C programming Language

Chapter 4:

1. Structures

What is a Structure?

- How can we define something like an array, where each element contains all the information about student: name, age, telephone, address, etc.
- What is the type of this sort of array?
- We can define multiple data types, but for some students we have to define multiple parallel arrays - for each type.
- It turns out that what we need is a **structure**.
- A structure is a data aggregate, consisting of elements of different types (these elements are called members or fields).

Structure Usage

- Structures are widely used. For instance:

Person
fname
lname
address
salary

Process
priority
total time
status
id

File
name
device
sector
Block
open mode

Structure Declaration

- As mentioned above, structure is a type defined by the programmer.
- In order to use a structure, we should declare it. Then we can use it as any other primitive type.
- Declaration:

```
struct structure_name
{
   members;
};
```

- For the declaration, no memory is allocated.
- Syntax comment: don't forget the semicolon after the struct block (like for any declaration).

Structure Declaration

```
struct Student
{
  char name[20];
  int id;
};
```

- In this example we declared a new type named Student and provided information on its structure.
- Now we can define new variables of Student type, exactly as we define variables of primitive types:

```
int          x,     y,     *pInt = NULL;
struct Student  std1,std2,*pStudent = NULL;
```

Each one of the variables (std1, std2) contains 2 fields.

Structure Initialization

```
struct Student
{
 char name[20];
 int id;
};
```

- Initialization can be achieved via definition:

```
struct Student std = {"arie", 222};
```

- The initialization parameter list corresponds to the structure's (member) fields.

Structure Member Operator .

- We can't refer to the structure as a whole. However, the structure member operator . is used to refer to each one of the structure's fields.
- Reference a field via the variable std1 using operator . :

```
std1.name
std1.id
```

- Reference a field via the pointer pStudent:

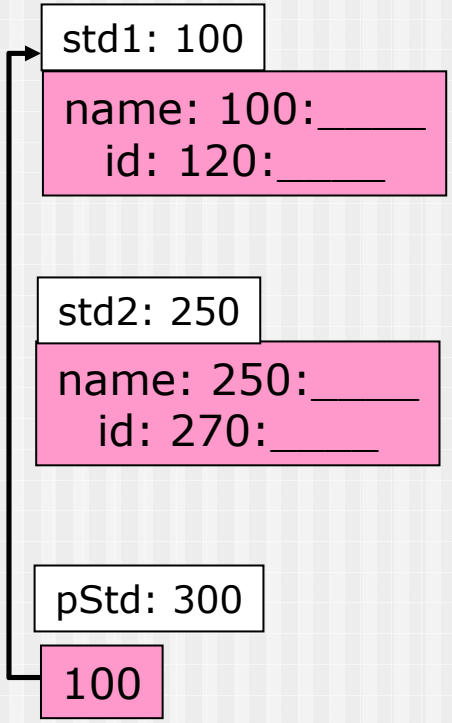
```
(*pStudent).name
(*pStudent).id
```

- <u>Question</u> : Why are the parentheses necessary?
- Another, more aesthetic way for referencing a field via a pointer is using the (shorthand for (*pointer).) operator -> (arrow):

```
pStudent->name
pStudent->id
```

Example

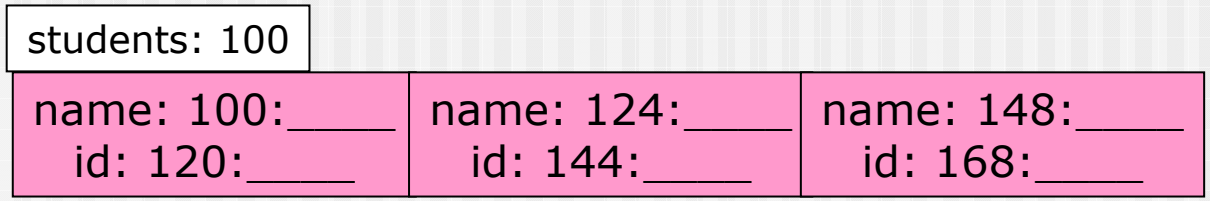
```
struct Student
{
    char name[20];
    int id;
};
void main()
{
    struct Student std1, std2, *pStd;
    printf("enter name and id of std1 and
       std2\n");
    scanf("%s %d", std1.name, &(std1.id));
    scanf("%s %d", std2.name, &(std2.id));
    pStd = &std1;
    printf("%s %d", pStd->name, pStd->id);
}
```



Array of Structures

- Definition: `struct struct_name array_name[num_of_items]`
- Example:

```
struct Student students[3];
students[2].id = 222;
(*(students+2)).id = 333;
(students+2)->id = 444;
```



- Initialization via definition:

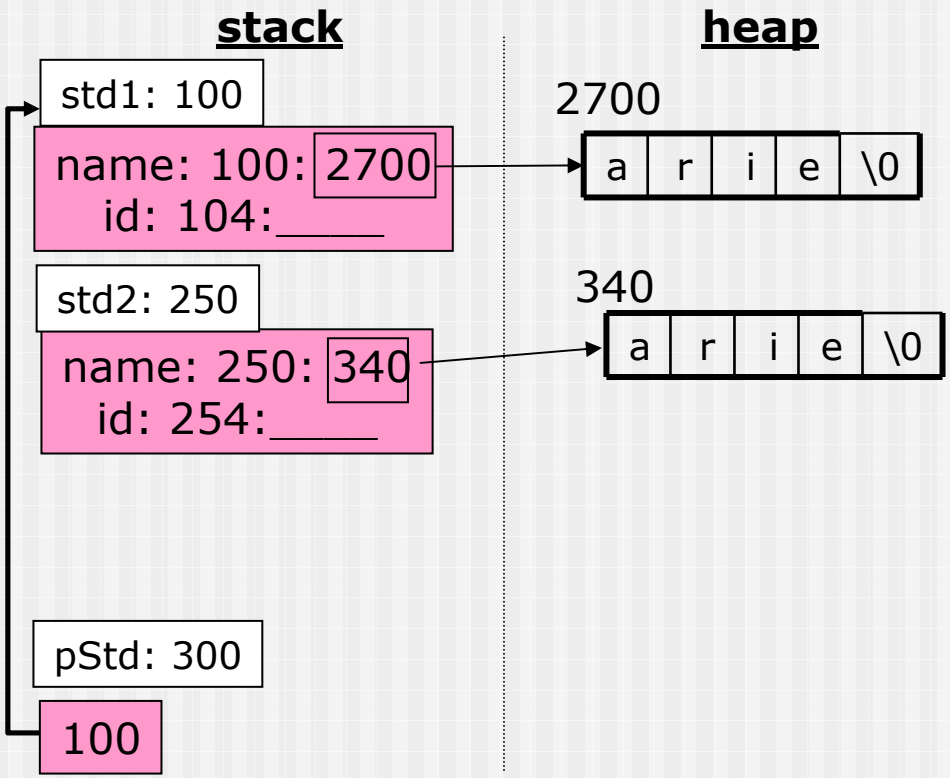
```
struct Student students[3] = {     {"ami", 222},
                                   {"dana", 333},
                                   {"haia", 444}      };
```

- If internal braces are missing, what will happen???

Pointer as Field

- A field may be a pointer also. We can refer to it by operator *(). or by ->, and then assign it to any address.
- For example:

```
typedef struct
{
    char *name;
    int id;
} Student;
void main()
{
    Student std1, std2, *pStd;
    std2.name = "arie";
    pStd = &std1;
    pStd->name =
        (char *) malloc(5);
    strcpy(pStd->name, "arie");
}
```



Structure as Parameter to Function

- A structure can be passed to a function by value or by address.
- Upon passing it by value, it is copied to the function. Any changes in the copy will not affect the original.
- Upon passing it by address, only its address is passed. Any changes on the pointed value will affect the original.
- For example:

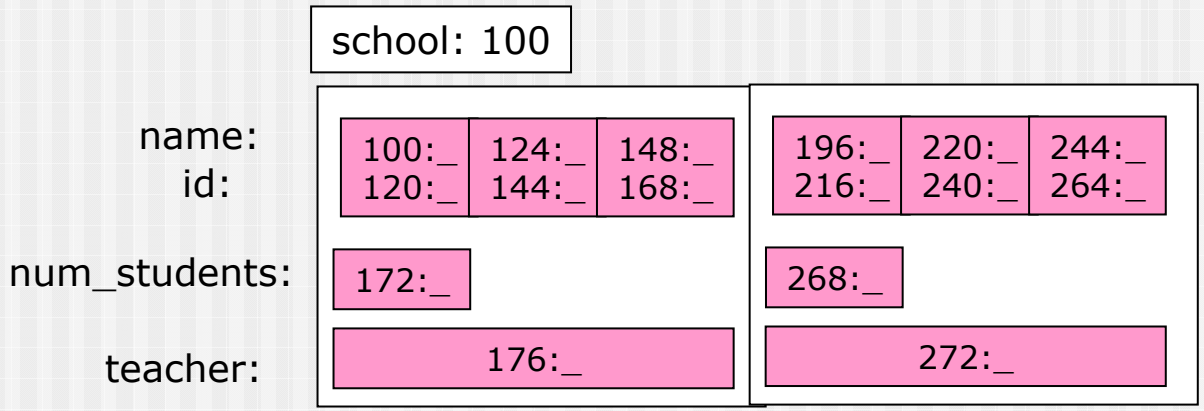
```
typedef struct
{
    char name[20];
    int id;
} Student;
void main()
{
    Student std1;
    input_student(&std1);
    output_student(std1);
}
```

```
void input_student(Student *std)
{
    scanf("%s", std->name);
    scanf("%d", &(std->id));
}
void output_student(Student std)
{
    printf("%s", std.name);
    printf("%d", std.id);
}
```

Nested Structures

- A member of a structure may be itself a structure.
- For example, a student could be a member of a class.

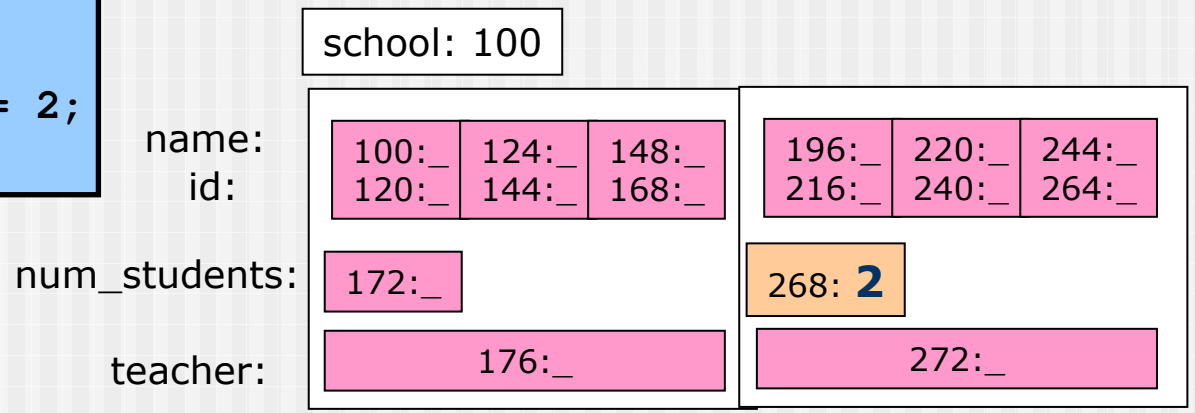
```
typedef struct
{
    char name[20];
    int id;
} Student;
typedef struct
{
   Student students[3];
   int num_students;
   char teacher[20];
} Class;
void main()
{
    Class school[2];
}
```



Nested Structures

Access to the fields of Class can be done via the structure member operator . or via -> as well as the access to Student's fields.

```
void main()
{
   Class school[2];
   school[1].num_students = 2;
}
```



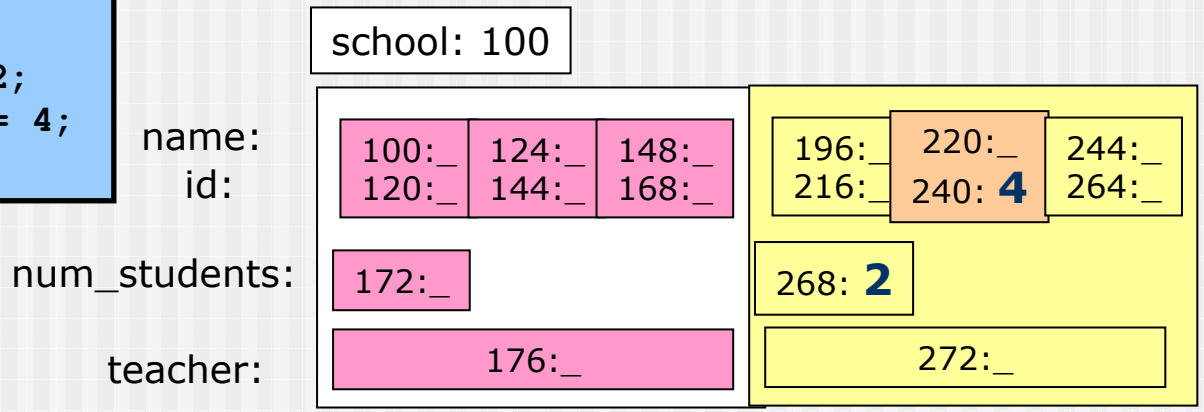
Nested Structures

Access to the fields of Class can be done via the structure member operator . or via -> as well as the access to Student's fields.

```
void main()
{
  Class school[2];
  school[1].num_students = 2;
  school[1].students[1].id = 4;
}
```

What is its type?

Class



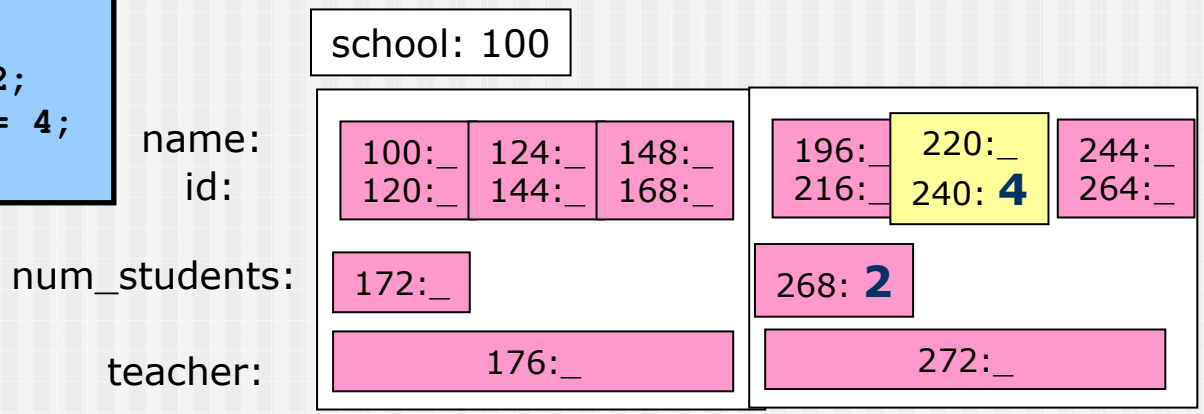
Nested Structures

Access to the fields of Class can be done via the structure member operator . or via -> as well as the access to Student's fields.

```
void main()
{
  Class school[2];
  school[1].num_students = 2;
  school[1].students[1].id = 4;
}
```

What is its type?

Student



Nested Structures

- Type recognition is very important for passing a structure as argument to a function.
- For instance: passing the classes to input and output functions.

```
void input_class(Class *cls)
{
   int i;
   scanf("%s", cls->teacher);
   scanf("%d", &(cls->num_students));
   for(i=0; i<cls->num_students; i++)
     input_student(&(cls->students[i]));
}
void output_class(Class cls)
{
    int i;
    puts(cls.teacher);
    for(i=0; i<cls.num_students; i++)
      output_student(cls.students[i]);
}
```

```
void main()
{
    Class school[2];
    int i;
    for(i=0; i<2; i++)
      input_class(&school[i]);
    //input_class(school+i);
    for(i=0; i<2; i++)
      output_class(school[i]);
}
```

Nested Structures

- Example of dynamic allocation of structures:

```
typedef struct
{
    char name[20];
    int id;
} Student;
typedef struct
{
   Student *students;
   int num_students;
   char teacher[20];
} Class;
typedef struct
{
   Class *classes;
   int num_classes;
} Education;
```

```
void main()
{
 int i, j;
 Education edu;
 puts("how many classes?");
 scanf("%d", &edu.num_classes);
 edu.classes = (Class *)
     malloc(edu.num_classes*sizeof(Class));
 for(i=0; i<edu.num_classes; i++)
 {
  puts("enter num of students");
  scanf("%d", &(edu.classes[i].num_students));
  edu.classes[i].students = (Student *) malloc
   (edu.classes[i].num_students*sizeof(Student));
  for(j=0; j<edu.classes[i].num_students; j++)
  input_student(&(edu.classes[i].students[j]));
//input_student(((edu.classes)+i)->students+j);
 }
}
```