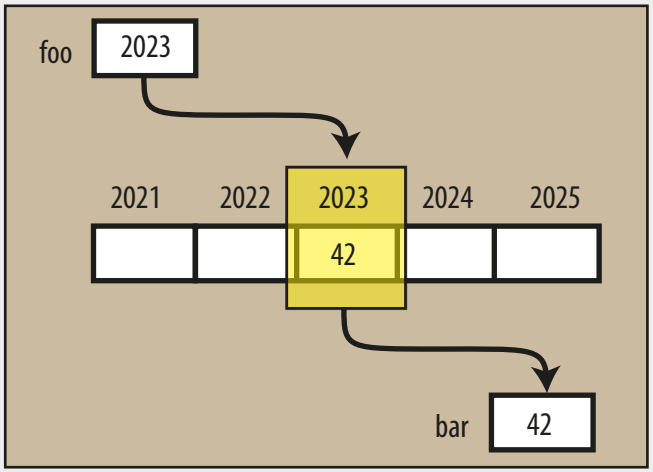
Computer Architecture

Paul-André Melliès

Lecture 6 : Programming in C

C. pointers, structures and unions



Pointers

Modifying an array by calling a function

```
void print(int a[], int size){
  int i;
  for (i = 0; i < size; i++) printf("%d ", a[i]);
}

void erase(int a[], int size){
  int i;
  for (i = 0; i < size; i++) a[i] = 0;
}

int main() {
  int array[] = {3, 38, 23, 17};
  print(array, 4);  // "3 38 23 17 "
  erase(array, 4);
  print(array, 4);  // "0 0 0 0 "   (?!...)
  return 0
}
```

Call by reference

Key fact :

When a function is called with an array as parameter,
the modification performed by the function on the array
alter the value of the array transmitted by the caller.

But why ?

And what is the type of `a` in `void erase(int a[], ... )` ?

In the function call `erase(array,4)` what is exactly transmitted ?

- the « value of an array » ? no ...
- something else ? yes, the address of the array `array` in memory !

The parameter `a` is in fact of type « address of the first element
of an array of integers » which reduces to : « pointer to int ».

Variables vs. physical addresses

A basic principle already mentioned several times :

Variables are **locations** in the computer's memory which can be accessed by their identifier (= name).

In this way, the program does not need to care about the physical address of the data in memory: it simply uses the identifier whenever it needs to access the variable.

Hence, whenever a variable is declared, the memory needed to store its value is assigned a specific location in memory (= its memory address).

Address-of-operator &

An important point to remember :

In general, the C program does not decide the exact memory addresses where its variables are stored: this is the task of the operating system.

It may be useful however for the program to obtain the address of a variable at run-time in order to access data cells that are at a certain position relative to it in the memory.

Address-of-operator `&`

So, the address of a variable can be obtained by preceding the name of the variable with an ampersand sign `&` known as address-of-operator :

```
ptr = &myvar
```

The important point here is that by preceding the variable **myvar** by an address-of-operator (the operator ampersand, noted `&`) we are no longer assigning the content of the variable myvar, but its address !

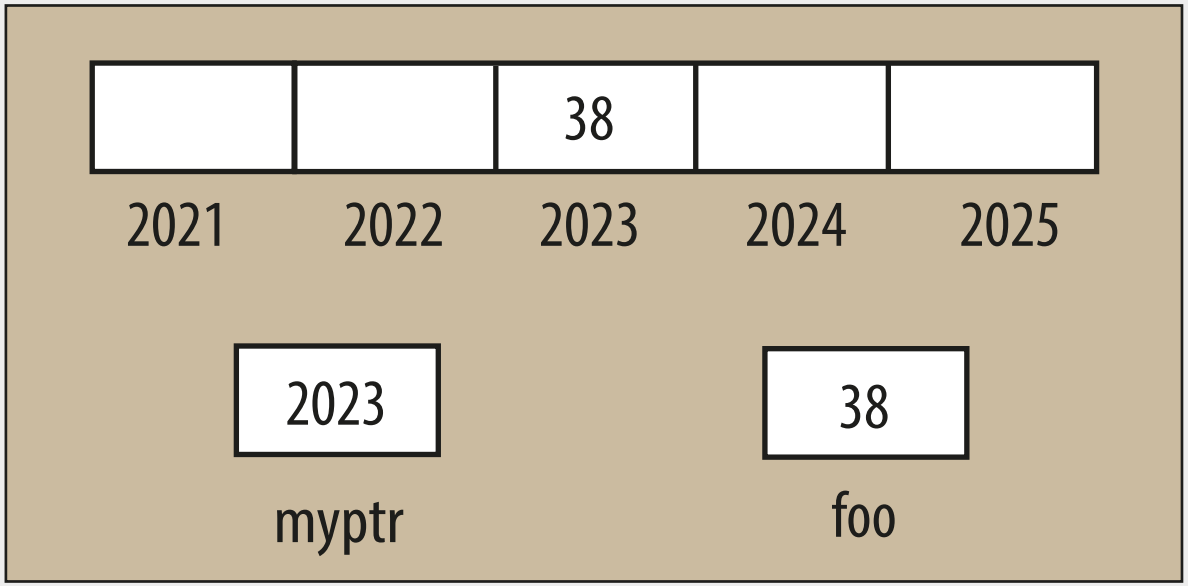
Of course, this address is not known to the program before runtime, but for the sake of explanation, we may suppose that it is assigned at run-time the specific memory address 2023.

Address-of-operator &

In that case, if we compile and run the C program below:

```
myvar = 38;
myptr = &myvar;
foo   = myvar;
```

we will reach the following state of the memory :



Pointers

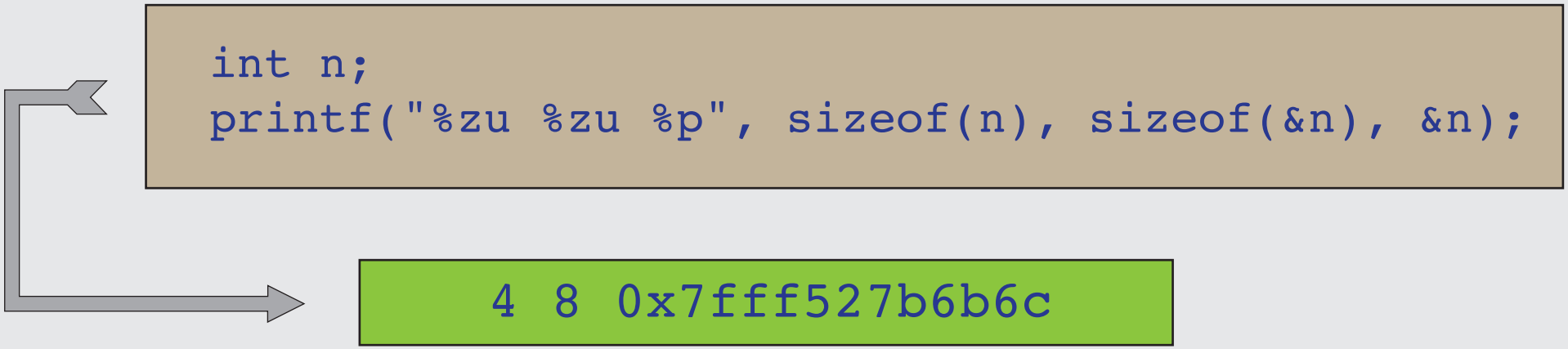
A variable like `myptr` which stores the address of another variable is called **a pointer**.

A pointer is said to « point at » the variable whose address it stores.

Key idea : pointers can be used to access the variable they point at, even if the program only knows the address at run-time !

The address of an integer variable

By convention, the address of a variable is the address of its first memory cell :



```
int n;
printf("%zu %zu %p", sizeof(n), sizeof(&n), &n);
```

```
4 8 0x7fff527b6b6c
```

`sizeof(n)`	:	the size of the integer variable `n`
`sizeof(&n)`	:	the size of the address of the variable `n`
`&n`	:	the address of its first memory cell printed here in hexadecimal

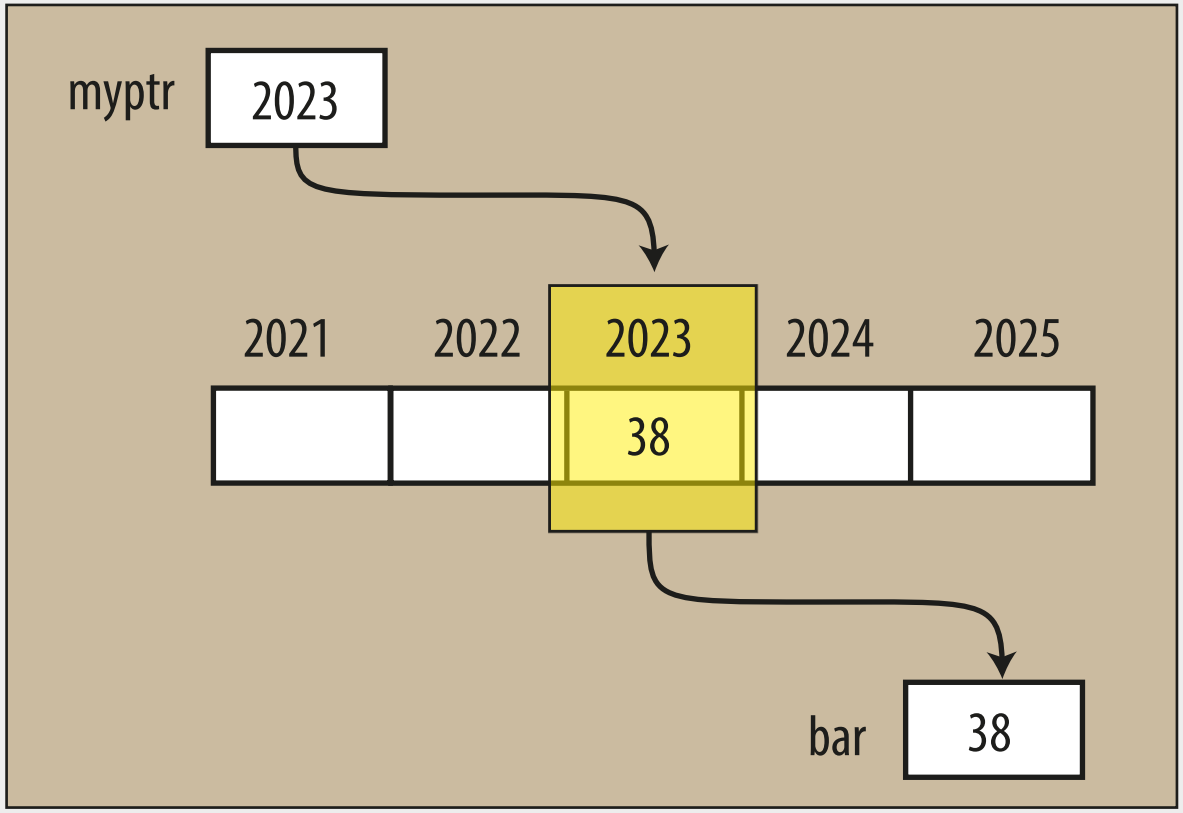
The dereference operator *

The following statement :

```
bar = *myptr;
```

should be read
« the variable `bar` receives the value of the variable pointed by `myptr` »

In the previous situation,
one thus gets at run-time :



Duality between reference and dereference

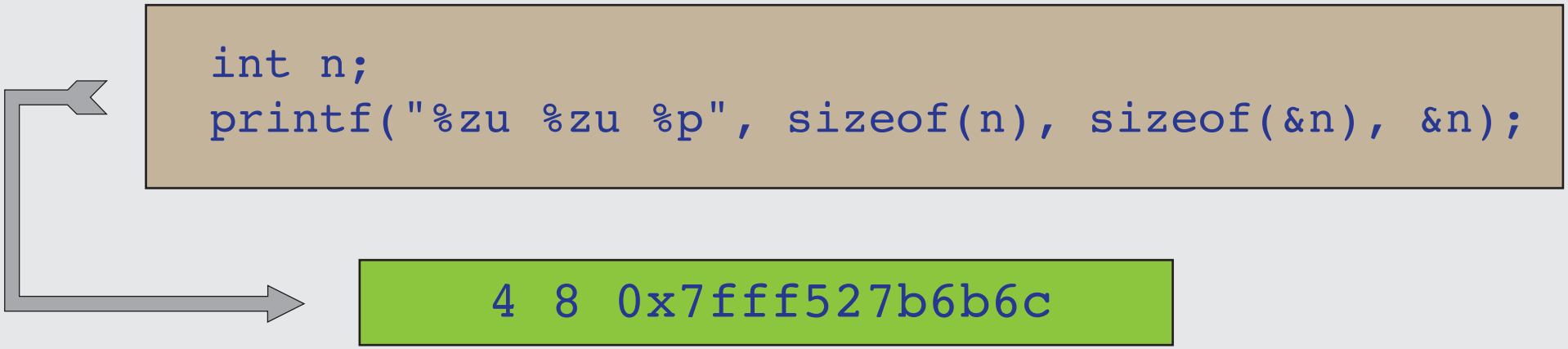
The address-of-operator `&` can be read as « address of »

The dereference-operator `*` can be read as « the value pointed by »

```
myvar == 38
&myvar == 2023

foo == 2023
*foo == 38
```

The address of an integer variable

By convention, the address of a variable is the address of its first memory cell :



```
int n;
printf("%zu %zu %p", sizeof(n), sizeof(&n), &n);
```

```
4 8 0x7fff527b6b6c
```

`sizeof(n)`	:	the size of the integer variable `n`
`sizeof(&n)`	:	the size of the address of the variable `n`
`&n`	:	the address of its first memory cell printed here in hexadecimal

Declaration of pointers

The pointer variables are declared in the following way :

```
int * myptrint;
char * myptrchar;
double * myptrdouble;
```

Although they point to variables of different data types, these variables are likely to occupy the same amount of space in memory, corresponding to the size of a memory address in the machine.

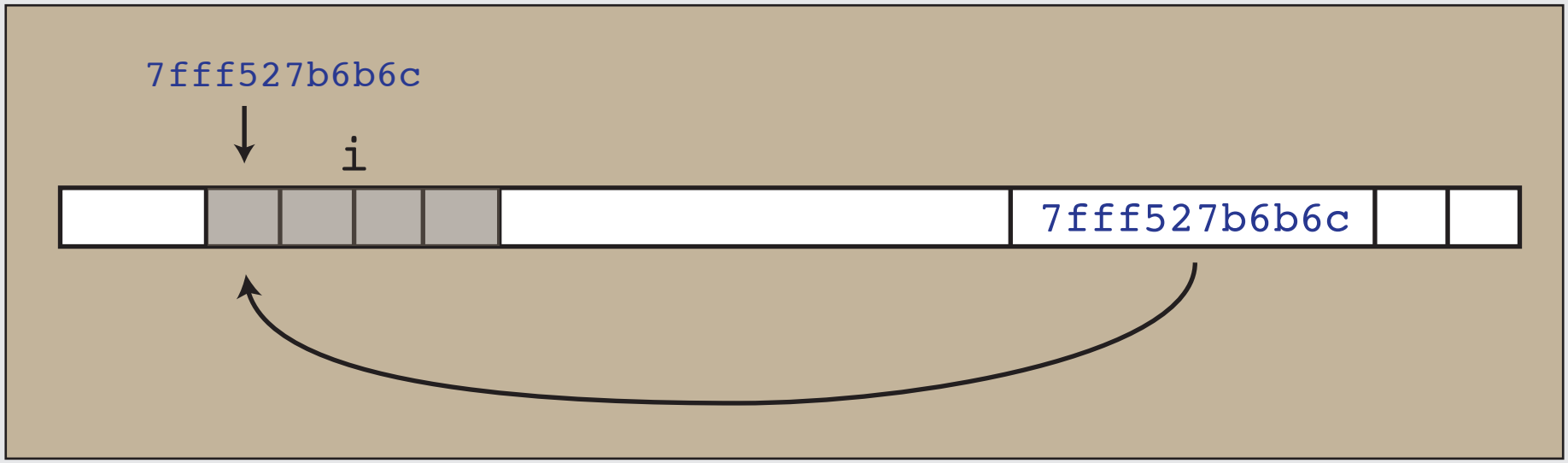
However, these pointers point to variables with different types and thus which do not occupy the same amount of space in memory.

Declaration and initialization of pointers

```
int i = 42;      // variable int
int *p;          // pointer towards an int variable

p = &i;          // the address of i is stored in p

printf("value of i         : %d\n", i);   // print 42
printf("value of i, via p : %d\n", *p);  // print 42

*p = 17;         // store a value in the address of i
                 // and thus in i

printf("value of i         : %d\n", i);   // print 17
printf("value of i, via p : %d\n", *p);  // print 17
```

Basic principle

As long as the pointer `p` points to the variable `i`
the notations `*p` and `i` are **equivalent**
from an operational point of view (they have the same effect)



Typically : when `i` is located at the address `7fff527b6b6c`

`i = 38;` write `38` in the variable `i` at the address `7fff527b6b6c`

`*p = 38;` write `38` at the address stored in `p` (at `7fff527b6b6c`)

The value of a pointer can be altered

```
int i = 38, j; // int variables
int *p;        // pointer to int, not initialized

p = &i;        // p points to i
*p = *p + 1    // equivalent to i = i+1;

printf("value of i : %d\n", i);   // print 39

p = &j;        // p points now to j
*p = i + 1;    // equivalent to j = i+1;

printf("i : %d, j : %d\n", i, j);   // print 39, 40
```

Another example

```
int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  printf("%d\n",firstvalue);
  printf("%d\n",secondvalue);
}
```

```
firstvalue == 10
secondvalue == 20
```

Two pointers can point to the same variable

```
int i = 17;  // int variable
int *p, *q;  // pointers to int, not initialized

p = &i;      // p points to i
q = p;       // the address of i is copied in q

*p = *p + 1  // equivalent to i = i + 1;
*q = *q + 1  // equivalent to i = i + 1;

printf("i : %d\n", i);   // print 19
```

Another introductory example

```
int main ()
 {
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20

  printf("my first value is equal to %d\n", firstvalue);
  printf("my second value is equal to %d\n", secondvalue);
 }
```

```
my first value is equal to 10
my second value is equal to 20
```

Pointers as parameters

Pointers as parameters of a function

```
void increment (int *p)
{
  *p = *p + 1;
  // access to the content of the variable
  // whose address is stored in the pointer p
  // add 1 and store it in the same variable
}

int main()
{
  int i = 17;
  increment(&i);
  printf("i : %d\n", i);   // print 18
  return 0;
}
```

Here, the parameter `*p` of the function `increment` is a of type « pointer of int »

A typical application

Imagine that you want to program a function which swaps two integers...

How would you define it ?

The basic function would not work because every function is call-by-value :

does not work !!!

```
swap (a,b)
```

does not work !!!

```
void swap (int x, int y)
 {
   int temp;

   temp = x;
   x = y;
   y = temp;
 }
```

A typical application

Imagine that you want to program a function which swaps two integers...

How would you define it ?

The correct implementation is by « call by reference » and uses pointers :

```
swap (&a,&b)
```

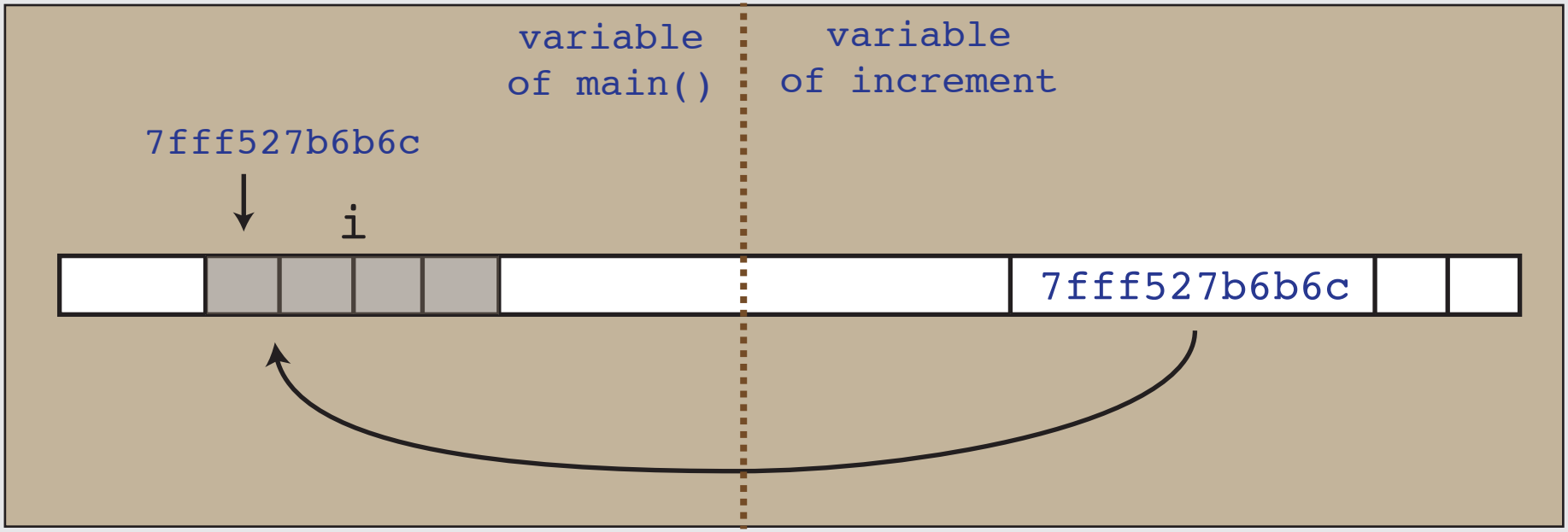
```
void swap (int * xptr, int * yptr)
 {
   int temp;

   temp  = *xptr;
   *xptr = *yptr;
   *yptr = temp;
 }
```

A typical application

```
void swap (int * xptr, int * yptr)
 {
   int temp;

   temp = *xptr;
   *yptr = *xptr;
   *xptr = temp;
 }

int main(){
  int a=0, b=1;
  printf("a=%d b=%d, a, b);
  // print "a=0 b=1"
  swap(&a,&b);
  printf("a=%d b=%d, a, b);
  // print "a=1 b=0"
  return 0;}
```

Call by reference

The principle that `*p` is equivalent to the variable `i` also holds when the pointer `p` points to a variable `i` in another function !



How the function call of `increment` works :

- the pointer `&i` is copied and allocated in a new variable `p`
- after the copy, the pointer `p` points on the variable `i` of `main()`
- the instruction `*p = *p + 1` is equivalent to `i = i + 1`

Pointers of pointers of pointers of ...

A pointer is a variable and is thus located at an address :

```
int i,          // int
    *p,         // pointer of int
   **q,         // pointer of pointer of int
  ***r;         // pointer of pointer of pointer of int

   p = &i;     // p points on i
   q = &p;     // q points on p
   r = &q;     // r points on q

***r = 17;     // equivalent to i=17
```

```
      *r ≡ q
    **r ≡ *q ≡ p
***r ≡ **q ≡ *p ≡ i
```

Immediate initialization of pointer

The instruction :

```
int i, *p = &i, *q = p;
```

is equivalent to the sequence of declarations and assignments:

```
int i, *p, *q;

p = &i;
q = p;
```

Remark that there is no star `*` in the two last assignments:
in a declaration, the star `*` is not an access to a pointed variable
but only an indication about the type of `p` and `q` (that is : '`int *`')

Something to keep in mind

The size of a pointer to an integer :

```
int i, *p = &i;
printf("%zu\n",sizeof(p)); // print 8
```

is the same as the size of a pointer to a double :

```
double i, *p = &i;
printf("%zu\n",sizeof(p));  // print 8
```

and the same as the size of a pointer to a char :

```
double i, *p = &i;
printf("%zu\n",sizeof(p));  // print 8
```

Correspondence between arrays and pointers

```
int array[4], *p;

// (1) automatic conversion of the name of the array a
//     in its allocated address, of type pointer of int.
// (2) store this address in p

p = array;

p[0] = 38;  // access to the elements of the array
p[1] = 0;   // with the usual notations
p[2] = 17;  // via the pointer p
p[3] = 12;

// the content of the array is now {38, 0, 17, 12}
```

Conversion rule on arrays

Every name of array which appears inside an expression
is automatically converted into its « address » ---
which is the address of the first byte of its first element.

The array also receives the type « pointer to the type τ »
where τ is the type of the elements of the array.

```
p = array;
```

The array `array` is converted into a pointer to its « address »
and is given automatically the type « pointer to `int` ».

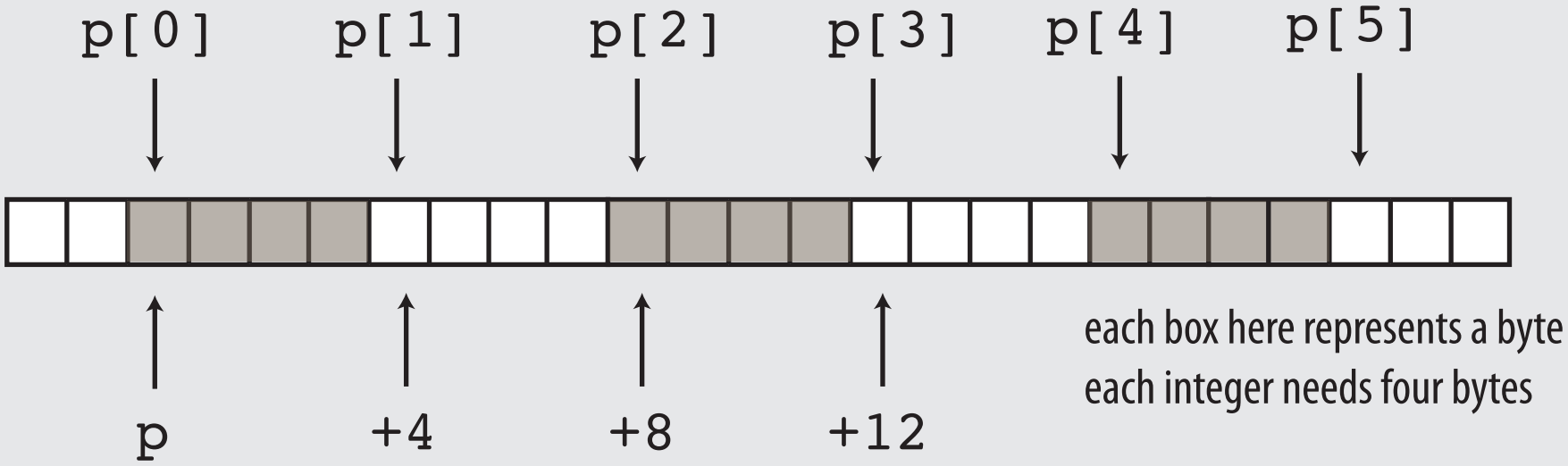
The pointer `p` is of type « pointer to `int` »
hence the assignment is correct from the point of view of typing.

Indexed notation for pointers

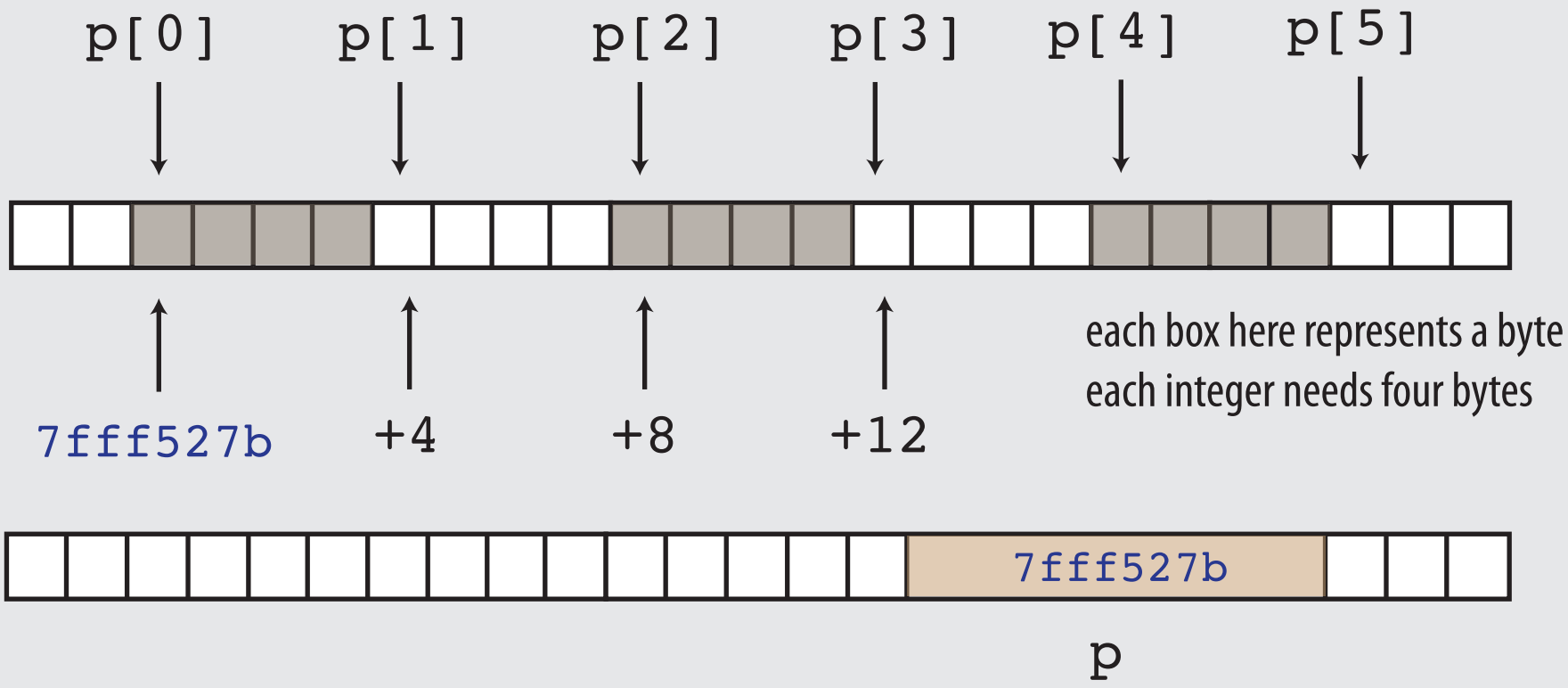
When a pointer `p` of type « pointer to τ » contains an address, the notation `p[i]` enables one to access the address `p` shifted by `i` times the size of the elements of type τ.

Size means the number of bytes necessary to allocate an element of type τ.

Typically, the size of an integer is 4 bytes. Hence :



Indexed notation for pointers



For the array `a` starting at address `7fff527b`
and the pointer `p` containing the same address `7fff527b`

the integer `p[0]` is located at the address `7fff527b`
the integer `p[1]` is located at the address `7fff527b + 4`
the integer `p[2]` is located at the address `7fff527b + 8`

Compare two arrays ?

What does the code below print ?

```
int t[] = {38, 10, 12};
int u[] = {38, 10, 12};

if (t == u)
  printf("equal!\n");
else
  printf("different!\n");
```

Compare two arrays ?

Evaluation of `(t == u)` :

The names of arrays `t` and `u` are converted into their addresses
and the equality test compares the addresses of the two arrays

The two arrays `t` and `u` have the same content
but they are located at different addresses in the memory

For that reason, `(t == u)` is equal to 0 (= false).

So, in order to compare the content of two arrays, there is no other choice than test the elements one after the other using a loop.

Assignment of a variable of array type ? Simply impossible !!!

An important exception to the conversion rule is :

> On the left of an assignment, a name of array is not converted to a pointer.
>
> It thus remains of type « an array with ... elements of type ... »

Guess what happens when one tries to compile the following code :

```
int main(){
  int t[4] = {38, 10, 12}, u[4];
  u = t;  //  ?????
  return 0;
```

Assignment of a variable of array type ? Simply impossible !!!

The message of `gcc` :

```
error: incompatible types when assigning
to type ‘int[4]’ from type ‘int *’
```

In other word, an error of typing...

```
u = t;
```

- `u` conserves its type of array with 4 integer elements
- `t` is converted to its address of type « pointer of int »

Because of the conversion on the right, it is impossible to assign to the array `u` an expression of the appropriate type !!!

Conversion during compilation

During compilation, the arrays given as parameters to functions are treated as pointers --- and may be thus written as pointers :

```
void erase (int a[], int size){ ... }
```

is in fact equivalent to

```
void erase (int *a, int size){ ... }
```

Note that the second notation is more usual than the first one

This means that the array `a` is treated as a pointer inside the function `erase`. Accordingly, the notation `a[i]` used for arrays in `erase` is in fact an indexed notation on a pointer to `int` (as described above).

Conversion during compilation

```
void erase (int a[], int size){ ... }

int main(){
  int array[100];
  erase(array, 100);
  return 0;
}
```

Execution of the function call `erase(array)` :

- the name `array` is converted into a pointer to its address
- this pointer is copied in a new variable `a`
- in `erase` the instructions `a[i]=0` erase the content of `array`

There is no operational difference between `a[i]=0` and `array[i]=0` since `a` has the same address as `array`.

The ampersand & operator

Key principle :

The ampersand operator `&` can be applied to any expression which can be **on the left side** of an assignment.

In particular, it can be applied to `array[0]`, `array[1]`, etc ...

Typically : `&(array[i])` is equal to `array` shifted `i` times the size of an element of the array.

In particular :

`p = array` and `p = &(array[0])` are equivalent

Similarly, once `p` has been assigned as `p = &(array[2]);`

- `*p` and `p[0]` and `array[2]` are equivalent
- `p[1]` and `array[3]` are equivalent
- `p[2]` and `array[4]` are equivalent

Size of array vs. size of pointer

For every variable `var`, the expression `sizeof(var)` is equal to the size of the variable in bytes.

Another exception to the conversion rule :

> When applied to `sizeof` an array variable remains a variable of type « array of ... elements of type ... »

```
int array[100], *p;

printf("size of array %zu", sizeof(array));
// print 400 = 100 * sizeof(int)

p = array;
printf("size of pointer %zu", sizeof(p));
// print 8 = sizeof(int *)
```

Pointer Arithmetic

Pointer arithmetic

```
int array[3], *p;

p = array;    // p is equal to the address of array
*p = 38;      // write 38 in array[0]

p = p + 1     // explicit shift of p :
              // one adds to p the size of int
              // more generally the size of
              // the element it points to
              // in that case 4 bytes
              // because sizeof(int)=4

*p = 17;      // store 17 in array[1]
p = p + 1     // new shift
*p = 3;       // store 3 in array[2]

// now array contains {38, 17, 3}
```

Pointer arithmetic

☞ `p + i ≡ pointer + integer_expression`

The value of a typed address of the same type as the pointer `p`

Equal to the value of the pointer `p`
shifted `i` times of the size of an element pointed by `p`

For `p` of type `int *` and of value `7fff527b6b6c`

`p + 0` is equal to `7fff527b6b6c`

`p + 1` is equal to `7fff527b6b6c + 4`

`p + 2` is equal to `7fff527b6b6c + 8`

Back to the indexed accesses

The indexed notation is in fact a syntactic short-cut :

`p[i]` is an abbreviation for `*(p + i)`

- `p + i` is equal to `7fff527b6b6c + i x sizeof(int)`

`*(p + i)` is equal to `p[i]`

- `p[i] = 38;` means `*(p + i) = 38;`

in other words :

store 38 in the address `7fff527b6b6c + i x sizeof(int)`

- Similarly, `array[i] = 38;` means `*(array + i) = 38;`

Structures

Structure type declaration

```
struct coord {     // type of structure to represent
  char name;       // coordinates of the plane
  int abs;         // labelled by chars.
  int ord;
};

int main(){
  struct coord a;  // new variable of type struct coord
  // storing values in the fields of a
  a.name = 'a';
  a.abs = 10;
  a.ord = 10;

  // read and print the values of the fields of a
  printf("%c %d %d", a.name, a.abs, a.ord);
  return 0;
}
```

Structure type declaration

```
struct name_of_the_structure {
...
field_type field_name
...
};
```

```
struct coord{
  char name;
  int abs;
  int ord;
};              // the semicolon is necessary
```

Outside any function : declares a structure type ---
that is : **create a new type,** without allocating any variable yet.

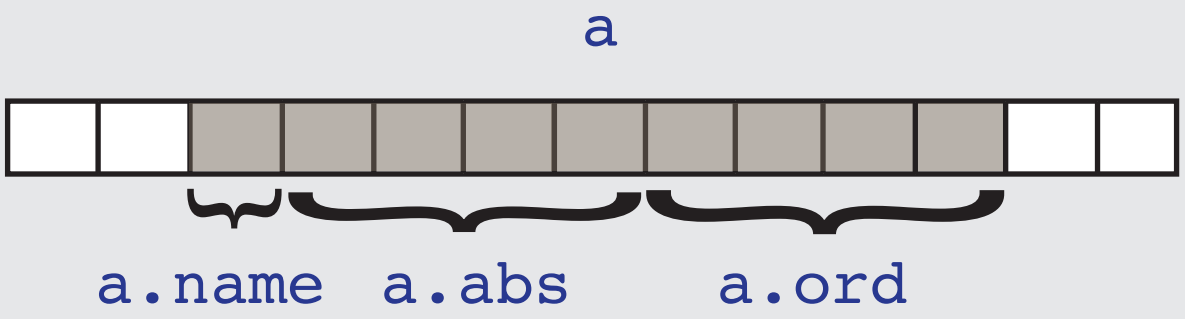
Specifies the shape of the declared variables of that type:
each variable will have three fields --- one char and two int.

Structure type declaration

☞ `struct coord a;` in the function `main()`

Allocate in memory a region of memory sufficiently large to store a char and two ints.

for instance :



☞ `a.name, a.abs, a.ord` `var_name.field.name`

Gives access to the fields of the variable `a`.

Immediate and partial initialization

- `struct coord a;`

The fields of `a` are allocated but not initialized (undefined value)

- `struct coord a = {'a', 38, 50};`

The fields of `a` are all initialized : the values are received by the field in the order the field were declared :

`name` then `abs` then `ord.`

- `struct coord a = {'a'};`

The first field of `a` is initialized,
the following fields take the value 0 of their type.

Assignment of structure variables

```
struct coord a = {'a', 38, 17} , b;

b = a;     // copy the values of the fields of a
           // into the fields of b

b.name = 'b';
b.abs = b.abs + 10;
b.ord = b.ord + 10;

printf("%c %d %d\n", a.name, a.abs, a.ord);
printf("%c %d %d\n", b.name, b.abs, b.ord);

// a 38 17
// b 48 27
return 0;
}
```

Structures as function parameters

```
struct coord translate(struct coord c, int da, int do){
  c.abs = c.abs +da;
  c.ord = c.ord +do;
return c;
}

int main(){
  struct coord a = {'a', 38, 17}, b;
  printf("%c %d %d\n", a.name, a.abs, a.ord);
  // a 38 17
  b = translate(a, 10, 10);
  b.name = 'b';
  printf("%c %d %d\n", b.name, b.abs, b.ord);
  // b 48 27
  return 0;
}
```

Here, the structure is sent as parameter and returned as return value.

Values « nearly » like values of basic types

Structures can be copied, sent to a function as parameter, returned to a function as returned value.

However, structures cannot be compared by `==` or `!=`

So, in order to determine whether two structures are equal one needs to compare their fields :

```
if (a.name == b.name &&
    a.abs  == b.abs  &&
    a.ord  == b.ord) { ... }
```

Structures vs. addresses as parameters

It is in general more efficient to give a pointer as parameter than a whole structure. Typically :

```
void shifter(struct coord *ps, int da, int do){
  (*ps).abs += da;
  (*ps).ord += do;
}
```

The function may be called in `main()` by

```
shifter(&a, 10, 10);
```

The function then modifies « in-place » the fields of the structure `a` .

A useful notation

This recipe is sufficiently important (and also natural)
to come equipped with an abbreviated notation :

`(*ps).name` can be alternatively written `ps -> name`

`(*ps).abs` can be alternatively written `ps -> abs`

`(*ps).ord` can be alternatively written `ps -> ord`

```
void translater(struct coord *ps, int da, int do){
  (ps -> abs) += da;
  (ps -> ord) += do;
}
```

A typical application of structures

Instead of using an array of arrays, a matrix of integers of size *height* x *width* can be represented as an array of integers with the convention that :

`m(i,j)`	is stored in	`array[i * width + j]`

In order to access to the content of a matrix formulated in this way, the function should know :

- the address of the first element (of type « pointer to integer »)
- its width (number of columns to compute the index)
- its height (number of lines to know the maximal index)

Except when one wants to access `m(0,0)` one always needs to pass these three parameters to the function.

Advantage of this representation : the function can treat matrices of arbitrary dimension.

First version : pass the three parameters

```
void print_matrix (int *content, int height, int width){
  int i,j;
  for (i = 0; i < height; i++){
    for (j = 0; j < width; j++){
      printf("%3d ", content[(width * i) + j]);
    }
    printf("\n");
  }
}

int main(){
  int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
  print_matrix (array, 3, 4);
  // print:    0   1   2   3
  //           4   5   6   7
  //           8   9  10  11
  return 0;
}
```

Second version :
put the three parameters together

```
struct matrix {
  int height;
  int width;
  int *content;
};
```

Example of declaration and initialization of a matrix in `main` :

```
int main() {
  // declare the content
  int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
  // declare the associated structure
  struct matrix m = {3, 4, array};
}
```

Access to the matrix

The access to the content of the matrix can be performed by a function of the form :

```
// generic access
int element(struct matrix *pm, int i, int j){
  return pm -> content[(pm -> width * i) + j];
}
```

The function `element` is then called in the following way :

```
// value of the element m(i,j)
element(&m, i, j);
```

For this function as for many others, one applies the convention : pass the address of the structure rather than the structure itself

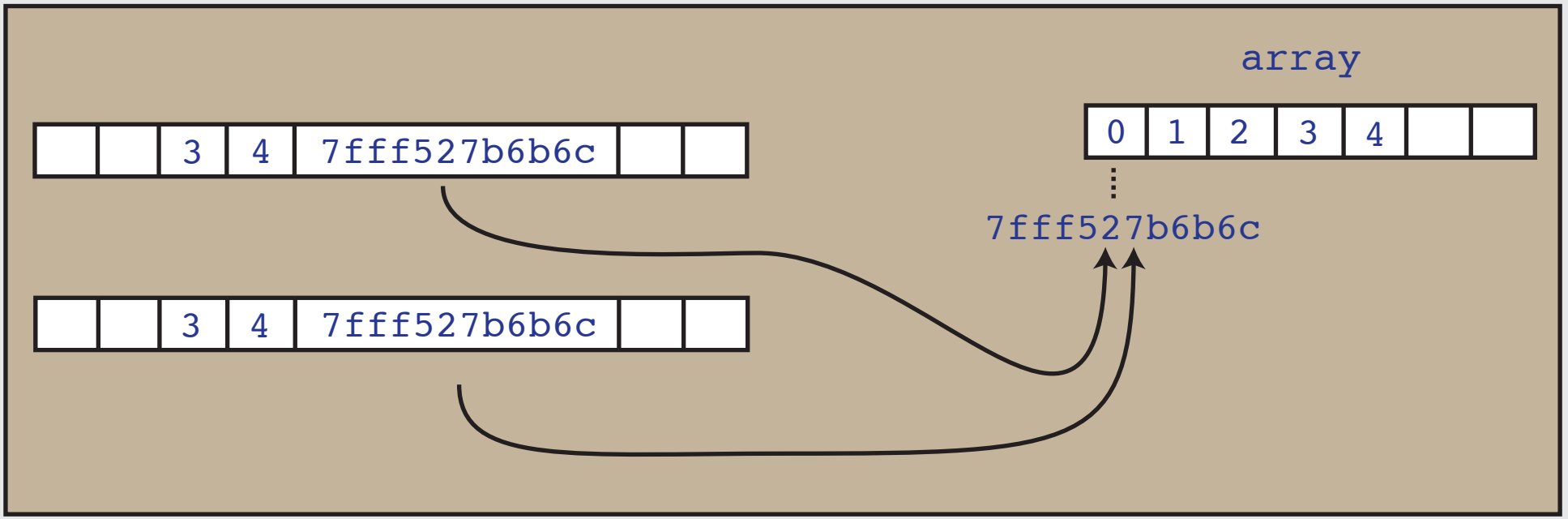
New code for the function

```
void print_matrix (struct matrix *pm){
  int i,j;
  for (i = 0; i < pm -> height; i++){
    for (j = 0; j < pm -> width; j++){
      printf("%3d ", pm -> content[(pm -> width * i) + j]);
    }
    printf("\n");
  }
}

int main(){
  int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
  struct m = {.height = 3, .width = 4, .content = array};
  print_matrix (&m);
  // print:    0   1   2   3
  //           4   5   6   7
  //           8   9  10  11
  return 0;
}
```

Remark

Copying these structures by assignment does not duplicate the associated array, only its address :

```
int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
  struct m = {.height = 3, .width = 4, .content = array};
  struct n = m;
```



Here, `m.content` and `n.content` contain the same address : the address of `array` hence `m` and `n` share the same content.

Linked lists

Linked lists
and other recursive data types

linked lists

```
struct cell {
        char value;
        struct cell * next;
};
```

binary trees

```
struct bnode {
        char value;
        struct bnode * left;
        struct bnode * right;
};
```

Unions

The unions

```
union int_or_double {    // declare the union type
  int integer;           // fields
  double with_dot;
};

int main() {
  union int_or_double u;

  u.integer = 42;              // access to u as a variable
  printf("%d",u.integer);      // of type int

  u.with_dot= 3.14;            // access to u as a variable
  printf("%d",u.with_dot);     // of type double
  return 0;
}
```

Unions

```
union int_or_double {    // declare the union type
  int integer;           // fields
  double with_dot;
};
```

is a union type declaration which creates a new type
without allocating any variable.

At a given time, a variable of that type will contain a value
of type `int` or of type `double` (but not the two at the same time)

The choice of a name of a field like `integer` or `with_dot`
specifies the way one wants to read/write in the union.

Acknowledgements and references

The slides of this lecture are largely based on a very nice and cleverly crafted introductory course on the programming language C given by Vincent Padovani in my University Paris Diderot.

Padovani is a pedagogical master and I am greatly endebted to his art here !

I have also inserted in the lecture a series of slightly more demanding examples extracted from the marvelous and so instructive book by Kernighan and Ritchie.

Please read these programs carefully and try to understand what they do...
You will discover their beauty, and learn a lot from them !