PROGRAMMING IN C: POINTERS AND MEMORY MANAGEMENT (2)

COMP1206 - PROGRAMMING II

Enrico Marchioni
e.marchioni@soton.ac.uk
Building 32 - Room 4019

ARRAYS AND POINTERS

- Recall that a pointer is a variable that holds a memory address.
- You declare a pointer as follows:

```
type *pointername;
```

- You can initialize a pointer assigning the memory address extracted from a variable (which you already declared) by using the & operator:

```
pointername = &variablename;
```

- You can change/read the value that lives at the address the pointer points at by using *:

```
*pointername = value;
```

- You can initialize a pointer assigning the address of an array:

```
pointername = arrayname;
```

- You can use pointers to do almost everything you can do with an array, except declare an array and its size.

- Let's look at two examples.

- In the first, we define an array `array[5]` and display its elements by going through them explicitly.

- In the second, we declare a pointer `p_array` to the array, and use pointer mathematics to go through different memory blocks.

```c
#include <stdio.h>

int main()
{
    int array[5] = { 1, 2, 4, 8, 16};
    int x;


    for(x=0;x<5;x++)
    {
        printf("array[%i] = %i\n",x,array[x]);
    }


    return(0);
}
```

```
array[0] = 1
array[1] = 2
array[2] = 4
array[3] = 8
array[4] = 16
```

```c
#include <stdio.h>

int main()
{
    int array[5] = { 1, 2, 4, 8, 16};
    int x, *p_array;
    p_array = array;

    for(x=0;x<5;x++)
    {
        printf("array[%i] = %i\n",x,*p_array);
        p_array++;
    }

    return(0);
}
```

```
array[0] = 1
array[1] = 2
array[2] = 4
array[3] = 8
array[4] = 16
```

- In the previous example, a pointer is used to store the location of an array, access and display its content.

- At each cycle of the loop

  `*p_array`

  displays the value that lives in a certain memory block and corresponds to the value of the array.

- Pointers can be used also to fill an array with values.

- We are going to see two examples of how this can be done.

```c
#include <stdio.h>

int main()
{
     int cent[8];
     int x;


     for(x=0;x<8;x++)
     {
          cent[x] = (x+1) * 100;
          printf("cent[%i] = %i\n",x,cent[x]);
     }
     return(0);
}
```

```
cent[0] = 100
cent[1] = 200
cent[2] = 300
cent[3] = 400
cent[4] = 500
cent[5] = 600
cent[6] = 700
cent[7] = 800
```

```c
#include <stdio.h>

int main()
{
     int cent[8];
     int x, *c;
     c = cent;

     for(x=0;x<8;x++)
     {
          *c = (x + 1) * 100;
          printf("cent[%i] = %i\n", x, *c++);
     }
     return(0);
}
```

```
cent[0] = 100
cent[1] = 200
cent[2] = 300
cent[3] = 400
cent[4] = 500
cent[5] = 600
cent[6] = 700
cent[7] = 800
```

```
for(x=0;x<8;x++)
   {
        *c = (x + 1) * 100;
        printf("cent[%i] = %i\n", x, *c++);
   }
```

- For each `x`

  ```
  *c = (x + 1) * 100;
  ```

  initializes the element `x` of the array.
- At the beginning of the loop, the first element is located where the pointer was initialized.
- The statement

  ```
  printf("cent[%i] = %i\n", x, *c++);
  ```

  prints the value of element `*c` and then jumps to the following memory block with `*c++`.
- This is the same as

  ```
  printf("cent[%i] = %i\n", x, *c);
  c++;
  ```

```c
#include <stdio.h>

int main()
{
    int cent[8];
    int x, *c;
    c = cent;

    for(x=0;x<8;x++)
    {
        *(c + x) = (x + 1) * 100;
        printf("cent[%i] = %i\n", x, *(c + x));
    }
    return(0);
}
```

```
cent[0] = 100
cent[1] = 200
cent[2] = 300
cent[3] = 400
cent[4] = 500
cent[5] = 600
cent[6] = 700
cent[7] = 800
```

- `++` is more binding than `*`
- So, `*c++` tells the compiler to apply `++` to `c`, and then to extract the content of the memory address with `*`.
- Notice that the jump to the next memory block will happen after `c` is used, since `++` occurs after `c`.
- What happens if we use `++*c`?
- `++*c` tells the compiler to apply `++` to `*c`.
- So, the compiler reads what lives at the address of `c` and immediately adds 1 to the value it read.

```c
#include <stdio.h>

int main()
{
     int cent[8];
     int x, *c;
     c = cent;

     for(x=0;x<8;x++)
     {
          *c = (x + 1) * 100;
          printf("cent[%i] = %i\n", x, ++*c);
     }
     return(0);
}
```

```
cent[0] = 101
cent[1] = 201
cent[2] = 301
cent[3] = 401
cent[4] = 501
cent[5] = 601
cent[6] = 701
cent[7] = 801
```

Pointer	*Memory Address*	*Memory Contents*
`p`	Yes	No
`*p`	No	Yes
`*p++`	Incremented after value is read	Unchanged
`*(p++)`	Incremented after value is read	Unchanged
`(*p)++`	Unchanged	Incremented after it's used
`*++p`	Incremented before value is read	Unchanged
`*(++p)`	Incremented before value is read	Unchanged
`++*p`	Unchanged	Incremented before it's used
`++(*p)`	Unchanged	Incremented before it's used
`p*++`	Not a pointer	Not a pointer
`p++*`	Not a pointer	Not a pointer

- What happens when we initialize a pointer with an array of conflicting type?
- Take the following code

```
short int array[10];
double *pointer;
pointer = array;
```

- `pointer` points to the address of a double.
- We initialize `pointer` with the address of an array of 10 integers.
- This creates a situation of type incompatibility and the compiler gives as a warning:

```
test.c:11:13: warning: incompatible pointer types assigning
    to 'double *' from
      'int [10]' [-Wincompatible-pointer-types]
    pointer = array;
              ^ ~~~~~
1 warning generated.
```

- Still, the code successfully compiles, and moving between blocks of memory using the pointer does not match moving between elements of the array.

```c
#include <stdio.h>

int main() {
    short int array[10];
    double *pointer;

    for (int x = 0; x < 10; ++x) {
        array[x] = x;
    }

    pointer = array;

    printf("%p -- %p -- %f -- %i\n", pointer, &array[0], *pointer, array[0]);
    printf("%p -- %p -- %f -- %i\n", pointer+1, &array[1], *(pointer+1), array[1]);
    printf("%p -- %p -- %f -- %i\n", pointer+2, &array[2], *(pointer+2), array[2]);

    return 0;
}
```

```
0x7ffeefbff570 -- 0x7ffeefbff570 -- 0.000000 -- 0
0x7ffeefbff578 -- 0x7ffeefbff572 -- 0.000000 -- 1
0x7ffeefbff580 -- 0x7ffeefbff574 -- 0.000000 -- 2
Program ended with exit code: 0
```

MEMORY ALLOCATION

- C manages memory statically, locally and dynamically.
- Static variables are allocated in the main memory and persist for the lifetime of the program.
- Automatic variables are allocated and deallocated to the stack automatically (for instance, when calling functions).
- Dynamic memory is explicitly managed an it can be allocated from the heap (i.e. the free store).

- In C it is possible to access the heap, allocate some memory, assign it to a pointer.
- When this memory is not needed anymore, you can deallocate it and return it to the heap.
- The C standard library `stdlib.h` supports the definition of specific functions for dynamic memory management: `malloc`, `calloc`, `realloc`, `free`.
- To use these functions, you need to include the standard library, i.e.:

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

- `malloc` is a function that takes one argument that specifies number of bytes to be reserved.
- The function returns a pointer to the beginning of the allocated storage area in memory.
- It has the following form:

```
int *p;
p = malloc( n * sizeof(int));
```

- We want to allocate some free memory to a pointer `p` of type `int`

- `int` gives us the basic size of the memory block we need for our data type
- `n` is the amount of memory blocks we need.
- If we just need space for a float

```
float *p;
p = malloc( sizeof(float));
```

- If we need space for a float array of size 50:

```
float *p;
p = malloc( 50 * sizeof(float));
```

- It is possible that `malloc` cannot allocate the requested memory and might return a null pointer.
- A null pointer `NULL` is a special pointer with a reserved value that indicates that the pointer does not point to anything.
- It is good practice to check this possibility

```
float *p;
p = malloc( 50 * sizeof(float));
    if (p == NULL)
    {
        printf("Malloc failed!\n");
        return -1;
    }
```

- When you have no more use for the memory you requested, you can return it to the heap.
- The `free` function serves this purpose:

```
free(p);
```

```c
#include <stdio.h>                  //STORE AND FETCH FIRST 1000 MULTIPLES OF 12
#include <stdlib.h>

int main(){
    int *p, n;                      //ALLOCATE 4000 BYTES
    p = malloc (1000 * sizeof (int));

    if (p == NULL){
        printf("Malloc failed!\n");
        return -1;}

    for (int x = 0; x<1000; x++) {      //FILL ARRAY WITH FIRST 1000 MULTIPLES OF 12
        *(p + x) = (x + 1)*12;}

    printf("Please enter a number between 1 and 1000\n");//REQUEST ARRAY ELEMENT
    scanf("%i", &n);

    if (n<1 || n>1000) {
        printf("Wrong value!\n");
        return 1;}

    printf("The value you requested is %i\n", *(p + n - 1));     //PRINT VALUE
    free(p);              //DEALLOCATE MEMORY

    return 0;}
```

```
Please enter a number between 1 and 1000
5
The value you requested is 60
Program ended with exit code: 0
```

- `malloc` allocates the requested chunk of memory from the heap.
- This memory is located at some address, but we do not know what lives there.
- `calloc` is a function that takes two arguments that specify the number of elements to be reserved and the size of each element in bytes.
- The function returns a pointer to the beginning of the allocated storage area in memory.
- The storage area is also automatically set to 0.

- `calloc` can be used as follows:

```
int *p;
p = calloc( n, sizeof(int));
```

- If we need space for a float array of size 50:

```
float *p;
p = calloc( 50, sizeof(float));
```

allocates 50 blocks of memory space for floats and initializes all bytes to zero.

```c
                        //ALLOCATE MEMORY WITH CALLOC
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *p;
    p = calloc (10, sizeof (int));       //ALLOCATE MEMORY

    for (int x = 0; x<10; x++) {
        printf("%i lives at %p\n", *(p+x), p+x); //PRINT MEMORY ADDRESS AND  VALUE
    }

    free(p);

    return 0;
}
```

```
0 lives at 0x1028143a0
0 lives at 0x1028143a4
0 lives at 0x1028143a8
0 lives at 0x1028143ac
0 lives at 0x1028143b0
0 lives at 0x1028143b4
0 lives at 0x1028143b8
0 lives at 0x1028143bc
0 lives at 0x1028143c0
0 lives at 0x1028143c4
Program ended with exit code: 0
```

```c
                                //ALLOCATE MEMORY WITH MALLOC (COMPARE WITH PREVIOUS EXAMPLE)
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *p;
    p = malloc (10*sizeof (int));        //ALLOCATE MEMORY

    for (int x = 0; x<10; x++) {
        printf("%i lives at %p\n", *(p+x), p+x); //PRINT MEMORY ADDRESS AND  VALUE
    }

    free(p);

    return 0;
}
```

```
0 lives at 0x100400220
0 lives at 0x100400224
0 lives at 0x100400228
0 lives at 0x10040022c
1162412048 lives at 0x100400230
1093614931 lives at 0x100400234
1768714352 lives at 0x100400238
1769234787 lives at 0x10040023c
796094063 lives at 0x100400240
1685021528 lives at 0x100400244
Program ended with exit code: 0
```

- Sometimes you need to change the amount of memory you are using to store data.
- `realloc` is a function that changes the size of previously allocated memory (with `malloc` or `calloc`) and returns a pointer to the new block.
- It has the following form:

```
int *p;
p = malloc( 10 * sizeof(int));
p = realloc( p , 20 * sizeof(int));
```

- We have allocated 40 bytes to store some integers but we need more space.
- `realloc` allocates space for 20 integers, copies the content of the previous 10 memory blocks and returns a memory address that is assigned to a pointer.

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

int main()
{
    int *p;
    p = malloc (5*sizeof (int)); // Allocate memory to p

    printf("First allocation\n"); // Initialize blocks at p
    for (int x = 0; x<5; x++) {
        *(p + x) = x * x;
        printf("%i lives at %p\n", *(p + x), p + x);
    }

    printf("Reallocation\n"); // Reallocate memory and increase
        size of p
    p = realloc(p, 10*sizeof(int));
    for (int y = 0; y<10; y++) {
        printf("%i lives at %p\n", *(p + y), p + y);
    }

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

```
First allocation
0 lives at 0x100676410
1 lives at 0x100676414
4 lives at 0x100676418
9 lives at 0x10067641c
16 lives at 0x100676420
Reallocation
0 lives at 0x1006763a0
1 lives at 0x1006763a4
4 lives at 0x1006763a8
9 lives at 0x1006763ac
16 lives at 0x1006763b0
0 lives at 0x1006763b4
0 lives at 0x1006763b8
131072 lives at 0x1006763bc
0 lives at 0x1006763c0
0 lives at 0x1006763c4
Program ended with exit code: 0
```