Structures & unions

Outline

- Structures
- Initialization
- Alignment of structure members
- Nested structures
- Bit fields
- Unions
- Passing structures as function arguments
- Returning structures from the functions
- Linked list

Structures

- A structure is like an array except that each element can have a different data type. Moreover, elements in a structure have names instead of subscript values.

- Without structures, a single person's record would be declared as:

```
char name[20], tcno[11];
short day, month, year;
```

```
strcpy(name, "John Smith");
strcpy(tcno, "01322222654");
day=26;
month=11;
year=1957;
```

- What about multiple people's records?

```
char name[1000][20], tcno[1000][11];
short day[1000], month[1000], year[1000];
```

- Three ways to define a structure
  - using a TAG name
  - without a TAG name
  - using a typedef name

```
// Define the template and var. together WITH a tag name:
struct personalstat{
   char ps_name[20], ps_tcno[11];
   short ps_day, ps_month, ps_year;
} ps, psarr[1000], *ptrps;
```

1

```
// Define the template and var. together WITHOUT a tag name:
struct {
     char ps_name[20], ps_tcno[11];
     short ps_day, ps_month, ps_year;
  } ps;
```

2

```
structure personalstat{
   char ps_name[20], ps_tcno[11];
   short ps_day, ps_month, ps_year;
};
// *Declare a variable from above template
struct personalstat ps;
struct personalstat psarr[1000], *ptrps;
ptrps=&psarr[10]; // e.g. use of pointers
```

1

```
typedef struct {
   char ps_name[20], ps_tcno[11];
   short ps_day, ps_month, ps_year;
} PERSONALSTAT;
// *Declare a variable from above template
PERSONALSTAT ps;
```

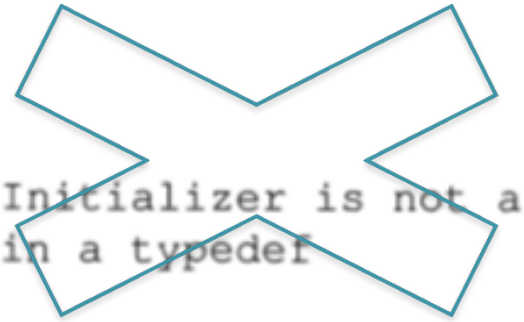
3

Initialization

PERSONALSTAT ps = { "George Smith", "002340671",
3, 5, 1946 };

PERSONALSTAT psarr[] = { {},
{}
};

```
typedef struct
{
    int a;
    float b;
} s = { 1, 1.0 };   /* Initializer is not allowed
                     * in a typedef
                     */
```



Referencing structure members & Arrays

- ps.ps_day=15;
- ps.ps_month=3;
- ps.ps_year=1987;

- If *ptrps is a pointer:
  - (*ptrps).ps_day
  - ptrps ->ps_day

- Array of structures is declared with structure's typedef name and array name:
- PERSONALSTAT psarr[10];

Array of Structures vs Pointer of Structures - I

```
#include "pstat.h" // contains declaration of
                        PERSONALSTAT typedef
//count the number of people in a certain age group
int agecount(PERSONALSTAT psarr[], int size, int
low_age, int high_age, int current_year){
    int i, age, count=0;
    for(i=0; i<size; i++){
            age=current_year - psarr[i].ps_year;
            if(age>=low_age && age<=high_age)
                count++;
    }
    return count;
}
```

```
#include "pstat.h" // contains declaration of
                        PERSONALSTAT typedef
//count the number of people in a certain age group
int agecount(PERSONALSTAT psarr[], int size, int
low_age, int high_age, int current_year){
    int i, age, count=0;
    for(i=0; i<size; ++psarr, i++){
            age=current_year - psarr->ps_year;
            if(age>=low_age && age<=high_age)
                count++;
    }
    return count;
}
```

Array of Structures vs Pointer of Structures-2

```
#include "pstat.h" // contains declaration of
                        PERSONALSTAT typedef
//count the number of people in a certain age group
int agecount(PERSONALSTAT psarr[], int size, int
low_age, int high_age, int current_year){
    int i, age, count=0;

    for(i=0; i<size; ++psarr, i++){
          age=current_year - psarr->ps_year;
          if(age>=low_age && age<=high_age)
             count++;
    }
    return count;
}
```

```
#include "pstat.h" // contains declaration of
                        PERSONALSTAT typedef
//count the number of people in a certain age group
int agecount(PERSONALSTAT psarr[], int size, int
low_age, int high_age, int current_year){
    int age, count=0;
    PERSONALSTAT *p=psarr, *plast=&psarr[size]
    for( ; p<plast; ++p){
          age=current_year - p->ps_year;
          if(age>=low_age && age<=high_age)
             count++;
    }
    return count;
}
```

Nested structures

```
typedef struct {
   char day;
   char month;
   short year;
} DATE;

typedef struct {
   char ps_name[20], ps_tcno[11];
   DATE ps_birth_date;
} PERSONALSTAT;

// *Declare an array from above definition:
PERSONALSTAT psarr[1000];
psarr[j].ps_birth_date.day=25;
```

- You are permitted to declare pointers to structures that have not yet been declared.
- This feature enables you to create self-referential structures and also to create mutually referential structures:

```
struct s1{
    int a;
    struct s2 *b;
};
```

```
struct s2{
    int a;
    struct s1 *b;
};
```

- This is known as forward referencing, is one of the few instances in C where you may use an identifier before it has been declared.
- Note that forward references are not permitted within typedefs. The following produces a syntax error:

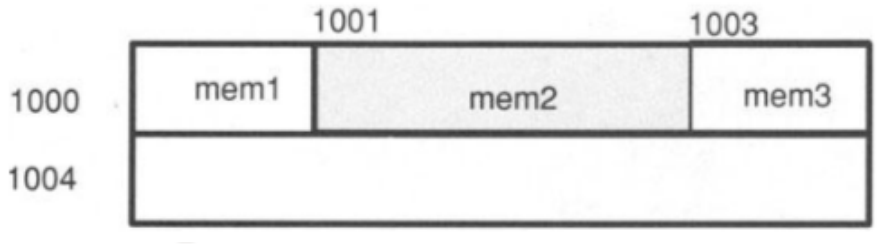
```
typedef struct {
    int a;
    FOO *ptr; // ERROR: FOO is not yet declared
} FOO;
```

Alignment of structure members

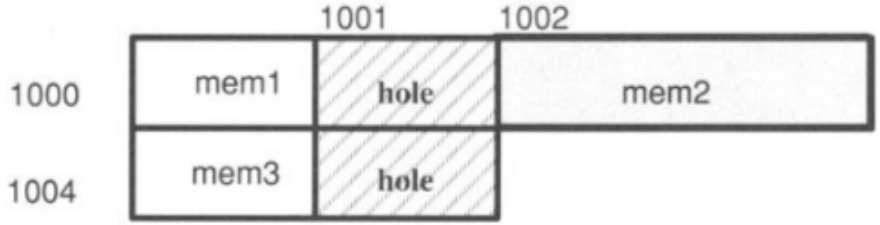
- Some computers require that any data object larger than a char must be assigned an address that is a multiple of a power of 2 (all objects > than a char be stored at even addresses).
- Normally, these alignment restrictions are invisible to the programmer. However, they can create holes, or gaps, in structures.
- Consider how a compiler would allocate memory for the following structure:

```
structure ALIGN_EXAMP{
    char mem1;
    short mem2;
    char mem3;
} s1;
```

If the computer has no alignment restrictions, s1 would be stored as:



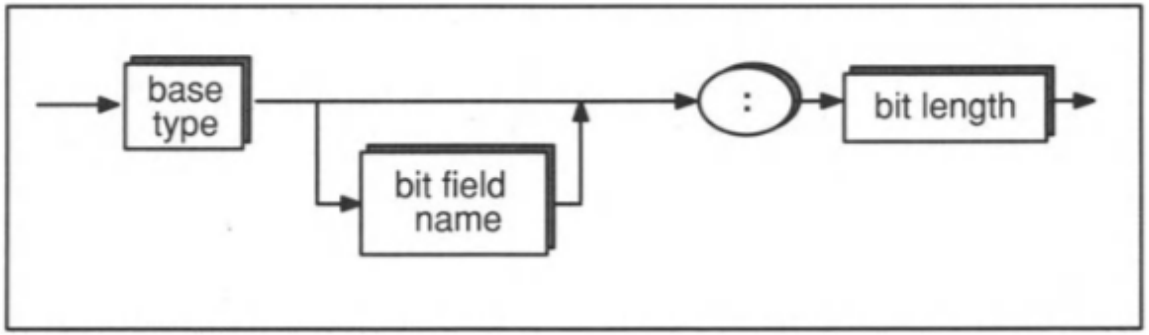
If the computer requires objects > a char to be stored at even addresses, s1 would be stored as:



*This storage arrangement results in a 1-byte hole between mem1 and mem2 and following mem3.

Bit fields

- The smallest data type that C supports is char(8 bits)
- But in structures, it is possible to declare a smaller object called a *bitfield*.
- Bit fields behave like other int variables, except that:
  - You cannot take the address of a bit field and
  - You cannot declare an array of bit fields.
- Syntax:



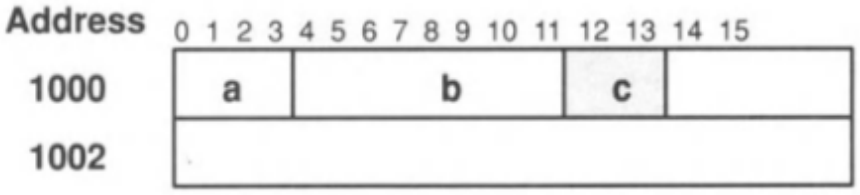
- The base type may be **int**, **unsigned int**, or **signed int**.
- If the bit field is declared as int, the implementation is free to decide whether it is an unsigned int or a signed int (**For portable code, use the signed or unsigned qualifier**).
- The *bit length* is an integer constant expression that may not exceed the length of an int.
- On machines where ints are 16 bits long, e.g. the following is illegal: int too_long: 17;

Bit fields -2

- Assuming your compiler allocates 16-bits for a bit field, the following declarations would cause *a, b,* and *c* to be packed into a single 16-bit object

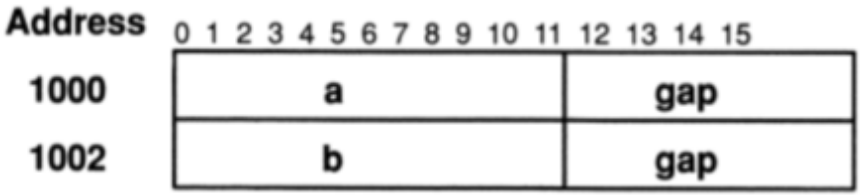
```
struct
{
   int a : 3;
   int b : 7;
   int c : 2;
} s;
```

Each implementation is free to arrange the bit fields within the object in either increasing or decreasing order



- If a bit field would located in an **int** boundary, a new memory area may be allocated, depending on your compiler. For instance, the declaration might cause a new 16-bit area of memory to be allocated for *b:*

```
struct
{
   int a : 10;
   int b : 10;
} s;
```



Bit fields -3

- Consider *DATE* structure example:

```
struct DATE{
   unsigned int day : 5;
   unsigned int month : 4;
    unsigned int year : 11;
};
```

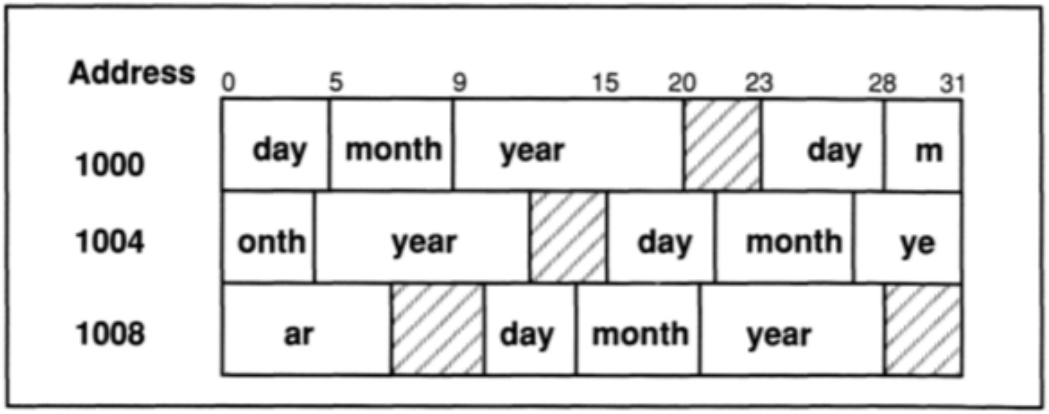
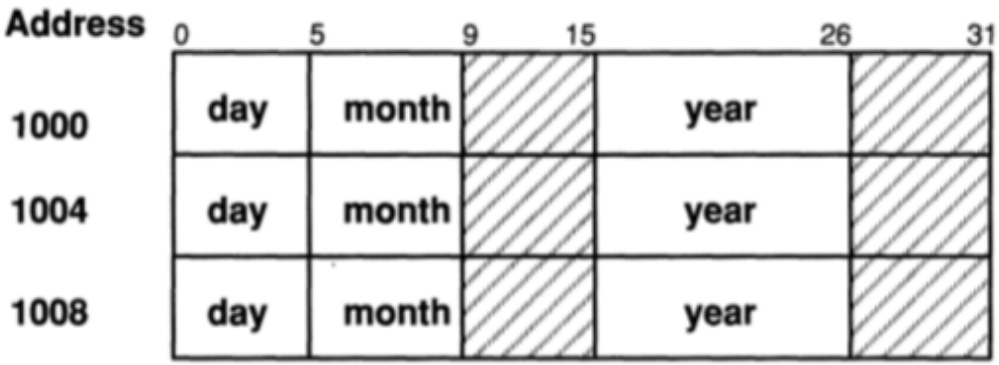


Figure 8-8. Storage of the DATE *Structure with Bit Fields. This figure assumes that the compiler packs bit fields to the nearest* **char** *and allows fields to span* **int** *boundaries.*



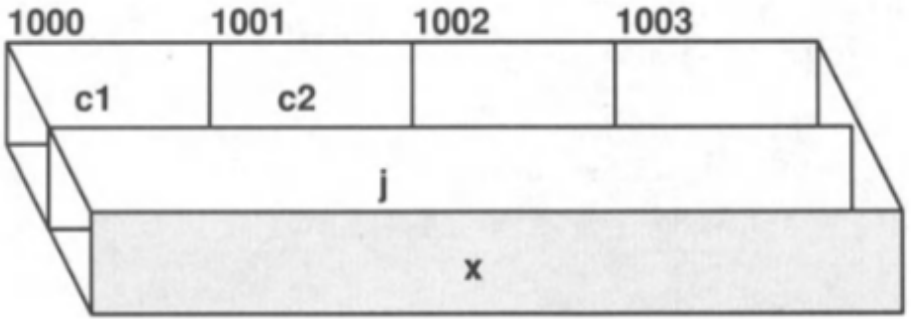
Alternative Storage of the DATE Structure with Bit Fields. This figure assumes that the compiler packs bit fields to the nearest **short** and **does not allow fields to span int boundaries**.

Unions

- Unions are similar to structures except that the members are overlaid one on top of another, so members share the same memory.
- There are two basic applications for unions:
  - Interpreting the same memory in different ways.
  - Creating flexible structures that can hold different types of data.
- Example:

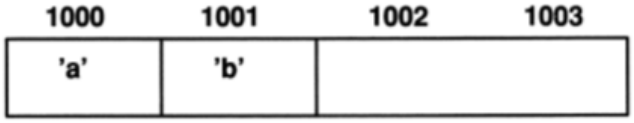
```
typedef union
{
  struct
  {
    char c1, c2;
  } s;
  long j;
  float x;
} U;

U example;
```



- Usage:

```
example.s.c1 = 'a';
example.s.c2 = 'b';
```



* If you make the assignment:
example.j = 5; // it overwrites the 2 chars, using all 4 bytes to store value 5.

Real life example for Unions in Structures

- Consider our PERSONALSTAT example (name, tcno, birth_date), we want to add additional information as follows:
  - Are you T.C. citizen?
  - If you are a T.C. citizen, in which city were you born?
  - If not a T.C. citizen, what is your nationality?

```
typedef struct {
   unsigned int day : 5;
   unsigned int month : 3;
   unsigned int year : 11;
} DATE;

typedef struct {
   char ps_name[20], ps_tcno[11];
   DATE ps_birth_date;
  // Bit field for TC citizenship:
   unsigned int TCcitizen : 1;
   char nationality[20];
   char city_of_birth[20];
} PERSONALSTAT;
```

Real life example for Unions in Structures

- Consider our PERSONALSTAT example (name, tcno, birth_date), we want to add additional information as follows:
  - Are you T.C. citizen?
  - If you are a T.C. citizen, in which city were you born?
  - If not a T.C. citizen, what is your nationality?

```
typedef struct {
   unsigned int day : 5;
   unsigned int month : 3;
   unsigned int year : 11;
} DATE;

typedef struct {
   char ps_name[20], ps_tcno[11];
   DATE ps_birth_date;
  // Bit field for TC citizenship:
   unsigned int TCcitizen : 1;
   char nationality[20];
   char city_of_birth[20];
} PERSONALSTAT;
```

```
typedef struct {
   unsigned int day : 5;
   unsigned int month : 3;
   unsigned int year : 11;
} DATE;

typedef struct {
   char ps_name[20], ps_tcno[11];
   DATE ps_birth_date;
  // Bit field for TC citizenship:
   unsigned int TCcitizen : 1;
   union{
      char nationality[20];
      char city_of_birth[20]
   } location;
} PERSONALSTAT;
```

Passing structures as function arguments

- There are two ways to pass structures as arguments:
  - **Pass the structure itself (pass by value):**

    PERSONALSTAT ps;

    …

    func(ps); // Pass by value. Passes an entire copy of the structure
  - **Pass a pointer to the structure (pass by reference):**

    …

    func(&ps); // Pass by reference. Passes the address of the structure

- Passing the address of a structure is usually faster because only a single pointer is copied to the argument area.
- Passing by value, on the other hand, requires that the entire structure be copied.
- There are only two circumstances when you should pass a structure by value:
  - The structure is very small (i.e., approximately the same size as a pointer).
  - You want to guarantee that the called function does not change the structure being passed. When an argument is passed by value, the compiler generates a copy of the argument for the called funct. The called function can only change the value of the copy

Passing structures as function arguments -2

- Depending on which method you choose, you need to declare the argument on the receiving side as either a structure or a pointer to a structure:
  - *func (PERSONALSTAT ps)* // Pass by value - the argument is a structure
  - *func (PERSONALSTAT* ptrps)* // Pass by reference - the argument is a pointer to a structure.

- **Note that** the argument-passing method you choose determines which operator you should use in the function body:
  - the dot operator if a structure is passed by value
  - the right-arrow operator if the structure is passed by reference.

Returning structures from the functions

- Just as it is possible to pass a structure or a pointer to a structure, it is also possible to return a structure or a pointer to a structure.

- As with passing structures, you generally want to return pointers to structures because it is more efficient.

- // Define a function that returns a struct:

```
struct tagname func1  (struct tagname st){
   …
   return st;  // Return an entire struct
```

- // Define a function that returns a pointer to a struct:

```
struct tagname * func2 (){
    static struct tagname pst;
    return &pst;  // Return the address of a struct
```

- Note, however, that if you return a pointer to a structure, the structure must have fixed duration. Otherwise, it may not be valid once the function returns.

Trigonometric functions example with structures