Engineering School

Computer Architecture and Operating Systems Department

Degree: Artificial Intelligence

Subject: Fundamentals of Programming II

Dynamic memory

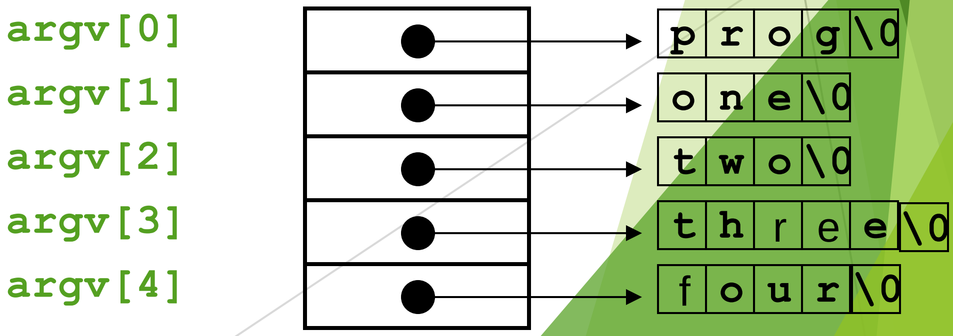
Program arguments

```
$ gcc prog.c -o prog
$ prog one two three four
```

Number of arguments

List of arguments

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i;
  printf("argc is: %d\n", argc);
  for( i = 0; i < argc; i++ )
  {
    printf(“Parameter %d is: %s\n", i, argv[i]);
  }
  return 0;
}
```



Content

- Static structures
  - Array as parameters
  - Static Structure creation
- Dynamic structures
  - Pointers
  - How to allocate and free memory
  - Structures

Array as parameter

- Arrays (and matrices) can only be passed by reference.
- Necessary to pass the address of the first element of the array.

```
void VisualizeArray (int [], int); /* prototype */

int main()
{
  int array[5]={1,2,4,6,10}
  VisualizeArray(&array[0],5);
  return 0;
}

void VisualizeArray (int vec[], int len)
{
  int i;
  for (i = 0; i < len; i++)
    printf("%d", vec[i]);
}
```

Content

- Static structures
  - Array as parameters
  - Static Structure creation

- Dynamic structures
  - Pointers
  - How to allocate and free memory
  - Structures

Structures

struct: a way to compose existing types into a structure

`struct` complex data structure that contains a set of fields with different types

```
struct [label]
{
      type field1;
      type field2;
      ...
};
```

Structures

```
struct person
{
   char  name[20];
   int   age;
   float weight;
};

struct person he={“John Smith”,31,80};
struct person all[20];

               printf(“His name is %s\n”, he.name);
               all[2].age=20;
```

Content

- Static structures
  - Array as parameters
  - Static Structure creation

- Dynamic structures
  - Pointers
  - How to allocate and free memory
  - Structures

Dynamic structures

Problems with static data structures:

- Difficulty predicting the required memory - 100 elements or 50000 elements?
- When inserting and removing, it may be necessary to move a large part of the elements

We need to be able to build data structures that we can easily traverse and where we can add new elements without imposing a maximum number of elements on coding time.

Dynamic structures

Introduce dynamic data structures

- The memory space they use is variable (dynamic) and adapts to the number of real elements we have at any given time.
- This type of structure is called dynamic data structure
- The mechanism that allows us to implement dynamic data structures are the pointers.

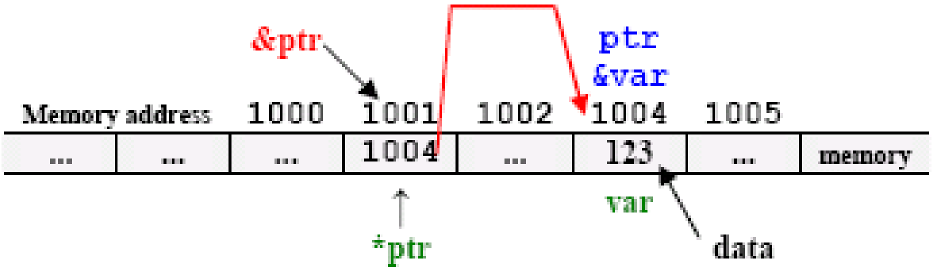
Stack

Queue

List

Pointers

- A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.
- A pointer refers to another variable in memory.




```
*ptr    -   a pointer variable
var     -   a normal variable
// declare a pointer variable ptr of type int
int *ptr;
// declare and initialize normal variable var of type int
int var = 123;
// assign the address of normal variable var to pointer ptr.
ptr = &var;
```

Pointers

*, &, ->

Return the content of the variable (*).

Return the address of the operand (&).

Access to a field of a structure (->).

Operations

Assign (=)

Compare (==, !=)

Inicialize (NULL)

Increment (++), Decrement (--)

Declaration

```
<tipus>* <nom_variable>;
```

Exemples:

```
int* pInteger;
float* pFloat
Student* pStudent;
```

Pointers

```
int n1=3, n2=10;
int *pn1, *pn2;

pn1 = pn2 = NULL;

pn1 = &n1;          /* ‘pn1’ points ‘n1’ */
pn2 = &n2;          /* ‘pn2’ points ‘n2’ */

if (pn1 != pn2)
{
  printf(“pn1 and pn2 points different memory
          positions \n”);
  printf(“The content of pn1 is: %d \n”, *pn1);
  printf(“ The content of pn2 is: %d \n”, *pn2);
}
```

Pointers

```
int n1=3, n2=10;
int *pn1, *pn2;

pn1 = pn2 = NULL;

pn1 = &n1; /* ‘pn1’ points ‘n1’ */
pn2 = pn1; /* ‘pn2’ points ‘n1’ */
if(pn1 == pn2)
{
  printf(“pn1 and pn2 points the same memory position \n”);
  printf(“The content of pn1 and pn2 is: %d\n”,*pn1);
}
```

Pointers

```
int main()
{
  int x, *p;

  x  = 10;
  *p = x;
  return 0;
}
```

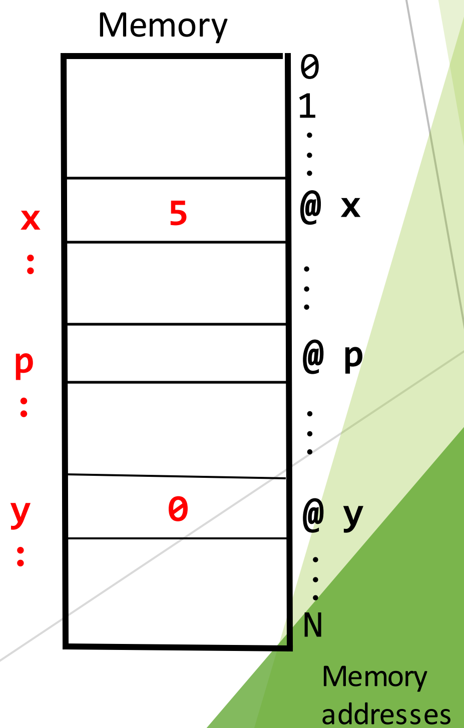
Is it a correct code?

```
int main()
{
  int x, *p;
  x = 10;
  p = &x;
  printf(“%d”, *p);
  return 0;
}
```

What is the value of *p?

Pointers

```
int x, y;
int *p;

x = 5;
y = 0;
```

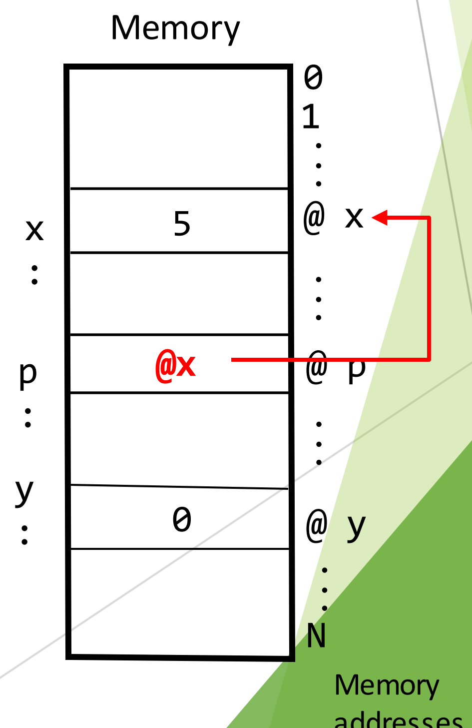


Pointers

```c
int x,y;
int *p;

x = 5;
y = 0;

p = &x;
```

Operator &

&x: return the address of the memory of the variable x



- A variable of a *pointer type p* contains an address of the memory
- If p contains the address of x (*p = &x*) we say
  - *p* points *x*

  p -> *x*

Pointers

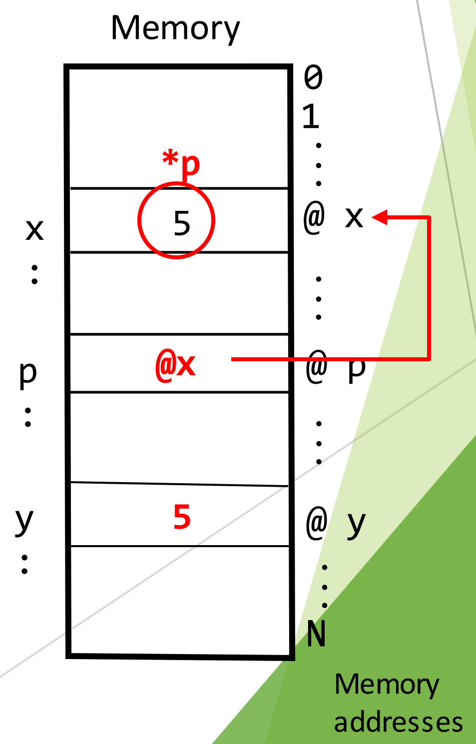
```c
int x,y;
int *p;

x = 5;
y = 0;

p = &x;
y = *p;
*p = 10;
```

Operator &

&x: return the address of the memory of the variable x

Operator *

*p: return the value of the variable which p points to



- A variable of a *pointer type p* contains an address of the memory
- If p contains the address of x (*p = &x*) we say
  - *p* points *x*
  - *p* references *x (*p iqual to x)*

Pointers

```
int x,y;
int *p;

x = 5;
y = 0;

p = &x;
y = *p;
*p = 10;
```

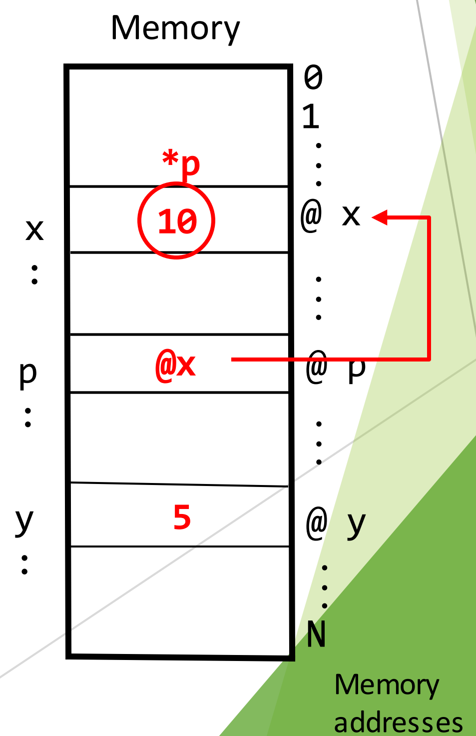
Operator &

&x: return the address of the memory of the variable x

Operator *

*p: return the value of the variable which p points to

- A variable of a *pointer type p* contains an address of the memory
- If p contains the address of x (*p = &x*) we say
  - *p* points *x*
  - *p* references *x (*p iqual to x)*



Pointers

```
int x, y;
int *p, *q;

x = 5;
y = 0;

p = &x;
y = *p;
*p = 10;
q = p;
p = &y;
*p = *q + 2;
```

	x	y	p	q
1	5	-	-	-
2	5	0	-	-
3	5	0	@x	-
4	5	5	@x	-
5	10	5	@x	-
6	10	5	@x	@x
7	10	5	@y	@x
8	10	12	@y	@x

```
x  &x
y  &y
*p p  &p
*q q  &q
```

Pointers

```c
int x,y;
int *p;

x = 5;
y = 0;

p = &x;
y = *p;
*p = 10;
p = NULL;
```

Operator &

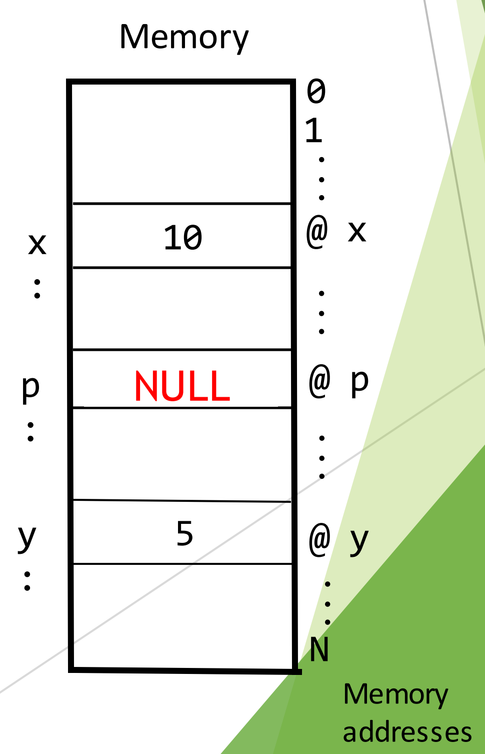
&x: return the address of the memory of the variable x

Operator *

*p: return the value of the variable which p points to

Value NULL

Special value that indicates that the pointer does not point any valid address



Pointers

```c
int x,y;
int *p;

x = 5;
y = 0;

p = &x;
y = *p;
*p = 10;
p = NULL;
*p = 0;
```

NOT CORRECT!!!

Operator &

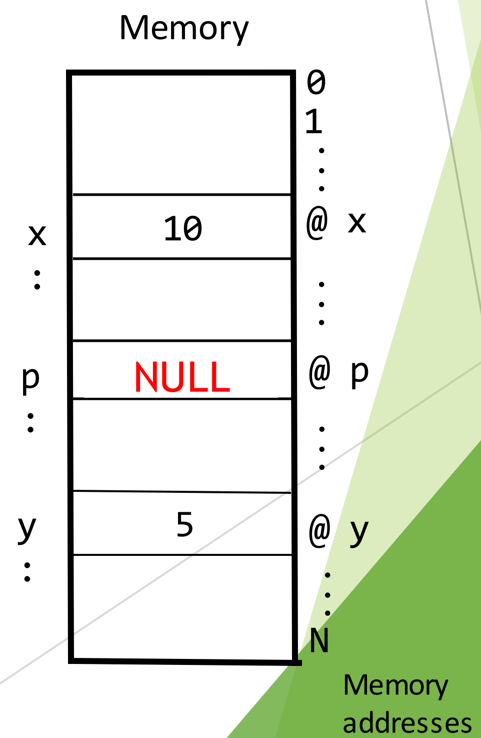
&x: return the address of the memory of the variable x

Operator *

*p: return the value of the variable which p points to

Value NULL

Special value that indicates that the pointer does not point any valid address



Pointers

```
void interchange (int* p_x, int* p_y)
{
   int tmp;

   tmp = *p_x;
   *p_x = *p_y;
   *p_y = tmp;
}
```

```
int main()
{
    int x=4, y=3;
    interchange (&x, &y);
    return 0;
}
```

Content

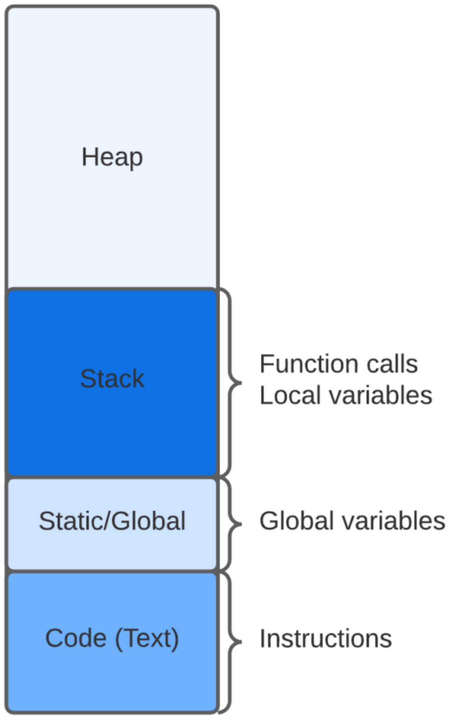
- Static structures
  - Array as parameters
  - Static Structure creation
- Dynamic structures
  - Pointers
  - How to allocate and free memory
  - Structures

How to allocate and free memory

Function	Task
malloc( )	Allocate request size of bytes & return a pointer to the first byte of the allocated space. And contains garbage values.
calloc( )	Allocate space for an array of elements, initialize them to zero and returns a pointer to the first byte of allocated space.
realloc( )	Modify the size of previously allocated space.
free( )	Free the previously allocated space.

How to allocate and free memory

```
#include <stdlib.h>
int global=0;
int main()
{
  int *ptr;
  ptr = (int *) malloc(sizeof(int));
  if (ptr != NULL)
    *ptr = 10;
  free(ptr);
  return 0;
}
```



How to allocate and free memory

```
#include <stdlib.h>
int main()
{
  int *ptr;
  ptr = malloc(15*sizeof(int));  // 15*sizeof(*ptr)
  if (ptr != NULL)
    *(ptr + 5) = 45;
  free(ptr);
  return 0;
}
```

How to allocate and free memory

```
#include <stdlib.h>
int main()
{
  int *ptr;
  //a block of 15 integers
  ptr = malloc(15*sizeof(int));

  if (ptr != NULL)
  {
    //assign 45 to sixth integer
    *(ptr + 5) = 45;
    printf("Value of the 6th integer is %d",*(ptr + 5));
  }
  free(ptr);
}
```

How to allocate and free memory

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

int main()
{
    int* ptr;
    int size;

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

    //  Memory allocates dynamically
    ptr=(int*)malloc(size*sizeof(int));

    // Checking for memory allocation
    if (ptr == NULL)
    {
        printf("Memory not allocated.\n");
        return 1;  //exit(1);
    }
```

How to allocate and free memory

```
printf("Memory successfully allocated.\n");

// Initialize the elements of the array
for (int j = 0; j < size; ++j) {
    ptr[j] = j + 1;   // *(ptr + j) = j + 1
}

printf("The elements of the array are: ");
for (int k = 0; k < size; ++k) {
    printf("%d, ", ptr[k]);  // *(ptr + j)
}

// Free the memory
free(ptr);
printf("Malloc Memory successfully freed.\n");

return 0;
}
```

Content

- Static structures
  - Array as parameters
  - Static Structure creation

- Dynamic structures
  - Pointers
  - How to allocate and free memory
  - Structures

Structures

```
struct person
{
   char  name[20];
   int   age;
   float weight;
} you;

struct person he={“John Smith”,31,80};
struct person all[20];
struct person *she;

printf(“His name is %s\n”, he.name);
all[2].age=20;
she=&all[2];
printf(“Her age is %d\n”,she->age);
```

Fields are accessed using ‘.’ notation or -> in case of pointers to the structure.

Structures

```
typedef struct person
{
   char  name[20];
   int   age;
   float weight;
} Person;
```

```
// struct person all[20];
Person * all; //struct person *all;

all=(Person *)malloc(20*sizeof(Person))
all->age=5; // (all+0) equivalent to &all[0]
(all+1)->weight=10.5; // (all+1) equivalent to &all[1]
printf("First age is %d\n", all->age);
printf("Second weight is %d\n", (all+1)->weight);
```

Structures

```
typedef struct Pixel
{
   unsigned char R;
   unsigned char G;
   unsigned char B;
} Pixel;
```

variable pix is defined as a pointer to a Pixel:
Pixel * pix;

Expression	Meaning
pix	a pointer to a Pixel struct
*pix	the Pixel struct itself
(*pix).R	the R field of the Pixel struct
pix->R	an alternate way to reference the R field of the Pixel struct