

C Programming

Lecture 7: struct, union and enum

Name	Gender	Age
Tom	Male	22
Jack	Male	21
Jane	Female	21

Lecturer: *Dr. Wan-Lei Zhao*

Autumn Semester 2022

Opening Discussion

- Given we have following informatio for 40 students
 - 1 student number
 - 2 name
 - 3 age
 - 4 gender
 - 5 height
 - 6 GPA
- We now want to build records for all the students

```
1 int main()  
2 {  
3     char std1nm[64], std2nm[64], ...;  
4     char std1nb[11], std2nb[11], ...;  
5     int std1ag, std2ag, ...;  
6     char std1gd[5], std1gd[5], ...;  
7     ...  
8 }
```

1 struct

2 union

3 enum

Composite Data Types

- It is valid/OK to do it in the way we learned
- However, it is not convenient
- C provides us the way to extend current data types
- We can combine primitive types into one type

```
1 struct STD {  
2     char stdNm[64];  
3     char stdNb[11];  
4     int age;  
5     char gender[5];  
6 };
```

- “`struct STD`” is a new data type
- Its role is similar as `int`, or `float`,...

```
struct structTag {  
    type1 member1;  
    type2 member2;  
  
    ...  
    typeN memberN; };
```

- Keyword “**struct**” is required, it tells C you are going to define a composite type
- **structTag** gives a **unique** tag for this new type
- You list all the members and their corresponding types
- “**;**” is required at the end
- Keep in your mind, you define a **type** instead of a variable/constant

struct: define variable of composite type (1)

```
struct structTag record;
```

- Keyword “struct” and structTag are required
- “record” is the variable name of structTag type

struct: define variable of composite type (2)

`struct structTag record;`

- Keyword “`struct`” and `structTag` are required
- “`record`” is the variable name of `structTag` type

```
1 struct STD {  
2     char stdNm[64];  
3     char stdNb[11];  
4     int age;  
5     char gender[5];  
6 };  
7 int main()  
8 {  
9     struct STD record;  
10    struct STD stds[40];  
11 }
```

struct: initialize variable of composite type (1)

- Each member in the composite type variable is treated as a variable
- They are visited via “var.member1”

```
1 struct STD {  
2     char stdNm[64];  
3     char stdNb[11];  
4     int age;  
5     char gender[5];  
6 };  
7 int main()  
8 {  
9     struct STD record;  
10    strcpy(record.stdNm, "Min_Li");  
11    strcpy(record.stdNb, "11201522031");  
12    record.age = 20;  
13    strcpy(record.gender, "male");  
14 }
```


struct: initialize variable of composite type (2)

```
1 #include <stdio.h>
2 #include <string.h>
3 struct STD {
4     char stdNm[64];
5     char stdNb[11];
6     int age;
7     char gender[5];};
8 int main()
9 {
10     struct STD std;
11     strcpy(std.stdNm, "Min_Li");
12     strcpy(std.stdNb, "22031");
13     std.age = 20;
14     strcpy(std.gender, "male");
15     printf("Name: _%s\n", std.stdNm);
16     printf("Numb: _%s\n", std.stdNb);
17     printf("Age: _%d\n", std.age);
18     printf("Gender: _%s\n", std.gender);
19 }
```

struct: exmaple (1)

- Please build a struct type for date (Year, month and day)
- Work out which day it is of the year
 - ① We need `struct` type to keep date inform
 - ② We need to calculate which day of the year is
 - ③ It depends on year (whether it is a leap year)
 - ④ Depends on the month
 - ⑤ Depends on the date

5 minutes to think about it...

[General procedure]

- ① Accept input, save the information to a date structure
- ② Check whether the year is leap year or not
- ③ Check which month it is
- ④ We need an array to keep the days of months

[General procedure in more detail]

- 1 Define a date struct
- 2 Accept input, save the information to a date structure
- 3 Initialize of days of months (12 months)
- 4 If it is leap year and date.month ≥ 3
- 5 Plus 1 day to the total
- 6 End-If
- 7 For i from 1 to (date.month-1)
- 8 sum up days of months before current month
- 9 End-for

struct: example (4)

```
1 struct DATE {  
2     int day, month, year;  
3 };  
4  
5 int main()  
6 {  
7     struct DATE date;  
8     int dyMonth[]={31,28,31,  
9     30,31,30,31,31,  
10    30,31,30,31};  
11    int i = 1, dayth = 0;  
12    printf("Year:");  
13    scanf("%d", &date.year);  
14    printf("Month:");  
15    scanf("%d", &date.month);  
16    printf("Day:");  
17    scanf("%d", &date.day);
```

```
17     if (isLeap(date.year))  
18     {  
19         dayth += 1;  
20     }  
21     for (; i < date.month; i++)  
22     {  
23         dayth += dyMonth[i - 1];  
24     }  
25     dayth += date.day;  
26     return 0;  
27 }
```

Is there anything wrong?? Two mistakes!!

struct: exmample (5)

```
1 struct DATE {  
2     int day, month, year;  
3 };  
4 int isLeap(int year)  
5 {     if(year%4==0) {  
6         if(year%400==0){  
7             return 1;  
8         } else if(year%100==0){  
9             return 0;  
10        }  
11        return 1;  
12    } else {  
13        return 0;  
14    } //end-if-else  
15 } //end-isLeap  
16 int main()  
17 {     struct DATE date;  
18     int dyMonth[]={31,28,31,  
19     30,31,30,31,31,  
20     30,31,30,31};
```

```
21     int i = 1, dayth = 0;  
22     printf("Year:");  
23     scanf("%d", &date.year);  
24     printf("Month:");  
25     scanf("%d", &date.month);  
26     printf("Day:");  
27     scanf("%d", &date.day);  
28     if(isLeap(date.year)&&date  
29         .month>2)  
30     {  
31         dayth += 1;  
32     }  
33     for (; i<date.month; i++)  
34     {  
35         dayth+= dyMonth[i-1];  
36     }  
37     dayth += date.day;  
38     return 0;  
39 }
```

struct: size of the struct type (1)

- Now let's consider another problem
- What is the size (bytes occupied) of struct type variable

```
1 struct DATE {  
2     int day, month, year;  
3 };  
4 struct STD{  
5     char Name[10];  
6     int age;  
7     char gender[6];  
8 };  
9 int main()  
10 {  
11     printf("%d\n", sizeof(  
12         struct DATE));  
13     printf("%d\n", sizeof(  
14         struct STD));  
15     return 0;  
16 }
```

[Output]

```
1 12  
2 24
```

- Can you figure out why?

struct: size of the struct type (2)

- Now let's consider another problem
- What is the size (bytes occupied) of struct type variable

```
1 struct DATE {  
2     int day, month, year;  
3 };  
4 struct STD{  
5     char Name[10];  
6     int age;  
7     char gender[6];  
8 };  
9 int main()  
10 {  
11     printf("%d\n", sizeof(  
12         struct DATE));  
13     printf("%d\n", sizeof(  
14         struct STD));  
15     return 0;  
16 }
```

[Output]

```
1 12  
2 24
```

- **Name** will be given with **12** bytes instead of 10
- **gender** will be given with **8** bytes instead of 6
- For the convenience of memory allocation
- This could be different from one compiler to another

struct example: complex number (1)

- Given two complex number $a = 2+3i$ and $b = 4-i$
- You are asked to define a struct of Complex
- Fulfill $c = a+b$

Think about it in 5 minutes ...

struct example: complex number (2)

- Define the `struct`

```
1 struct Complex {  
2     float real, virt;  
3 };
```

struct example: complex number (3)

```
1 struct Complex {  
2     float real, virt;  
3 };  
4  
5 struct Complex add( struct Complex a, struct Complex b){  
6     //fill by yourself  
7 }
```

struct example: complex number (4)

```
1 struct Complex {
2     float real, virt;
3 };
4
5 struct Complex add( struct Complex a, struct Complex b){
6     struct Complex c;
7     c.real = a.real + b.real;
8     c.virt = a.virt + b.virt;
9     return c;
10 }
11
12 int main(){
13     struct Complex a = {2,3} , c;
14     struct Complex b = {4,-1};
15     c = add(a, b);
16     printf("c = %f + %fi", c.real, c.virt);
17     return 0;
18 }
```

struct: `typedef` to save code (1)

- In “struct STD”, “`struct`” has been repeated everywhere
- We can use “`typedef`” to save up our typing

```
1 struct DATE {  
2     int day, month, year;  
3 };  
4 struct STD{  
5     char Name[10];  
6     int age;  
7     char gender[6];  
8 };  
9 typedef struct STD StdType;  
10 typedef struct DATE DatType;  
11 int main()  
12 {  
13     DatType date;  
14     StdType std;  
15     printf("%d\n", sizeof(  
16         DatType));  
17     ...  
18 }
```

- During compiling stage
- “StdType” is replaced by “`struct STD`”

struct: `typedef` to save code (2)

- You can apply `typedef` to any type

```
1 #include <stdio.h>
2 typedef unsigned int uint;
3
4 int main()
5 {
6     uint a = 32768;
7     printf("%d\n", a);
8     printf("%d\n", sizeof(uint));
9     return 0;
10 }
```

- During compiling stage
- “uint” is replaced by “`unsigned int`”
- You actually give a **nickname** to the type by `typedef`

1 struct

2 union

3 enum

- Sometimes it is not necessary to reserve a field for each struct member
- Several fields are allowed to share the same block of memory
- This special type of structure is called **union**

```
1 struct Data {  
2     short i;  
3     float f;  
4     char str[20];  
5 };
```

```
1 union Data {  
2     short i;  
3     float f;  
4     char str[20];  
5 };
```


union: definition (1)

```
union [union tag] {  
    type1 member1;  
    type2 member2;  
  
    ...  
};
```

- It is basically very similar as `struct`
- However, the members are kept in different way

```
1 struct Data1 {  
2     short i;  
3     float f;  
4     char str[20];  
5 };
```

```
1 union Data2 {  
2     short i;  
3     float f;  
4     char str[20];  
5 };
```

union: definition (2)

```
1 struct Data1 {
2     short i;
3     float f;
4     char str[10];
5 };
6
7 union Data2 {
8     short i;
9     float f;
10    char str[10];
11 };
12 int main()
13 {
14     Data1 d1;
15     Data2 d2;
16     printf("Size of d1 %d", sizeof(d1));
17     printf("Size of d2 %d", sizeof(d2));
18     return 0;
19 }
```

[Output:???

union: definition (3)

```
1 int main()  
2 {  
3     Data1 d1;  
4     Data2 d2;  
5     printf("Size of d1 %d", sizeof(d1));  
6     printf("Size of d2 %d", sizeof(d2));  
7     return 0;  
8 }
```

Size of d1: 20

Size of d2: 12

- Can you figure out why??

union: definition (4)

```
1 int main()  
2 {  
3     Data1 d1;  
4     Data2 d2;  
5     printf("Size of d1_%d", sizeof(d1));  
6     printf("Size of d2_%d", sizeof(d2));  
7     return 0;  
8 }
```

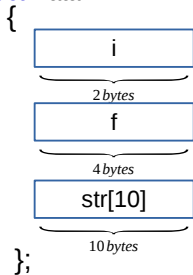
Size of d1: 20

Size of d2: 12

- For the convenience of memory allocation
- `str` will be given 12 bytes instead of 10

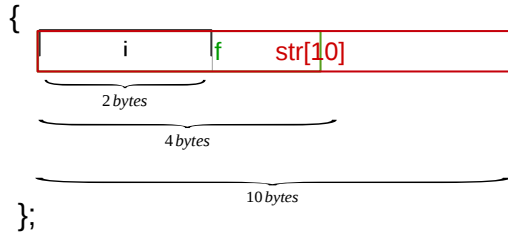
union: how they are kept in the memory

struct Data



(a) struct

union Data



(b) union

union: learn by example (1)

```
1 #include <stdio.h>
2 #include <string.h>
3 union Data {
4     int i;
5     float f;
6     char str[20];
7 };
8 int main()
9 {
10     union Data data;
11     data.i = 10;
12     data.f = 220.5;
13     strcpy( data.str , "C_Programming" );
14
15     printf( "data.i: %d\n" , data.i );
16     printf( "data.f: %f\n" , data.f );
17     printf( "data.str: %s\n" , data.str );
18     return 0;
19 }
```

union: learn by example (2)

```
1 #include <stdio.h>
2 #include <string.h>
3 union Data {
4     int i;
5     float f;
6     char str[20];
7 };
8 int main(){
9     union Data data;
10    data.i = 10;
11    data.f = 220.5;
12    strcpy( data.str , "C_Programming" );
13    printf( "data.i : %d\n" , data.i );
14    printf( "data.f : %f\n" , data.f );
15    printf( "data.str : %s\n" , data.str );
16 }
```

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming

union: learn by example (3)

```
1 #include <stdio.h>
2 #include <string.h>
3 union Data {
4     int i;
5     float f;
6     char str[20];
7 };
8 int main(){
9     union Data data;
10    data.i = 10;
11    strcpy( data.str , "C_Programming" );
12    data.f = 220.5;
13    printf( "data.i: %d\n" , data.i );
14    printf( "data.f: %f\n" , data.f );
15    printf( "data.str: %s\n" , data.str );
16 }
```


union: learn by example (4)

```
1 #include <stdio.h>
2 #include <string.h>
3 union Data {
4     int i;
5     float f;
6     char str[20];
7 };
8 int main(){
9     union Data data;
10    data.i = 10;
11    strcpy( data.str , "C_Programming" );
12    data.f = 220.5;
13    printf( "data.i : %d\n" , data.i );
14    printf( "data.f : %f\n" , data.f );
15    printf( "data.str : %s\n" , data.str );
16 }
```

data.i : 1130135552

data.f : 220.500000

data.str :

union: learn by example (5)

```
1 #include <stdio.h>
2 #include <string.h>
3 union Data {
4     int i;
5     float f;
6     char str[20];
7 };
8 int main(){
9     data.i = 10;
10    printf( "data.i: %d\n", data.i);
11    data.f = 220.5;
12    printf( "data.f: %f\n", data.f);
13    strcpy( data.str, "C_Programming");
14    printf( "data.str: %s\n", data.str);
15 }
```

union: learn by example (6)

```
1 #include <stdio.h>
2 #include <string.h>
3 union Data {
4     int i;
5     float f;
6     char str[20];
7 };
8 int main(){
9     data.i = 10;
10    printf( "data.i : %d\n", data.i);
11    data.f = 220.5;
12    printf( "data.f : %f\n", data.f);
13    strcpy( data.str, "C_Programming");
14    printf( "data.str : %s\n", data.str);
15 }
```

data.i : 10

data.f : 220.500000

data.str : C Programming

1 struct

2 union

3 enum

- Sometimes, we feel it is more meaningful
- with symbols: Janurary, Feburary ,..., December
- than numbers: 1, 2, ..., 12
- enum allows us to do a kind of correlating
- Numbers are assigned with readable symbols

```
enum enumName{memb1, memb2, memb3,...};
```

- You enumerate all the members' name inside “{ }”
- They are symbols
- They will be related to integer 0, 1, 2,... automatically

enum: definition (2)

`enum enumName{memb1, memb2, memb3};`

```
1 enum Month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,  
    Nov, Dec};  
2 int main()  
3 {  
4     ...  
5 }
```

- You enumerate all the members' name inside “{ }”
- They are symbols
- They will be related to integer 0, 1, 2,... automatically

enum: how to use it

```
1 #include <stdio.h>
2 enum Month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
   Nov, Dec};
3 int main()
4 {
5     enum Month m;
6     m = Feb;
7     printf("Month is: %d\n", m);
8     return 0;
9 }
```

[Output]

```
1 Month is: 1
```

- **Feb** is a symbol instead of a string
- They will be related to integer 0, 1, 2,... automatically

enum: learn by example (1)

```
1 #include <stdio.h>
2 enum Week {Mon=1, Tue=1, Wed=3,
3   Thu=5, Fri, Sat=4, Sun};
4 int main()
5 {
6     enum Week wk;
7     wk=Wed;
8     printf("Wed: %d\n", wk);
9     wk=Fri;
10    printf("Fri: %d\n", wk);
11    wk=Sun;
12    printf("Sun: %d\n", wk);
13    return 0;
14 }
```

[Output]

```
1 Wed: ?
2 Fri: ?
3 Sun: ?
```

enum: learn by example (2)

```
1 #include <stdio.h>
2 enum Week {Mon=1, Tue=1, Wed=3,
3   Thu=5, Fri, Sat=4, Sun};
4 int main()
5 {
6     enum Week wk;
7     wk=Wed;
8     printf("Wed: %d\n", wk);
9     wk=Fri;
10    printf("Fri: %d\n", wk);
11    wk=Sun;
12    printf("Sun: %d\n", wk);
13    return 0;
14 }
```

[Output]

```
1 Wed: 3
2 Fri: 6
3 Sun: 5
```

- Can you figure out why??
- This way is valid, but NOT suggested

enum: learn by example (2)

```
1 #include <stdio.h>
2 enum Week {Mon=1, Tue, Wed,
3   Thu, Fri, Sat, Sun};
4 int main()
5 {
6     enum Week wk;
7     wk=Wed;
8     printf("Wed: %d\n", wk);
9     wk=Fri;
10    printf("Fri: %d\n", wk);
11    wk=Sun;
12    printf("Sun: %d\n", wk);
13    return 0;
14 }
```

[Output]

```
1 Wed: 3
2 Fri: 5
3 Sun: 7
```

- This is the right way

enum: learn by example (3)

```
1 #include <stdio.h>
2 enum Week {Mon=1, Tue, Wed,
3   Thu, Fri, Sat, Sun};
4 typedef enum Week WkType;
5 int main()
6 {
7     WkType wk;
8     wk=Wed;
9     printf("Wed: %d\n", wk);
10    wk=Fri;
11    printf("Fri: %d\n", wk);
12    wk=Sun;
13    printf("Sun: %d\n", wk);
14    return 0;
15 }
```

[Output]

```
1 Wed: 3
2 Fri: 5
3 Sun: 7
```

- You can use “`typedef`” to save up your coding efforts