Computer Programming using C Lecture 8: Arrays, memory allocation, and pointers to pointers

Prof. Stephen Smith

E-mail: stephen.smith@york.ac.uk

Based on lecture notes by Dr Julian Miller

Contents

- Pointer arithmetic
- Arrays and pointers
  - strings and character pointers
- Arrays *of* pointers
- Allocating memory at run time
- Two-dimensional arrays (pointers to pointers)

Pointer arithmetic

- Pointers are *variables*
  - that hold *memory addresses*
- Variables can be incremented
  - and so can pointers
- Consider

```
int *p;
p++;
```

- The memory address held by **`p`** will be incremented
  - *to a value that points to the next integer in memory*

Arrays and pointers

- Consider the following array declaration

  `a[10];`

- The *name* of the array (without brackets)
  - Returns the *address* of a[0]
- **`a[i]`** is equivalent to **`*(a+i)`**

Consider

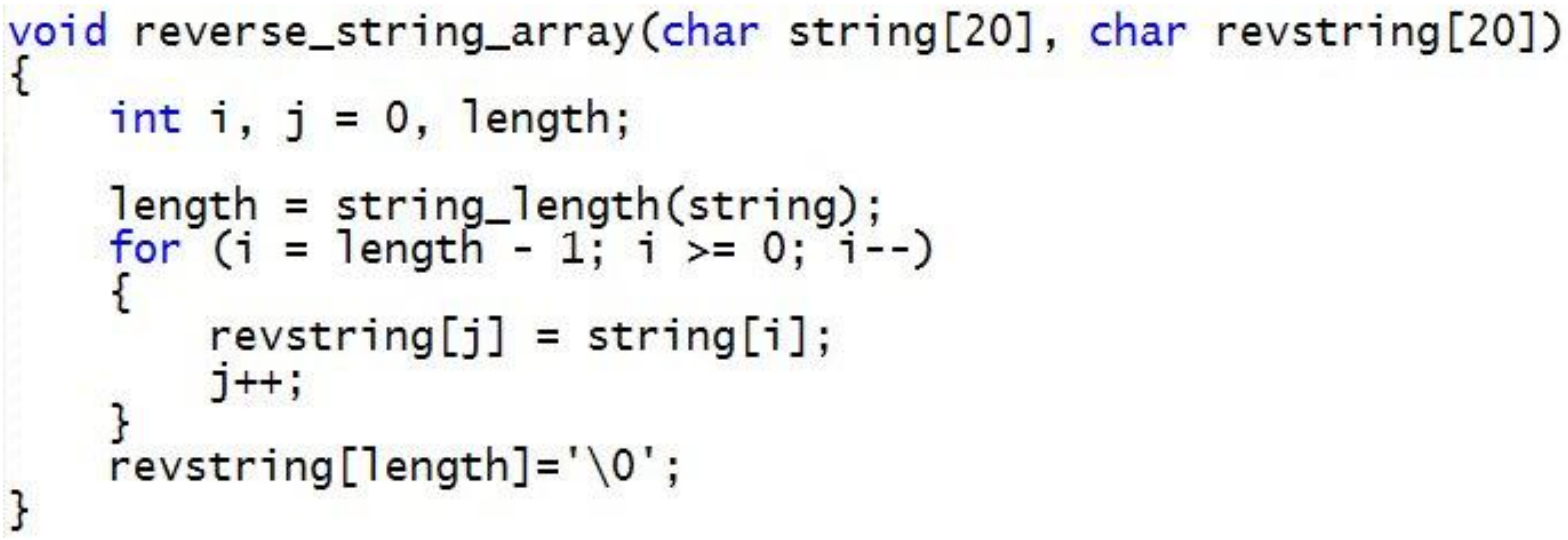
```
int  a[5], *p;
```

`p = a;` (**`p`** now points to the start of array **`a`**)

`p = a + 1;` (**`p`** now points to the 2nd element of **`a`**)

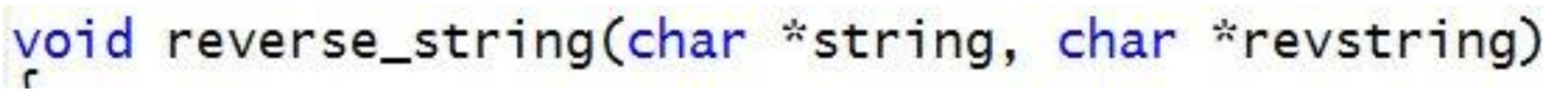
Strings and character pointers

- Character pointers and strings are equivalent
  - Consider the following function with char array arguments



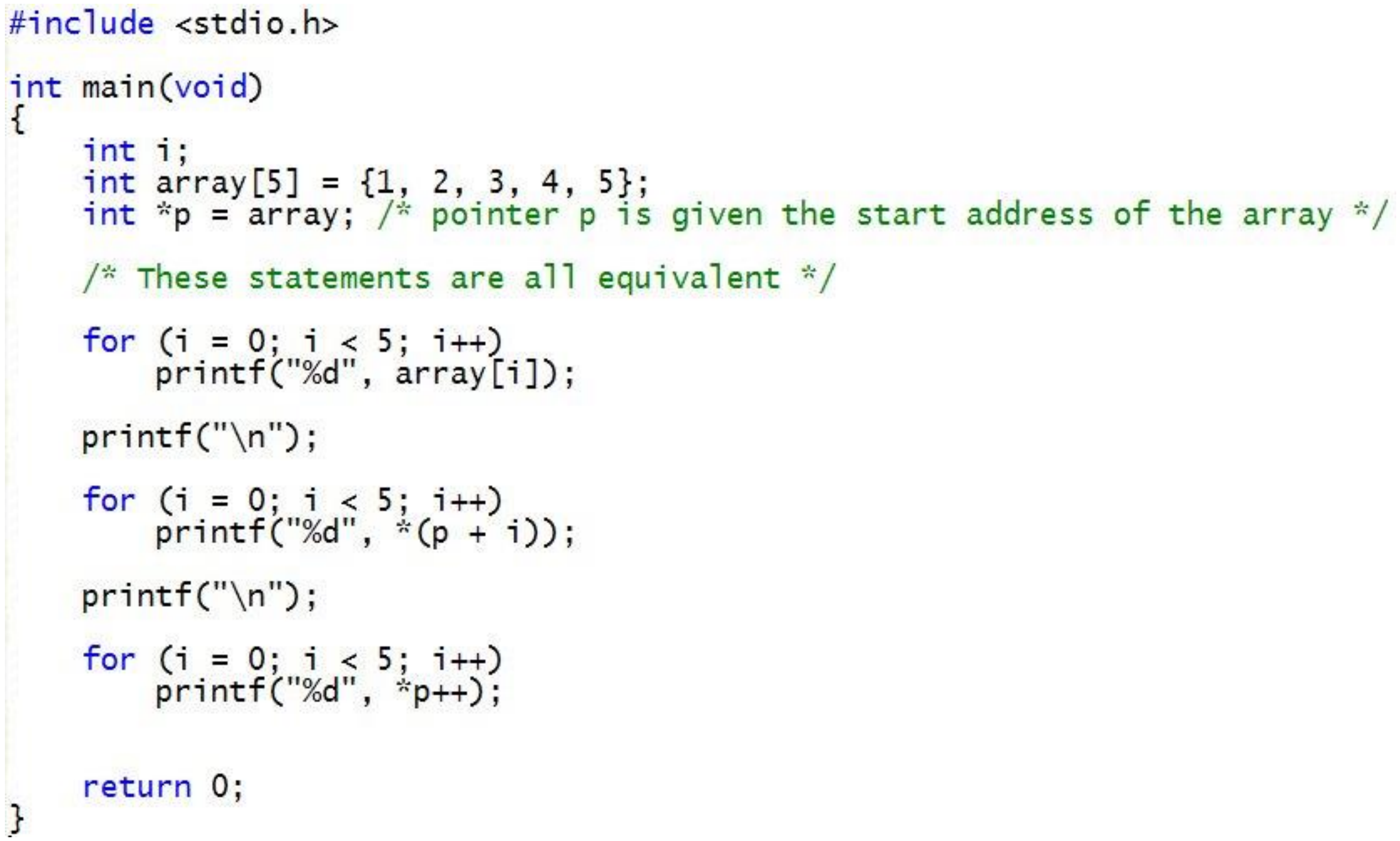
```c
void reverse_string_array(char string[20], char revstring[20])
{
    int i, j = 0, length;

    length = string_length(string);
    for (i = length - 1; i >= 0; i--)
    {
        revstring[j] = string[i];
        j++;
    }
    revstring[length]='\0';
}
```

- Is equivalent to the following function header
  - with pointer to char parameters



```c
void reverse_string(char *string, char *revstring)
```

Pointers and Arrays: equivalents



```c
#include <stdio.h>

int main(void)
{
    int i;
    int array[5] = {1, 2, 3, 4, 5};
    int *p = array; /* pointer p is given the start address of the array */

    /* These statements are all equivalent */

    for (i = 0; i < 5; i++)
        printf("%d", array[i]);

    printf("\n");

    for (i = 0; i < 5; i++)
        printf("%d", *(p + i));

    printf("\n");

    for (i = 0; i < 5; i++)
        printf("%d", *p++);


    return 0;
}
```

Arrays of pointers

```
int x = 1, y;
int *b[10];
```

- **`b`** is an *array* of integer pointers

```
b[2] = &x;
```

- The third element of **`b`** holds the *address* of variable **`x`**

```
y = *b[2];
```

- The integer variable **`y`** is assigned the *contents* of the variable whose *address* is stored in **`b[2]`**

Fixed dimension arrays are inconvenient

- So far we have declared arrays to be of fixed dimension, e.g.

```
int array[100];
```

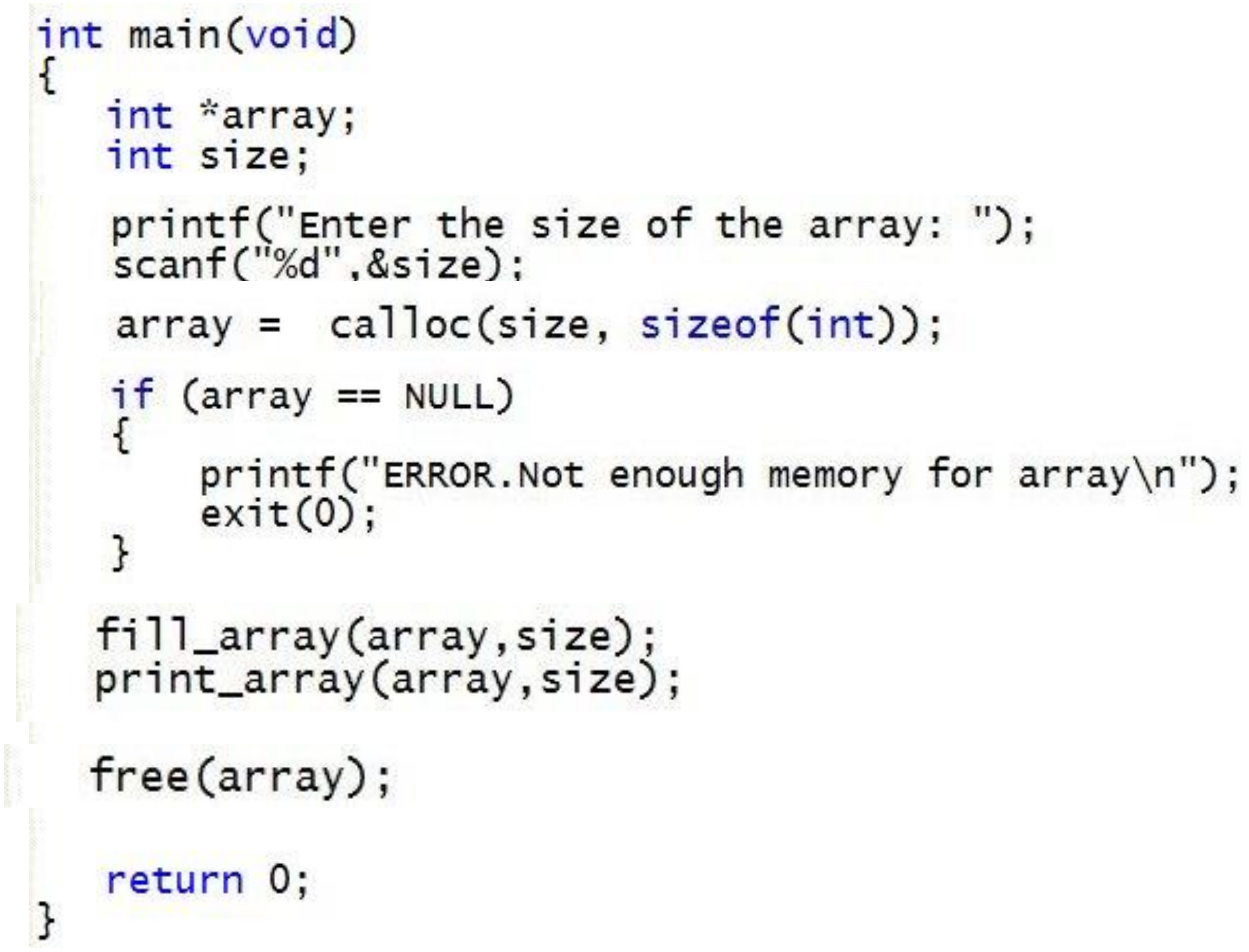
- This can be inconvenient
  - as often one doesn’t know in advance how much memory to allocate

Memory Allocation functions

- **`sizeof(my_data_type)`**
  - predefined operator that returns the amount of memory (in bytes) that **`my_data_type`** requires

- Following functions are defined in **`stdlib.h`**
- **`malloc(mysize*sizeof(mydatatype))`**
  - Reserves memory for **`mysize`** elements of size **`mydatatype`** and returns the address in memory of the start of the memory allocated
- **`calloc(mysize, sizeof(mydatatype))`**
  - Acts exactly like **`malloc()`**, except the contents of the memory allocated is set to zero
- **`realloc(myarraypointer,mysize))`**
  - This allows you to resize some memory that has previously been requested and allocated to **`myarraypointer`**
- **`free(mypointer)`**
  - Releases memory whose start address was given in **`mypointer`**

Allocating space for a one-dimensional array



```
int main(void)
{
    int *array;
    int size;

    printf("Enter the size of the array: ");
    scanf("%d",&size);

    array =  calloc(size, sizeof(int));

    if (array == NULL)
    {
        printf("ERROR.Not enough memory for array\n");
        exit(0);
    }

    fill_array(array,size);
    print_array(array,size);

    free(array);

    return 0;
}
```

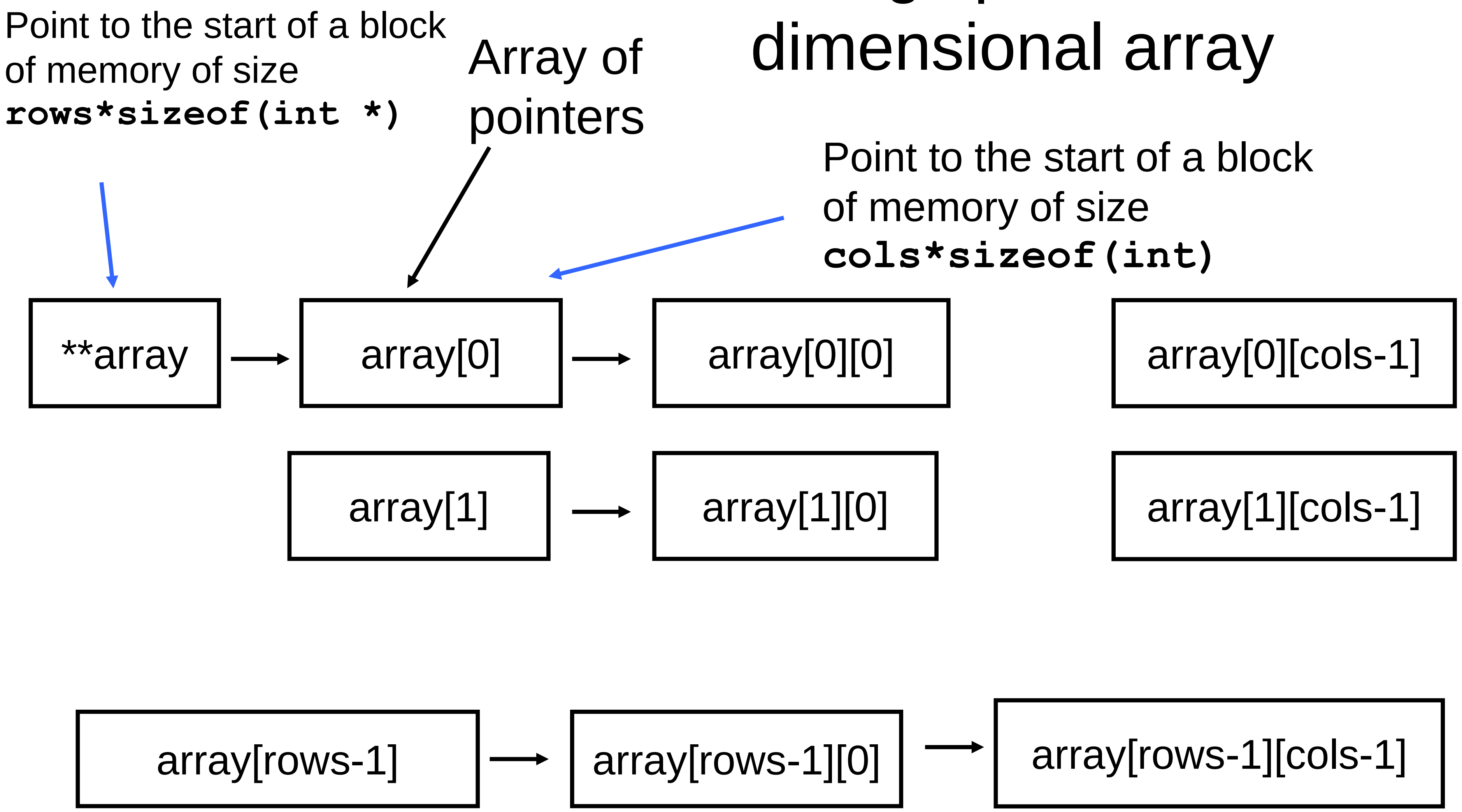
Pointers to pointers

- A pointer is a variable that hold the address of another variables
- But a pointer *itself* is another variable so one should be able to store the address of the pointer in a variable too.

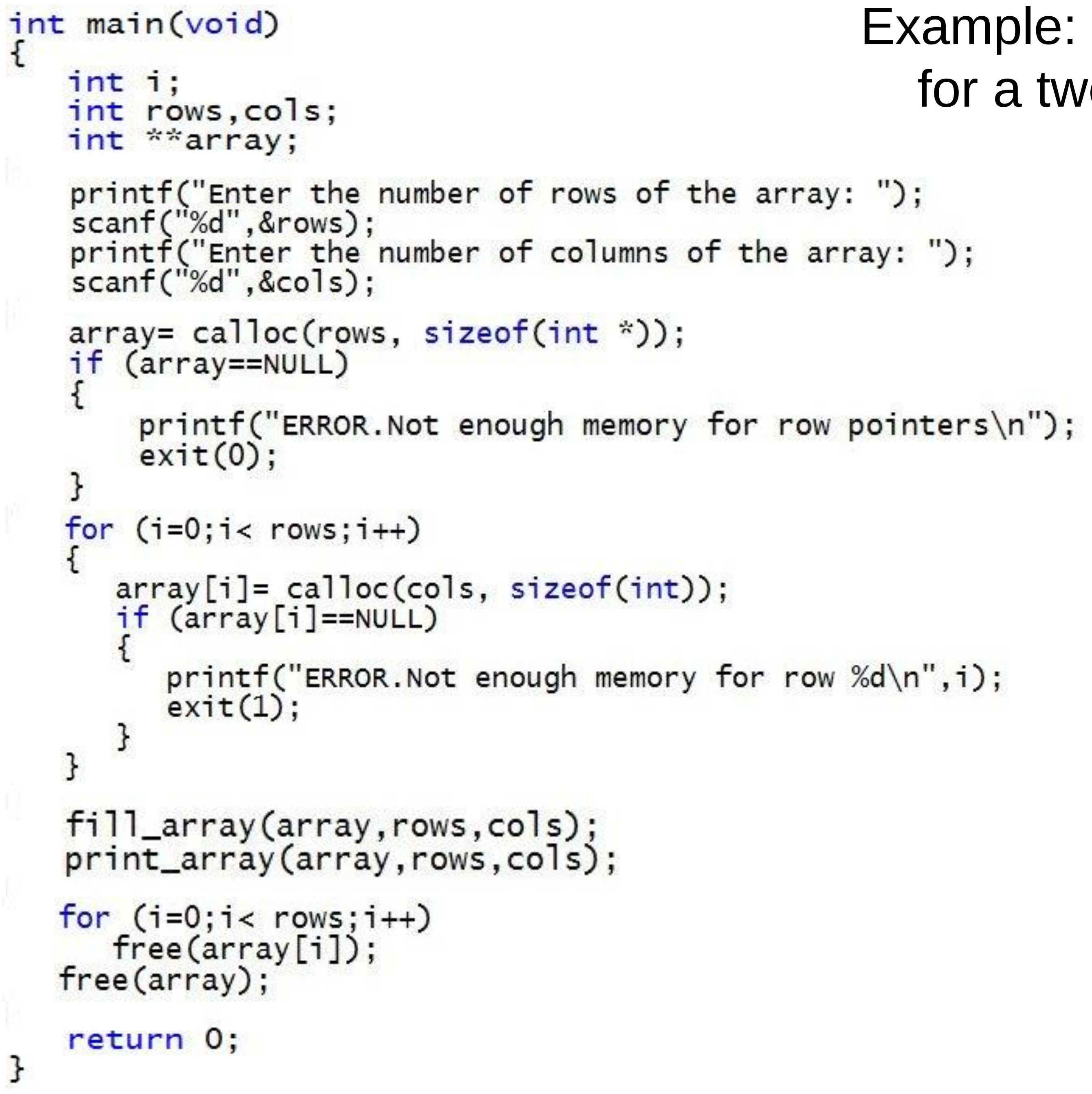
```
int i = 0, *p, **q;
p = &i;
*p = 3;
q = &p;
**q = *p+2;
```

- One common use of pointers to pointers is in the allocation of memory at execution time for *multidimensional* arrays

Allocating space for a two dimensional array



Example: allocating space for a two dimensional array



```c
int main(void)
{
    int i;
    int rows,cols;
    int **array;

    printf("Enter the number of rows of the array: ");
    scanf("%d",&rows);
    printf("Enter the number of columns of the array: ");
    scanf("%d",&cols);

    array= calloc(rows, sizeof(int *));
    if (array==NULL)
    {
        printf("ERROR.Not enough memory for row pointers\n");
        exit(0);
    }
    for (i=0;i< rows;i++)
    {
       array[i]= calloc(cols, sizeof(int));
       if (array[i]==NULL)
       {
          printf("ERROR.Not enough memory for row %d\n",i);
          exit(1);
       }
    }

    fill_array(array,rows,cols);
    print_array(array,rows,cols);

   for (i=0;i< rows;i++)
      free(array[i]);
   free(array);

    return 0;
}
```

Summary

- Pointer arithmetic
- Arrays and pointers
- Looked at how to allocate memory for arrays at run time
- Examined two dimensional arrays via pointers to pointers
- In Lab 8:
  - arrays and strings and pointers, memory allocation