Structures, array of structures, array of pointers to structures, sorting

sort an array of structures:

We start with a structure having an array of char and an int fields. We develop a function to initialize such a structure.
We then create an array of such structures. We initialize each of the structures by calling the init function.
We then call a function to disp an array of structures.
We sort them. The comparison is based on the predicate function comp_name which compares the structures based on the array of char field.
Please check the comp_name and disp functions.

Selection sort:

Sorting by itself is a huge field – there is a book on this topic itself. In this example we are using an algorithm called selection sort.
Let us say that the first i elements are already sorted in non-decreasing order of the predicate – the comparison function. We find the smallest element based on the predicate between i + 1 and n – 1 elements. We than swap this element with the ith element. We select the smallest element each time and bring it to the right position.
The algorithm is as follows.

```
- for i ← 0 to n – 2 do // arrange the ith element correctly
// if n – 1 elements are in order the last one has to be in order
        pos ← i   // pos of the smallest
-       for j ← i + 1 to n – 1 do
                if a[j] < a[pos] then // use comparator functions
                        pos ← j
        swap a[i], a[pos] // make the ith element the smallest between i to n -1 elements.
```

Sort using an array of pointers to structures:

Create an array of structures. Populate them.
Create an array of pointers to structures. Initialize the array of pointers to structures such that each element of the array of pointer to structures points to the corresponding element in the array of structures.

Check the code below.

```
init_ptr(x, p, n);

void init_ptr(person_t x[], person_t* p[], int n)
{
	for(int i = 0; i < n; ++i)
	{
		p[i] = &x[i];
	}
}
```

Now, we can display the array of structures by traversing the array of pointers to structures.

Check this code.

```
void disp_all_ptr(person_t* p[], int n)
{
	for(int i = 0; i < n; ++i)
	{
		// p[i]->name
		disp(p[i]);
	}
}
```

We can sort the array structures indirectly through the array of pointers to structures. We will not change the array of structures at all. Whenever the elements are not in order, we swap the pointers in the array of pointers to structures.

Observe the following two functions.

This swap function swaps two pointers.

```
void myswap_ptr(person_t **a, person_t **b)
{
	person_t* temp;
	temp = *a; *a = *b; *b = temp;
}
```

This sort function sorts using selection sort, uses comp as the comparator, accesses the elements through the array of pointers to structures, swaps the pointers in the array of pointer to structures whenever comparator requires swapping.

```
void sort_using_ptr(person_t* p[], int n, int (*comp)(const person_t*, const
person_t*))
```

```
{
	int i; int pos; int j;
	for(i = 0; i < n - 1; ++i)
	{
		// smallest bet ith and n -1th pos
		pos = i;
		for(j = i + 1; j < n; ++j)
		{
			if(comp(p[pos], p[j]))
			{
				pos = j;
			}
		}
		if(i != pos)
		{
			myswap_ptr(&p[i], &p[pos]);
		}
	}
}
```

The input array is never changed. But the elements are accessed through the array of pointers, the elements are in order of the predicate.

```
#include <stdio.h>
#include "1_struct.h"
int main()
{
#if 0
	person_t x;
	init(&x, "sir mv", 102);
	disp(&x);
#endif
#if 0
	person_t x[20];
	int n = 5;
	init(&x[0], "MV", 102);
	init(&x[1], "GV", 108);
	init(&x[2], "AN murthy rao", 103);
	init(&x[3], "Ramanujan", 32);
	init(&x[4], "Shankaracharya", 32);
	disp_all(x, n);
	sort(x, n, comp_name);
	disp_all(x, n);
#endif
	person_t x[20];
	person_t* p[20];
	int n = 5;
	init(&x[0], "MV", 102);
	init(&x[1], "GV", 108);
```

```
	init(&x[2], "AN murthy rao", 103);
	init(&x[3], "Ramanujan", 32);
	init(&x[4], "Shankaracharya", 32);

	init_ptr(x, p, n);

	disp_all_ptr(p, n); printf("\n");
	disp_all(x, n); printf("\n");
	sort_using_ptr(p, n, comp_name);
	disp_all_ptr(p, n); printf("\n");
	disp_all(x, n); printf("\n");

}



#ifndef PERSON_H
#define PERSON_H
struct person
{
	char name[20];
	int age;
};
typedef struct person person_t;
void init(person_t* ptr_person, char* name, int age);
void disp(const person_t* ptr_person);
void disp_all(const person_t* x, int n);
int comp_name(const person_t* lhs, const person_t* rhs);
void myswap( person_t* lhs,  person_t* rhs);
void sort(person_t x[], int n, int (*comp)(const person_t*, const person_t*));

void init_ptr(person_t x[], person_t* p[], int n);
void disp_all_ptr( person_t* p[], int n);
void sort_using_ptr(person_t* p[], int n, int (*comp)(const person_t*, const
person_t*));
void myswap_ptr(person_t **a, person_t **b);
#endif

#include <stdio.h>
#include <string.h>
#include "1_struct.h"

void init(person_t* ptr_person, char* name, int age)
{
	strcpy(ptr_person->name, name);
	ptr_person->age = age;
}
void disp(const person_t* ptr_person)
{
	printf("%s %d\n", ptr_person->name, ptr_person->age);
}
int comp_name(const person_t* lhs, const person_t* rhs)
```

```
{
	return strcmp(lhs->name, rhs->name) > 0;
}
void disp_all(const person_t* x, int n)
{
	for(int i = 0; i < n; ++i)
	{
		disp(&x[i]);
	}
}
#if 0
// selection sort:
//	also an exchange sort
//	first i elements are in order; find the smallest amongst i + 1 to n - 1
element
//	bring this to ith position by swapping
void sort(person_t a[], int n, int (*comp)(const person_t*, const person_t*))
{
	int i; int pos; int j;
	for(i = 0; i < n - 1; ++i)
	{
		// smallest bet ith and n -1th pos
		pos = i;
		for(j = i + 1; j < n; ++j)
		{
			if(comp(&a[pos], &a[j]))
			{
				pos = j;
			}
		}
		if(i != pos)
		{
			myswap(&a[i], &a[pos]);
		}
	}
}
void myswap( person_t* lhs,  person_t* rhs)
{
	person_t temp = *lhs; *lhs = *rhs; *rhs = temp;
}
#endif

void init_ptr(person_t x[], person_t* p[], int n)
{
	for(int i = 0; i < n; ++i)
	{
		p[i] = &x[i];
	}
}
void disp_all_ptr(person_t* p[], int n)
{
	for(int i = 0; i < n; ++i)
```

```
	{
		// p[i]->name
		disp(p[i]);
	}
}
void myswap_ptr(person_t **a, person_t **b)
{
	person_t* temp;
	temp = *a; *a = *b; *b = temp;
}
void sort_using_ptr(person_t* p[], int n, int (*comp)(const person_t*, const
person_t*))
{
	int i; int pos; int j;
	for(i = 0; i < n - 1; ++i)
	{
		// smallest bet ith and n -1th pos
		pos = i;
		for(j = i + 1; j < n; ++j)
		{
			if(comp(p[pos], p[j]))
			{
				pos = j;
			}
		}
		if(i != pos)
		{
			myswap_ptr(&p[i], &p[pos]);
		}
	}
}
```

You may display the original array to make out that the original array is untouched.

Pointers demystified again:

```
#include <stdio.h>
void f1(int x)
{
	x = 20;
}

void f2(int* p)
{
	int x = 20;
	p = &x;
}

void f3(int **qq)
{
	printf("f3 called\n");
```

```
	int x = 20;
	//*qq = &x; // creates  a dangling pointer
	**qq = x;

}
void myswap(int *a, int *b)
{
	int *temp = a; a = b; b = temp;
}

void myswap1(int **mm, int **nn)
{
	int temp = **mm; **mm = **nn; **nn = temp;
}
void myswap2(int **mm, int **nn)
{
	int* temp = *mm; *mm = *nn; *nn = temp;
}
int main()
{
	int a = 10; int b = 20;
	int *p = &a; int *q = &b;
	printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
	myswap2(&p, &q);
	printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
#if 0
	int a = 10; int b = 20;
	int *p = &a; int *q = &b;
	printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
	myswap1(&p, &q);
	printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
#endif
#if 0
	int a = 10; int b = 20;
	int *p = &a; int *q = &b;
	myswap(p, q);
	printf("%d %d %d %d\n", a, b, *p, *q);
#endif
#if 0
	int a = 10;
	f1(a);
	printf("a : %d\n", a);

	f2(&a);
	printf("a : %d\n", a);

	//f3(& &a); // error

	int *p = &a;
	int **pp = &p;
	f3(pp);
	printf("a : %d\n", a); // 20
```

```
#endif

}
```

Example 1:

```
	int a = 10;
	f1(a);
	printf("a : %d\n", a); // 10
```

We are passing a copy of a as argument. So, no matter what f1 does, the argument a can never be changed.

Example 2:

```
void f2(int* p)
{
	int x = 20;
	p = &x;
}


f2(&a);
printf("a : %d\n", a); // 20
```

p points to a. The function f2 does not change what p points to. p is changed to point to a local variable. Both p and x will die at the end of the function call. The variable a remains unchanged.

Example 3:

```
void f3(int **qq)
{
	printf("f3 called\n");
	int x = 20;
	**qq = x;

}

	int *p = &a;
	int **pp = &p;
	f3(pp);
	printf("a : %d\n", a); // 20
```

p points to a. pp points to p. qq gets a copy of pp on the call to the function f3. The variable qq is a pointer to pointer to int. **qq is an int. In this case, it is same as a. The assignment **qq = x; will change a.

Example 4:

```
void myswap(int *a, int *b)
{
	int *temp = a; a = b; b = temp;
}


	int a = 10; int b = 20;
	int *p = &a; int *q = &b;
	myswap(p, q);
	printf("%d %d %d %d\n", a, b, *p, *q);
```

p points to a and q points b. The function myswap gets two pointers. The pointers are swapped and not what they point to. The variables a and b remain unchanged.

Example 5:

```
void myswap1(int **mm, int **nn)
{
	int temp = **mm; **mm = **nn; **nn = temp;
}


	int a = 10; int b = 20;
	int *p = &a; int *q = &b;
	printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
	myswap1(&p, &q);
	printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
```

myswap1 receives pointer to pointer. But it swaps two integers. Observe the type of the local variable temp. So a and b are swapped. The variables p and q remain unchanged.

Example 6:

```
void myswap2(int **mm, int **nn)
{
	int* temp = *mm; *mm = *nn; *nn = temp;
}


	int a = 10; int b = 20;
	int *p = &a; int *q = &b;
```

```
    printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
    myswap2(&p, &q);
    printf("%d %d %d %d %p %p\n", a, b, *p, *q, p, q);
```

The function myswap2 swaps pointers to int – swaps p and q.

You may write the activation records to understand how these work.