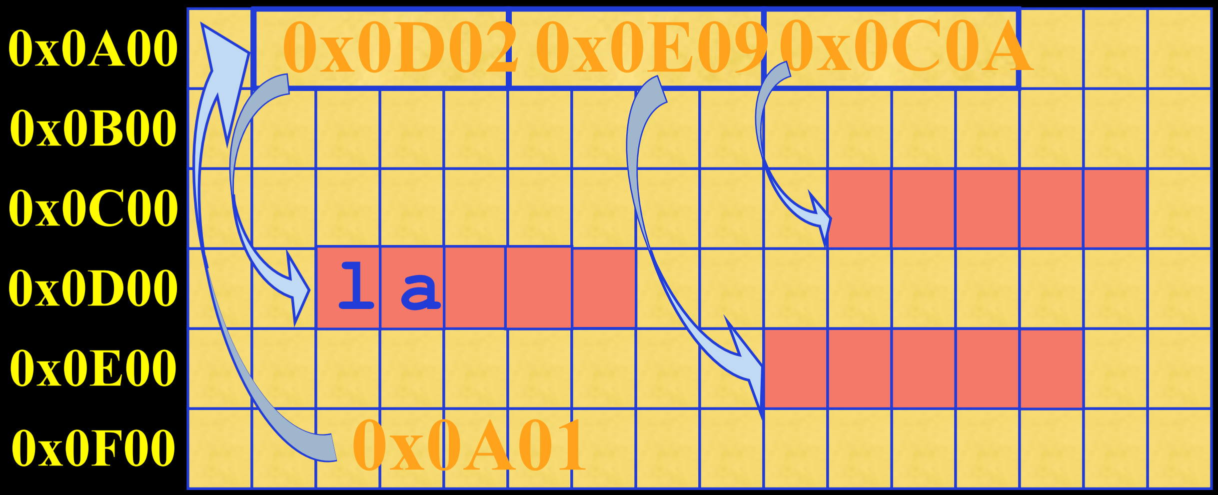
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
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’;
```

Many dimensions (suite)

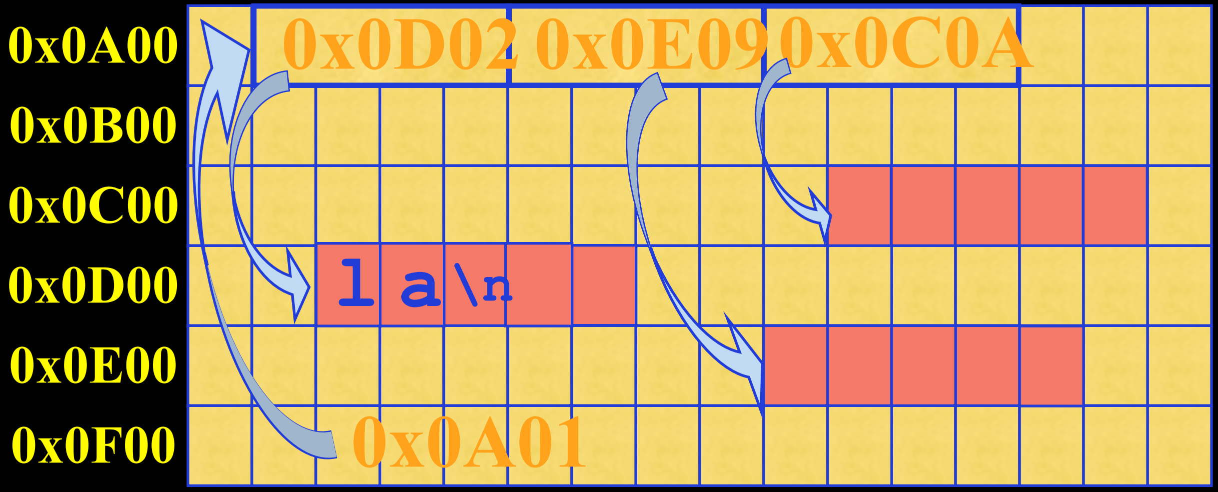
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
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’;
```

Many dimensions (suite)

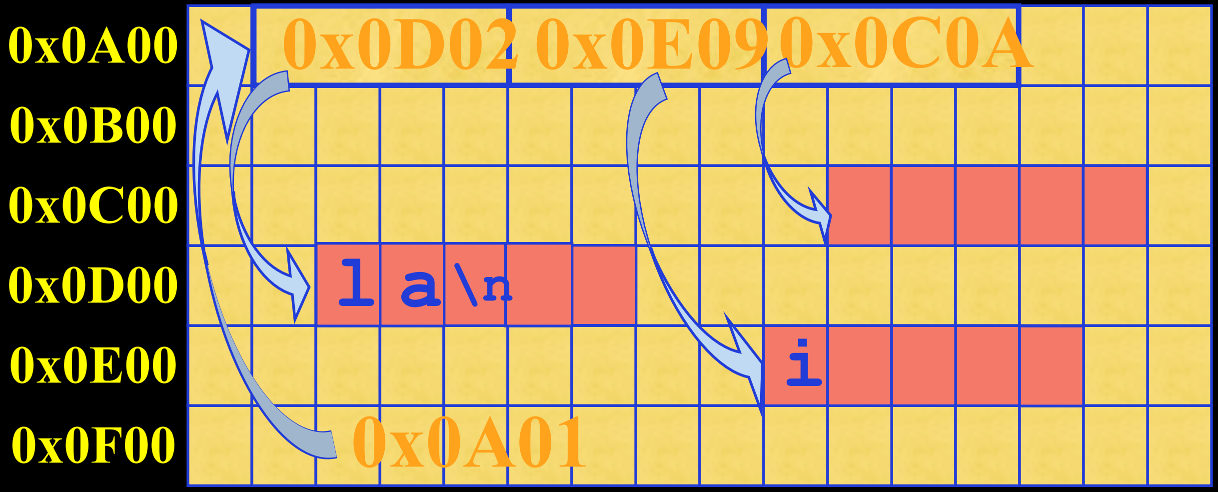
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
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’;
```

Many dimensions (suite)

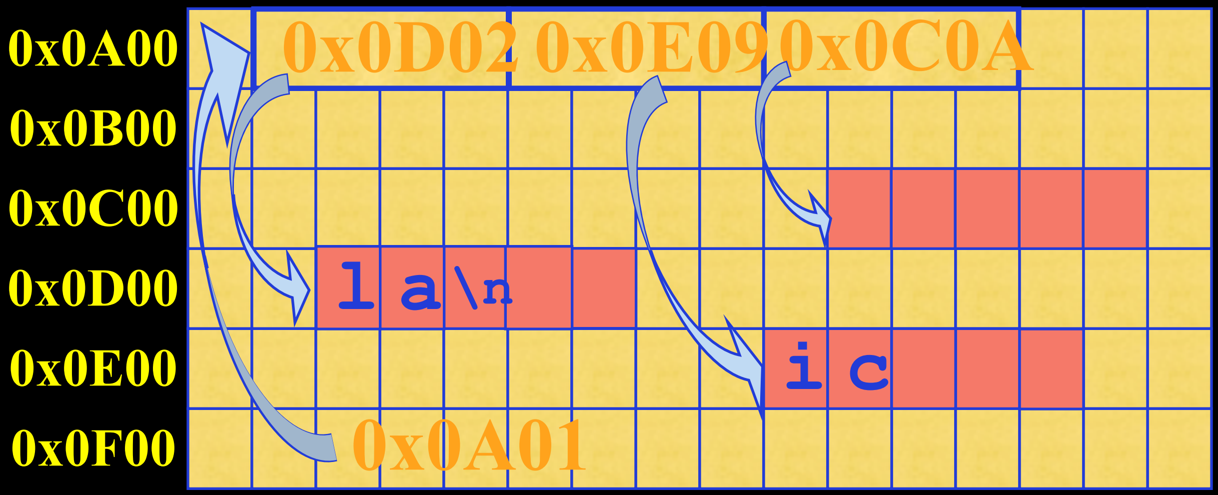
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
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’;
```

Many dimensions (suite)

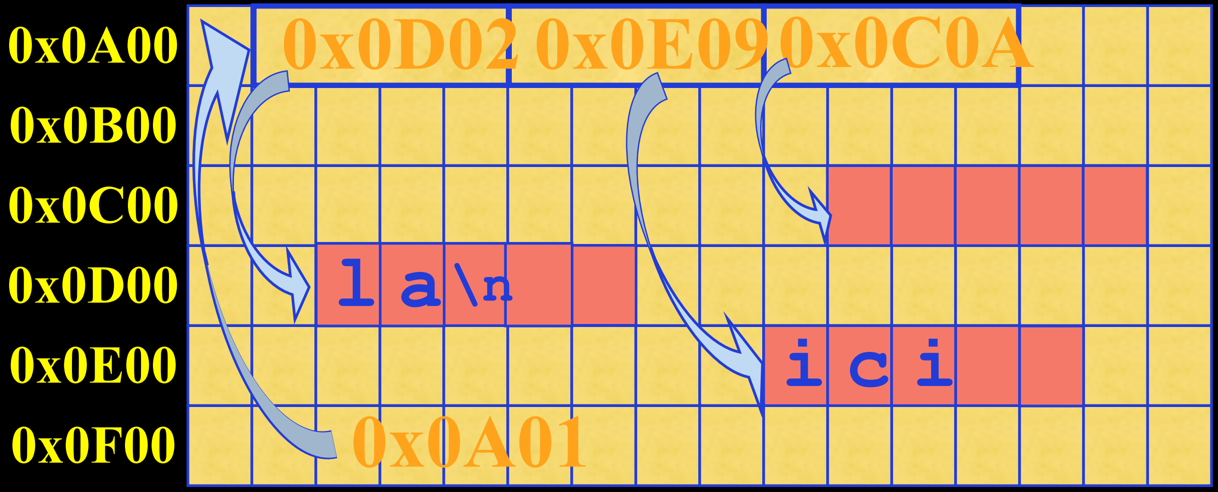
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
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';
```

Many dimensions (suite)

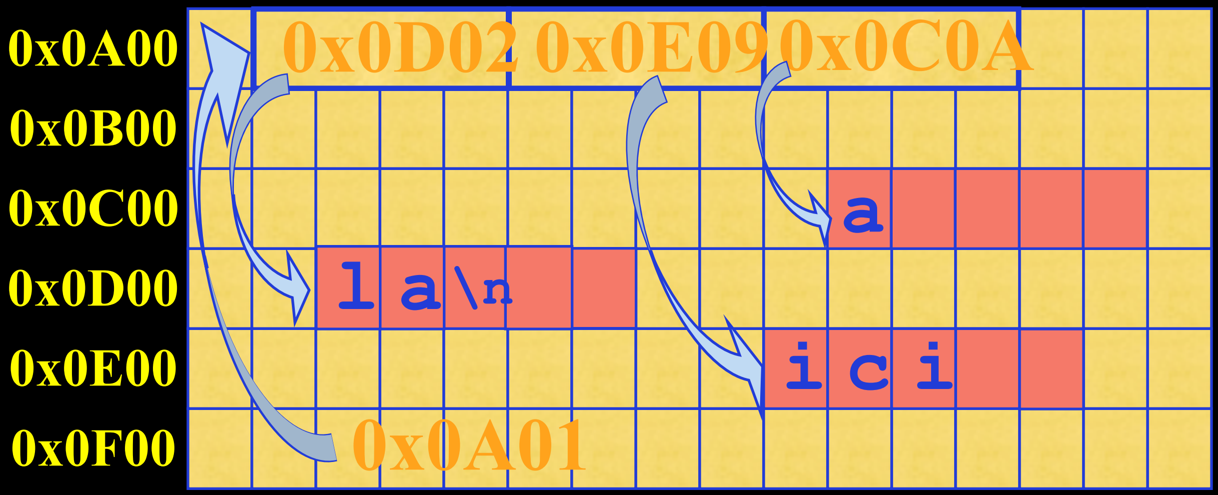
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
T[0][0] = '1'; 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)

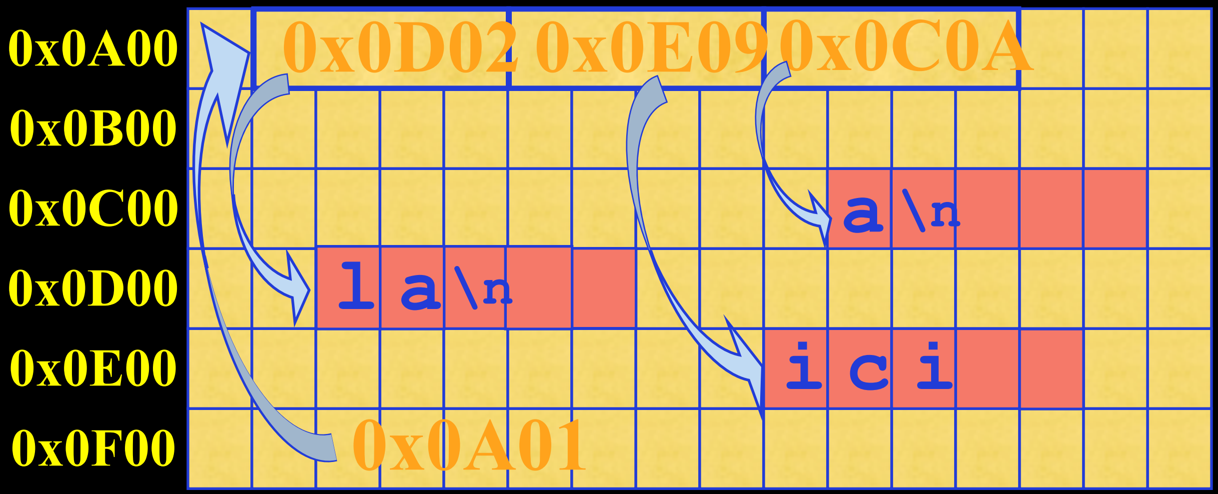
```
int i, m = 3, n = 5;
char **T = malloc(m * sizeof(char *));
for (i=0; i<m; i++) T[i] = malloc(n * sizeof(char));
```



```
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’;
```

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’;
```

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 : `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`**

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

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`**

```
int *pa = malloc((sizeof(int));
          /*allocation*/
```

Memory Liberation : `free`

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

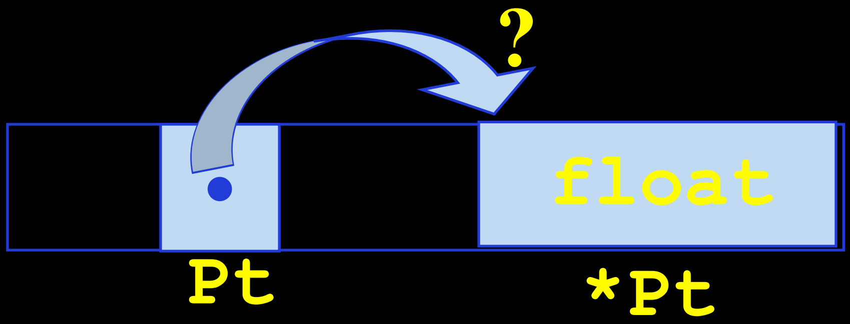
We can decide to free this memory for other usage (the memory of a computer is not infinite)

⇒ function **`free`**

```
int *pa = malloc((sizeof(int));
            /*allocation*/
 free(pa);                /*free*/
```

Functions and pointers

`float *Pt;`



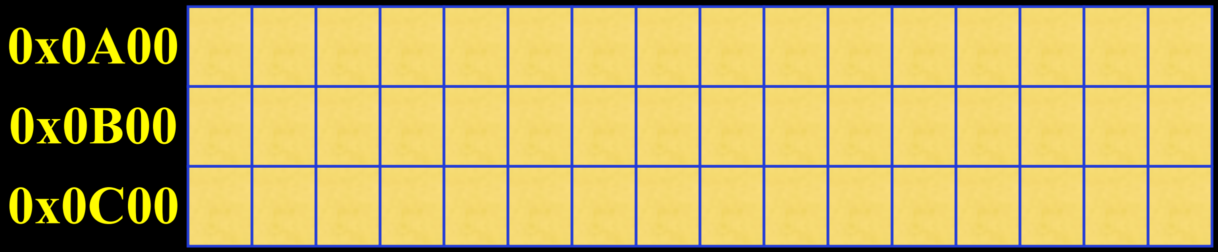
A function manipulates **copies of the values of its arguments**

⇒ cannot modify the value of its arguments

```
void function(int A, double *B)
```

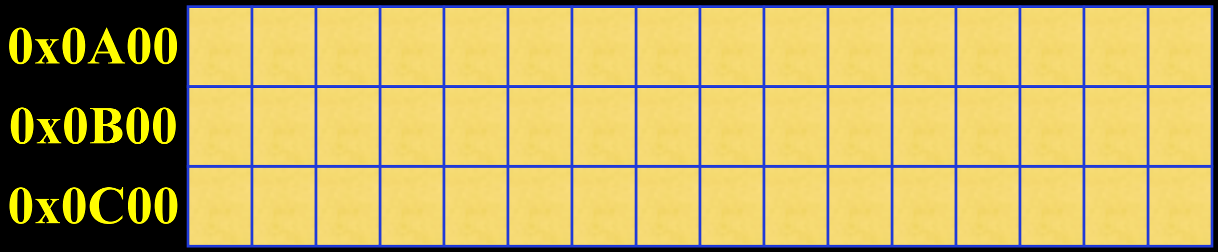
- **`int A,`** **`A`** receives a copy of 1st arg.
- **`double *B,`** **`B`** receives a copy 2nd arg., a pointer on a **`double`**

Calling by address



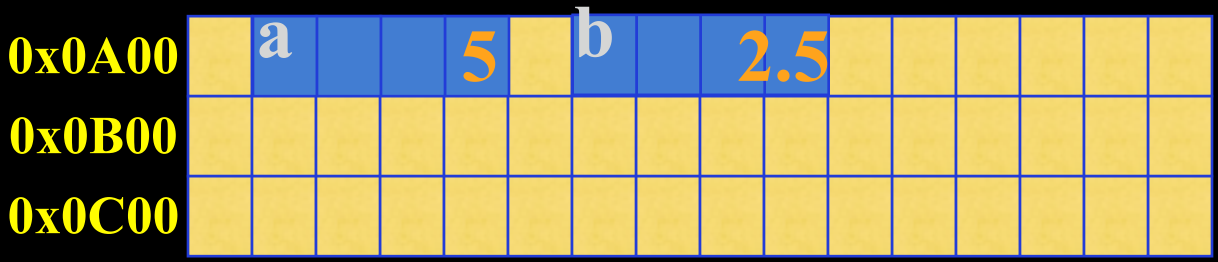
Calling by address

```
void function(int A, double *B) {…}
function(a, &b);
```



Calling by address

```
void function(int A, double *B) {…}
function(a, &b);
```



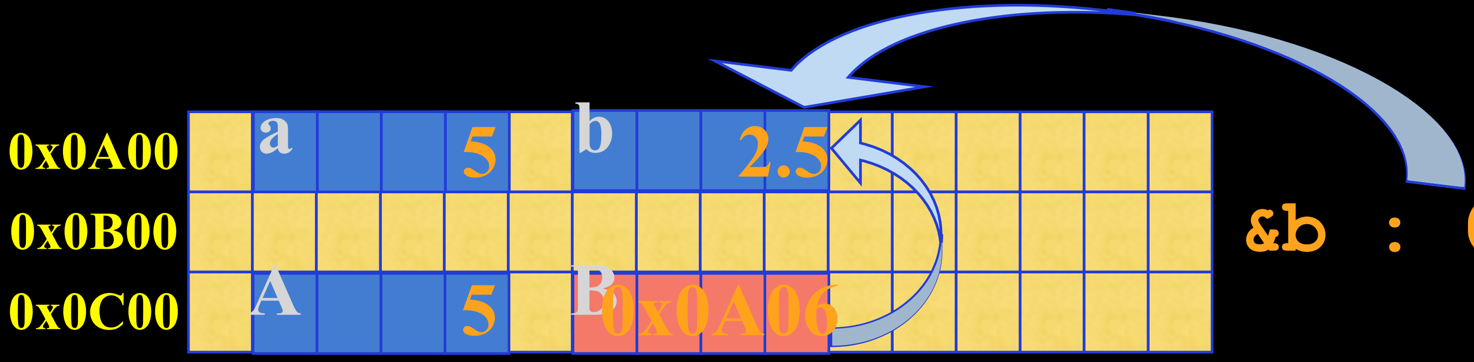
`&b : 0x0A06`

Calling by address

```
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`



`&b : 0x0A06`

Calling by address

```
void function(int A, double *B) {…}
function(a, &b);
```

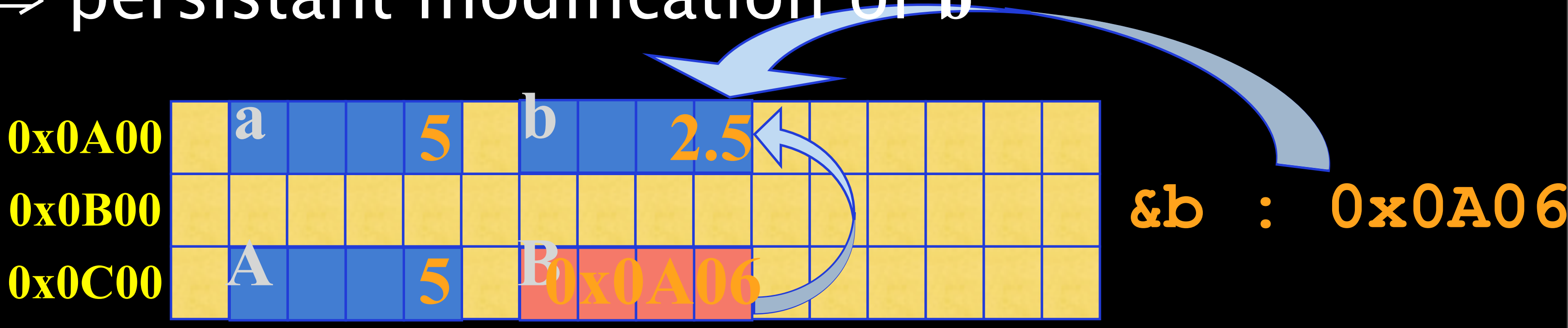
`A` is a copy of `a`

`B` is a copy of `&b`, a pointer on `b`

⇒modification of `*B` will modify

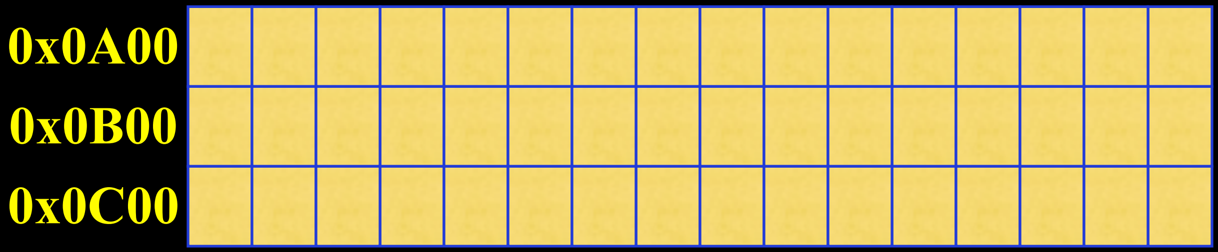
the object pointed by `B`, which is `b`

⇒ persistant modification of `b`



Example II

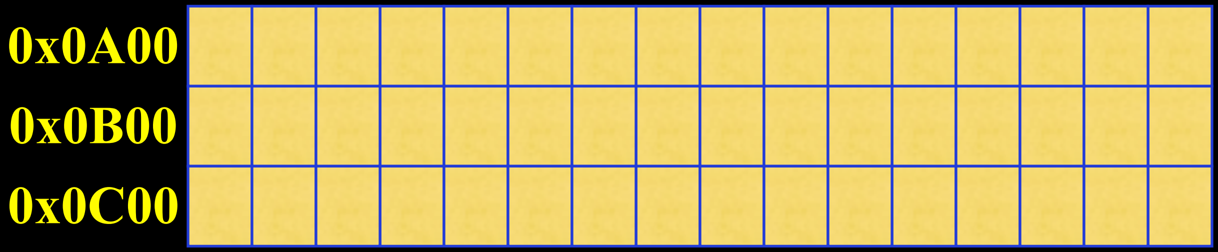
```
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```



Example II

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

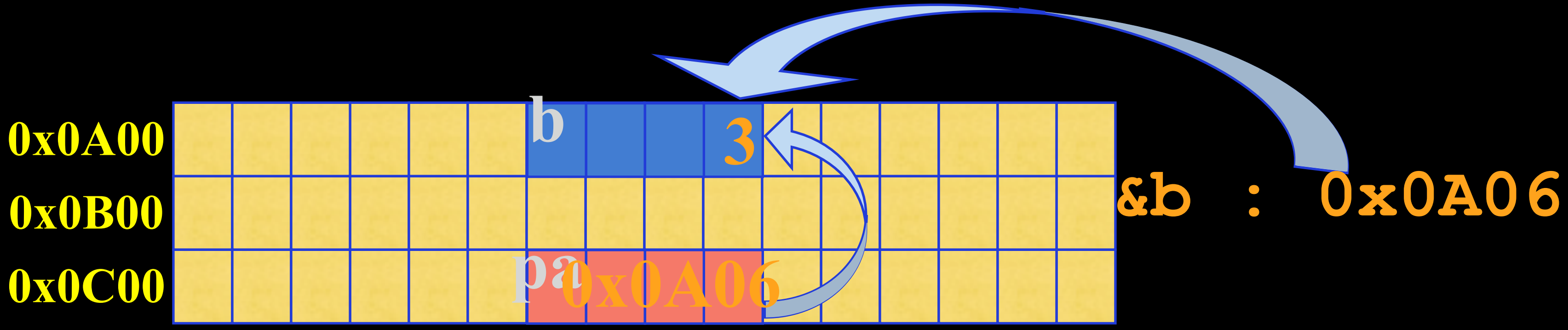
```
     b = 3;
incrementation(&b);
```



Example II

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

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



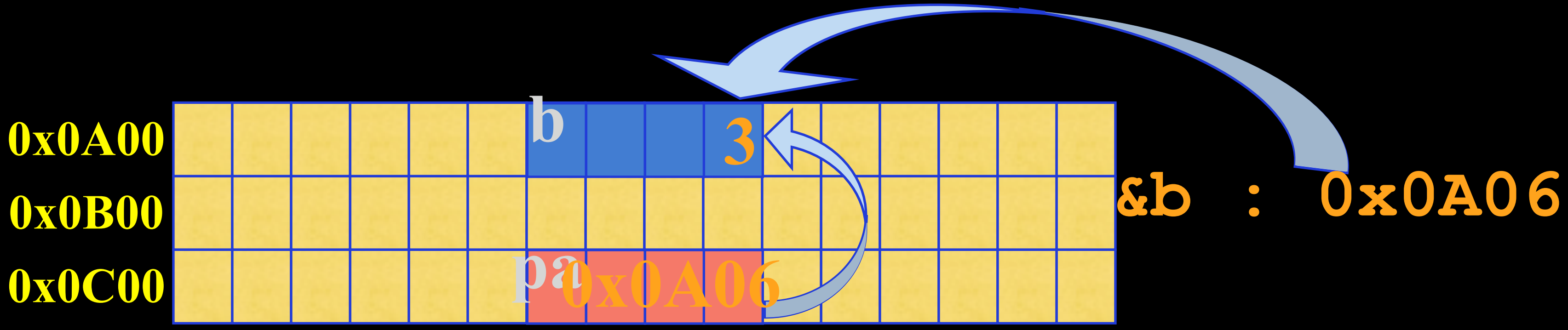
Example II

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

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

A modification of `*pa` will modify `b` :

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



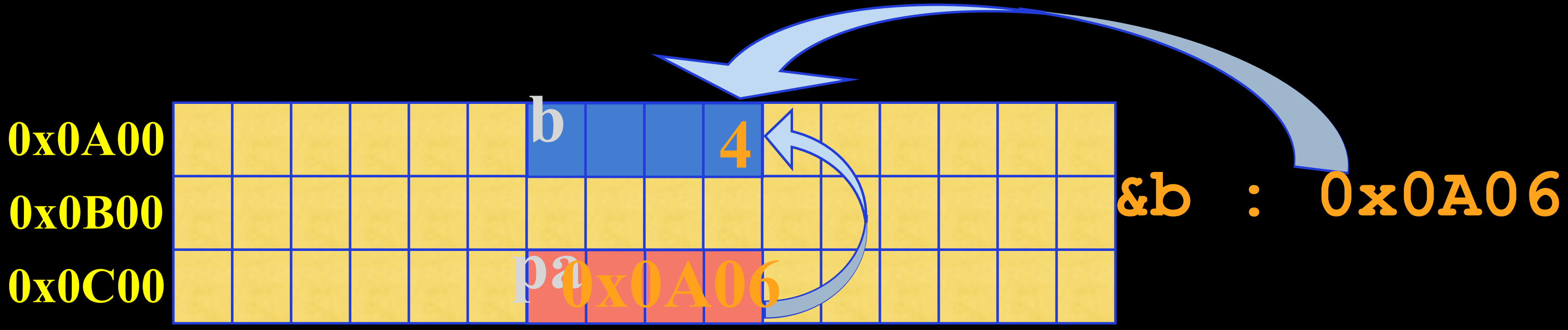
Example II

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

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

A modification of `*pa` will modify `b` :

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



Example II

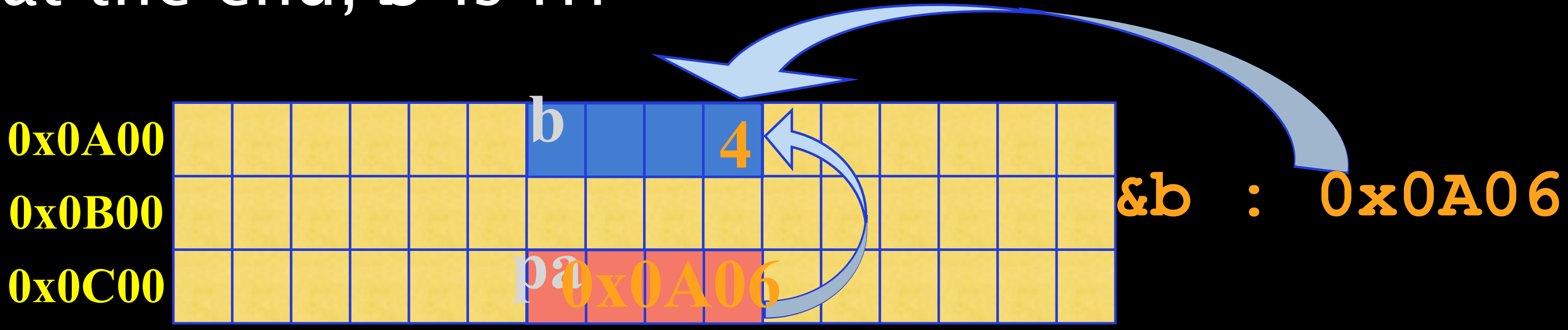
```
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```

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

at the end, `b` is ???

A modification of `*pa` will modify `b` :

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



Example II

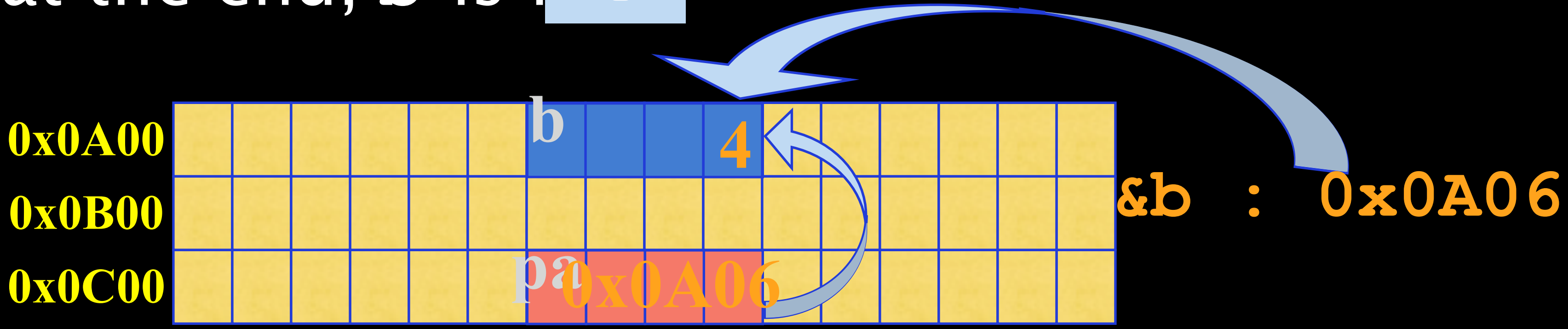
```
void incrementation(int *pa) {
    (*pa) = (*pa)+1;
}
```

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

A modification of `*pa` will modify `b` :

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

at the end, `b` is ? **4**



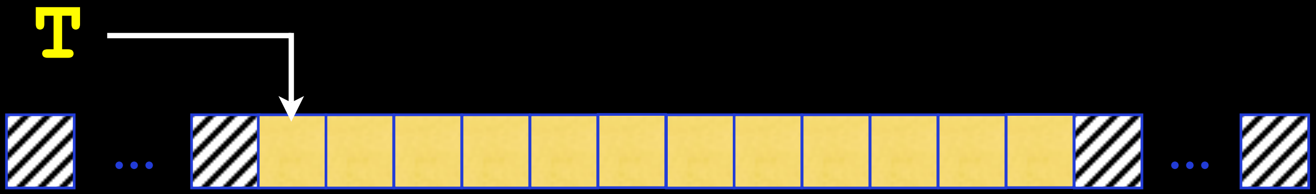
The array

The function manipulate only copies of its arguments

```
void initialisation(int *tab, int n){…}
initialisation(T,1);
```

⇒ the value of an array `T` is a pointer on the first cell (address of the first cell)

`T` is a memory address



Permanent Allocation

```
#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

```
#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

```
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

```
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

```
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

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

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

```
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