Subject #15: Pointers

- The following program is similar to our first program in C (without the input part), but it uses a function to swap numbers:

```
main()
{
    int i,j ;
    void swap(int *pi, int *pj) ;

    i = 5 ;
    j = 17 ;
    printf("At the beginning, i = %d and j = %d\n", i, j) ;

    swap(&i, &j) ;
    printf("At the end,       i = %d and j = %d\n", i, j) ;
}


void swap(int *pi, int *pj)
{
    int temp ;

    temp = *pi ;
    *pi = *pj ;
    *pj = temp ;
}
```

- `&i` is a *pointer* to the *int* variable `i`. It holds the memory address at which that variable is stored. Because C function calls pass parameters by value, in order for a function to access and change a variable in the function that called it, we must give it pointers as parameters. This is exactly what is done when we read input with the **scanf** function.
- The unary operator `&` is called the *address* operator.
- We can also define variables which are pointers to any legal type themselves. The declaration for a pointer to an *int*, for example, looks like

```
int *pi ;
```

  The declarations of the parameters of the function **swap** are exactly of this type.

- If `pi` is a pointer to an *int*, then `*pi` is the integer it points to. The unary operator `*` is called the *indirection* operator.
- Make sure now that you understand the above program!

- Generally in C, the syntax of declarations and the syntax of use both agree. The declaration for a pointer to *int* is `int *pi` because `*pi` is an *int*. The declaration of an *int* array is, say, `int arr[10]` because `arr[0]` is an *int*.
- One should use pointers with care, since an inappropriate use of a pointer might result in the program crashing or exhibiting an erratic behavior.
- Structures, as any other kind of variable, may be passed as arguments to functions. However, because they contain a lot of information it is usually better to pass a pointer to the structure, instead of passing the structure itself. Here is an example:

```
void summarize_student(struct student *s)
{
    s->final_grade = (s->exam_grade + s->project_grade)/2;
    printf("Name: %s, ID: %d, Grade: %d\n",
                  s->name, s->id, s->final_grade);
}
```

  The function call would look like this:

```
struct student st;

   .
   .
   .

summarize_student(&st);
```

- The expression:

```
s->name
```

  has the same meaning as:

```
(*s).name
```

Classwork: Minimum and maximum of an array

- Write a function `void minmax(int a[], int n, int *min, int *max)` that finds the minimum and maximum of the elements in the array `a[]`, which contains `n` elements. The function should return the results through `min` and `max`.
- Write a short program to test the function.

Subject #16: More on Pointers

- Pointers and arrays are very close in C. In fact, the name of an array is a pointer. Therefore, `&arr[0]` is identical with `arr`. An array points to a constant address which cannot be changed. The memory for all the array elements is allocated at compilation time.
- Using *address arithmetic*, we can use a pointer to a certain place in an array to get a pointer to another place in it. For example, to get the fifth element in the array `arr`, we can write `*(arr+4)` just as we can write `arr[4]`.
- The valid pointer operations are
  - assignment of pointers of the same type.
  - assigning zero (or the symbolic constant **NULL**, defined in **<stdio.h>**) to a pointer. This is done for marking only — the address 0 doesn't point anywhere.
  - adding an integer to or subtracting an integer from a pointer (yielding a pointer).
  - subtracting two pointers to members of the same array (resulting in an integer).
  - comparing such two pointers with the relational operators.
  - comparing a pointer to zero.
- We shall give now an example of a function computing the length of a string (not including the '\0', which marks the end of the string), using simple address arithmetic.

```
/* strlen: return length of string s */
int strlen(char *s)
{
    int n ;

    for (n = 0; *s != '\0' ; s++)
        n++ ;
    return n ;
}
```

- Since `s` is a pointer, incrementing it is perfectly legal; `s++` has no effect on the character string in the function that called **strlen**, but merely increments **strlen**'s private copy of the pointer. That means that calls like

```
strlen("hello\n") ;  /* string constant  */
strlen(array) ;      /* char array[40] ; */
strlen(ptr) ;        /* char *ptr ;      */
```

all work.

- This is another possible version of the **strlen** function:

```
/* strlen: return length of string s */
int strlen(char *s)
{
    char *t ;

    for (t = s; *t != '\0' ; t++) ;
    return t-s ;
}
```