Structures in C

Heterogeneous Data

- A piece of information that describe the characteristics of an entity (object, person, company, etc.) are grouped together in a user-defined data type called struct (stands for structure).

- Imagine writing a program dealing with student data

- Each student record would consist of
  - last name
  - first name
  - an ID number
  - Gender
  - phone Number
  - Etc.

- It allows groups of data to be passed around in a single variable

  - It is also know as an abstract data type

  - This is the first step of data abstraction, which is encapsulation of a group of related data in one entity.

Defining Structure Types in C

- The syntax for a struct **definition** (not declaration) is as follows:

  struct [tag_name] {member_declaration_list};

- In other words:

  ```
  struct Struct_Name
  {
      <dataType1> identifier1;
      <dataType2> identifier2;
      ...
      <dataTypeN> identifierN;
  };
  ```
  Don't forget the semi-colon. This is part of the syntax!

- The new type is called **Struct_Name** and contains **data members** (or **member variables**) named **identifier1**, **identifier2**, etc.

Defining Structure Types in C

- To illustrate, here's the definition of a struct to hold a student record

```
struct studentRecord
{
    char firstName[30];
    char lastName[30];
    int idNum;
    char gender;
    char phoneNumber[20];
};
```

The studentRecord struct contains seven data members.

- This is only a definition; it does not allocate space for a variable structure type.

Using Structure Types in C

- Struct definitions generally go either outside the functions and at the top of the program or go into a header file:

```
#include <stdio.h>

struct Name
{
    char firstName [30];
    char middleInitial;
    char lastName [30];
};

void PrintName( struct Name name );

int main()
{
    struct Name myName;
    strcut Name herName;
    struct Name x, y, z;
    ...
}
```

- As function prototype and main function show, you can declare **struct** variables like any other primary data types:

Accessing Struct Data

- To access the members of a struct, use a period ("**.**") which is called the ***member access operator***, also called the ***dot operator***
  - In ENGG 233 we have seen this with strings and vectors:
- The syntax is as follows — Member access operator

> `<struct variable name>.<member name>`

- Notice that we do not include the struct type, but rather the ***name of a variable*** whose type is a struct.

- Example:

  struct Name theName;

  theName.middleInitial = 'S';

- How the firstName and the lastName must be assigned?

Example Accessing Struct Data

```c
struct Complex_number{
    double real;
    double imag;
};

int main(void){
  struct Complex_number cplx  ;

  printf("Enter the real part of a complex number: ");
  scanf("%lf", &cplx.real);

  printf("Enter the imaginary part of a complex number: ");
  scanf("%lf", &cplx.imag);

  printf("The real part is: %f and the imaginary part is: %f\n",
                                  cplx.real, cplx.imag);
  return 0;
}
```
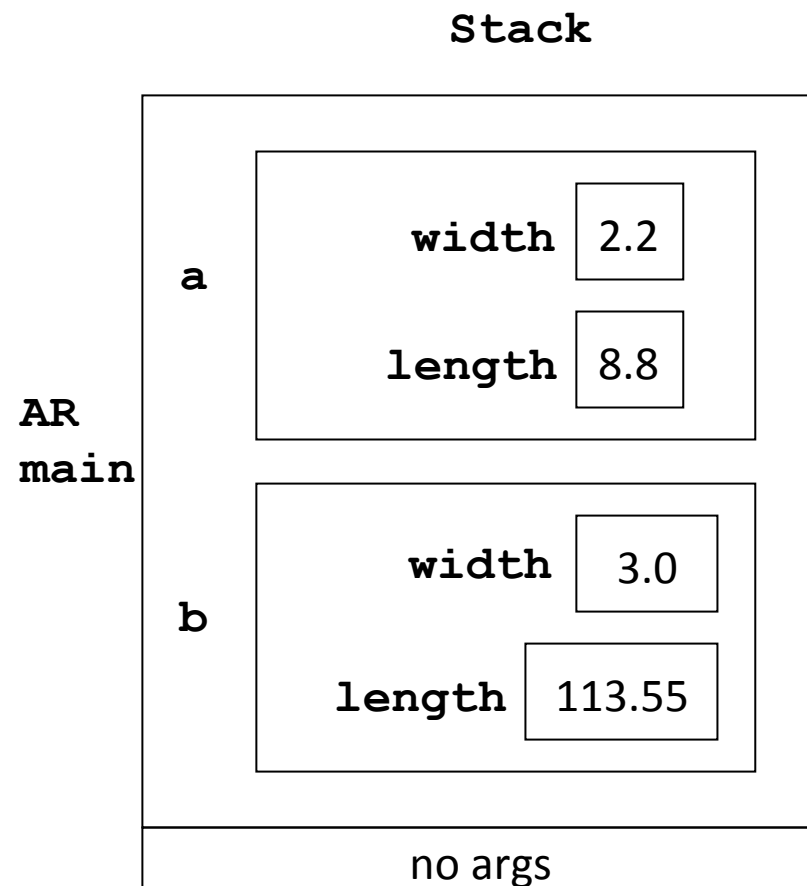
Structure Instance on the Memory

- Every instance of a structure type has its own set of member variables on the memory:

```
struct Rectangle {
    double width;
    double length;
};

int main(void) {
    struct Rectangle a;
    struct Rectangle b;
    a.width = 2.2;
    a.length = 8.8;
    b.width = 3.0;
    b.length = 113.55;

    // point 1

    return 0;
}
```

Point 1

Stack

a
  width 2.2
  length 8.8

AR main

b
  width 3.0
  length 113.55

no args

Pointer to Structure Type

Pointers to User-defined Data Types

- A pointer in C can point to any addressable memory location, including user-defined data types such as structures.

- Consider the following definition of structure Point:

```
struct Point {
    double x;      // Point's x-coordinate
    double y;    // Point's y-coordinate
    char label;
};
```

- Consider the following statements:

```
struct Point centre;
struct Point *pst; // pointer to struct Point
pst = &centre;       // pst pointing to centre
```

Pointers to User-defined Data Types

- To access the structure's data members via a pointer, you need to dereference the pointer first and then access the member:

  ```
  (*pst).x = 57.66;
  (*pst).y = 99.00;
  (*pst).label = 'A';
  ```

  Note: The parentheses are required because dot operator has higher precedence

- A better and possibly easier option is to use an arrow operator **->** (a dash followed by greater sign)

  ```
  pst -> x = 57.66;
  pst -> y = 99.00;
  pst -> label = 'A';
  ```

Pointers to User-defined Data Types

- The following statements produce the same output:

```
printf("label = %c  x = %f  y= %f.",
                    centre.label, centre.x, centre.y);


printf("label = %c  x = %f  y= %f.",
                      pst -> label, pst->x, pst ->y);


printf("label = %c  x = %f  y= %f.",
                      (*pst).label,(*pst).x, (*pst).y);
```

- You may also assign a pointer to any members of a C structure:

```
double *pd;
char *pc;
pd = &centre.x;
pc = &centre.label;
printf("label = %c  x = %f.", *pc, *pd);
```

Passing Structure as Arguments of a Function

Structure Types as a Function Argument

- Like any other data types in C, a structure data type can be passed to a function either **by value**, or by its **address**.

- Consider the following definition of structure type Staff and the given main function:

```c
struct Staff {
    char Fname [30];
    char Lname [30];
    int age;
    double salary;
};

// prototype of function print goes here
int main(){
    struct Staff st = { "Judy", " Moor", 45, 3000.00};
    print (&st);
    return 0;
}
```

- Further details on initializing structures will be discussed later in this set of slides.

- Now lets write the implementation and the prototype of function print.

Structure Types as a Function Argument

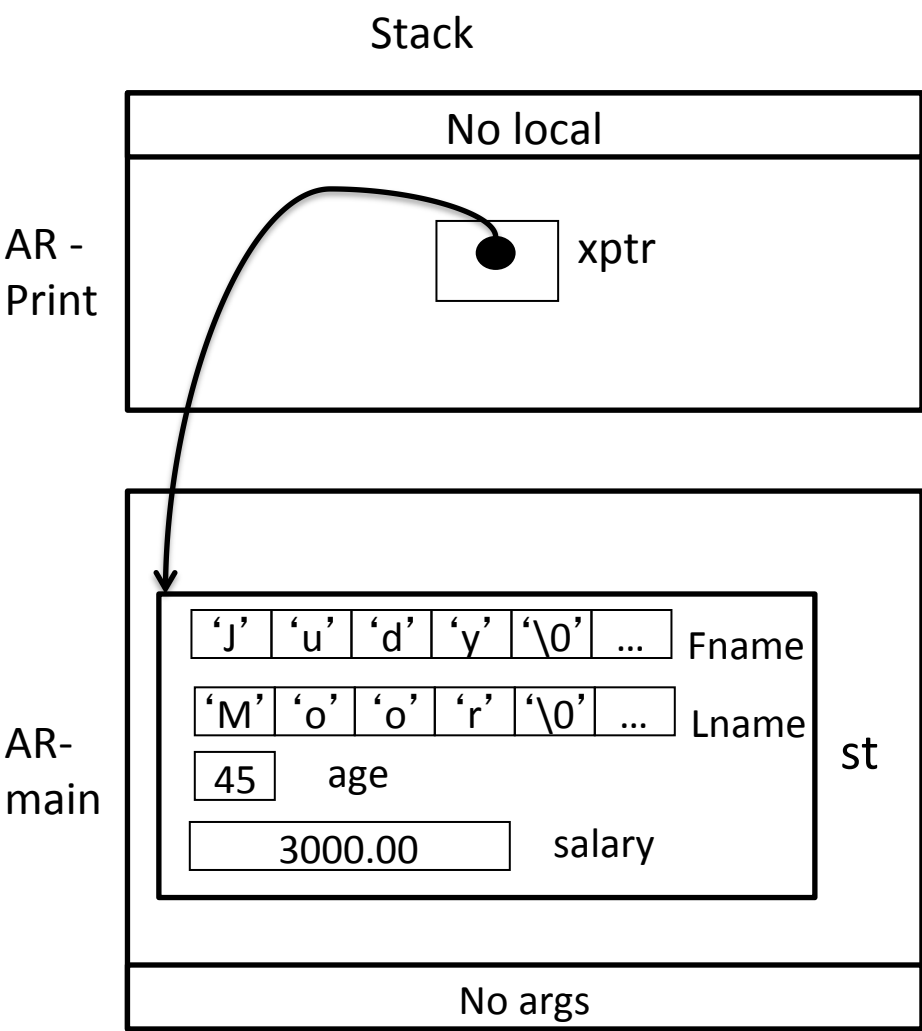- Here is the prototype of the function print

```
void print ( const struct Staff *xptr);
```

- And, here is the implementation of function print:

```
void print ( const struct Staff *xptr)
{
  // point one
  printf("%s", xptr -> Fname);
  printf("%s", xptr -> Fname
  printf("%d", xptr -> age);
  printf("%f", xptr -> salary);
  double *yptr = &(xptr -> salary);
  *yptr = 4000.00;
  // point two
}
```
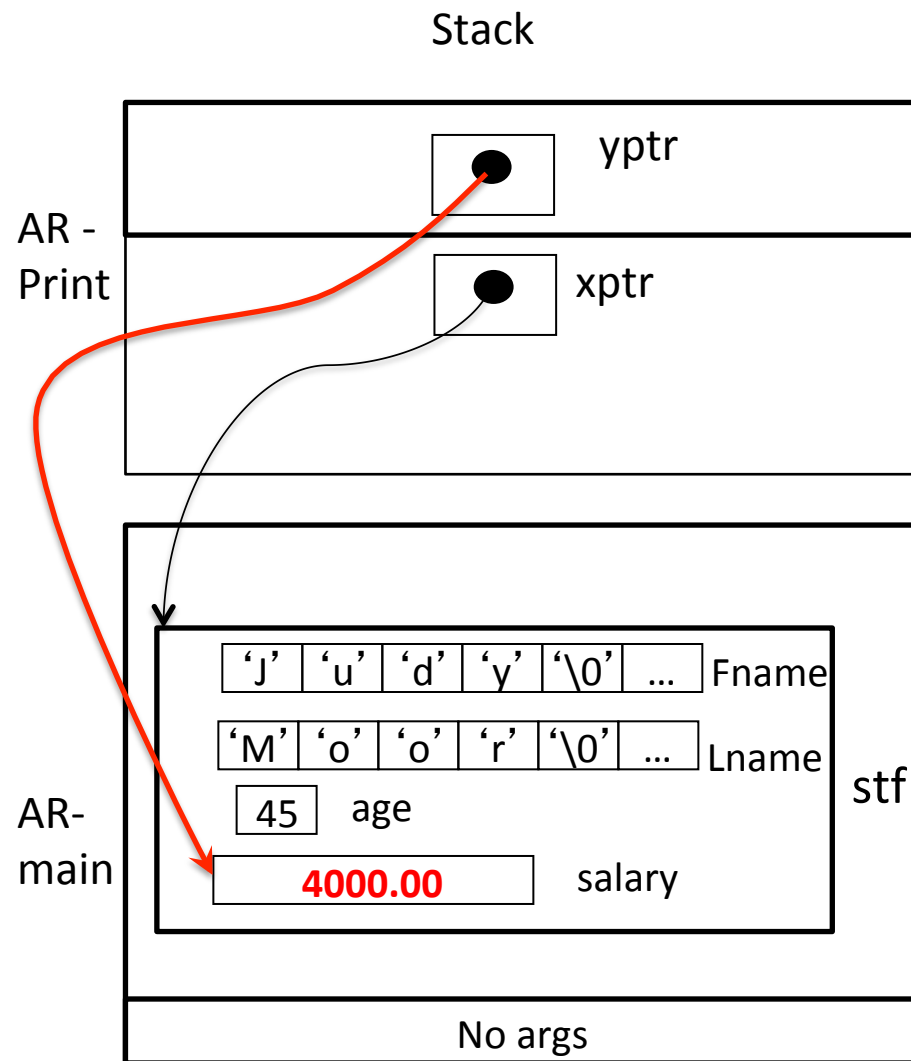
- In general passing by address is preferred

AR for Point One in function print

Stack

AR -
Print

No local	
	xptr

AR-
main

'J' 'u' 'd' 'y' '\0' ...	Fname
'M' 'o' 'o' 'r' '\0' ...	Lname
45	age
3000.00	salary

st

No args

AR at Point Two function print

Stack

Copying Structure Instances

- You are allowed to assign structure variables of the ***same type***:

```
struct Rectangle box1, box2;
struct Point centre, corner;


box1.width = 10;
box1.length = pow( box1.width, 3 );


// copy the first box object
box2 = box1;
// WARNING: BAD STATEMENTS
box2 = centre;      // Type mismatch: Different types!
corner = box1;      // Type mismatch: Different types!
```

Functions Returning Structure Data Type

- C function can also return a structure data type.

- The following function returns a point object that represents a point in the middle of two points:

```
struct Point mid_point(const struct Point *a,
                        const struct Point *b)
{
    struct Point middle;
    middle.x = (a -> x + b ->x) / 2;
    middle.y = (a -> y + b ->y) / 2;
    middle.lable = 'M';
    return middle;
}
```
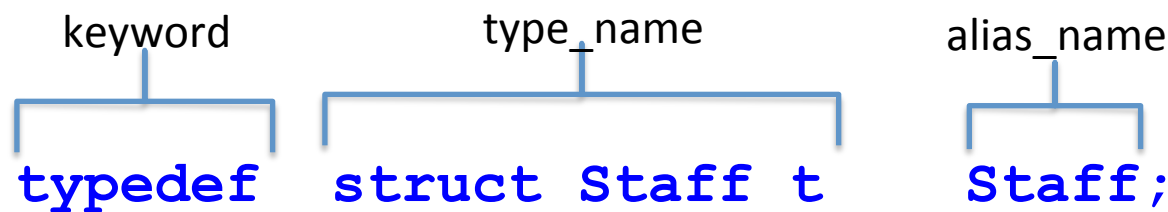
Using **typedef**

Using Typedef with Structure Types

- In C, you can use **typedef** to simplify the declaration of structure types.
- General format is:

typedef   type_name   alias_name;


```
struct Staff_t {
    char Fname[SIZE];          // First Name
    char Lname[SIZE];          // Last Name
    int age;
    double salary;
};
```


keyword   type_name   alias_name

typedef   struct Staff_t   Staff;

Using Typedef with Structure Types

- Now, you can use Both Staff and struct Staff_t as a type to declare your variables:

  ```
  Staff x, y, z;

  Staff *p;

  struct Staff_t  w, *p2;
  ```

- You can also use typedef with the definition of the structure:

  ```
  typedef struct Staff_t {
      char  Fname[SIZE];        // First Name

      char Lname[SIZE];         // Last Name

      int age;

      double salary;
  } Staff;
  ```

Initializing Structure Instances

- You can initialize a structure type as follows:
  ```
  int main()
  {
      struct Name x, y;
      struct Name z = {"Bill", 'S', "Horstmann"};
      return 0;
  }
  ```

- The initialization of structure type follows the usual initialization rules:

  – automatic storage class: member will have indeterminate initial.

  – static storage: initial values will be zero, and pointers will be null pointer.

- Notice that initialization is allowed only at the time declaration. The following assignment is NOT allowed;
  ```
  struct Name w;
  W = {"Judy", 'E', "Moor"};   // ERROE
  ```

Initializing Structures Instances

- You may also specify fewer initializer than the number of data member. In this case any remaining member are initialized to zero:

- Example:

```
typedef struct Point {
    double x_coordinate;
    double y_coordinate;
    char label;
} Point;


Point centre = { 100 };
```

- X_coordinate is initialized to 100 and two other members y_coordinate and label to zero.

Initializing Structures

- C99 standard also allows us to explicitly initialize certain members of a structure type:

  ```
  Point center = {.x = 100.05, .y = 205.00};
  ```

Note: you cannot initialize the structure data members at the time of definition. The following definition and initialization is **NOT** allowed:

```
struct Point {
   double x = 100;
   double y = 100;
   char label = ' ';
   };
```

Structures and Arrays

```c
#define SIZE 20

typedef struct Staff {
            char Fname[SIZE];        // First Name
             char Lname[SIZE];        // Last Name
            int age;
            double salary;
    } Staff;
```

- Array Declaration

  ```c
  Staff a[100];
  ```

  – Declares an array with 100 elements of struct variable.
  – All the rules that applies to arrays of simple data types, applies to an array of structure too.
    - Use index number to access each element:

      ```c
      printf ("%d" a[i].age);// prints the value of age
      ```

      The name of an array of structure holds the address of the first element.

Initialization of Array of Structures

- You can initialize array of structures, as follows:

```
Staff arr [3] = {
            {"Judy", "Moore", 18, 2450.00},
            {"Jack", "Wong", 19, 3450.00},
            {"Tim", "Lewis", 21, 1450.00}
};
```

- To display the second character in the first element of array **arr**, we can use the following syntax. **What is the output**?

```
printf("%c", arr[1].Lname[0]);
```

Reading Data into Members of Structure

- You can use scanf to read values from keyboard into the data members of an struct variable:

```
Staff x;

printf ("Enter your first name: );
scanf("%s", x.Fname);

printf ("Enter your last name: );
scanf("%s", x.Lname);

printf ("Enter your age: );
scanf("%d", &x.age);

printf ("Enter your salary: );
scanf("%lf", &x.salary);
```

- Notice the & (address operator) in front of the numeric variable names, but not string variables.

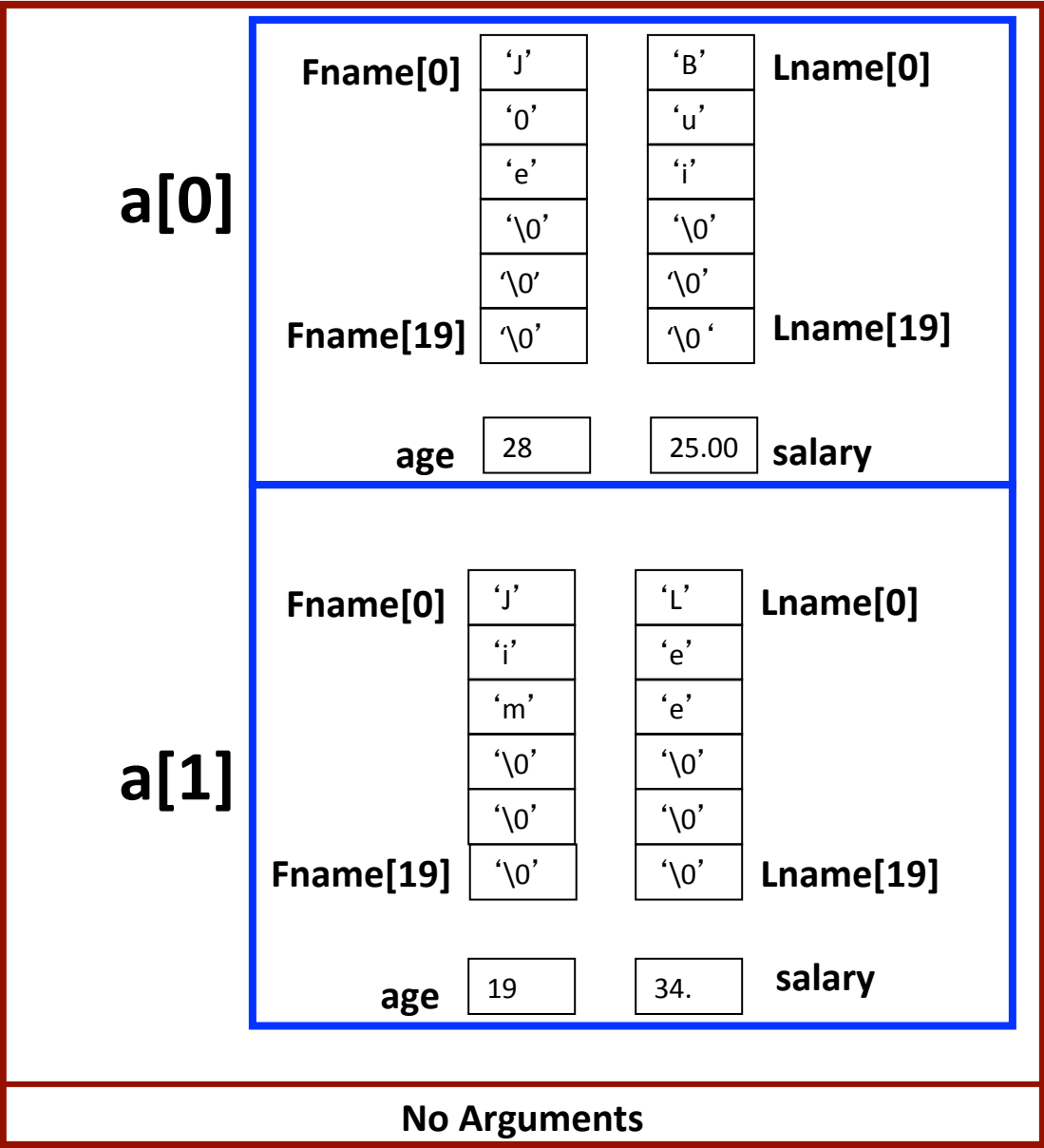## Array of Structure on the Memory

```
int main(){

    Staff a[2] =
                    {
                        {"Joe", "Bui", 28, 25},
                        {"Jim", "Lee", 19, 34}

                    };


    // point 1
     return o;
 }
```

- See the AR diagram for this program in the next slide

AR Diagram – Point 1 in main

Stack

AR main

a[0]

Fname[0]	'J'	'B'	Lname[0]
	'o'	'u'	
	'e'	'i'	
	'\0'	'\0'	
	'\0'	'\0'	
Fname[19]	'\0'	'\0 '	Lname[19]

age 28   25.00 salary

a[1]

Fname[0]	'J'	'L'	Lname[0]
	'i'	'e'	
	'm'	'e'	
	'\0'	'\0'	
	'\0'	'\0'	
Fname[19]	'\0'	'\0'	Lname[19]

age 19   34. **salary**

No Arguments

M. Moussavi

30

Nested Structure

Nested Structures

- A structure may have another structure object as one of its members.

- Consider the definitions of struct Date and struct Person in a header files called date.h, and person.h.

- Notice how an instance or a pointer of type Date is defined in struct Person.

Header file date.h	Header file person.h
| ```
#ifndef nested_Structure_date_h
#define nested_Structure_date_h

    typedef struct Date
    {
       int day,  month,  year;
     } Date;


#endif
``` | ```
#ifndef nested_Structure_person_h
#define nested_Structure_person_h
#include "date.h"

typedef struct Person{
   char name[30];
   Date birthday;
   Date* graduation_day;
} Person;


#endif
``` |

Using a Nested Structure

```c
#include <stdio.h>
#include "person.h"
#include "date.h"

int main()
{

    Person p1 = {"Jack", 12, 12, 1983, NULL};
    Date d = {23, 11, 1972};
    Person p2 = {"Judy", 8, 10, 1982, NULL};
    p2.graduation_day = &d;

    printf("Person's name is: %s, birthday: %d-%d-%d, and graduation date:"
            " %d-%d-%d\n", p2.name, p2.birthday.day, p2.birthday.month,
        p2.birthday.year, p2.graduation_day->day, p2.graduation_day->month,
        p2.graduation_day->year);

    return 0;
}
```

Program's Output:

Person's name is: Judy, birthday: 8-10-1982, and graduation date: 23-11-1972

A structure that contains an array of another structure

Header file: department.h	Implementation file: main.c
| ```c
#ifndef nested_Structure_department_h
#define nested_Structure_department_h

#include "person.h"

#define STAFF_SIZE 100
#define NAME_LENGTH 30

typedef struct Department
{

  char dept_name[NAME_LENGTH];
  Person staff_list[STAFF_SIZE];

} Department;

#endif
``` | ```c
#include <stdio.h>
#include <string.h>
#include "date.h"
#include "department.h"
// here is a simplified version of a possible main
// function that uses some of these structures
int main(void)
{
  Department dept;
  Date d;
  dept.staff_list[0].graduation_day = &d
  dept.staff_list[0].graduation_day->day = 20;

  dept.staff_list[0].birthday.day= 23;
  strcpy (dept.staff_list[0].name, "Julia");
  strcpy (dept.dept_name, "ABC Engineers");
  return 0;
}
``` |