Structure and Union

SDC, OSW CSE 3541

Department of Computer Science & Engineering
ITER, Siksha 'O' Anusandhan Deemed To Be University
Jagamohan Nagar, Jagamara, Bhubaneswar, Odisha - 751030

Text Book(s)

Jeri R. Hanly & Elliot B. Koffman

Problem Solving and Program Design in C

Seventh Edition, Pearson Education

Robert Love

LINUX System Programming

Second Edition, SPD, O'REILLY

Contents

Introduction

- A mechanic is good enough who knows how to repair only one type of vehicle. Other mechanic is too enough to repair different types of vehicles. Think the better one.
- To handle real world data, all **int**s, or all **float**s or all **char**s or **array**s are not good enough at a time.
- However, we require another kind of programming tool, **structure** or **union**, that could able to handle one or more variables of same or different types.
- Unlike an array, a structure or union can have individual components that contain data of different types.
- **For example**, a single variable **planets** can store a planet's name, diameter, number of moons, the number of years to complete one solar orbit, and the number of hours to make one rotation on its axis.
- Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure.

STRUCTURE

- A structure is a collection of one more variables, possibly of different types, grouped together under a single name for convient handling.
- It is a user-defined data type.
- The keyword `struct` introduces a structure declaration.

Defining a Structure

A structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defined as

❶ Syntax: Structure Declaration

```
struct <structure name/tag> {
 data_type 1 member 1;
 data_type 2 member 2;
 . . . . .
 data_type n member n;
};
```

Example

```
struct sdata{
   int regdno;
   float mark;
   char name[30];
   int *pp;
};
```

Description

- **`struct`** is a required keyword.
- **`structure name/tag`** is a name that identifies structures of this type.
- **`member 1`**, ... , **`member n`** are individual member declarations with their associated data types.
- The individual members can be ordinary variables, pointers, arrays, or other structures.
- The member names within a particular structure must be **distinct** from one another, though a member name can be the same as the name of a variable that is defined outside of the structure.
- A storage class, however, cannot be assigned to an individual member and individual members cannot be initialized within a structure type declaration.

Did I know?

Assume a structure is named **`account`**. It contains four members: an integer quantity (**`acct-no`**), a single character (**`acct-type`**), an 80-element character array (**`name [80]`**), and a floating-point quantity (**`balance`**). Declare the required structure.

Did I know?

Assume a structure is named **account**. It contains four members: an integer quantity (**acct-no**), a single character (**acct-type**), an 80-element character array (**name [80]**), and a floating-point quantity (**balance**). Declare the required structure.

Structure Declaration

```
struct account {
    int acct-no;
    char acct-type;
    char name[80];
    float balance;
};
```

Did I know?

Assume a structure is named **account**. It contains four members: an integer quantity (**acct-no**), a single character (**acct-type**), an 80-element character array (**name [80]**), and a floating-point quantity (**balance**). Declare the required structure.

Structure Declaration

```
struct account {
    int acct-no;
    char acct-type;
    char name[80];
    float balance;
};
```

Try one more

The basic object in graphics is a point. A point is described as an x-coordinate and y-coordinate, where both are integers. Declare a structure with name **point** to describe the point.

Did I know?

Assume a structure is named **account**. It contains four members: an integer quantity (**acct-no**), a single character (**acct-type**), an 80-element character array (**name [80]**), and a floating-point quantity (**balance**). Declare the required structure.

Structure Declaration

```
struct account {
    int acct-no;
    char acct-type;
    char name[80];
    float balance;
};
```

Try one more

The basic object in graphics is a point. A point is described as an x-coordinate and y-coordinate, where both are integers. Declare a structure with name **point** to describe the point.

Your answer

```
struct point{
    int x;
    int y;
};
```

Declaring Structure Variable

Once the new structure data type has been defined one or more variables can be declared to be of that type. Structure variable can be declared in various ways as;

❶ Structure Variable Declaration

```
struct sdata {
 int regdno;
 float mark;
 char name[80];
};
struct sdata s1,s2;
```

❷ Structure Variablr Declaration

```
struct sdata {
 int regdno;
 float mark;
 char name[80];
}s1,s2;
```

❸ Structure Variablr Declaration

```
struct {
 int regdno;
 float mark;
 char name[80];
}s1,s2;
```

Description

- **scruct** the keyword.
- **sdata** name of the structure name.
- **regdno**, **mark** and **name** are structure members or components.
- **s1** and **s2** are declared as structure variables.

Placement of Structure & Structure Variable

Placement of structure definition in a program

- **In global scope:** Structure can be declared outside of the function main or others.
- **In local scope:** Structure can be declared inside of the function main or others.

Placement of structure variable in a program

- **In global scope:** structure variable(s) can be declared outside of the required function.
- **In local scope:** structure variable(s) can be declared inside of the required function.

Example: Global Scope

```
struct sdata{
 int regdno;
 float mark;
 char name[80];
};
struct sdata s1,s2;
int main(){
   .....
   return 0;
}
```

Example: Local Scope

```
struct sdata{
 int regdno;
 float mark;
 char name[80];
};
int main(){
   struct sdata s1,s2;
   .....
   return 0;
}
```

Structure Variable Initialization

- The members of a structure variable can be assigned initial values in much the same manner as the elements of an array.
- The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

```
struct struct_name var_name={value 1, value 2,...., value n};
```

- where **value 1** refers to the value of the first member, **value 2** refers to the value of the second member, and so on.
- A structure variable, like an array, can be initialized only if its storage class is either **external** or **static**. So the general form can re-write as;

```
storage_class struct struct_name var_name={value 1,value 2,....,value n};
```

Variable initialization

```
struct sdata{
 int regdno;
 float mark;
 char name[80];
}s1={23,45.50,"CSE"};
int main(){
  .....
  return 0;}
```

Variable initialization

```
struct sdata{
 int regdno;
 float mark;
 char name[80];
};
int main(){
 struct sdata s1={10,5.5,"cse"};
 return 0;}
```

Assignment of initial values to the members of a structure variable

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure.

```
static struct account customer = {1234,'S',"Alok K.",586.30,
                                                    5,24,90};
```

structure & it's initialization

```
struct date {
int month;
int day;
int year;
};
struct account (
int acct-no;
char acct-type;
char name[80];
float balance ;
struct date lastpayment;
};
```

Description

- **customer** is a static structure variable of type **account**, whose members are assigned initial values.
- Values to the structure members are assigned in order; first member-to-first value, 2nd member-to-second value and so on.
- The last member is itself a structure that contains three integer members (**month**, **day** and **year**). Therefore, the last member of customer is assigned the integer values 5, 24 and 90.

Array of Structures

It is also possible to define an array of structures; i.e., an array in which each element is a structure.

Array of structures declaration

```
struct sdata{
 int regdno;
 float mark;
 char name[80];
};
struct sdata s[100];
```

Description

- **s** is a 100-element array of structures. Hence, each element of the array **s** is a separate structure of type **sdata**.
- Each structure of type **s** includes an array (i.e. name[80]).

An array of structures can be assigned initial values just as any other array. Remember that each array element is a structure that must be assigned a corresponding set of initial values.

Illustration: Array of Structures Initialization

Array of structures

```
struct sdata{
 int regdno;
 float mark;
 char name[80];
};
```

❶ Initialization

```
struct sdata s[]={10, 45.6, "AD",
                  12, 34.5, "BD",
                  23, 10.4, "SJ",
                  56, 34.7, "SB"
                 };
```

Some programmers may prefer to embed each set of constants within a separate pair of braces, in order to delineate the individual array elements more clearly. This is entirely permissible. Thus, the array declaration can be written;

❷ Initialization

```
struct sdata s[]={
                    {10, 45.6, "AD"},
                    {12, 34.5, "BD"},
                    {23, 10.4, "SJ"},
                    {56, 34.7, "SB"}
                  };
```

Verify: Valid or Invalid Declarations

Two different structures, called **first** and **second**, are declared in a program. Check whether this kind of declarations are valid in a program or not.

❶

```
struct first {
 float a;
 int   b;
 char c;
};
```

❷

```
struct second {
char a;
float b, c;
};
```

Verify: Valid or Invalid Declarations

Two different structures, called **first** and **second**, are declared in a program. Check whether this kind of declarations are valid in a program or not.

❶

```
struct first {
 float a;
 int   b;
 char c;
};
```

❷

```
struct second {
char a;
float b, c;
};
```

- Notice that the individual member names a, b and c appear in both structure declarations, but the associated data types are different.
- This duplication of member names is permissible, since the scope of each set of member definitions is confined to its respective structure.
- Within each structure the member names are distinct, as required.

Manipulating Individual Members/Components of a Structured Type

The members/components of a structure can be manipulated in the following ways;

- **Using direct component selection (i.e. dot (.) operator):** A period is placed between a structure type variable and a member/component name to create a reference to the member/component.

Structure Template

```
struct sdata{
 int regdno;
 float mark;
 char name[80];
};
```

Initialization

```
struct sdata s;
s.regdno=123;
s.mark=12.34;
strcpy(s.name, "AKS");
/* scanf may be used to read from user*/
```

```
struct acc{int b=4;};//Not allowed:no memory is allocated for it.
```

- **Using indirect component selection operator (i.e. arrow (->) operator):** The character sequence **->** is placed between a pointer variable and a component name creates a reference that follows the pointer to a structure and selects the component.

An Example: Dot(.) Operator to Initialize structure

Consider the given structure

```
struct planet_t{
  char name[80];
  double diameter;
  int moons;
  double orbit_time,
       rotation_time;
};
int main(){
 struct planet_t blank_planet = {"",0,0,0,0};
 struct planet_t current_planet;
 strcpy(current_planet.name, "Jupiter");
 current_planet.diameter = 142800;
 current_planet.moons = 16;
 current_planet.orbit_time = 11.9;
 current_planet.rotation_time = 9.925;
 ...
 return 0;
}
```

Usage of . operator

Variable `blank_planet`, a structure of type `planet_t`

.name	\0 ? ? ? ? ? ? ? ?
.diameter	0.0
.moons	0
.orbit_time	0.0
.rotation_time	0.0

current_planet

initialize using . operator

Variable `current_planet`, a structure of type `planet_t`

.name	J u p i t e r \0 ? ?
.diameter	142800.0
.moons	16
.orbit_time	11.9
.rotation_time	9.925

Structure Pointer/ Pointer to Structure

- Pointer which points to the address of the memory block that stores a structure.
- **Structure pointer declaration:**

```
struct struct_name *ptrname;
```

- The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

Structure Pointer

```
struct sdata{
   int regdno;
   float mark;
   char name[80];
};
int main(){
   struct sdata s;
   struct sdata *ptr;
   ptr=&s;
   ......
   return 0;
}
```

Description

- **sdata** the structure.**s** the structure variable.
- **ptr** the structure pointer that holds the address of the structure.
- **Arrow(->) operator:** To access structure member using pointer.
- **->** and **(*).** both represent the same.
- Syntax: **ptr -> struct_member** `or` **(*ptr) . struct_member**

```
int main(){
   struct sdata s,*ptr;
   ptr=&s;
   ptr->mark=10.34;
   return 0;}
```

An Example: Usage of dot(.) and arrow(->)

❶ Using Dot operator

```
struct sdata{
  int regdno;
  float mark;
  char name[80];
};
int main(){
  struct sdata s;
  scanf("%d",&s.regdno);//10
  scanf("%f",&s.mark);//23.45
  scanf("%s",s.name);//CSE-C
  printf("%d\n",s.regdno);
  printff("%f\n",s.mark);
  printff("%s\n",s.name);
  return 0;
}
```

❷ Using arrow operator

```
struct sdata{
  int regdno;
  float mark;
  char name[80];
};
int main(){
  struct sdata s,*ptr;
  ptr=&s;
  scanf("%d",&ptr->regdno);//10
  scanf("%f",&ptr->mark);//23.45
  scanf("%s",ptr->name);//CSE-C
  printf("%d\n",ptr->regdno);
  printf("%f\n",ptr->mark);
  printf("%s\n",ptr->name);
  return 0;}
```

```
int main(){
  struct sdata s,*ptr;ptr=&s;scanf("%d",&(*ptr).regdno);//10
  scanf("%f",&(*ptr).mark);/*23.45*/scanf("%s",&(*ptr).name);//CSE-C
  printf("%d\n",(*ptr).regdno);printff("%f\n",(*ptr).mark);
  printf("%s\n",(*ptr).name);return 0;}
```

Structure & Dynamic Memory Allocation

- We know dynamic memory allocation mechanism; **`malloc`**, **`calloc`**, **`realloc`** and **`free`**.
- The user-defined data type can be dynamically allocated and subsequently processed using pointer to the memory allocated.
- **Dynamic memory allocation using `malloc`:**

```
struct struct_name *ptrname;
ptrname=(struct struct_name *)malloc(sizeof(struct struct_name));
```

- The pointer, **`ptrname`**, tells the address of a structure in memory (i.e. in HEAP).

Structure and `malloc`

```
struct sdata{ int regdno;
  float mark;
  char name[80];
};
int main(){ struct sdata *ptr;
  ptr=(struct sdata *)malloc(sizeof(struct sdata));
  ptr->regdno=20;
  printf("Regdno:%d\n",ptr->regdno);
  ......
  return 0;}
```

Structure and Function

We can pass structure member(s) and/or the entire structure as function argument. Also function can return as structure type.

- **Passing structure member(s) as function argument:**
  - Pass/Call by value method (i.e. function input parameter(s))
  - Pass by address/pointer/reference (i.e. function output parameter(s))
- **Passing entire structure as function argument:**
  - Pass/Call by value method (i.e. function input parameter(s))
  - Pass/Call by address/pointer/reference (i.e. function output parameter(s))

Consider the structure

```
#define STRSIZ 10
struct planet_t{
 char name[STRSIZ];
 double diameter;
 int moons;
 double orbit_time,
   rotation_time;
};
```

Structure variable & initialization

```
int main(){
  struct planet_t current_planet;
  strcpy(current_planet.name, "Jupiter");
  current_planet.diameter = 142800;
  current_planet.moons = 16;
  current_planet.orbit_time = 11.9;
  current_planet.rotation_time = 9.925;
  ..................................
  return 0;
}
```

Passing Structure Member as Call by Value

Calling `passval(...);` function from main

```
int main(){
struct planet_t current_planet;
passval(current_planet.moons);
return 0;
}
```

Structure member as input parameter

```
void passval(int nm){
   printf("%d\n", nm);
   nm=nm+34;
   printf("%d\n", nm);
}
```

- Like **`current_planet.moons`** other structure members can be passed by value.
- Any update of formal parameter inside the function will **not affect** the original value of the structure member **`moons`** in main.

Passing Structure Member as Call by Address

Calling `passval(...);` function from main

```
int main(){
struct planet_t current_planet;
passaddr(&current_planet.moons);
return 0;
}
```

Structure member as output parameter

```
void passaddr(int *nm){
   printf("%d\n", *nm);
   *nm=*nm+34;
   printf("%d\n", *nm);
}
```

- Like **`current_planet.moons`** other structure members can be passed by address.
- Any update of formal parameter inside the function using indirection will **affect** the original value of the structure member **`moons`** in main.

Pointer to structure	Actual argument	Formal parameter
`ptr`	`ptr->struct_mem`	`struct_mem_data_type var_name`
`ptr`	`&ptr->struct_mem`	`struct_mem_data_type *var_name`

Structure Type Data as Input Parameters

When a structured variable is passed as an input argument to a function, all of its component values are copied into the components of the function's corresponding formal parameter.

To display the value of our structure **`current_planet`**;

Calling `print_planet(current_planet);` function from main

```
int main(){
......
print_planet(current_planet);
return 0;
}
```

Function with a Structured Input Parameter

```
void print_planet(planet_t pl){ /* input - one planet structure */
printf("%s\n", pl.name);
printf("Equatorial diameter: %.0f km\n", pl.diameter);
printf("Number of moons: %d\n", pl.moons);
printf("Time to Complete one orbit of the sun:%.2f years\n",
                                                pl.orbit_time);
printf("Time to complete one rotation on axis:%.4f hours\n",
                                              pl.rotation_time);
}
```

Structure Type Data as Output Parameters

When a structure variable is used as an output argument(i.e. pass by address), the address-of operator must be applied to the structure variable.

Calling `scan_planet(&current_planet);` function from main

```
int main(){
......
status = scan_planet(&current_planet);
return 0;
}
```

Function with a Structured Output Argument

```
int scan_planet(planet_t *plnp){ /* output - address of planet_t
                                            structure to fill */
int result = scanf("%s%lf%d%lf%lf",(*plnp).name, &(*plnp).diameter,
                                   &(*plnp).moons, &(*plnp).orbit_time,
                                   &(*plnp).rotation_time);
if(result == 5) {result=1;}
else if(result != EOF) {result = 0;}
return (result);
}
```

Function Return as Structure

When a structure variable is used as an output argument(i.e. pass by address), the address-of operator must be applied to the structure variable.

Calling `get_planet();` function from main

```
int main(){
struct planet_t current_planet;
current_planet = get_planet();
return 0;
}
```

Function return as structure, `planet`

```
/* Gets and returns a planet_t structure */
planet_t
get_planet(void){
 planet_t planet;
 scanf("%s%lf%d%lf%lf", planet.name,&planet.diameter,
                        &planet.moons,&planet.orbit_time,
                        &planet.rotation_time);
 return (planet);
}
```

Nested Structure

- Structure written inside another structure is called as nesting of structures.
- We can write one structure inside another structure as member of that structure.

❶ One Structure inside other

```
struct sdata{
  int regdno;
  float mark;
  char name[80];
  struct date{
  int d;
  int m;
  int y;
}b1,b2;
}s1,s2;
-------- ACCESS -------
(1) s1.b1.d-For inner structure
(2) s1.regdno-For self
(3) Same procedure for pointer
```

NOTE: if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing
`variable.member.submember`

❷ Structure as member of another structure

```
struct date{
  int d;
  int m;
  int y;
};
struct sdata{
  int regdno;
  float mark;
  char name[80];
  struct sdate b1,b2;
}s1,s2;
-------- ACCESS -------
(1) s1.b1.d-For inner structure
(2) s1.regdno-For self
(3) Same procedure for pointer
(4) struct date can be accessed
    independently
```

User-defined Data Types (`typedef`)

- The **`typedef`** is a keyword in C.
- The **`typedef`** feature allows users to define new data-types that are equivalent to existing data types.
- Once a user-defined data type has been established, then new variables, arrays, **structures**, etc. can be declared in terms of this new data type.
- In general terms, a new data type is defined as

```
typedef existing-type new-type;
```

  where **`existing-type`** refers to an existing data type (either a standard data type, or previous user-defined data type), and **`new-type`** refers to the new user-defined data type.
- The new data type will be **new** in name only. In reality, this new data type will not be fundamentally different from one of the standard data types.

An Example:

Here is a simple declaration involving the use of typedef- `typedef int age;` In this declaration **`age`** is a user-defined data type, which is equivalent to type **`int`**.

Hence, the variable declaration **`age male, female;`**

is equivalent to **`int male, female;`**

typedef few more Examples

❶ Normal declaration

```
float men[100], women[100];
```

❶ Using typedef

```
typedef float height;
height men[100], women[100];
```

❷ Normal declaration

```
float men[100], women[100;
```

❷ Using typedef

```
typedef float height[10];
height men, women;
```

❸ Normal declaration

```
int a=10,*ptr;
ptr=&a;
printf("%d",*ptr);
```

❸ Using typedef

```
typedef int* ptrvar;
ptrvar ptr;
ptr=&a;
printf("%d",*ptr);
```

typedef feature can also be used for defining structures. Using **typedef** structure can eliminates the need to repeatedly write **struct** keyword whenever a structure is referenced. Hence, the structure can be referenced more concisely.

typedef to Define Structure

❶ Syntax: Structure Declaration

```
typedef struct{
 data_type 1 member 1;
 data_type 2 member 2;
 . . . . .
 data_type n member n;
}new-type;
```

Example: structute using typedef

```
typedef struct {
char name[20];
double diameter;
int moons;
double orbit_time,
            rotation_time;
} planet_t;
```

Description

- where **new-type** is the user-defined structure type. Structure variables can then be defined in terms of the **new-type**
- In right example, **planet_t** is the structure type. This structure type has five distinct components.
- **The typedef statement itself allocates no memory**.
- A variable declaration is required to allocate storage space for a structured data object
- Variable declaration for **planet_t** structure type:

```
planet_t current_planet, //current_planet variable of planet_t
         previous_planet,  //variable of planet_t
 blank_planet = {"", 0, 0, 0, 0}; // var with initialization
```

- **Processing:** Same way using dot(.), indirection and arrow(->) operators.

Did I know?

Assume a structure is named **`account`**. It contains four members: an integer quantity (**`acct-no`**), a single character (**`acct-type`**), an 80-element character array (**`name [80]`**), and a floating-point quantity (**`balance`**). Declare a user-defined structure using.

Did I know?

Assume a structure is named **account**. It contains four members: an integer quantity (**acct-no**), a single character (**acct-type**), an 80-element character array (**name [80]**), and a floating-point quantity (**balance**). Declare a user-defined structure using.

Structure Declaration

```
typedef struct{
    int acct-no;
    char acct-type;
    char name[80];
    float balance;
}account_t;
```

Did I know?

Assume a structure is named **account**. It contains four members: an integer quantity (**acct-no**), a single character (**acct-type**), an 80-element character array (**name [80]**), and a floating-point quantity (**balance**). Declare a user-defined structure using.

Structure Declaration

```
typedef struct{
    int acct-no;
    char acct-type;
    char name[80];
    float balance;
}account_t;
```

Try one more

The basic object in graphics is a point. A point is described as an x-coordinate and y-coordinate, where both are integers. Declare a user-defined structure with name **point_t** to describe the point.

Did I know?

Assume a structure is named **account**. It contains four members: an integer quantity (**acct-no**), a single character (**acct-type**), an 80-element character array (**name [80]**), and a floating-point quantity (**balance**). Declare a user-defined structure using.

Structure Declaration

```
typedef struct{
    int acct-no;
    char acct-type;
    char name[80];
    float balance;
}account_t;
```

Try one more

The basic object in graphics is a point. A point is described as an x-coordinate and y-coordinate, where both are integers. Declare a user-defined structure with name **point_t** to describe the point.

Your answer

```
typedef struct{
    int x;
    int y;
}point_t;
```

Self-Referential Structure

- A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind.
- They are widely used in dynamic data structures like trees, linked list, etc.

Declaration

```
struct account{
    int acct-no;
    char acct-type;
    char name[80];
    float balance;
    struct account *nextacc;
}account[100];
```

- Unlike a static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self referential structure can dynamically be expanded or contracted.
- Self referential structures may have either single pointer of same structure type or have multiple pointers of same type depending on applications.

UNION

- **union** a data structure that overlays components in memory, allowing one chunk of memory to be interpreted in multiple ways
- It is a user-defined data type.
- The keyword **union** is used for union declaration.
- Same procedures are followed for union as like structure to access it's member and for processing.

A Critique of Structure vs Union

Similarities Between Structure and Union

- Both are user-defined data types used to store data of different types as a single unit.
- Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
- Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
- Dot(.) operator or selection operator (->) is used for accessing members for both the user-defined data types.
- A structure can be placed inside a union and vice-versa.

Differences between Structure and Union- SELF REALIZATION

- Keyword
- Memory management and storage
- Syntax
- Accessing
- Initialization
- With pointer
- With function

Try Yourself

1. Declare an array of 40 student_t structures, and write a code segment that displays on separate lines the names (last name, first name) of all the students in the list.
2. Write a program to use a union type component in a structured variable when the needed structure components vary depending on the value of one component.
3. Define a structure type called subscriber_t that contains the components name , street_address , and monthly_bill (i.e., how much the subscriber owes).
4. Define a data structure to store the following student data: gpa, major, address (consisting of street address, city, state, zip), and class schedule (consisting of up to six class records, each of which has description, time, and days components). Define whatever data types are needed.
5. Define a structure type element_t to represent one element from the periodic table of elements. Components should include the atomic number (aninteger); the name, chemical symbol, and class (strings); a numeric field for the atomic weight; and a seven-element array of integers for the number of electrons in each shell. The following are the components of an element_t structure for sodium. **`11 Sodium Na alkali_metal 22.9898 2 8 1 0 0 0 0`**.

Exercise

Identify the following statements as possibly valid or definitely invalid. If invalid, explain why.

Structure declaration

```
typedef struct {
char fst_name[20],
last_name[20];
int score;
char grade;
} student_t;
. . .
student_t stu1, stu2;
```

```
(a) student_t stulist[30];
(b) printf("%s", stu1);
(c) printf("%d %c", stu1.score,
     stu1.grade);
(d) stu2 = stu1;
(e) if (stu2.score == stu1.
    score)
        printf("Equal");
(f) if (stu2 == stu1)
       printf("Equal structures
           ");
(g) scan_student(&stu1);
(h) stu2.last_name = "Martin";
```

THANK YOU