Introduction to Programming in C

(IS-FEE-10061S)

Białystok University of Technology
Faculty of Electrical Engineering
Academic year 2023/2024

Workshop no. 11 (16.05.2024)

Jarosław Forenc, PhD

Topics

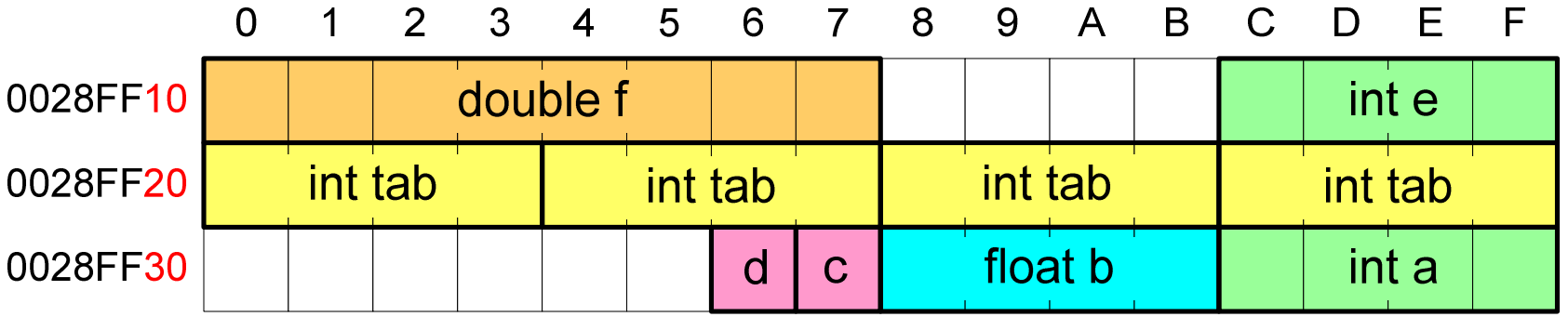
- Pointers
  - declaration, value assignment
  - relation to arrays, operations on pointers
- Dynamic memory allocation
  - calloc(), malloc(), free() functions
  - memory allocation for structure, vector and matrix

Pointers: what is a pointer?

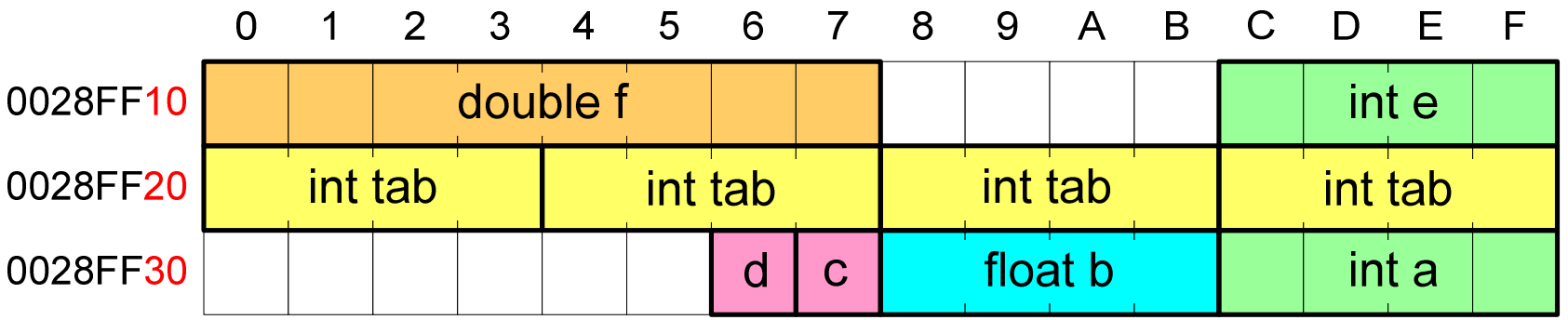
- Pointer - a variable that may contain the address of a memory area - usually the address of another variable (object)

```
int a;
float b;
char c, d;
int tab[4], e;
double f;
```

- Variables stored in the computer's memory



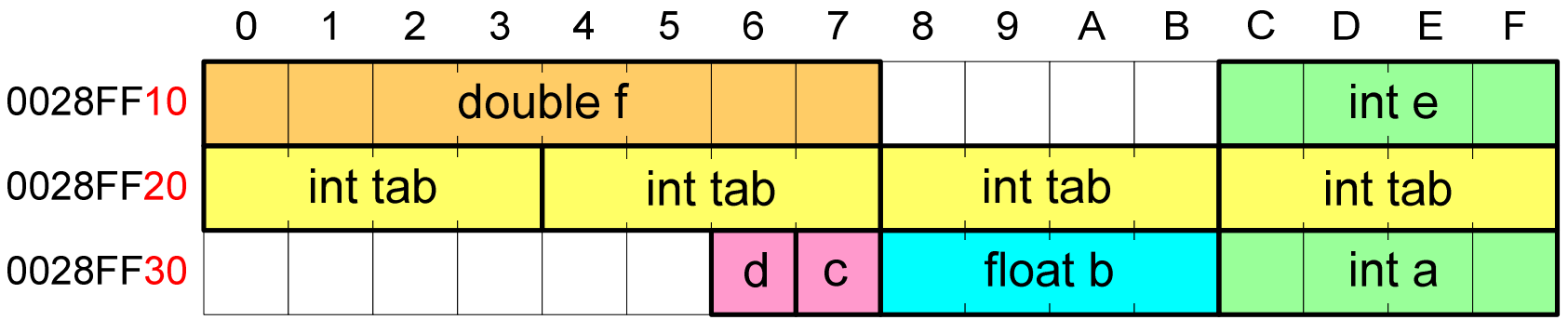
Pointers: what is a pointer?



- Each variable is located at a specific address in memory and, depending on the type, occupies a certain number of bytes
- During program compilation, all variable names are replaced with their addresses
- Printing the variable address:

```
printf("The address of the variable a: %p\n", &a);
printf("The address of the array tab:  %p\n", tab);
```

Pointers: what is a pointer?



- Each variable is located at a specific address in memory and, depending on the type, occupies a certain number of bytes
- During program compilation, all variable names are replaced with their addresses
- Printing the variable

```
The address of the variable a: 0028FF3C
The address of the array tab:  0028FF20
```

```
printf("The address of the variable a: %p\n", &a);
printf("The address of the array tab:  %p\n", tab);
```

Pointers: declaration

- When declaring a pointer (pointing variable), we must specify the type of object to which it points
- A pointer declaration looks the same as any other variable, except that its name is preceded by an asterisk (*)

```
type *variable_name;
```

or

```
type* variable_name;
```

or

```
type * variable_name;
```

or

```
type*variable_name;
```

Pointers: declarations

- Declaration of pointer to type int

```
int *ptr;
```

- We say that the type of ptr is: pointer to int
- To store the address of a double variable, we must declare a variable of type: pointer to double

```
double *ptrd;
```

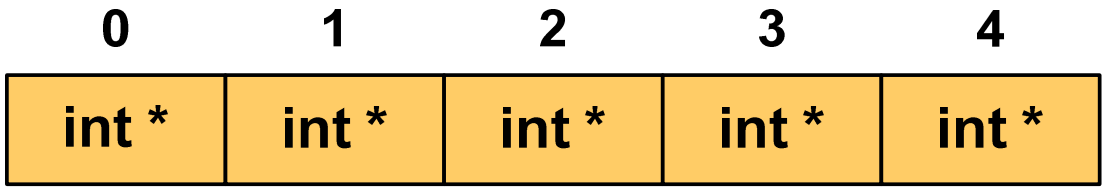
- We can declare pointers to data of any type, including pointer-to-pointer-to…

```
char **ptrc;
```

Pointers: declarations

- We can declare arrays of pointers - the tab_ptr variable is an array containing 5 pointers to int type

```
int *tab_ptr[5];
```



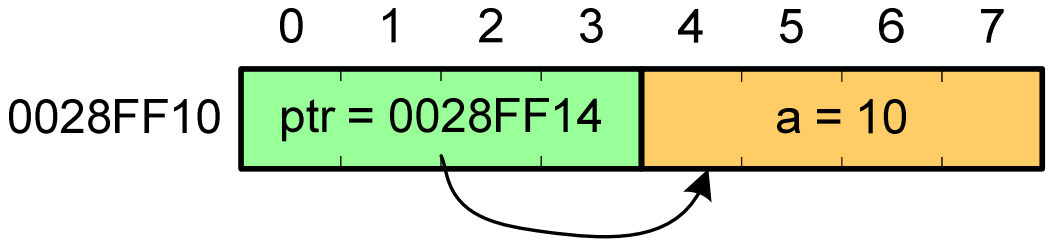
- The ptr_tab variable, on the other hand, is a pointer to a 5-element array of int

```
int (*ptr_tab)[5];
```

Pointers: assigning values to pointers

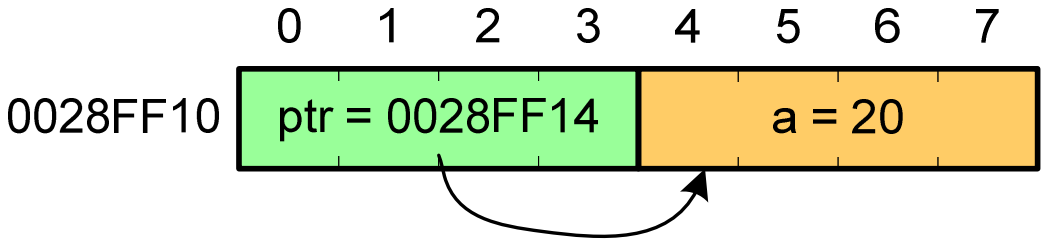
- We can assign an address of variable to a pointer
- Such addresses are created using the address operator (&)

```
int a = 10;
int *ptr;
ptr = &a;
```



- Having the address of a variable, we can "get" to its value using the dereference operator (indirection operator) - asterisk (*)

```
*ptr = 20;
```



Pointers: null pointer

- A null pointer is a special value, distinct from all other pointer values, for which inequality is guaranteed with a pointer to any object
- An integer expression with a value of zero (0) is used to write a null pointerro (0)

```
int *ptr = 0;
```

- Instead of the value 0, the symbolic constant NULL can be used, which is changed to 0 during program compilation

```
int *ptr = NULL;
```

Example: assigning values to pointers

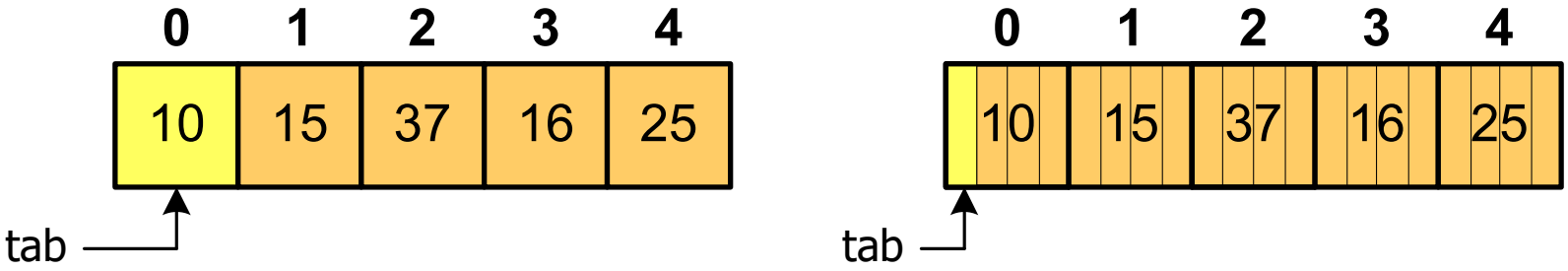
```c
#include <stdio.h>

int main(void)
{
   int x = 15;
   int *ptri = NULL;

   printf("x =    %d\n",x);
   printf("ptri = %p\n",ptri);

   ptri = &x;              // address assignment
   printf("ptri = %p\n",ptri);

   *ptri = *ptri + 10;  // x = x + 10
   printf("x =    %d\n",x);
   printf("x =    %d\n",*ptri);

   return 0;
}
```

```
x =    15
ptri = 0000000000000000
ptri = 00000000010FF960
x =    25
x =    25
```

Pointers and arrays

- The name of the array is its address (more precisely - the address of the element with index 0)

```
int tab[5] = {10,15,37,16,25};
```

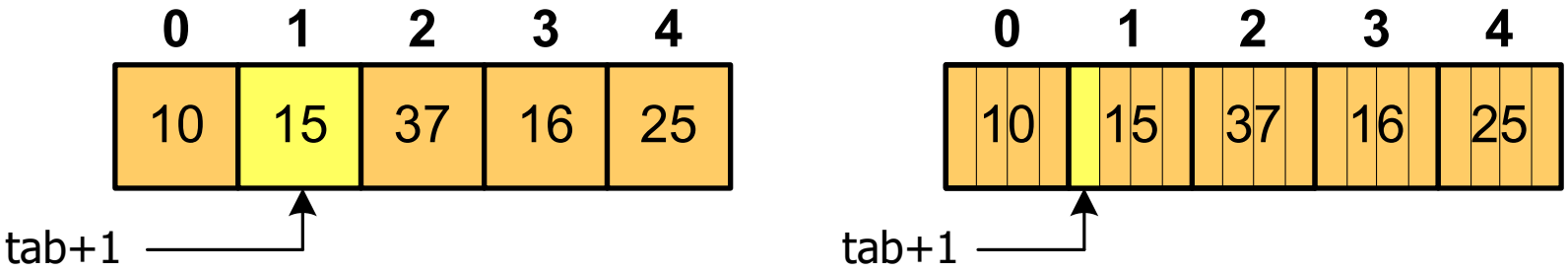


- Using the operator * before the name of the array allows us to "get" to the contents of the element with index 0

*tab is equivalent to tab[0]

Pointers and arrays

- Adding 1 to the array address takes us to the array element at index 1



therefore: *(tab+1) is equivalent to tab[1]

in general: *(tab+i) is equivalent to tab[i]

- The *(tab+i) notation requires parentheses because the * operator has a very high precedence

Pointers and arrays

- Omitting the parentheses results in invalid access to array elements

```
int tab[5] = {10,15,37,16,25};
int x;

x = *(tab+2);
printf("x = %d",x);          /*  x = 37 */

x = *tab+2;
printf("x = %d",x);          /*  x = 12 */
```

x = *(tab+2); is equivalent to x = tab[2];

x = *tab+2; is equivalent to x = tab[0]+2;

Dynamic memory allocation

- When is dynamic memory allocation used?
  - when the size of the array will be known only during program execution and not during its compilation
  - when the size of the array is very large
- The following functions are used for dynamic memory allocation:
  - calloc()
  - malloc()
- Memory is allocated in the heap
- The allocated memory should be freed by calling the function:
  - free()

Dynamic memory allocation

CALLOC stdlib.h

```
void *calloc(size_t num, size_t size);
```

- Allocates a num*size block of memory (capable of holding an array of num-elements, each occupying size bytes)
- Returns a pointer to the allocated memory block
- If memory cannot be allocated, it returns NULL
- Allocated memory is initialized to zeros (bitwise)
- The returned pointer value must be cast to the correct type

```
int *tab;
tab = (int *) calloc(10,sizeof(int));
```

Dynamic memory allocation

MALLOC stdlib.h

```
void *malloc(size_t size);
```

- Allocates a block of memory containing size bytes
- Returns a pointer to the allocated memory block
- If memory cannot be allocated, it returns NULL
- Allocated memory is not initialized
- The returned pointer value must be cast to the correct type

```
int *tab;
tab = (int *) malloc(10*sizeof(int));
```

Dynamic memory allocation

FREE stdlib.h

```
void *free(void *ptr);
```

- Frees the memory block pointed to by the ptr parameter
- The ptr value must be the result of a calloc() or malloc() function call

```
int *tab;
tab = (int *) calloc(10,sizeof(int));
/* ... */
free(tab);
```

Example: dynamic memoy allocation for one variable

```
value = 123.45
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   float *ptr;

   ptr = (float *) calloc(1,sizeof(float));
   if (ptr == NULL)
   {
      printf(" Memory allocation error\n");
      return 0;
   }

   *ptr = 123.45f;
   printf("value = %g\n",*ptr);

   free(ptr);
   return 0;
}
```

Example: dynamic memory allocation for stucture

```
10,20 - 30,40
```

```c
#include <stdio.h>
#include <stdlib.h>

struct point
{
   int x, y;
};

int main(void)
{
   struct point p, *ptr_p;

   ptr_p = (struct point*) malloc(sizeof(struct point));

   p.x = 10; p.y = 20;
   ptr_p->x = 30; ptr_p->y = 40;
   printf("%d,%d - %d,%d\n",p.x,p.y,ptr_p->x,ptr_p->y);

   free(ptr_p);
   return 0;
}
```

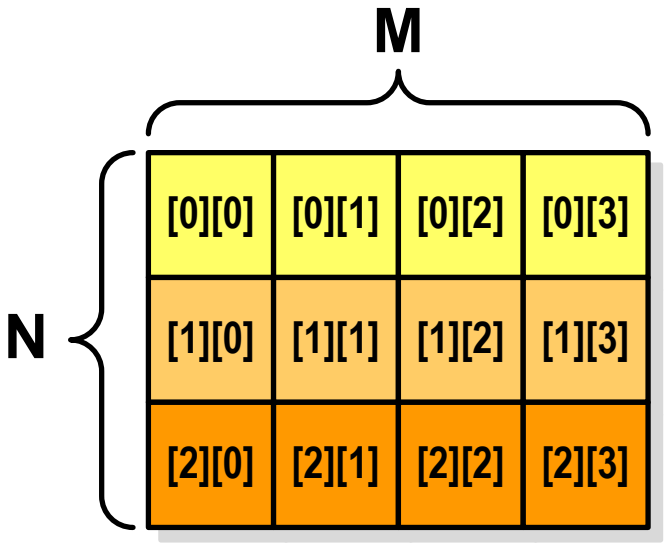
Example: dynamic memory allocation for vector

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int *tab, n = 10;

   tab = (int *) calloc(n,sizeof(int));

   for (int i=0; i<n; i++)
   {
       tab[i] = i*i;
       printf("tab[%d] = %d\n",i,tab[i]);
   }

   free(tab);

   return 0;
}
```

```
tab[0] = 0
tab[1] = 1
tab[2] = 4
tab[3] = 9
tab[4] = 16
tab[5] = 25
tab[6] = 36
tab[7] = 49
tab[8] = 64
tab[9] = 81
```

Dynamic memory allocation for matrix

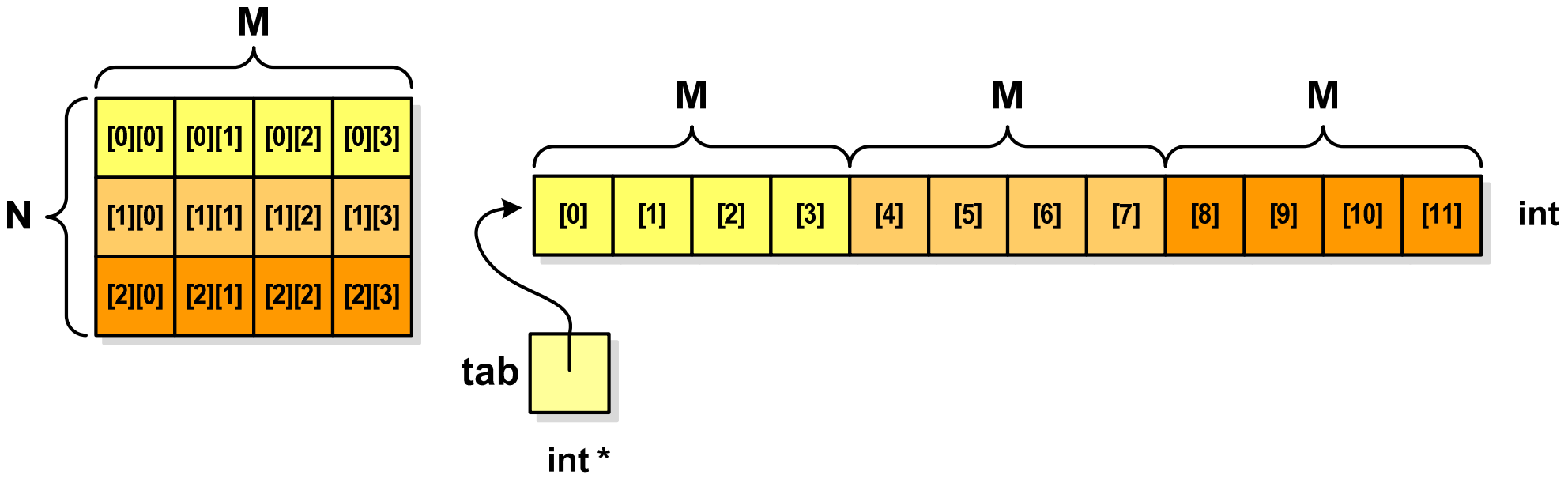
- The calloc() and malloc() functions directly allocate memory only for a vector of elements
- Dynamic memory allocation for array requires special methods
- We allocate memory for a matrix containing N-rows and M-columns



Dynamic memory allocation for matrix (1)

- N×M-element vector
- Memory allocation:

```
int *tab = (int *) calloc(N*M,sizeof(int));
```



Dynamic memory allocation for matrix (1)

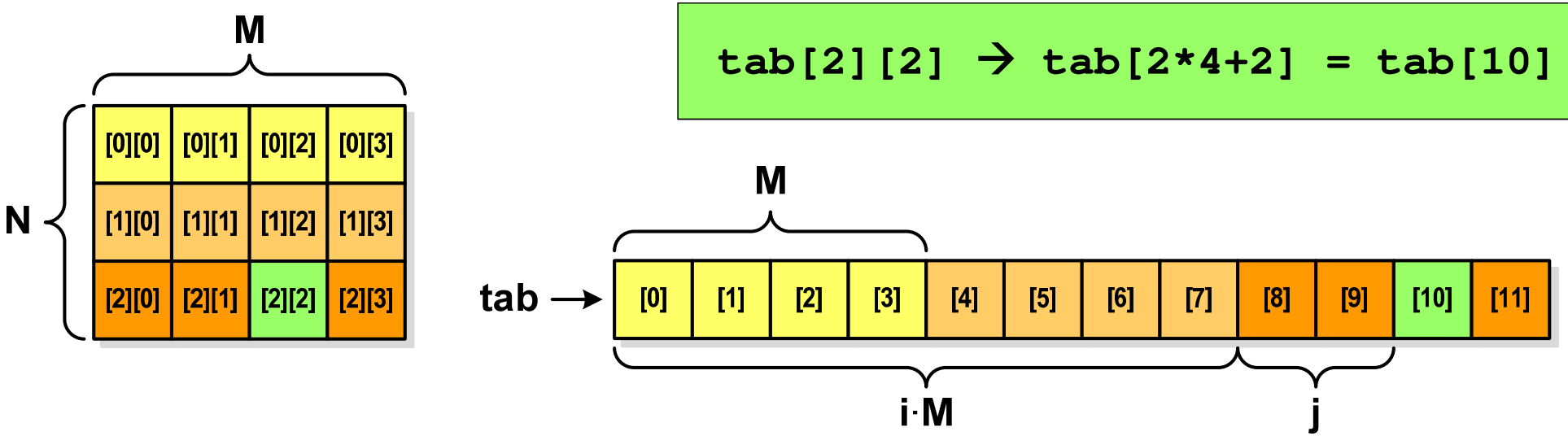
- Access to array elements:

```
tab[i*M+j]
```

lub

```
*(tab+i*M+j)
```

```
tab[2][2] → tab[2*4+2] = tab[10]
```



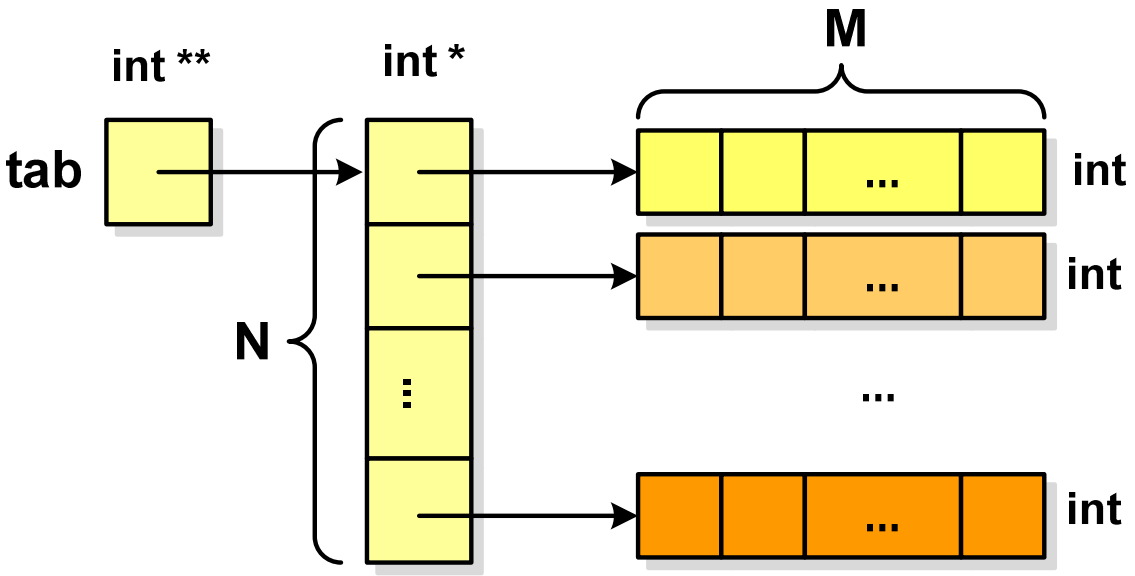
- Deallocation of memory:

```
free(tab);
```

Dynamic memory allocation for matrix (2)

- N-element vector of pointers + N vectors with M elements
- Memory allocation:

```
int **tab = (int **) calloc(N,sizeof(int *));
for (i=0; i<N; i++)
    tab[i] = (int *) calloc(M,sizeof(int));
```



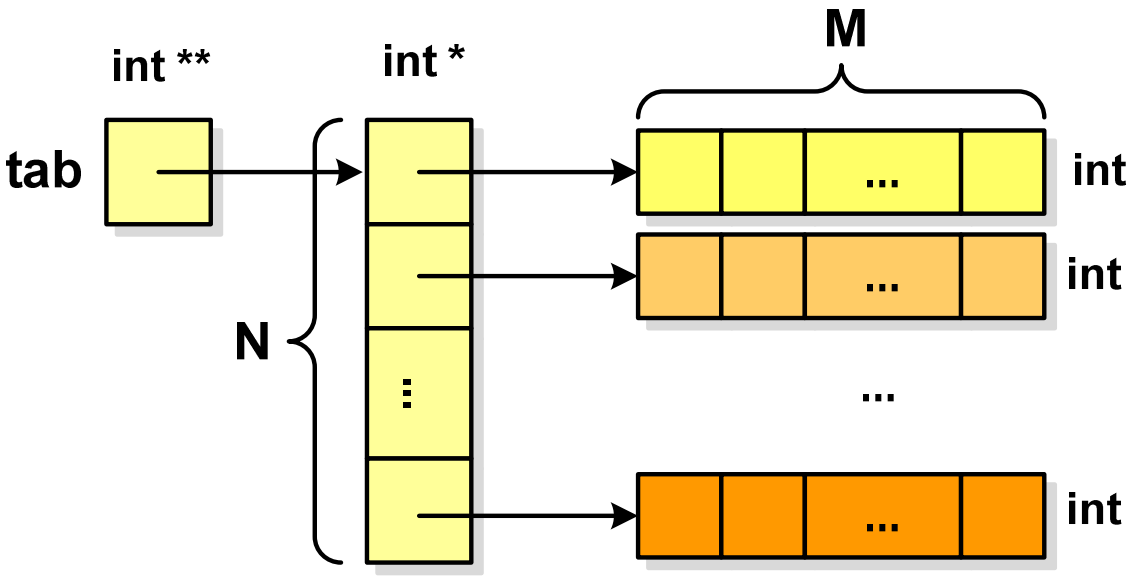
Dynamic memory allocation for matrix (2)

- Access to array elements:

```
tab[i][j]
```

- Deallocation of memory:

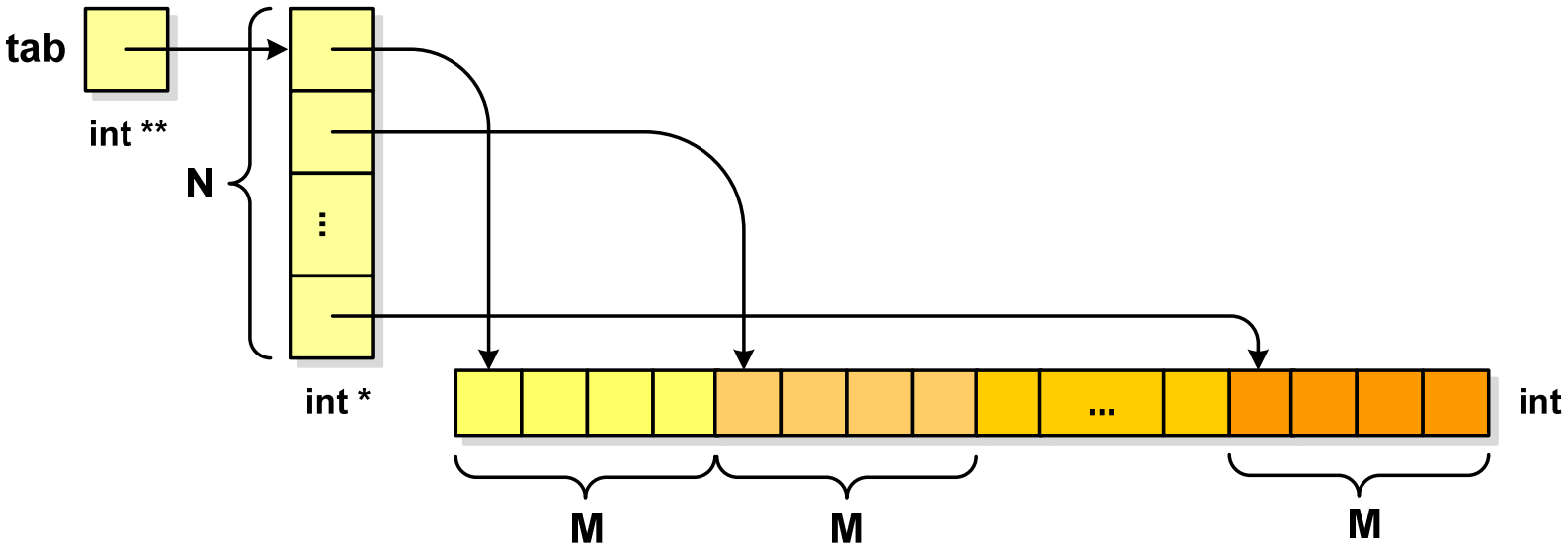
```
for (i=0; i<N; i++)
    free(tab[i]);
free(tab);
```



Dynamic memory allocation for matrix (3)

- N-element vector of pointers + N×M-element vector
- Memory allocation:

```
int **tab = (int **) malloc(N*sizeof(int *));
tab[0] = (int *) malloc(N*M*sizeof(int));
for (i=1; i<N; i++)
   tab[i] = tab[0]+i*M;
```



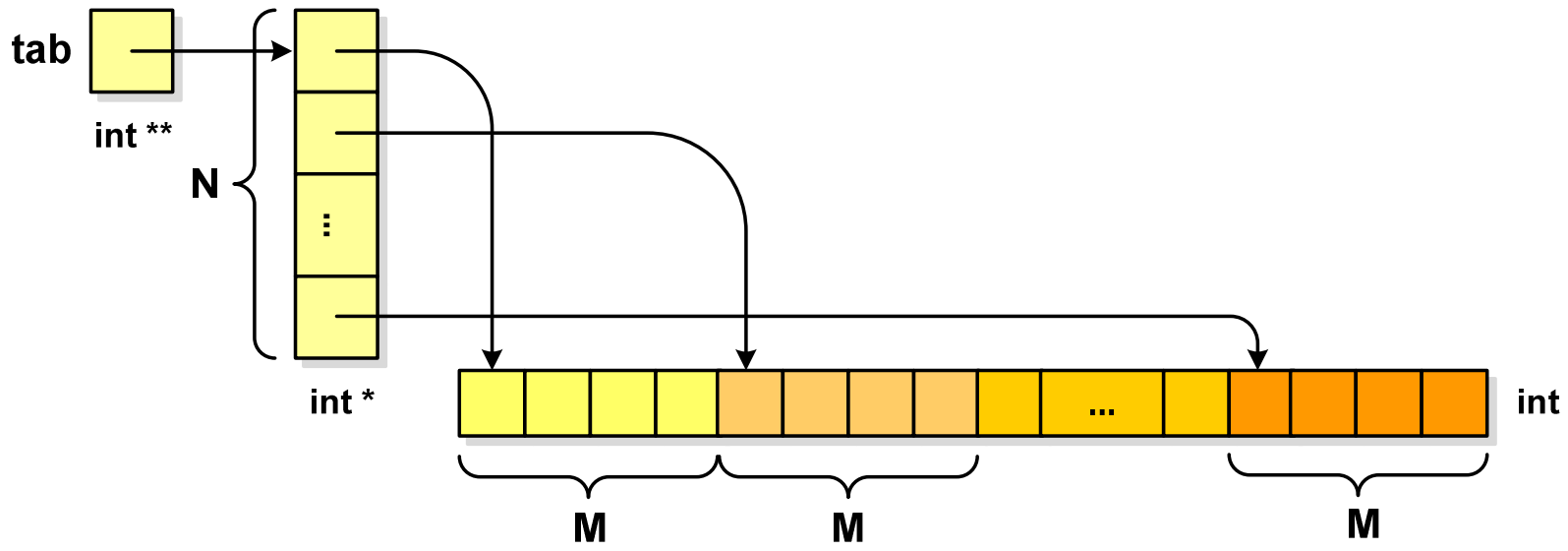
Dynamic memory allocation for matrix (3)

- Access to array elements:

```
tab[i][j]
```

- Deallocation of memory:

```
free(tab[0]);
free(tab);
```



End of workshop no. 11

Thank you for your attention!