CSE1142 – Structure and Union Types in C

Sanem Arslan Yılmaz

Some of the slides are from:
CMPE150 – Boğaziçi University
Deitel & Associates

Agenda

- Struct
  - Definitions & Syntax
  - Initialization
- Defining `struct` Variables
- Initializing `struct` Variables
- `struct` as a Parameter
- Array of `struct`
- `union` Syntax and Usage
- `typedef` Syntax and Usage
- Examples

Motivation

- When you want to store several properties about an entity, you need to define several variables.
  - Eg: If you want to store the name, ID, department, class, and GPA of a student, you need to define 5 variables as follows:

```
char name[41];
long int id;
char department[16];
short int class;
float GPA;
```

- Together with all other variables you need to define, it is difficult to keep track of this info; there is nothing that shows the association between these variables.

- Structures help you to better organize your code. (It will be more obvious when we start discussing arrays of structures.)

Motivation – cont.

- Structures
  - Collections of related variables (aggregates) under one name
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files
  - Combined with pointers, can create linked lists, stacks, queues, and trees
  - They are derived data types—they're constructed using objects of other types.

Structure Syntax

- Syntax:

```
struct structure_name {
   field_definition(s)
   . . .
} variable_list;
```

  - `struct` keyword introduces the structure definition.
  - `structure_name` is an identifier and used with `struct` keyword to declare variables of the structure type.
    - It is optional, but without it you will not be able to refer to this type once again (for new variables and parameter definitions, etc.)
  - Variables declared within the braces of the structure definition are the structure's **members**.
    - There should be at least one field, but of course it should typically be two or more.
  - `variable_list` is optional. You may define the variables later on.

Example 1 – Student information

- Gather all information about student #1 and student #2 under two variables, `stu1` and `stu2`.

```
struct stu_info {
   char name[41];
   long int id;
   char dept[16];
   short int class;
   float gpa;
} stu1, stu2;
```

Structure Usage

- Structure is a user-defined type (like enumerated types).

- Note that `stu_info` is the name of the structure type, but `stu1` is the name of a variable of that type.

- Analogy:
  - `stu_info` is the name of a type, just like `int`.
  - `stu1` is the name of a variable of the given type.

- Therefore, you cannot assign any value to `stu_info`; it is the type, not the variable.

Valid Operations

- Assigning a structure to a structure of the *same* type
- Taking the address (**&**) of a structure
- Accessing the members of a structure
- Using the `sizeof` operator to determine the size of a structure

Accessing Fields of a Structure

- Two operators are used to access members of structures:
  - structure member operator (.) — also called the dot operator
  - structure pointer operator (->) — also called the arrow operator

- You may access a field of a structure by using a period (.) as follows:

  `structure_variable_name.field_name`

- You cannot use a field name alone. A field name makes sense only in the context of a structure variable
  - i.e., `id=123;` is wrong (`id` is not defined)

  `stu1.id=123;` is correct

Updating Fields of a struct

- Note that **stu1** and **stu2** are two separate variables, each with 5 fields.

	name	id	dept	class	gpa
stu1					

	name	id	dept	class	gpa
stu2					

- You can refer to individual fields of **stu1** and **stu2** as follows:
  - ❑ **strcpy(stu1.name, "Ahmet");   stu1.id=123;**
  - ❑ **strcpy(stu2.name, "Ayse");   stu2.id=456;**
- Then, the variables will be as follows:

	name	id	dept	class	gpa
stu1	"Ahmet"	123			

	name	id	dept	class	gpa
stu2	"Ayse"	456			

Defining struct Variables

- There are two ways for defining a struct variable:
  - Define the variable while defining the structure.

```
struct {
   char name[21];
   int credit;
 }cse1142;  /*Define both type and variables*/
```

    - Note that the optional struct name has been omitted.
  - Define the variable after the structure has been defined.

```
struct course_type {
    char name[21];
    int credit;
};  /*Define only the type*/
struct course_type cse1142; /*Define variables*/
```

    - Note that the optional struct name cannot be omitted here since we need the struct name later to define variables of that type.
    - Also, note that you cannot simply say `course_type`; you have to say `struct course_type`.

struct as a Field of Another struct

- You may have a struct that has a field of struct type.
- Example:

```
struct A_type {
   int m, n;
};
struct B_type {
   int f;
   struct A_type g;
} t;
```

  - `t` has two fields: `f` and `g`. `f` is of type `int`, `g` is of type `struct A_type`. Thus, `t` has the following fields (and subfields):

```
t.f
t.g
t.g.m
t.g.n
```

struct as a Field of Another struct (cont.)

- *A structure cannot contain an instance of itself.*

  Example:

```
struct B_type {
   int f;
   struct B_type g;     //ERROR
   struct B_type *bPtr; // OK
} t;
```

- A pointer to struct itself; however, may be included.

- A structure containing a member that’s a pointer to the *same* structure type is referred to as a self-referential structure.

Example 2 – Point in 2D

- Define a type for a point in two-dimensional space.

```
struct point_type {
   int x, y;
};
```

- Define variables A and B of point type.

```
struct point_type A, B;
```

- Assign values to these variables

```
A.x=2; A.y=3;
B.x=1; B.y=2;
```

Example 3 – Triangle and Rectangle

- Define type for a triangle.

```
struct triangle_type {
   struct point_type A, B, C;
};
```

- Define and initialize variable t of triangle type.

```
struct triangle_type t={{1,3},{2,4},{1,6}};
```

- Define type for a rectangle.

```
struct rectangle_type {
   struct point_type A, B, C, D;
};
```

Initializing struct Variables

- You may initialize struct variables during definition (in a way similar to arrays).

```
struct A_type {
    int m, n;
} k={10,2};

struct B_type {
    int f;
    struct A_type g;
} t={5,{6,4}};
```

- The following kind of initialization is wrong.

```
struct A_type {
    int m=10, n=2;
} k;
```

Initializing struct Variables – cont.

- You cannot use the **{…}** format for initialization after initialization (just as in arrays)
- Example:

```
struct A_type {
  int m, n;
} k;
...
k={10,2};
```

is wrong.

Using Structures with Functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

struct as a Parameter

- You may have a struct parameter (just like any other parameter).

```
void func1(struct A_type r)
{   struct A_type s;
    s=r;
    s.m++;
}
```

- You may also have a variable parameter.

```
void func2(struct A_type *p)
{   (*p).m=10; // p->m = 10 is also OK
    (*p).n=3;  // p->n = 3 is also OK
}
```

Example 4 – Function Call

```
struct complex {
   float real;
   float imaginary;
} c={5.2,6.7}, d={3,4};

struct complex add(struct complex n1, struct complex n2){
   struct complex r;
   r.real = n1.real + n2.real;
   r.imaginary = n1.imaginary + n2.imaginary;
   return r;
}
```

Field of a struct pointer

- In the previous slide in `func2()`, instead of `(*p).m`, we could have used `p->m`.
  - The "`->`" operator works only if "`p`" is a pointer to a struct, which has a field named `m`.

- In other words, you cannot say `r->m` in `func1()`.

Example 5 – Struct Pointer

```
void func2(struct A_type *h)
{
   (*h).m=5; /* Equivalent of "h->m=5;"*/
}

int main()
{  struct A_type k={1,2};
   func2(&k);
   printf("%d  %d\n", k.m, k.n);
}
```

struct as the Return Type

- You may also have a return type of struct.

```
struct A_type func4()
{   struct A_type s={10,4};
    ...
    return s;
}
```

Array of struct

- Array of struct is straight forward. It is like array of any other type.
  - Arrays of structures—like all other arrays—are automatically passed by reference.

```
struct stu_info {
   char name[41];
   long int id;
   char dept[16];
   short int class;
   float gpa;
} stu1, stu2;

struct stu_info   class[100];
int    number[100];

number[3]      = 42;
class[3].id    = 42;
```

Example 6 – Cube

- Define type for a point

```
struct point_type {
   int x, y;
};
```

- Define type for a triangle.

```
struct triangle_type {
   struct point_type A, B, C;
};
```

- Define type for a cube

```
struct cube_type {
   struct point_type corner[8];
};
```

Example 7 – Student Info

- Write a program that collects info about 10 students in class and finds the average of their GPAs.

```
struct stu_info {
   char name[41];
   long int id;
   char dept[16];
   short int class;
   float gpa;
};

void getGrade(struct stu_info s[])
{...}

int main(){
   struct stu_info student[10];
   int i;   float avg=0;
   getGrade(student);
   for (i=0; i<10; i++)
      avg += student[i].gpa;
   printf("%f\n",avg/=10);
   return 0;
}
```

Size of a struct

- Assume in your system a short int occupies 2 bytes and an int occupies 4 bytes.
- What is the size of the following struct?

```
struct A {
   short int m;
   int n;
   char k;
};
```

- It is at least `2+4+1=7` bytes, but could be even larger → Depends on your system.

Unions

Motivation - union

- When you use a struct, all fields store values simultaneously.

- Sometimes, it is necessary to store one field or the other exclusively (i.e., not both).
  - That is why you need a union.

union

- union
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a union share same storage space
  - Only the last data member defined can be accessed

- union definitions
  - Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```

Valid Operations

- Assignment to `union` of same type: `=`
- Taking address: `&`
- Accessing union members: `.`
- Accessing members using pointers: `->`

union Syntax and Size

- The syntax is very similar to struct:

```
union union_name {
   field_definition(s)
   ...
} variable_list;
```

- Example:

```
union M {
   int a;
   float b;
   double c;
};
```

- The difference is that a single value is stored at a time.
- The size of the union is the size of the largest field (rather than the sum of the sizes of all fields).

Example 9 – union number

```
union number {
   int x;
   double y;
};

int main(void){
   union number value; // define union variable
   value.x = 100; // put an integer into the union
   printf("int:%d  double:%f\n",value.x, value.y);

   value.y = 100.0; // put a double into the same union
   printf("int:%d  double:%f\n",value.x, value.y);
}
```

Union type for a geometric figure:

```
struct circle_t {
    double area,
    perimeter,
    radius;
};

struct rectangle_t {
    double area,
    perimeter,
    width,
    height;
};

struct square_t {
    double area,
    perimeter,
    side;
};
```

```
union {
    struct circle_t circle;
    struct rectangle_t rectangle;
    struct square_t square;
} figure_data_t;
```

Example 10 – student or staff information

- Consider the following structs:

```
struct staff_info {
   char name[41];
   long int SSid;
   enum {assist, prof, personnel} status;
   int salary;
};
struct stu_info {
   char name[41];
   long int id;
   short int class;
   float gpa;
};
```

- Write a program that fills in an array of 100 elements where each element could be of type stu_info or staff_info.

Example 10 – struct and union definitions

```
#include <stdio.h>

struct person_info {
   enum {student, staff} type;
   union {
      struct stu_info {
         char name[41];
         long int id;
         char dept[16];
         short int class;
         float gpa;
      } student;
      struct staff_info {
         char name[41];
         long int SSid;
         enum {assist, prof, personnel} status;
         int salary;
      } staff;
   } info;
};
```

Example 10 – cont.

```
void read_student(struct stu_info *s)
{...}
void read_staff(struct staff_info *s)
{...}

int main()
{  struct person_info person[100];
   int i;

   for (i=0; i<100; i++)
   {  printf("Enter 0 for student, 1 for staff: ");
      scanf("%d", &person[i].type);
      if (person[i].type==student)
         read_student(&person[i].info.student);
      else
         read_staff(&person[i].info.staff);
   }
   return 0;
}
```

typedef Syntax and Usage

- You may define new names for existing types using `typedef`.
  - Note that `typedef` does not create a new type.
- Syntax:

```
typedef existing_type_name new_type_name(s);
```

- Example:

```
typedef int tamsayi;
typedef float real;
```

- Now, you can do the following:

```
tamsayi i;
tamsayi arr[50]={0};
real x = 1.5;
i=10; arr[3]=17;
```

When to Use typedef

- typedef is mostly useful for structures to avoid using the word "struct" in front of the structure name.
- Example:

```
typedef struct A_type  A_t;
A_t var1;

typedef struct {
   int x, y;
} nokta_t, noktalar_t[10];
nokta_t n;
noktalar_t N;
n.x = 5;
N[4].x = 8;
```