Structure and Union

Instructor: Jeeho Ryoo

Structures in C

- In C, we can create our own, complex data types.
- This is very convenient for modeling real-life objects by defining our own data types that represent structured collections of data pertaining to particular objects.
- int, double, char are types of variables defined in C. by using structures, you can create your own types – a nice way to extend your programming languages.
- Unlike array, a structure can have individual components that contain data of different types.
- Each of these data items is stored in a separate **component** of the structure and can be referred by using the component name.

Defining structure using struct

```
# include <stdio.h>
struct line {
                int x1, y1; // Coordinates of one endpoint of a line
                int x2, y2; // Coordinates of other endpoint of a line
};
int main() {
                struct line line1;
}
```

Structure line has 4 Components of type int

This defines the variable line1 to be a variable of type line

Variables may also be declared in the structure definition.

```
struct line {
                int x1, y1, x2, y2;
} line1, line2;
```

Defining structure using typedef

typedef allows us to associate a name with a structure. (where else we saw typedef ?)

```
typedef struct {
        int x1, y1; // Coordinates of one endpoint of a line
        int x2, y2; // Coordinates of other endpoint of a line
} line_t;

int main() {
        line_t line1;
}
```

line1 is now a variable of type line_t.

The typedef statement itself allocates no memory. A variable declaration is required to allocate storage space for a structured data object.

Accessing components of a structure

To access a component of a structure, we can use the **direct component selection operator**, which is a dot/period.

```
int main() {
        line_t line1;
        line1.x1 = 3;
        line1.y1 = 5;
        if(line1.y2 == 3) {
                printf("Y co-ord of end is 3\n");
        }
}
```

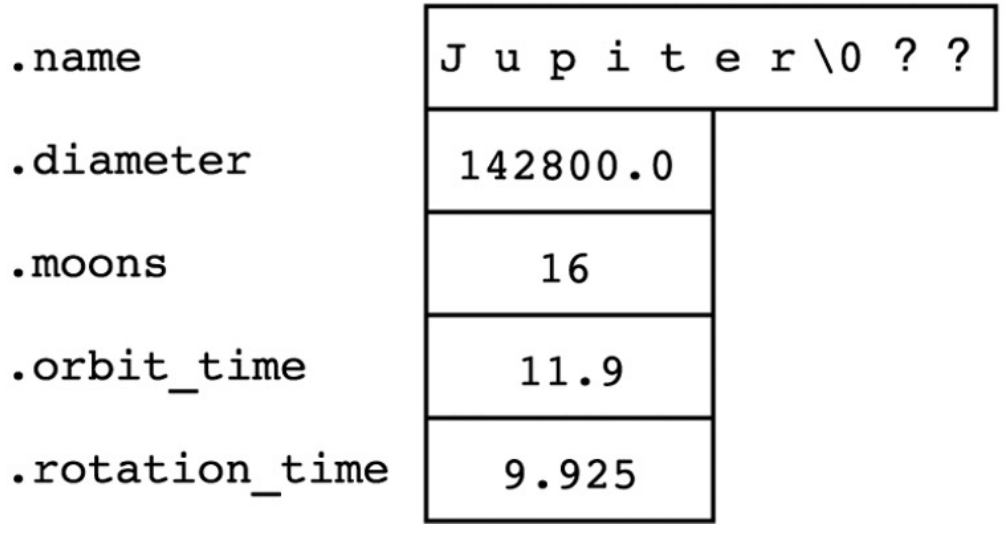
Direct component selection operator has the highest precedence.

Assigning values to the components of a structure (from text)

```
typedef struct {
  char name[10];
  double diameter;
  int moons;
  double orbit_time,

retation_time;
} planet_t;
```

```
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;
```

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



Using Structures

Structures can contain any C type, including arrays, pointers or even other structures as components.

Initialization of structures: (similar to arrays)

```
line_t line1 = {3, 5, 6, 7};
```

Assignment of entire structures:

```
line2 = line1;  // assign to each component of line2 a value of
                // the corresponding component of line1.
```

Although C permits copying of entire structure, the equality and inequality operator can not be applied to a structured type as a unit.

```
if (line1 == line2) // Invalid
```

Also you can't use structures as argument to printf and scanf statement.

Structures as Input parameter

We can pass structure as input argument to a function. We have to make sure that the function prototype is introduced to compiler **after** the structure is declared.

```
typedef struct {
                    char name[10];
                    double diameter;
                    int moons;
                    double orbit_time,
                    retation_time;
} planet_t;

void print_planet (planet_t p1);

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

```
1.  /*
2.   * Displays with labels all components of a planet_t structure
3.   */
4.  void
5.  print_planet(planet_t pl) /* input - one planet structure */
6.  {
7.        printf("%s\n", pl.name);
8.        printf("  Equatorial diameter: %.0f km\n", pl.diameter);
9.        printf("  Number of moons: %d\n", pl.moons);
10.       printf("  Time to complete one orbit of the sun: %.2f years
11.             pl.orbit_time);
12.       printf("  Time to complete one rotation on axis: %.4f hours
13.             pl.rotation_time);
14. }
```

current_planet is passed as input argument and all the component values of current_planet are copied to corresponding formal parameter p1 in function print_planet.

Structure as output parameter

Structures may contain large amount of data.
If a function needs to modify the content of a structure

Use pointers to pass address of the structure to functions instead of passing the structure by value. Example,
status = scan_planet(&current_planet); // Statement in function main

```
10. int
11. scan_planet(planet_t *plnp) /* output - address of planet_t structure
12.                                         to fill                        */
13. {
14.       int result;
15.
16.       result = scanf("%s%lf%d%lf%lf",  (*plnp).name,
17.                                        &(*plnp).diameter,
18.                                        &(*plnp).moons,
19.                                        &(*plnp).orbit_time,
20.                                        &(*plnp).rotation_time);
21.       if (result == 5)
22.             result = 1;
23.       else if (result != EOF)
24.             result = 0;
25.
26.       return (result);
27. }
```

Few notes about structure

Simple structure declaration

Syntax: structName varName; Example, planet_p p1;

A pointer to a structure

Syntax: structName * ptrName; Example: planet_p * p1_ptr;

Accessing a component of a structure

Syntax: varName.componentname; Example: p1.name

Accessing a component of a pointer to a structure

Syntax: (*ptrName).componentname;

Example: (*p1_ptr).name /* The brackets are important cause "." has higher priority than "*" */

Indirect component selection operator

- C provides a single operator that combines the function of the indirection (pointer dereference) and component selection operator.
- For a pointer to a structure, a component can be accessed by using indirection operator "->"
- Syntax: ptrName -> componentName; Example: p1_ptr -> name;

Structure as return type of a function

So far, we have seen that the structures are treated mostly like C's simple data types (int, char etc.). One exception though (Anybody?)
Comparatively, C's processing of array differs a lot from its handling of simple data types. For example, array can't be returned as a function result.
However, we can return structure as the function result. Returning a structure from a function means returning the values of all components.

```
1.  /*
2.   * Gets and returns a planet_t structure
3.   */
4.  planet_t
5.  get_planet(void)
6.  {
7.        planet_t planet;
8.
9.        scanf("%s%lf%d%lf%lf",  planet.name,
10.                               &planet.diameter,
11.                               &planet.moons,
12.                               &planet.orbit_time,
13.                               &planet.rotation_time);
14.       return (planet);
15. }
```

Structures as components of structures (1)

Structures can contain other structures as members.
Example: Employee data base

```
typedef struct  {
                  char dept_name[25];
                  int dept_no;
} department_t;

typedef struct  {
                  char name[25];
                  int employee_id;
                  department_t dept;
                  address_t * add_ptr;
                  double salary;
} emp_data_t;
```

Member structures must be defined beforehand, since the compiler must know their size.

The size of a pointer to a structure is just the size of the memory address and therefore is known. So struct address_t can be defined Later.

Structures as components of structures (2)

Send structure emp1 as input argument, modify it and then return the modified structure to the calling routine.

```
e = update1(emp1); // in main
...
emp_data_t update1(emp_data_t emp)
{
   printf("Enter department number: ");
   scanf("%d", &n);
   emp.dept.dept_no = n;
   ...
   return emp;
}
```

Involves lots of copying of structure components to and from the function.

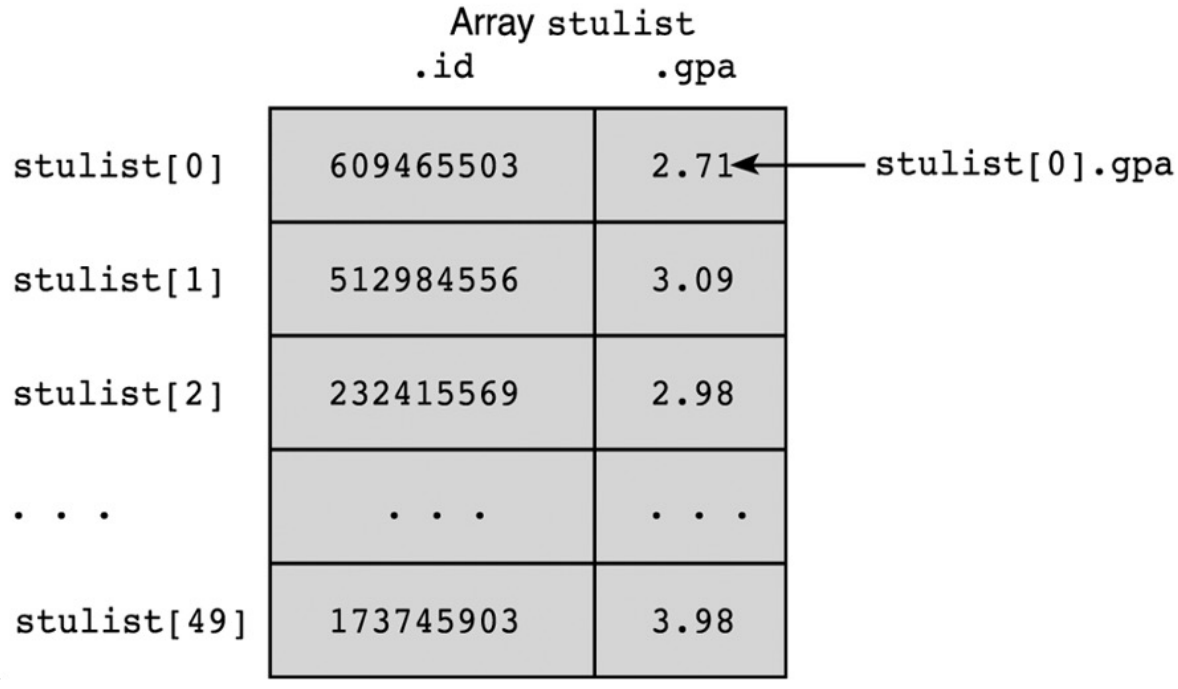
- Passing a pointer to a structure is more efficient.

```
update2 (&emp1); // in main
...
void update2 (emp_data_t *p)
{
   printf("Enter department number:
         ");
   scanf("%d", &n);
   p ->dept.dept_no = n;
   ...
}
```

- Use -> instead of . to access components of the structure, because p is a pointer to a structure.

Array of Structures

```
typedef struct {
    int id;
    double gpa;
} student_t;

student_t
stulist[50];
```



Accessing array elements

```
for (i = 0; i < 50; i++)
        printf ("%d\n", stulist[i].id);
```

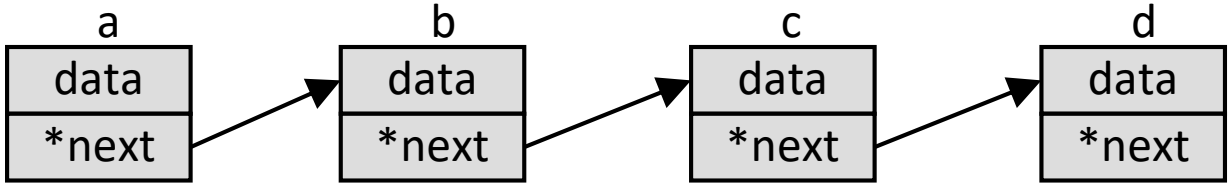
Self-referencing Structures

Structures may contain pointers to variables of their own type (recursive declaration).

```
struct list {
                    int data;
                    struct list *next;
} a, b, c, d;
```

This may look initially strange, but in real life it is a very very useful construction.

By using self-referencing structures, variables of that structure type may be linked as follows



Union (1)

C provides a data structure called union to deal with situations in which a data object can be interpreted an a variety of ways.
Like structure, union is a derived data type. On the other hand, union allows its components to share the same storage.
Example

```
typedef union {
                    int i;
                    float f;
} int_or_float;
int_or_float a, b, c, d;
```

Union (2)

Union provides a space in memory that can be interpreted in multiple ways.

```
union i_or_c {
        int i;
        char ch[4];
} n1, n2;
```

You can access n1 and n2 as either as an int or a char[].

```
n1.i = 10; n2.ch[1] = 'g';
```

Memory of a union is allocated according to the largest interpretation.

```
max(sizeof(int), 4*sizeof(char))
```

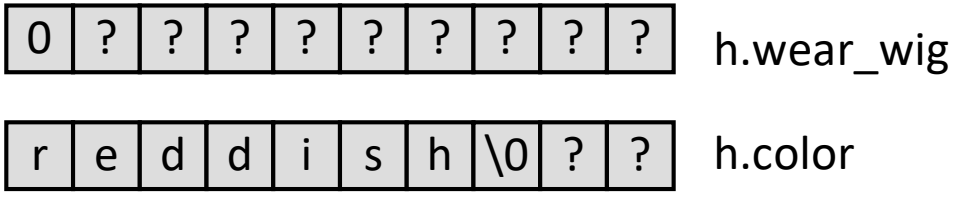
Union can help you save space in memory – allocate one space in memory and use it in multiple ways.

Union (3)

Unions are useful only if it is possible to determine within the program which interpretation is currently the valid interpretation.
Unions are mostly used as component of a larger structure, and the larger structure typically contains another component that determines which interpretation of the union is correct at the present time.

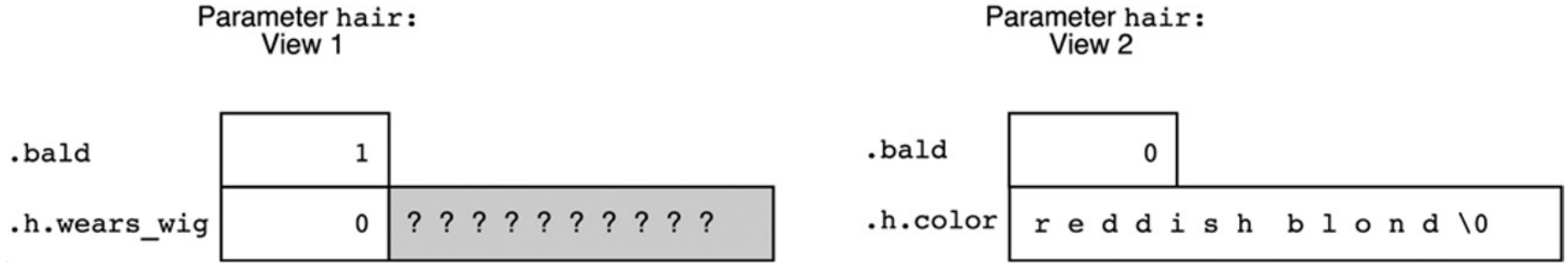
```
typedef union {
        int wear_wig;
        char color[10];
} hair_t;

typedef struct {
        int bald;
        hair_t h;
} hair_info_t;
```

Two interpretations of union h



Two interpretation of parameter hair

```
1.  void
2.  print_hair_info(hair_info_t hair) /* input - structure to display          */
3.  {
4.        if (hair.bald) {
5.              printf("Subject is bald");
6.              if (hair.h.wears_wig)
7.                    printf(", but wears a wig.\n");
8.              else
9.                    printf(" and does not wear a wig.\n");
10.       } else {
11.             printf("Subject's hair color is %s.\n", hair.h.color);
12.       }
13. }
```



Referencing the appropriate union component is programmer's responsibility.

struct Summary

- struct is a simple tool – combines simpler types into one larger type.
- Powerful for all sort of uses – used throughout C programs and operating systems.
- It makes modeling the real-life scenario easier.
- You can extend the language by defining your own types.

union Summary

Pros

- Very tight memory allocation
- Kind of polymorphism for types
- Useful for maintaining memory that can be interpreted in multiple ways

Cons

- You have to be careful with interpretation since types are very different at lower level.
- Can make code very confusing if used without enough documentation or improperly.