CH-230-A

Programming in C and C++

C/C++

Tutorial 4

Dr. Kinga Lipskoch

Fall 2021

Dynamic Memory Allocation

- What if we do not know the dimension of the array while coding?
- Dynamic memory allocation allows you to solve this problem
  - And many others
  - But can also cause a lot of troubles if you misuse it

Pointers and Arrays

There is a strong relation between pointers and arrays

- Indeed an array is nothing but a pointer to the first element in the sequence
- We are looking at this in detail

Specifying the Dimension on the Fly

To specify the dimension on the fly you can use the `malloc()` function defined in the header file `stdlib.h`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main() {
4    int *dyn_array, how_many, i;
5    printf("How many elements? ");
6    scanf("%d", &how_many);
7    dyn_array =
8      (int*) malloc(sizeof(int) * how_many);
9    if (dyn_array == NULL)
10     exit(1);
11   for (i = 0 ; i < how_many; i++) {
12     printf("\nInput number %d:", i);
13     scanf("%d", &dyn_array[i]);
14   } return 0;
15 }
```

The `malloc()` Function (1)

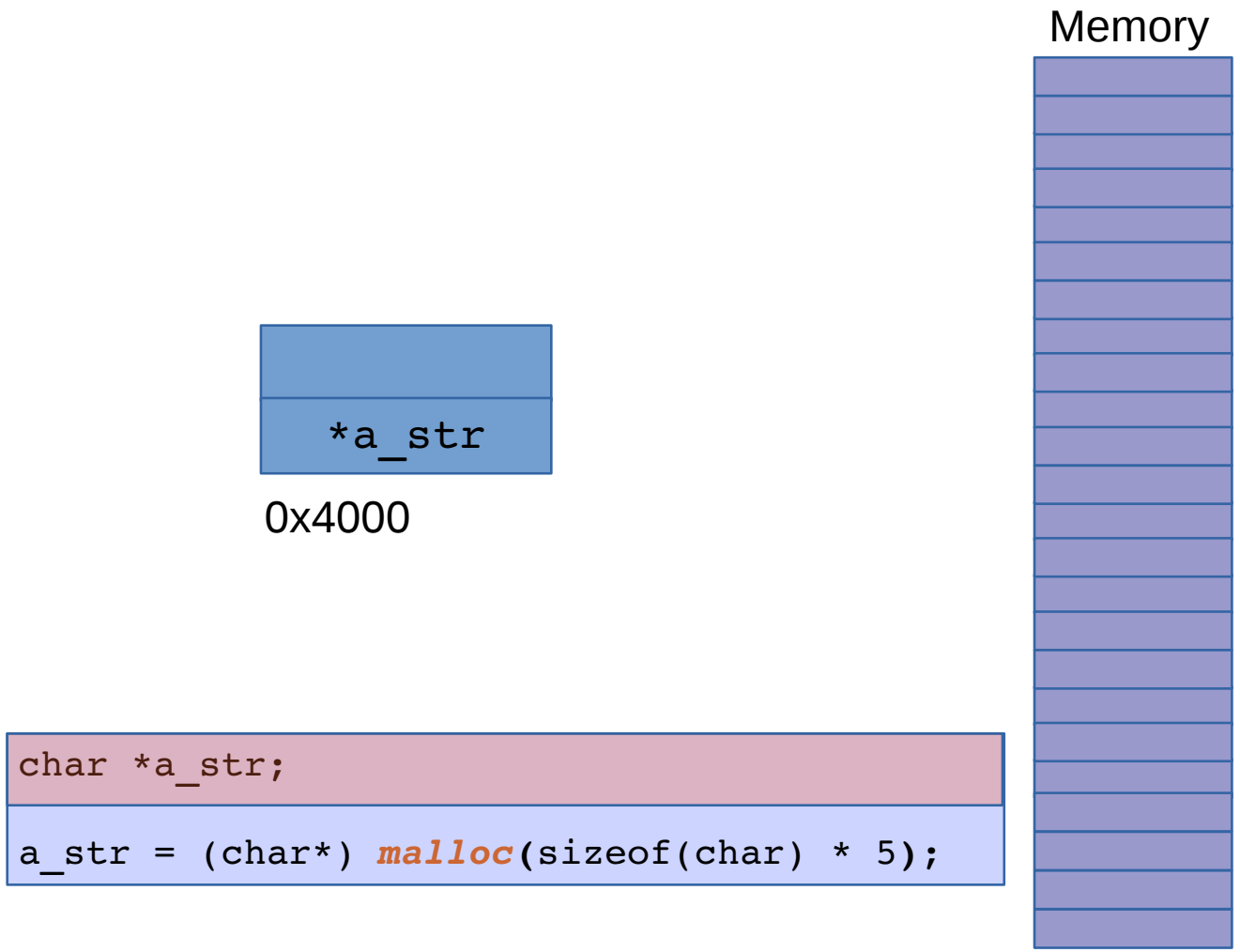
- `void * malloc(unsigned int);`
- `malloc` reserves a chunk of memory
- The parameter specifies how many bytes are requested
- `malloc` returns a pointer to the first byte of such a sequence
- The returned pointer must be forced (cast) to the required type

The malloc() Function (2)

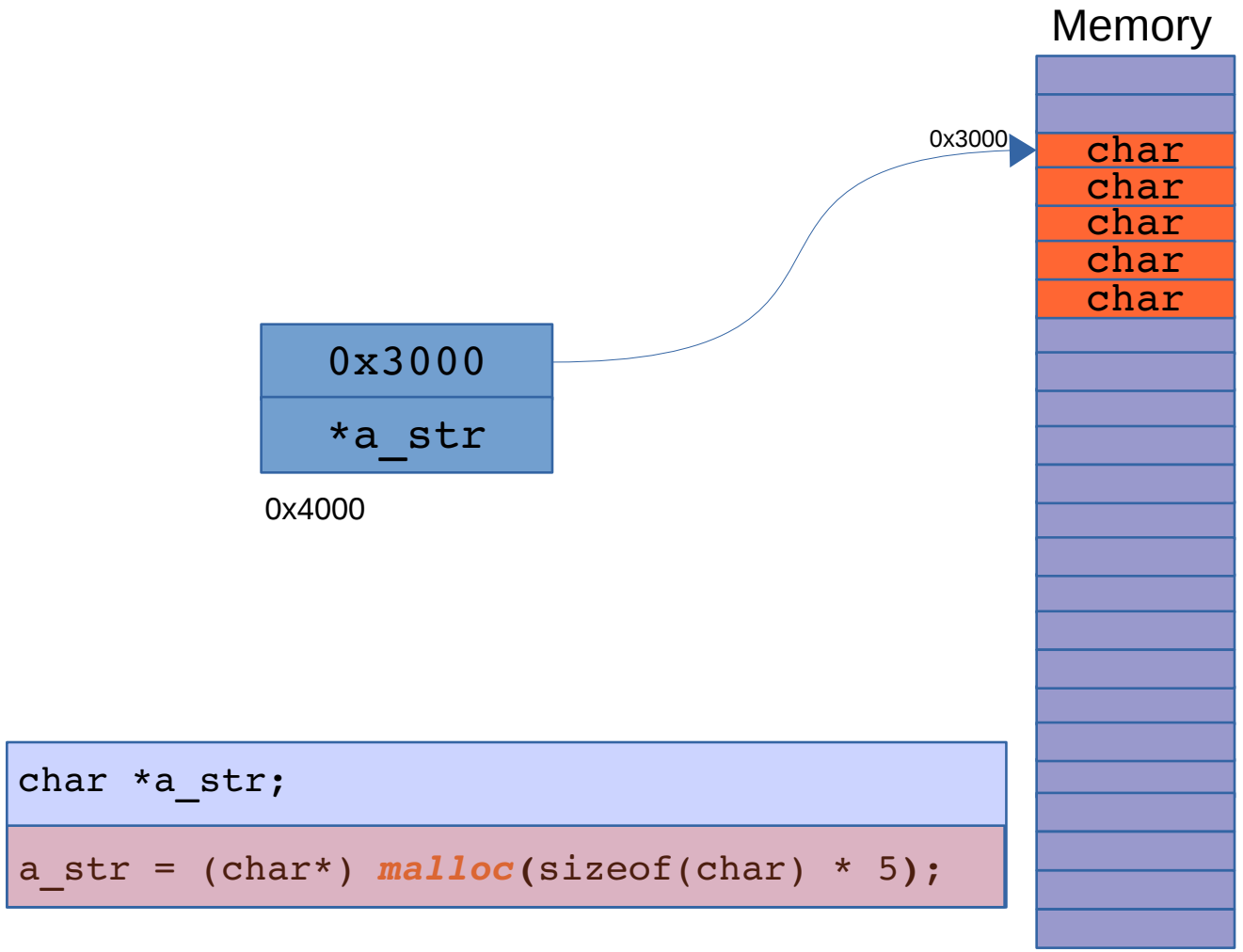
```
1 pointer   = (cast) malloc(number of bytes);
2
3
4 char* a_str;
5 a_str = (char*) malloc(sizeof(char) * how_many);
```

- malloc returns a void * pointer (i.e., a generic pointer) and this is assigned to a non void * pointer
- If you omit the casting you will get a warning concerning a possible incorrect assignment

Dynamically Allocating Space for an Array of `char`

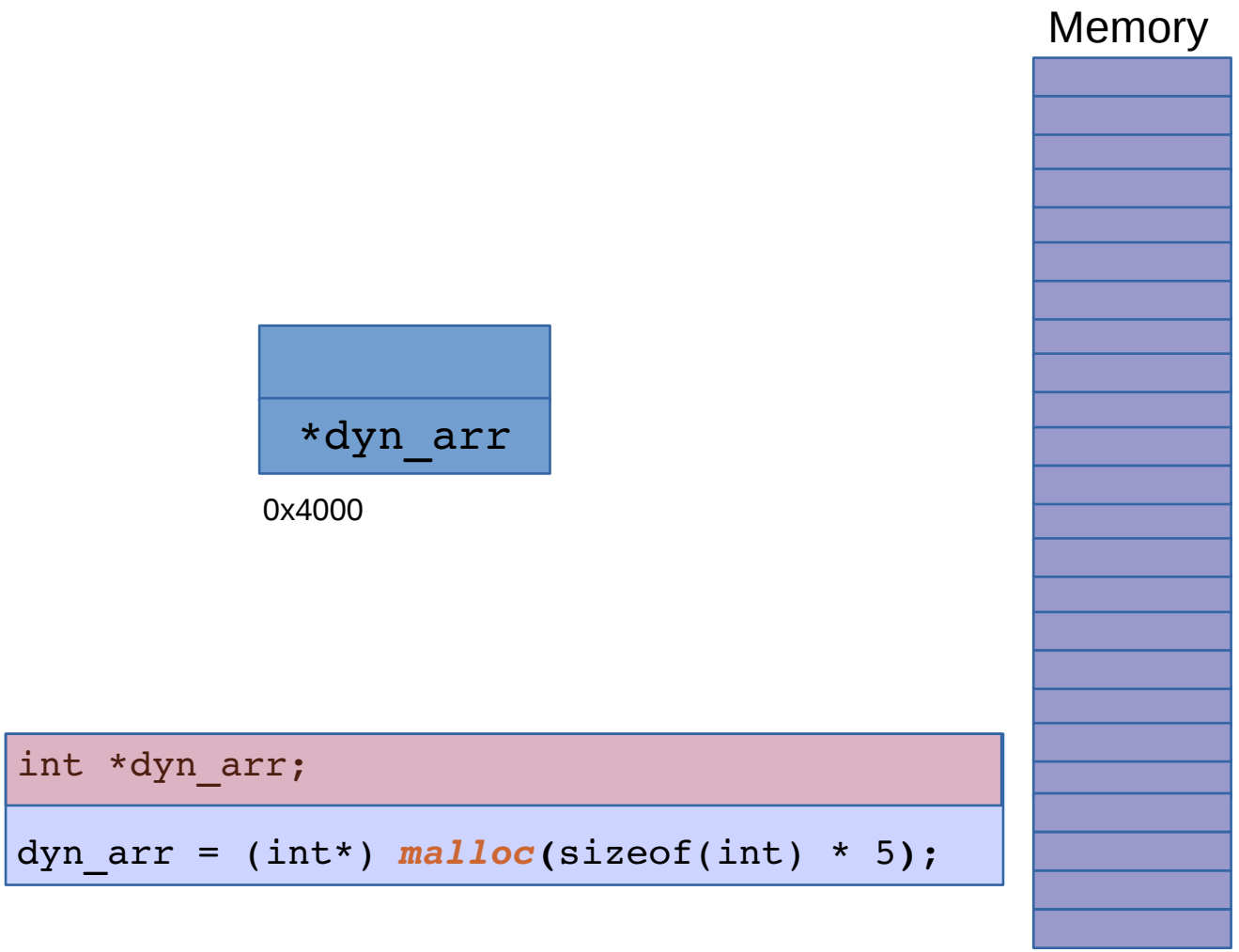


Dynamically Allocating Space for an Array of `char`

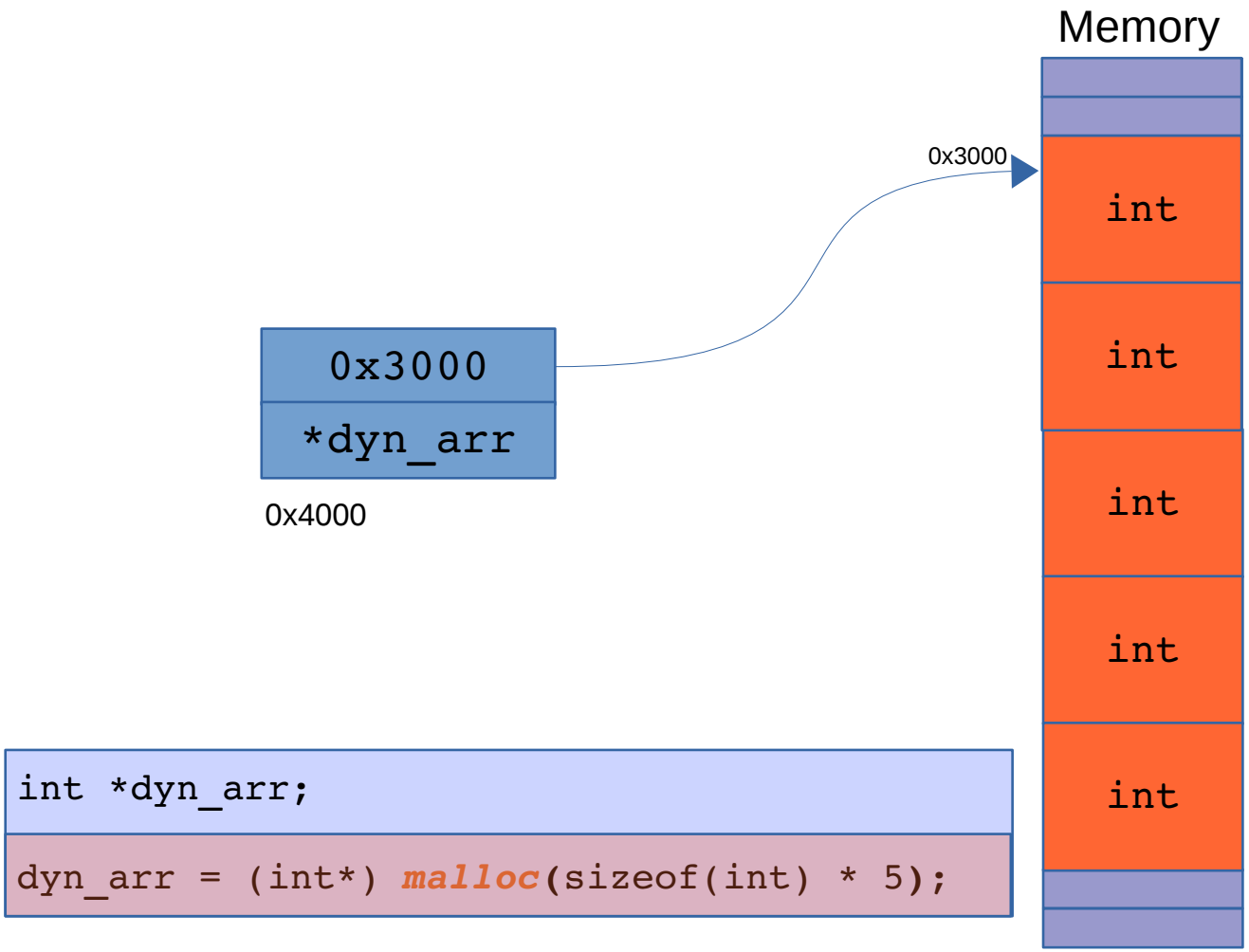



```
char *a_str;
a_str = (char*) malloc(sizeof(char) * 5);
```

Dynamically Allocating Space for an Array of `int`



Dynamically Allocating Space for an Array of `int`



malloc() and free()

- All the memory you reserve via `malloc`, must be released by using the `free` function
- If you keep reserving memory without freeing, you will run out of memory

```
1  float *ptr;
2  int number;
3  ...
4  ptr = (float*) malloc(sizeof(float) *
     number);
5  ...
6  free(ptr);
```

Rules for `malloc()` and `free()`

- The following points are up to you (the compiler does not perform any control)
  1. Always check if `malloc` returned a valid pointer (i.e., not `NULL`)
  2. Free allocated memory just once
  3. Free only dynamically allocated memory
- Not following these rules will cause endless troubles
- `sizeof()` is compile time operator, it does not work on allocated memory

Review: Pointers, Arrays, Values

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int length[2] = {7, 9};
  int *ptr1, *ptr2; int n1, n2;
  ptr1 = &length[0];
  // &length[0] is pointer to first elem
  ptr2 = length;
  // length is pointer to first elem therefore
  // same as above
  n1 = length[0];
  // length[0] is value
  n2 = *ptr2;
  // *ptr2 is value therefore same as above
  printf("ptr1: %p, ptr2: %p\n", ptr1, ptr2);
  printf("n1: %d, n2: %d\n", n1, n2);
  return 0;
}
```

Multi-dimensional Arrays

- It is possible to define multi-dimensional arrays
  - Mostly used are bidimensional arrays, i.e., tables or matrices
- As for arrays, to access an element it is necessary to provide an index for each dimension
  - Think of matrices in mathematics

Multi-dimensional Arrays in C

- It is necessary to specify the size of each dimension
  - Dimensions must be constants
  - In each dimension the first element is at position 0

```
1 int matrix[10][20];    /* 10 rows, 20 cols */
2 float cube[5][5][5];   /* 125 elements */
```

- Every index goes between brackets

```
1 matrix[0][0] = 5;
```

Multi-dimensional Arrays in C: Example

```c
#include <stdio.h>
int main() {
  int table[50][50];
  int i, j, row, col;
  scanf("%d", &row);
  scanf("%d", &col);
  for (i = 0; i < row; i++)
    for (j = 0; j < col; j++)
      table[i][j] = i * j;
  for (i = 0; i < row; i++)
  {
    for (j = 0; j < col; j++)
      printf("%d ", table[i][j]);
    printf("\n");
  }
  return 0;
}
```

The `main` Function (1)

- Can return an `int` to the operating system
  - Program exit code (can be omitted)
  - print exit code in shell: `$> echo $?`
- Can accept two parameters:
  - An integer (usually called `argc`)
  - A vector of strings (usually called `argv`)
  - `argc` specifies how many strings contains `argv`

The main Function (2)

```
1 #include <stdio.h>
2 int main(int argc, char *argv[]) {
3   int i;
4   for (i = 1; i < argc; i++)
5     printf("%d %s\n", i, argv[i]);
6   return 0;
7 }
```

- Compile it and call the executable `paramscounter`
- Execute it as follows:

  ```
  $> ./paramscounter first what this
  ```

- It will print first, what and this, one word per line
- Note that argc is always greater or equal than one
- The first parameter is the program's name

Pointers and Arrays

- Ex:
```
char array[5];
char *array_ptr1 = &array[0];
char *array_ptr2 = array;
// the same as above
```
- C allows pointer arithmetic:
  - Addition
  - Subtraction
- `*array_ptr` equivalent to `array[0]`
- `*(array_ptr+1)` equivalent to `array[1]`
- `*(array_ptr+2)` equivalent to `array[2]`
- What is `(*array_ptr)+1`?