C Developer

Pointers

SUPINFO

Course Objectives

- ✓ Define the notion of pointer
- ✓ Understand the relationship between a pointer and an array
- ✓ Study the case of pointers to a structure

Course Plan

1. Concept of Pointers

2. Pointers and Arrays

3. Pointers and Structures

1. Concept of Pointers

1. Concept of Pointers

Memory organization

- In a computer, the memory is organized in *cells*, which can be identified by their **addresses**

Address	Value
…	
66280	
66281	
66282	
66283	
66284	
66285	
…	

- Each cell contains one **byte**

1. Concept of Pointers

Memory organization

- Depending on the type of data stored, one or more *cells* are occupied
- For example, to store a variable of **char** type it will take one byte:

Address	Value
…	
66280	
66281	A
66282	
66283	
66284	
66285	
…	

1. Concept of Pointers

Memory organization

- To store a variable of **int** type it will take several bytes, which can vary according to the processor or the operating system:

Address	Value
…	
66280	
66281	233
66282	
66283	
66284	
66285	
…	

1. Concept of Pointers

Variables and memory

- To declare a variable of a certain type is the same as reserving enough memory space to store a data of the given type
- When the variable is used, a link is made between its name and the address where its content is stored
- This is called **direct access** to the data

1. Concept of Pointers

Variables and memory

- A **date** variable of the **int** type:

<table>
<tr><th>Address</th><th>Value</th></tr>
<tr><td>…</td><td></td></tr>
<tr><td>66280</td><td></td></tr>
<tr><td>66281</td><td rowspan="4">23101991</td></tr>
<tr><td>66282</td></tr>
<tr><td>66283</td></tr>
<tr><td>66284</td></tr>
<tr><td>66285</td><td></td></tr>
<tr><td>…</td><td></td></tr>
</table>

date

1. Concept of Pointers

Variables and memory

- Given a variable, we can also access the address from which its content is stored
- The address operator “**&**” is used for that
- Its use is very simple: this operator is followed by the variable name
- We already used it with **scanf**

1. Concept of Pointers

Variables and memory

- A **date** variable of the **int** type:

&date

<table>
<tr><th>Address</th><th>Value</th></tr>
<tr><td>…</td><td></td></tr>
<tr><td>66280</td><td></td></tr>
<tr><td>66281</td><td rowspan="4">23101991</td></tr>
<tr><td>66282</td></tr>
<tr><td>66283</td></tr>
<tr><td>66284</td></tr>
<tr><td>66285</td><td></td></tr>
<tr><td>…</td><td></td></tr>
</table>

date

1. Concept of Pointers

Variables and memory

```c
#include <stdio.h>

int main()
{
    int x = 10;
    printf("Value of x: %d\n", x);
    printf("Address %%d of x: %d\n", &x);
    printf("Address %%x of x: %x\n", &x);
    printf("Address %%p of x: %p\n", &x);
    return 0;
}
```

```
Value of x: 10
Address %d of x: 6422300
Address %x of x: 61ff1c
Address %p of x: 0061FF1C
```

```c
#include <stdio.h>

int main()
{
    char x, y;
    printf("&x: %p\n&y: %p\n", &x, &y);
    return 0;
}
```

```
&x: 0061FF1F
&y: 0061FF1E
```

1. Concept of Pointers

Definition and use of a pointer

- A pointer is a variable that contains a memory address
- When declaring it, we must specify the data type that will be stored from this address
- Syntax:

```
type *pt;
```

1. Concept of Pointers

Definition and use of a pointer

- If we want to access the address, we use the pointer under its name, as for a classical variable

- If we want to access the value, we use the **indirection operator** “*****”

- ***pt** is therefore the value stored from the **pt** address

1. Concept of Pointers

Definition and use of a pointer

- Same example with another point of view:

pt

<table>
<tr><th>Address</th><th>Value</th></tr>
<tr><td>…</td><td></td></tr>
<tr><td>66280</td><td></td></tr>
<tr><td>66281</td><td rowspan="4">23101991</td></tr>
<tr><td>66282</td></tr>
<tr><td>66283</td></tr>
<tr><td>66284</td></tr>
<tr><td>66285</td><td></td></tr>
<tr><td>…</td><td></td></tr>
</table>

*pt

```c
int *pt;
*pt = 23101991;
```

1. Concept of Pointers

Definition and use of a pointer

- Same example with another point of view:

pt

&date

<table>
<tr><th>Address</th><th>Value</th></tr>
<tr><td>…</td><td></td></tr>
<tr><td>66280</td><td></td></tr>
<tr><td>66281</td><td rowspan="4">23101991</td></tr>
<tr><td>66282</td></tr>
<tr><td>66283</td></tr>
<tr><td>66284</td></tr>
<tr><td>66285</td><td></td></tr>
<tr><td>…</td><td></td></tr>
</table>

*pt

date

```
int *pt;
int date = 23101991;
pt = &date;
```

1. Concept of Pointers

Definition and use of a pointer

- When manipulating a variable through a pointer containing its address, we say that we have an **indirect access** to the data stored under the name of the variable

```c
#include <stdio.h>

int main()
{
    int x = 10; int *px = &x;
    printf("Direct access x = %d & &x = %p\n", x, &x);
    printf("Indirect access *px = %d & px = %p\n", *px, px);
    *px = 666;
    printf("x updated through px: %d\n", x);
    return 0;
}
```

```
Direct access x = 10 & &x = 0061FF18
Indirect access *px = 10 & px = 0061FF18
x updated through px: 666
```

1. Concept of Pointers

Definition and use of a pointer

- When declaring a pointer, we first reserve a memory location able to contain an address

- A memory location is also reserved to contain a data of the expected type if you use:

```
type *pt;
```

1. Concept of Pointers

Definition and use of a pointer

- Therefore, a pointer is usually declared as follows:

```
type *p = NULL;
```

- In this case no memory is yet reserved to store the data and it will be only at the use of the pointer that this allocation will be done

- This allows you to use memory only when you really need it

- At the end of the use of the pointer, we will even free the allocated memory by reassigning the **NULL** value to the pointer

1. Concept of Pointers

Usefulness of pointers

- Dynamically manage the size of arrays
- Be able to modify the value of parameters passed to a function
- Create dynamic variables
- Be able to define data structures that are more flexible than arrays

1. Concept of Pointers

Dynamic variable allocation

- The goal is to allocate memory for variables when you really need it
- This can be useful to avoid wasting space
- We can also declare variables whose lifetime will be longer than that of the subroutines using them

1. Concept of Pointers

SUPINFO

Dynamic variable allocation

#include <stdlib.h>

1. We declare a pointer:

```
type *p = NULL;
```

2. When we need it, we reserve some memory for our variable:

```
p = malloc(sizeof(type));
```

3. When the use is over, we release the memory:

```
free(p);
```

1. Concept of Pointers

Dynamic variable allocation

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

int main()
{
    int *p = NULL;
    p = malloc(sizeof(int));
    *p = 123;
    printf("Value: %d\n", *p);
    free(p);
    printf("Value: %d\n", *p);
    return 0;
}
```

```
Value: 123
Value: 13115928
```

1. Concept of Pointers

Dynamic variable allocation

- To allocate a block of memory, use the **malloc** function

- Function signature:

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

  - It takes as parameter the size of the memory block, in bytes
  - The returned value is a pointer to the memory block allocated by the function; the type of this pointer is always **void***, which can be cast to the desired type of data pointer in order to be dereferenceable

1. Concept of Pointers

Dynamic variable allocation

- It can be useful to check if the dynamic memory allocation worked well, to check if there was enough space left
- Otherwise, the **malloc** function returns the **NULL** value to the pointer
- A simple test of nullity of the pointer will thus allow to know if we can really manipulate it
- You do not want to risk overwriting other data!

1. Concept of Pointers

Questions

2. Pointers and Arrays

2. Pointers and Arrays

Pointer arithmetic

- If we have an array **tab** and a pointer **pt** to the first element of this array, **pt + 1** will be a pointer to the second cell of the array
- The address contained in **pt + 1** is thus not the address contained in **pt** increased by 1
- There is an adjustment according to the type of the array elements

2. Pointers and Arrays

Pointer arithmetic

```c
#include <stdio.h>

int main()
{
    int tab[3] = {1, 2, 3};
    int *pi = &tab[0];
    printf("*pi = %d & *(pi+1) = %d & *(pi+2) = %d\n", *pi, *(pi+1), *(pi+2));
    float tab2[3] = {1.1, 2.2, 3.3};
    float *pf = &tab2[0];
    printf("*pf = %f & *(pf+1) = %f & *(pf+2) = %f\n", *pf, *(pf+1), *(pf+2));
    return 0;
}
```

```
*pi = 1 & *(pi+1) = 2 & *(pi+2) = 3
*pf = 1.100000 & *(pf+1) = 2.200000 & *(pf+2) = 3.300000
```

2. Pointers and Arrays

Pointer arithmetic

- On the same principle, you can use the “**++**” increment operator with pointers

- If **pt** is pointing to the first element of an array, after the operation **pt++**, it will be pointing to the second

- The difference between pre and post increment is still the same

2. Pointers and Arrays

Pointer arithmetic

```c
#include <stdio.h>

int main()
{
    int tab[3] = {1, 2, 3};
    int *pi = &tab[0];
    for (int k = 0; k < 3; k++) {
        printf("%d\t", *pi++);
    }
    return 0;
}
```

```
1       2       3
```

2. Pointers and Arrays

Pointer arithmetic

- The subtraction and decrement operations take place in the same way as the addition and incrementation operations, but in order for them to make sense the corresponding pointer must not refer to the first cell of the array

- You will not cause a compile error, but you will access other data; with the risk of uncontrolled modifications that this involves

2. Pointers and Arrays

Pointer arithmetic

```c
#include <stdio.h>

int main()
{
    int tab[3] = {1, 2, 3};
    int *pi = &tab[2];
    printf("%d\t", *pi);
    printf("%d\t", *--pi);
    printf("%d\n", *(pi-1));
    return 0;
}
```

```
3        2        1
```

2. Pointers and Arrays

Pointer arithmetic

- If you have declared an array:

```
type tab[...];
```

- The name of the array, here **tab**, is always the address of its first cell
- Thus, **tab** and **&tab[0]** are equivalent
- **tab** is therefore a pointer, but a constant pointer that will always point to the first element of the array (it cannot be assigned another value)

2. Pointers and Arrays

Pointer arithmetic

- Since **tab** is a constant pointer, operations like **tab++** are forbidden

- On the other hand, if **pt** is a pointer of the same type, we can make assignments of the following form:

```
pt = tab + 3;
```

- **&tab[n]** and **tab+n** are two identical addresses

- **tab[n]** and ***(tab+n)** are two identical values

2. Pointers and Arrays

Pointer arithmetic

```c
#include <stdio.h>

int main()
{
    int tab[3] = {1, 2, 3};
    printf("&tab[2] = %p & tab+2 = %p\n", &tab[2], tab+2);
    printf("tab[2] = %d & *(tab+2) = %d\n", tab[2], *(tab+2));
    return 0;
}
```

```
&tab[2] = 0061FF1C & tab+2 = 0061FF1C
tab[2] = 3 & *(tab+2) = 3
```

2. Pointers and Arrays

Pointer arithmetic

- If you have declared an array:

```
type tab[...];
```

- Then a pointer:

```
type *pt = tab;
```

- Then the writing **pt[n]** makes sense and is worth ***(pt+n)** (and thus also **tab[n]**)

2. Pointers and Arrays

Pointer arithmetic

- If you have declared an array:

```
type tab[...];
```

- Then a pointer:

```
type *pt = &tab[k]; //k -> array size
```

- Then the writing **pt[-n]** makes sense and is worth ***(pt-n)** (and thus also **tab[k-n]**)

2. Pointers and Arrays

Pointer arithmetic

```c
#include <stdio.h>

int main()
{
    int tab[3] = {1, 2, 3};
    int *pt = tab;
    printf("pt[2] = %d & *(pt+2) = %d\n", pt[2], *(pt+2));
    int *pt2 = &tab[2];
    printf("pt2[-1] = %d & *(pt2-1) = %d\n", pt2[-1], *(pt2-1));
    return 0;
}
```

```
pt[2] = 3 & *(pt+2) = 3
pt2[-1] = 2 & *(pt2-1) = 2
```

2. Pointers and Arrays

Constant strings and pointers

- An assignment like the one below, means that we initialize **ps** with the address of the "**Hello world!**" constant string:

```
char *ps = "Hello world!";
```

- You cannot change the value of a constant; this operation is therefore illicit:

```
ps[3] = 'X';
```

2. Pointers and Arrays

Constant strings and pointers

```c
#include <stdio.h>

int main()
{
    char *ps = "Hello world!";
    printf("%c\n", ps[3]);
    ps[3] = 'X';
    printf("%c\n", ps[3]);
    return 0;
}
```

```
l

Process returned -1073741819 (0xC0000005)
```

2. Pointers and Arrays

Constant strings and pointers

```c
#include <stdio.h>

int main()
{
    char tab[] = "Hello world!";
    printf("%c\n", tab[3]);
    tab[3] = 'X';
    printf("%c\n", tab[3]);
    char *ps = tab;
    ps[9] = 'X';
    printf("%s\n", ps);
    return 0;
}
```

```
l
X
HelXo worXd!

Process returned 0 (0x0)
```

2. Pointers and Arrays

Dynamic array allocation

- The goal here is to be able to define arrays whose dimension is known only at the program execution
- For example, if this dimension is entered by the user himself, or if it is the result of a calculation
- We need to use the **malloc** function

1. Concept of Pointers

SUPINFO

Dynamic array allocation

#include <stdlib.h>

- Same procedure as for a variable, but with **n** elements:

```
type *tabDyn = NULL;
tabDyn = malloc(n*sizeof(type));
free(tabDyn);
```

- The **sizeof** function allows you to know the number of bytes that a variable of a given type takes in memory; using it ensures the portability of the program, because from one computer to another the same type can be stored with different numbers of bytes

- As in the case of a dynamic variable, precautions can be taken by checking that the allocation is successful

1. Concept of Pointers

Dynamic array allocation

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

int main()
{
    int *tabDyn = NULL; int n;
    printf("Size: ");
    scanf("%d", &n);
    tabDyn = malloc(n*sizeof(int));
    for(int i = 0; i < n; i++) {
        tabDyn[i] = i*i;
        printf("%d ", tabDyn[i]);
    }
    free(tabDyn);
    return 0;
}
```

```
Size: 10
0 1 4 9 16 25 36 49 64 81
```

2. Pointers and Arrays

Static arrays of pointers

1. We declare a usual static array (not dynamic):

```
type *tab[dim];
```

2. For each row **i** we size the number **n** of columns:

```
tab[i] = malloc(ni*sizeof(type));
```

3. When the use is over, we release the memory on each row:

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

2. Pointers and Arrays

Static arrays of pointers

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

int main()
{
    int *tab[5]; int i, j;
    for(i = 0; i < 5; i++) {
        tab[i] = malloc((i+1)*sizeof(int));
        for(j = 0; j < i+1; j++) {
            tab[i][j] = i+1;
            printf("%d", tab[i][j]);
        }
        printf("\n");
    }
    for(i = 0; i < 5; i++) {
        free(tab[i]);
    }
    return 0;
}
```

```
1
22
333
4444
55555
```

2. Pointers and Arrays

Static arrays of pointers

```c
#include <stdio.h>

int main()
{
    char *tab[3]; int i;
    tab[0] = "Hello";
    tab[1] = "Bye";
    tab[2] = "MfmoIHY7Z-k";
    for(i = 0; i < 3; i++) {
        printf("%s\n", tab[i]);
    }
    return 0;
}
```

```
Hello
Bye
MfmoIHY7Z-k
```

2. Pointers and Arrays

Dynamic arrays of pointers

1. We declare a *double* pointer:

```
type **tab;
```

2. We then size the number **m** of rows:

```
tab = malloc(m*sizeof(type*));
```

3. For each row **i** we size the number **n** of columns:

$$tab[i] = malloc(n_i*sizeof(type));$$

2. Pointers and Arrays

Dynamic arrays of pointers

4. When the use is over, we release the memory on each row:

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

5. We then free the memory allocated to the array:

```
free(tab);
```

2. Pointers and Arrays

Dynamic arrays of pointers

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

int main()
{
    int **tab; int i, j, n;
    printf("Size: ");
    scanf("%d", &n);
    tab = malloc(n*sizeof(int*));
    for(i = 0; i < n; i++) {
        tab[i] = malloc((i+1)*sizeof(int));
        for(j = 0; j < i+1; j++) {
            tab[i][j] = i+1;
            printf("%d", tab[i][j]);
        }
        printf("\n");
    }
    for(i = 0; i < n; i++) {
        free(tab[i]);
    }
    free(tab);
    return 0;
}
```

```
Size: 7
1
22
333
4444
55555
666666
7777777
```

2. Pointers and Arrays

Exercise

- Make the user enter the size of an array of integers
- Fill it in randomly using the **rand** function
- Calculate the maximum of the array

2. Pointers and Arrays

Questions

3. Pointers and Structures

3. Pointers and Structures

Pointers to a structure

- We can define pointers to a structure:

```
struct structureName {
     type attribute1;
     type attribute2;
     ...
};
```

```
struct structureName variableName;
```

```
struct structureName *pt = &variableName;
```

- We can declare pointers to complex data types that we define ourselves

3. Pointers and Structures

Pointers to a structure

- There are two ways to access the attributes:
  - In the usual way:

```
(*pt).attribute1
```

  - With the arrow operator “**->**”:

```
pt->attribute1
```

- We will prefer the second syntax which is less cumbersome

3. Pointers and Structures

Pointers to a structure

```c
#include <stdio.h>

int main()
{
    struct user {
        int id;
        char name[55];
        int level;
    };
    struct user John;
    struct user *pt;
    pt = &John;
    John.id = 4;
    printf("(*pt).id = %d\n", (*pt).id);
    printf("pt->id = %d\n", pt->id);
    return 0;
}
```

```
(*pt).id = 4
pt->id = 4
```

3. Pointers and Structures

Dynamic arrays of structures

- Dynamic arrays of structures can be defined as with an elementary type
- The **sizeof** function also works with complex data types that we define

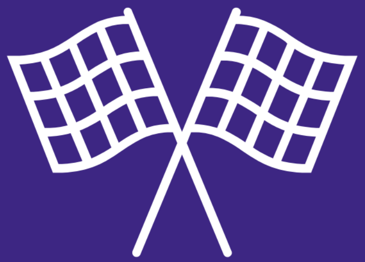
```c
#include <stdio.h>

int main()
{
    struct user {
        int id;
        char name[55];
        int level;
    };
    struct user *tabDyn;
    tabDyn = malloc(3*sizeof(struct user));
    tabDyn[0].id = 4;
    (tabDyn+1)->id = 5;
    (*(tabDyn+2)).id = 6;
    printf("%d %d %d\n", tabDyn[0].id, tabDyn[1].id, tabDyn[2].id);
    return 0;
}
```

```
4 5 6
```

3. Pointers and Structures

Questions

Pointers



Thank you for your attention