First Year Engineer in computer science

CHAPTER 10:

Pointers

2023-2024

Sid Ahmed BERRABAH

Definition

- Every variable is a memory location, and every memory location has a defined address that can be accessed using the ampersand operator (&), which means memory address.

- Consider the following C code:

```
int var1;
char var2[10];
printf("Address of variable var1 : %x\n", &var1) ;
printf("Address of variable var2 : %x\n", &var2) ;
```

Prints:

```
Address of variable var1 : b2c29be8
Address of variable var2 : b2c29bee
```

Definition

- A pointer is a variable that stores the address of another variable, which is the address of the memory location.
- Like any other variable, a pointer must be declared before it can be used. Therefore, a pointer is also a memory location.
- A pointer is declared as follows:

type *variable_name ;

Where **type** is the base type of the pointer variable, which is the type of data that the pointer can point to.

The following declarations are also correct:

```
type * variable_name;
type*  variable_name;
```

Definition

- Example of pointer declaration

```
int *ip ;    // pointer to an int

double *dp ;  // pointer to a double

float *fp ;    // pointer to a float

char *ch ;   // pointer to a  char
```

The value of all these pointers is always a long hexadecimal number that represents a memory address. The only difference between these pointers is the type of data they point to.

Initialize and manipulate pointers

- A good practice is to assign the value NULL (in capital letters) to the pointer during its declaration. This helps to avoid the risk of inadvertently modifying arbitrary memory locations, which could lead to a runtime error (possibly even a fatal runtime error).

  Example : **float *ptf = NULL ;**

This indicates that the pointer is not currently pointing to any variable.

- To be used, a pointer needs to point to an address. This is done by using the & operator to a variable.

  Example :

```
int x, *pt ;
pt = &x ;
```

Initialize and manipulate pointers

- Once a pointer point to an address, The operator * allows accessing the value contained at the pointed address.

Example :

```
int x = 25 ;
int *pt = &x;
printf("The value at the address pointed by pt is %d ", *pt ) ; //display 25
*pt = 30;
printf("the value of x is %d ", x ) ;   //display 30
printf("The value at the address pointed by pt is %d ", *pt ) ; //display 30
```

Pointers' Arithmetic

- **Addition/subtraction of an integer**

Let **p** be a pointer to a data of type **t** located at address **a**, and let **s** be the number of bytes occupied by a variable of type **t**.

- It is possible to define an expression in which we add or subtract an integer **k** to/from the pointer **p**.
- The value of this expression is an address of the same type as **p**, increased or decreased by **k × s** bytes.

p + k = $a + k \times s$ and p - k = $a - k \times s$

Arithmétique des pointeurs

- **Subtraction of pointers**
  - Only subtraction is defined between two pointers.
  - The two pointers must be of the same type.
  - The result of the operation is an integer representing the number of elements separating the pointed elements in memory.

Exercise1

Complete the situation table for the following program:

```
1.  short x,y,*p1,*p2;
2.  x = 4;
3.  p1 = &x;
4.  y = *p1;
5.  y = 8;
6.  p2 = &y;
7.  (*p1)++;
8.  y = p1;
9.  y++;
10. p2 = 12;
11. p1 = y;
```

	x	y	p1	*p1	p2	*p2
1						
2						
3						
4						
5						-
6						
7						
8						
9						
10						
11						

Exercise1

Complete the situation table for the following program:

```
1.  short x,y,*p1,*p2;
2.  x = 4;
3.  p1 = &x;
4.  y = *p1;
5.  y = 8;
6.  p2 = &y;
7.  (*p1)++;
8.  y = p1;
9.  y++;
10. p2 = 12;
11. p1 = y;
```

	x	y	p1	*p1	p2	*p2
1	-	-	NULL	-	NULL	-
2	4	-	NULL	-	NULL	-
3	4	-	&x	4	NULL	-
4	4	4	&x	4	NULL	-
5	4	8	&x	4	NULL	-
6	4	8	&x	4	&y	8
7	5	8	&x	5	&y	8
8	5	?	&x	5	&y	&x
9	5	?	&x	5	&y	&x+1
10	5	?	&x	5	12	?
11	5	?	?	?	12	?

Exercise1

Complete the situation table for the following program:

```
1.  short x,y,*p1,*p2;
2.  x = 4;
3.  p1 = &x;
4.  y = *p1;
5.  y = 8;
6.  p2 = &y;
7.  (*p1)++;
8.  y = p1;
9.  y++;
10. p2 = 12;
11. p1 = y;
```

In fact, the operations on lines 8, 10 and 11 are not valid

➔line 8 : we assign a short* value to a variable of type short

➔lines 10 and 11 : we assign a short values to pointers to shorts

Exercise2

Execute the following codes:

```
int main()
{
    int i =3, j =6;
    int ∗p1 , ∗ p2 ;
    p1 = & i ;
    p2 = &j ;
   ∗p1 = ∗p2;
   printf(“i = %d et j = %d”,i,j);
}
```

```
int main()
{
    int i =3, j =6;
    int ∗p1 , ∗ p2 ;
    p1 = & i ;
    p2 = &j ;
    p1 = p2;
   printf(“i = %d et j = %d”,i,j);
}
```

Exercise2

Execute the following codes:

```
int main()
{
    int i =3, j =6;
    int ∗p1 , ∗ p2 ;
    p1 = & i ;
    p2 = &j ;
   ∗p1 = ∗p2;
   printf(“i = %d et j = %d”,i,j);
}
```

i = 6 et j = 6

```
int main()
{
    int i =3, j =6;
    int ∗p1 , ∗ p2 ;
    p1 = & i ;
    p2 = &j ;
    p1 = p2;
   printf(“i = %d et j = %d”,i,j);
}
```

i = 3 et j = 6

Passing parameters by reference in a function

Example with passing parameters by value:

```
int swap(int a , int b)
{
     printf("in function swap : before swapping a = %d and b = %d ", a, b);
     int c = a;
     a = b ;
     b = c ;
     printf("in function swap : after swapping a = %d and b = %d ", a, b);
}

int main()
{
    int a = 5 , b = 8 ;
    printf("in function main : before call swap a = %d and b = %d ", a, b);
    swap(a , b) ;
    printf("in function main : after call swap a = %d and b = %d ", a, b);
    Return 0;
}
```

Passing parameters by reference in a function

Example with passing parameters by value:

```
int swap(int a , int b)
{
    printf("in function swap : before swapping a = %d and b = %d ", a, b);
    int c = a;
    a = b ;
    b = c ;
    printf("in function swap : after swapping a = %d and b = %d ", a, b);
}

int main()
{
   int a = 5 , b = 8 ;
   printf("in function main : before call swap a = %d and b = %d ", a, b);
   swap(a , b) ;
   printf("in function main : after call swap a = %d and b = %d ", a, b);
   Return 0;
}
```

Display :

```
in function main : before call swap a = 5 and b = 8
in function swap : before swapping a = 5 and b = 8
in function swap : after swapping a = 8 and b = 5
in function main : after call swap a = 5 and b = 8
```

Passing parameters by reference in a function

Example with passing parameters by reference:

```
int swap(int *a , int *b)
{
     printf("in function swap : before swapping a = %d and b = %d ", *a, *b);
     int c = *a;
     *a = *b ;
     *b = c ;
     printf("in function swap : after swapping a = %d and b = %d ", *a, *b);
}

int main()
{
    int a = 5 , b = 8 ;
    printf("in function main : before call swap a = %d and b = %d ", a, b);
    swap(&a , &b) ;
    printf("in function main : after call swap a = %d and b = %d ", a, b);
    Return 0;
}
```

Passing parameters by reference in a function

Example with passing parameters by reference:

```
int swap(int *a , int *b)
{
    printf("in function swap : before swapping a = %d and b = %d ", *a, *b);
    int c = *a;
    *a = *b ;
    *b = c ;
    printf("in function swap : after swapping a = %d and b = %d ", *a, *b);
}

int main()
{
   int a = 5 , b = 8 ;
   printf("in function main : before call swap a = %d and b = %d ", a, b);
   swap(&a , &b) ;
   printf("in function main : after call swap a = %d and b = %d ", a, b);
   Return 0;
}
```

Display :

```
in function main : before call swap a = 5 and b = 8
in function swap : before swapping a = 5 and b = 8
in function swap : after swapping a = 8 and b = 5
in function main : after call swap a = 8 and b = 5
```

Passing parameters by reference in a function

Note:

In the call : swap(&a , &b);

the parameters are always passed by value, but in this case, the values of the expressions &a and &b are passed to the function. These values are the addresses of the variables a and b in the calling function.

Also note that in the function swap(), we swapped the values of the variables pointed to by the addresses a and b:

```
c = *a;
*a = *b;
*b = c;
```

Passing parameters by reference in a function

Exercise : Provide a C function that finds the maximum and minimum of a vector passed as a parameter.

Passing parameters by reference in a function

Exercise : Provide a C function that finds the maximum and minimum of a vector passed as a parameter.

```
void maxmin(float T[], int N, float *max, float *min)
{
    *max = T[0];
    *min = T[0];
    for(int i = 0; i < N; i++)
    {
        if(T[i] > *max) *max = T[i];
        if(T[i] < *min) *min = T[i];
    }
}
```

Pointers and Arrays

When the identifier of an array is used alone (without indices), it is considered as a (constant) pointer to the beginning of the array. This means that the identifier can be used to access the first element of the array.

1. One dimension array

Example : int t[10]

- The notation t is then equivalent to &t[0].
- The identifier t is considered as being of type pointer to the type corresponding to the elements of the array, which in this case is int *
- The following notations are equivalent:

t+1	&t[1]
t+i	&t[i]
t[i]	*(t+i)

Pointers and Arrays

Traversing the elements of a one-dimensional array

Example : set to 1 the elements of an array

- First way

```
for (int i=0 ; i<10 ; i++)
    t[ i ] = 1 ;
```

- Second way

```
for (int i=0 ; i<10 ; i++)
    *(t+i) = 1 ;
```

Pointers and Arrays

- third way

```
int i ;
int * p :
for (p=t, i=0 ; i<10 ; i++, p++)
    *p = 1 ;
```

Or

```
for (p=t ; p<t+10 ; p++)
    *p = 1 ;
```

In this third way, we had to copy the value represented by t into a pointer named p. In fact, the identifier t represents a constant address.

Pointers and Arrays

Exercise : Let P a pointer pointing to an array A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

What values or addresses provide the following expressions: ?

1. *P+2
2. *(P+2)
3. &P+1
4. &A[4]-3
5. A+3
6. &A[7]-P
7. P+(*P-10)
8. *(P+*(P+8)-A[7])

Pointers and Arrays

Exercise : Let P a pointer pointing to an array A:

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

What values or addresses provide the following expressions: ?

1. *P+2 → 14
2. *(P+2) → 34
3. &P+1 → &A[1] (address of the element with value 23
4. &A[4]-3 → &A[1] (address of the element with value 23
5. A+3 → &A[3]
6. &A[7]-P → 7 (&A[7] = P + 7 so P + 7 – P = 7)
7. P+(*P-10) → &A[2] (*P = 12 so P+(12-10) = P+2
8. *(P+*(P+8)-A[7]) → 23 (*(P+90-89) = *(P+1))

Pointers and Arrays

Exercice : Provide a C function to sort (using selection sort) an array of size N passed as a parameter. Use pointers.

Pointers and Arrays

Exercice : Provide a C function to sort (using selection sort) an array of size N passed as a parameter. Use pointers.

```
void sortSelection(int *p, int N)
{
    int *t, *s, *m;
    int c;
    for(t = p; t < p + N; t++){
        m = t;
        for(s = t+1; s < p + N; s++){
            if(*s < *m) m = s;
        }
        printf("min = %d\n",*m);
        c = *m;
        *m = *t;
        *t = c;
    }
}
```

Pointers and Arrays

2. Two-dimensional arrays

a) Example 1: fixed size array

A function to set to 1 the elements of an array of dimension 10x15.

```
void init( int t[10][15])
{    int i, j ;
     for (i=0 ; i<10 ; i++)
          for (j=0 ; j<15 ; j++)
                    t[i][j] = 1 ;
}
```

Pointers and Arrays

2. Two-dimensional arrays

Example : A function to set to 1 the elements of an array of dimension N x M:

```
void initMat ( int t[][M], int N, int M)
{    int i, j ;
     int *p = t ;
     for (i=0 ; i<N ; i++)
          for (j=0 ; j<M ; j++)
                    *(p+i*N+j) = 1 ;
}
```

Or

```
void initMat(int *p, int N, int M)
{
          int *s = p+N*M;
          for(p; p < s; p++)
               *p = 1;
 }
```

Pointer to pointer

Just like any other object, a pointer also has an address. Therefore, it is possible to create an object pointing to this pointer: a **pointer to pointer**.

```
int a = 10;
int *pa = &a;
int **pp = &pa;
```

We can have two indirections:

- One to reach the referenced pointer: *pp gives the content of pa which is the address of a.
- A second to reach the variable on which the first pointer points: **pp gives the content of a.

generic pointer

void type

- We have already encountered the keyword void with functions, which allows us to indicate that a function does not use any parameters and/or does not return any value.

- void is also a type, like int or double, but it is said to be incomplete, which means that its size cannot be calculated.

generic pointer

Pointer to void :

- A **pointer to void** is considered a generic pointer, which means that it can reference any type of object.
- it is possible to assign any object address to a pointer to void and to assign a pointer to void to any other pointer (and vice versa).

```
int a;
double b;
void *p;
double *r;

p = &a; /* Correct */
p = &b; /* Correct */
r = p; /* Correct */
```

generic pointer

Utility of the pointer to void :

- Creating a function that must be able to work with any type of pointer

Example :

```
void permuter(void *p1, void *p2, int type)
{
    if(type ==0){
        int c = *(int *)p1;
        *(int *)p1 = *(int *)p2;
        *(int *)p2 = c;
    }

    else{
        float c = *(float *)p1;
        *(float *)p1 = *(float *)p2;
        *(float *)p2 = c;
    }
}
```

Array of pointers

Syntax :

type * array_name[number of elements];

Example : int * A[10]; // an array of 10 pointers to integers

Dynamic memory allocation

- Until now, the declaration of a variable automatically allocates the necessary memory space.
- The number of bytes required was known at compile time;
- the compiler calculates this value based on the data type of the variable.

Example :

```
int A ;  // allocation of 4 bytes
char T[20] ;   // allocation of 20 bytes
```

Dynamic memory allocation

Problematic:

- Often, we need to work with data whose number and size we cannot predict when writing the program.
- The size of the data is only known at runtime.
- We must avoid the waste of reserving the maximum predictable space. Memory is a limited resource, and reserving too much memory can lead to performance problems or even crashes.

Purpose: We are looking for a way to reserve or free memory space as needed during program execution.

malloc function

- The **malloc** function from the **stdlib** library helps us to locate and reserve memory during the execution of a program.

Syntax: void ***malloc**(size_t **size**)

- Allocates a block of memory of **size** bytes.
- Returns a void pointer to the allocated memory and must be type-cast to the appropriate data type.

Example: char * T = (char *)malloc(4000);

This returns the address of a block of 4000 available bytes and assigns it to T.

- If there is not enough memory, T is assigned the value NULL.

malloc function

- If we want to allocate space for data of a type whose size varies from one machine to another, we can use sizeof to determine the effective size in order to preserve the portability of the program.

sizeof <var>: provide the size in memory of the variable **var**.

sizeof <const>: provide the size in memory of the constant **const**.

sizeof <type>: provide the size in memory of an object of type **type**.

malloc function

Example : allocate memory for X values of type int where X is given by the user.

```
int X;
int *pNum;
printf(“ give the number of values : “);
scanf( “%d”, &X);
pNum = malloc(X*sizeof(int)) ;
```

realloc function

- The **realloc** function from the stdlib library helps us to resize an existing block of memory during the execution of a program.

Syntax: void ***realloc**(void *ptr, size_t **new_size**)

- Resizes an existing block of memory pointed to by ptr to **new_size** bytes.

- Returns a pointer to the resized block (may be a different memory location).

Example:
```
char * T = (char *) malloc(4000);
...
T = (char *) realloc( T, 100);
```

Resize the array T from 4000 bytes to 100 bytes.

- If there is not enough memory, T is assigned the value NULL.

Memory deallocation

- If we no longer need a block of memory that we have reserved using malloc, we can free it using the free() function from the stdlib library.

```
free(pointer); // pointer points to the block to be freed.
```

Avoid: Trying to free memory with free() that was not allocated by malloc(). This can cause unexpected behavior or even crash the program.

Attention: The free() function does not change the content of the pointer. The pointer will still point to the same memory address, even after the memory has been freed. It is advisable to assign the value NULL to the pointer immediately after freeing the block of memory to which it was attached.

- If the memory is not explicitly freed using free(), then it is automatically freed at the end of the program execution.

Exercise

Write a C program to read The N real values of a vector (N given by the user).

The program should then delete the negative values from the vector.

The size of the vector in the memory should be the exact memory size needed the store its elements.