中国民航大学
Civil Aviation University of China

Computer Programming

Sino-European Institute of Aviation Engineering

Module 8 Structures

Outline

- Declaring Structure Variables
- Initializing Structure Variables
- Operations on Structures
- Structure and function
- Arrays of Structure
- Pointers to Structure
- Unions
- Summary

Declaring Structure Variables

- The properties of a ***structure*** are different from those of an array.
  - The elements of a structure (its ***members***) aren’t required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.
- In some languages, structures are called ***records,*** and members are known as ***fields.***

Declaring Structure Variables

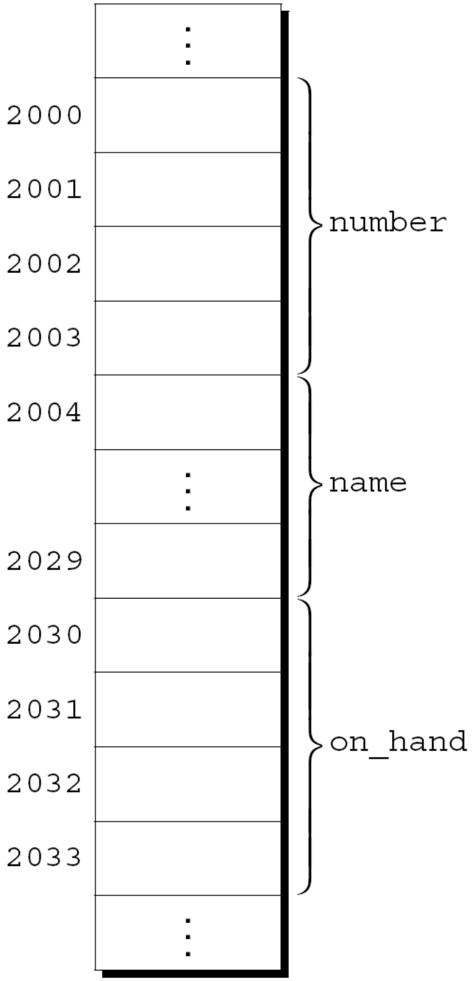
- A structure is a logical choice for storing a collection of related data items.
- A declaration of two structure variables that store information about parts in a warehouse:

```
struct  part
{
   int number;
   char name[NAME_LEN+1];
   int on_hand;
 };
sruct part  part1,part2;
```

```
struct  (part)
{
   int number;
   char name[NAME_LEN+1];
   int on_hand;
} part1, part2;
```

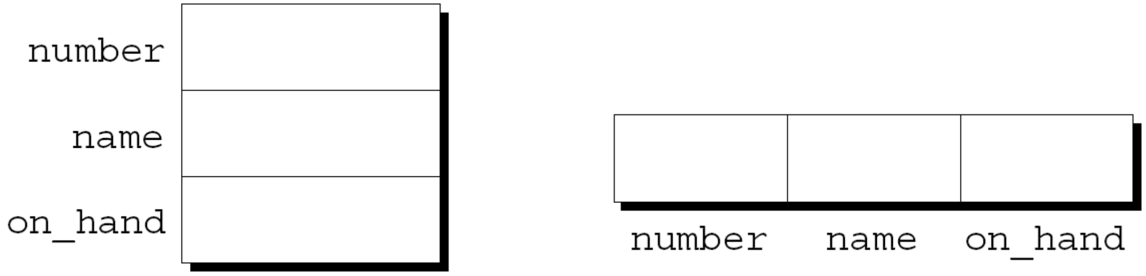
Declaring Structure Variables

- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1` →
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.



Declaring Structure Variables

- Abstract representations of a structure:



- Member values will go in the boxes later.

Declaring Structure Variables

- Each structure represents a new scope.
- Any names declared in that scope won’t conflict with other names in a program.
- In C terminology, each structure has a separate ***name space*** for its members.

Declaring Structure Variables

- For example, the following declarations can appear in the same program:

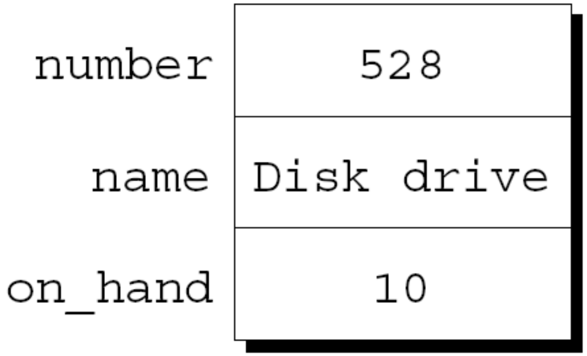
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct  (part)
 {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
 } part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```



number	528
name	Disk drive
on_hand	10

```
struct  part
 {
    int number;
    char name[NAME_LEN+1];
    int on_hand;
}
struct  part part1 = {528, "Disk drive", 10},
part2 = {914, "Printer cable", 5};
```

Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
- Expressions used in a structure initializer must be constant.
- An initializer can have fewer members than the structure it’s initializing.
- Any “leftover” members are given 0 as their initial value.

Operations on Structures

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the member.
- Statements that display the values of *part1*'s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operations on Structures

- The period used to access a structure member is actually a C operator.
- It takes precedence over nearly all other operators.
- Example:

```
scanf("%d", &part1.on_hand);
```

The . operator takes precedence over the & operator, so & computes the address of part1.on_hand.

Operations on Structures

- The other major structure operation is assignment:

  part2 = part1;

- The effect of this statement is to copy part1.number into part2.number, part1.name into part2.name, and so on.

Operations on Structures

- Arrays can’t be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.
- Some programmers exploit this property by creating “dummy” structures to enclose arrays that will be copied later:

```
struct { int a[10]; } a1, a2;
a1 = a2;
  /* legal, since a1 and a2 are structures */
```

Operations on Structures

- The = operator can be used only with structures of ***compatible*** types.
- Other than assignment, C provides no operations on entire structures.
- In particular, the == and != operators can't be used with structures.

- Two structures declared at the same time (as part1 and part2 were) are compatible.
- Structures declared using the same "structure tag" or the same type name are also compatible.

Structure and Function

- Functions may have structures as arguments and return values.
- A function with a structure argument:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

A call of print_part:

```
print_part(part1);
```

Structure and Function

- A function that returns a ***part*** structure:

```
struct part build_part(int number,
                       const char *name,
                       int on_hand)
{
  struct part p;

  p.number = number;
  strcpy(p.name, name);
  p.on_hand = on_hand;
  return p;
}
```

A call of build_part:

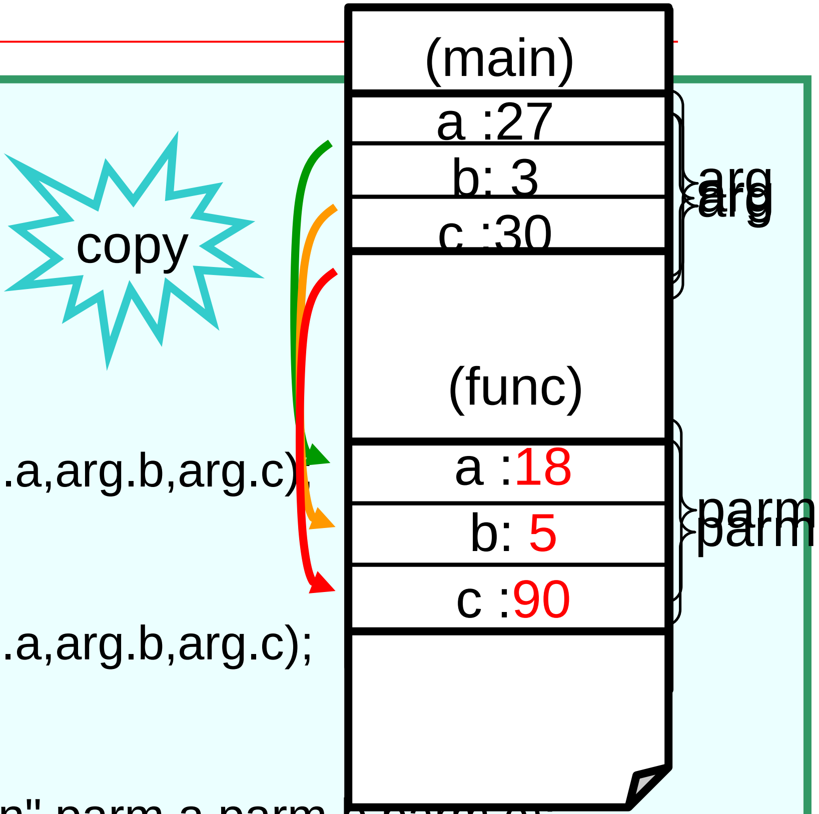
```
part1 = build_part(528, "Disk drive", 10);
```

Structure and Function

- Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure.
- To avoid this overhead, it’s sometimes advisable to pass a pointer to a structure or return a pointer to a structure.

Structure and Function

```
struct data
{   int a, b, c; };
main()
{   void func(struct data);
    struct data arg;
    arg.a=27;   arg.b=3;    arg.c=arg.a+arg.b;
    printf("arg.a=%d arg.b=%d arg.c=%d\n",arg.a,arg.b,arg.c);
    printf("Call Func()....\n");
    func(arg);
    printf("arg.a=%d arg.b=%d arg.c=%d\n",arg.a,arg.b,arg.c);
}
void func(struct data parm)
{   printf("parm.a=%d parm.b=%d parm.c=%d\n",parm.a,parm.b,parm.c);
    printf("Process...\n");
    parm.a=18;     parm.b=5;    parm.c=parm.a*parm.b;
    printf("parm.a=%d parm.b=%d parm.c=%d\n",parm.a,parm.b,parm.c);
    printf("Return...\n");
}
```



Nested Structures

- Nesting one structure inside another is often useful.
- Suppose that *person_name* is the following structure:

```
struct person_name {
  char first[FIRST_NAME_LEN+1];
  char middle_initial;
  char last[LAST_NAME_LEN+1];
};
```

Nested Structures

- We can use *person_name* as part of a larger structure:

```
struct student {
  struct person_name name;
  int id, age;
  char sex;
} student1, student2;
```

- Accessing student1’s first name, middle initial, or last name requires two applications of the . operator:

```
strcpy(student1.name.first, "Fred");
```

Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of *part* structures capable of storing information about 100 parts:

  struct part inventory[100];

Arrays of Structures

- Accessing a part in the array is done by using subscripting:

  print_part(inventory[i]);

- Accessing a member within a *part* structure requires a combination of subscripting and member selection:

  inventory[i].number = 883;

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:

inventory[i].name[0] = '\0';

Arrays of Structures

- Initializing an Array of Structures
  - Initializing an array of structures is done in much the same way as initializing a multidimensional array.
  - Each structure has its own brace-enclosed initializer; the array initializer wraps another set of braces around the structure initializers.

Arrays of Structures

- One reason for initializing an array of structures is that it contains information that won’t change during program execution.
- Example: an array that contains country codes used when making international telephone calls.
  - The elements of the array will be structures that store the name of a country along with its code:

```
struct dialing_code {
  char *country;
  int code;
};
```

Arrays of Structures

```
const struct dialing_code country_codes[] =
  {{"Argentina",            54}, {"Bangladesh",      880},
   {"Brazil",               55}, {"Burma (Myanmar)",  95},
   {"China",                86}, {"Colombia",         57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",            20},
   {"Ethiopia",            251}, {"France",           33},
   {"Germany",              49}, {"India",            91},
   {"Indonesia",            62}, {"Iran",             98},
   {"Italy",                39}, {"Japan",            81},
   {"Mexico",               52}, {"Nigeria",         234},
   {"Pakistan",             92}, {"Philippines",      63},
   {"Poland",               48}, {"Russia",            7},
   {"South Africa",         27}, {"South Korea",      82},
   {"Spain",                34}, {"Sudan",           249},
   {"Thailand",             66}, {"Turkey",           90},
   {"Ukraine",             380}, {"United Kingdom",   44},
   {"United States",         1}, {"Vietnam",          84}};
```

- The inner braces around each structure value are optional.

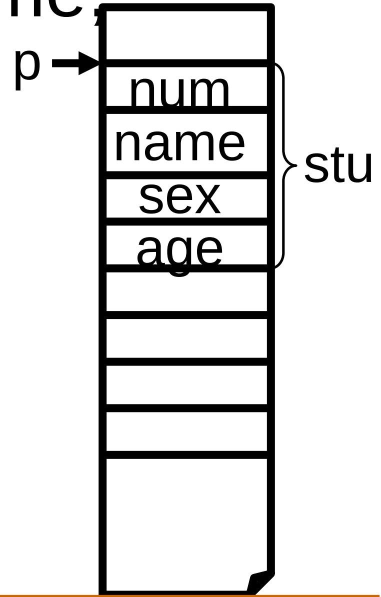
Pointers to Structures

Store the beginning address of the structure variable in the memory

- Format
  - struct structure_name *pointer_name;

    *struct student *p;*

```
struct  student
     {     int  num;
             char name[20];
             char sex;
             int age;
     }stu;
struct  student   *p=&stu;
```



```
     int    n;
     int   *p=&n;
    *p=10;       ⇔ n=10
```

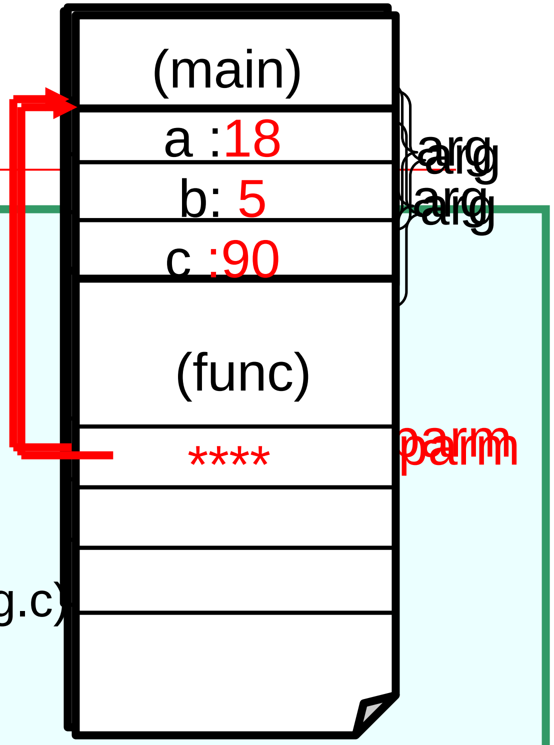
```
struct    student      stu1;
struct    student      *p=&stu1;
stu1.num=101;   ⇔   (*p).num=101
```

Pointers to Structures

- Like structure variable, functions may have pointers to structures as arguments

Argument	Content Transferred
Member of structure variable	Value
Structure variable	Multi-value
Pointers to structures	Address

Pointers to Structures



```
struct data
{   int a, b, c; };
main()
{   void func(struct data  *parm);
    struct data arg;
    arg.a=27;   arg.b=3;    arg.c=arg.a+arg.b;
    printf("arg.a=%d arg.b=%d arg.c=%d\n",arg.a,arg.b,arg.c)
    printf("Call Func()....\n");
    func(&arg);
    printf("arg.a=%d arg.b=%d arg.c=%d\n",arg.a,arg.b,arg.c);
}
void func(struct data  *parm)
{   printf("parm->a=%d parm->b=%d parm->c=%d\n",parm->a,parm->b,parm->c);
    printf("Process...\n");
    parm->a=18;     parm->b=5;    parm->c=parm->a*parm->b;
    printf("parm->a=%d parm->b=%d parm->c=%d\n",parm->a,parm->b,parm->c);
    printf("Return...\n");
}
```

Unions

- A ***union,*** like a structure, consists of one or more members, possibly of different types.
- The compiler allocates only enough space for the largest of the members, which overlay each other within this space.
- Assigning a new value to one member alters the values of the other members as well.

Unions

- An example of a union variable:

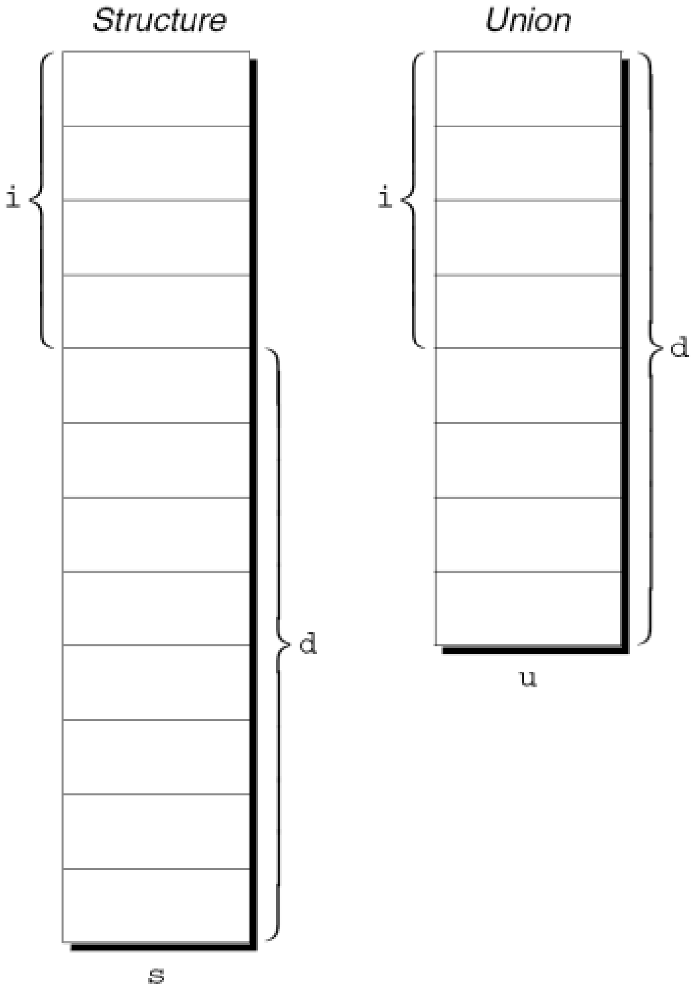
```
union {
  int i;
  double d;
} u;
```

- The declaration of a union closely resembles a structure declaration:

```
struct {
  int i;
  double d;
} s;
```

Unions

- The structure *s* and the union *u* differ in just one way.
- The members of `s` are stored at different addresses in memory.
- The members of `u` are stored at the same address.



Unions

- Initialization
  - Only be initialized with a value of the type of it’s first member

```
union
  {   int i;
        char ch;
        float f;
   }a={1,’a’,1.5};     (×)
```

```
union
  {   int i;
        char ch;
        float f;
   }a;
   a=1;           (×)
```

```
float  x;
  union
   {   int i;   char ch;  float f;
    }a,b;
    a.i=1;   a.ch=‘a’;   a.f=1.5;
    b=a;      (√)
    x=a.f;    (√)
```

Unions

- Changing one member of a union alters any value previously stored in any of the other members.
  - Storing a value in `a.ch` causes any value previously stored in `a.i` to be lost.
  - Changing `a.i` corrupts `a.ch`.

Unions

Format1:

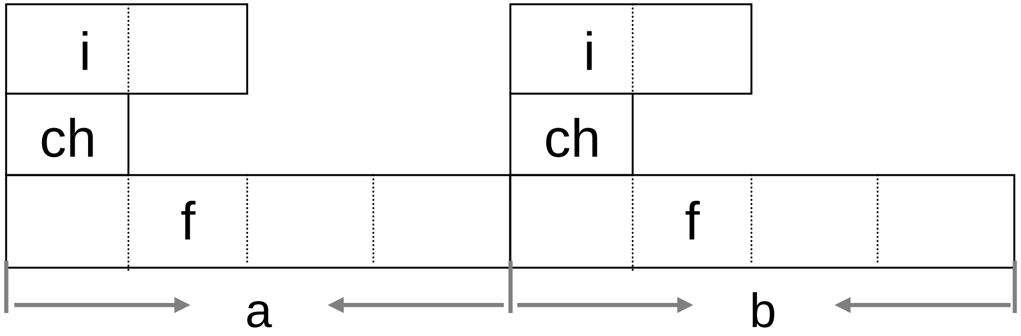
```
union data
   {    int i;
        char ch;
        float f;
    }a,b;
```

Format2:

```
union
     {    int i;
          char ch;
          float f;
      }a,b,c;
```

Format3:

```
union data
       {    int i;
            char ch;
            float f;
        };
       union data a,b,c,*p,d[3];
```



A union is a structure in which all members have offset zero from the base, the structure is big enough to hold the "widest" member, and the alignment is appropriate for all of the types in the union.

Unions

- Operation on unions

```
union data
       {   int i;
            char ch;
            float f;
        };
       union data a,b,c,*p,d[3];
```

a.i   a.ch   a.f

p->i   p->ch p->f

(*p).i   (*p).ch (*p).f

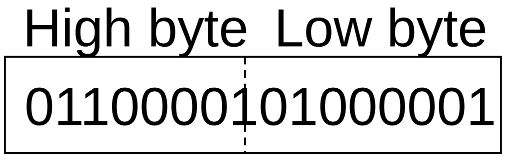
d[0].i   d[0].ch   d[0].f

Unions

- The properties of unions are almost identical to the properties of structures.
- We can declare union tags and union types in the same way we declare structure tags and types.
- Like structures, unions can be copied using the = operator, passed to functions, and returned by functions.

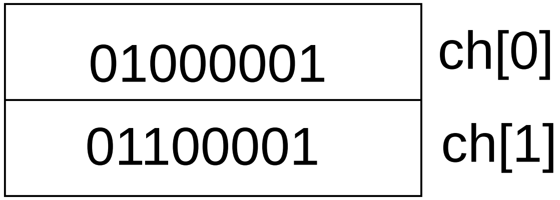
Unions

```
main()
{  union  int_char
   {   int i;
       char ch[2];
   }x;
   x.i=24897;
   printf("i=%o\n",x.i);
   printf("ch0=%o,ch1=%o\n
               ch0=%c,ch1=%c\n",
                 x.ch[0],x.ch[1],x.ch[0],x.ch[1]);
}
```



High byte Low byte

0110000101000001



01000001	ch[0]
01100001	ch[1]

Result：
i=60501
ch0=101,ch1=141
ch0=A,ch1=a

Summary

- □ Definition of structure variable and accessing of element (variable) of structure
- □ Nesting of structures
- □ Function with a structure argument or a structure return value
- □ Using array and pointer to structures

Thank you!