Structures

Structures

- Array
  - A derived type used to represent homogeneous data
- Structure
  - provides a means to aggregate variables of different types

a structure tag name

```
struct card {
    int pips;
    char suit;
};
```

- ✓ This declaration creates the derived data type struct card.
- ✓ A user-defined type
- ✓ Just a template, no storage allocated

Structures

-
  ```
  struct card {
          int pips;
          char suit;
  };
  struct card c1, c2;
  ```

-
  ```
  struct card {
          int pips;
          char suit;
  } c1, c2;
  ```

-
  ```
  struct card {
          int pips;
          char suit;
  };
  typedef  struct card  card;
  card c1, c2;
  ```

-
  ```
  struct card {
          int pips;
          char suit;
  } deck[52];
  ```
  - The identifier deck is declared to be an array of struct card

-
  ```
  typedef struct{
          float  re;
          float  im;
  } complex;
  complex a, b, c[100];
  ```
  - When using **typedef** to name a structure type, the tag name may be unimportant

Structure Assignment

- Structure assignment

  c1 = c2;

- Member access operator **.**

  structure_variable.member_name

  c1.pips = 3;
  c1.suit = ‘s’;

Structures

- Within a given structure, member names must be unique.
- Members in different structures can have the same name.

```
struct fruit {
        char *name;
        int   calories;
};
struct vegetable {
        char *name;
        int   calories;
};
struct fruit   a;
struct vegetable   b;


a.calories = 100;
b.calories = 120;
```

Structures

- If a tag name is not supplied, then the structure type cannot be used in later declarations.

```
struct {
    int   day, month, year;
    char day_name[4];
    char month_name[4];
} yesterday, today, tomorrow;
```

vs.

```
struct date {
    int   day, month, year;
    char day_name[4];
    char month_name[4];
};
struct date yesterday, today, tomorrow;
```

Accessing Members of a Structure

[class_info.h]

```
#define CLASS_SIZE 100
struct student{
        char *last_name;
        int   student_id;
        char grade;
};
```

[grade.c]

```
#include “class_info.h”
int main()
{
        struct student   tmp, class[CLASS_SIZE];

        tmp.grade = ‘A’;
        tmp.last_name = “Hong”;
        tmp.student_id = 910017;
        …
}
```

Accessing Members of a Structure

```
/* Count the failing grades. */

#include “class_info.h”
int fail(struct student class[])          ⇔ int fail(struct student *class)
{
    int i,cnt = 0;

    for (i=0; i<CLASS_SIZE; i++)
        cnt += class[i].grade == ‘F’;     ⇔ cnt += (class[i].grade == ‘F’);
    return cnt;
}
```

Accessing Members of a Structure

- The member access operator ->
  - access the structure members via a pointer

  pointer_to_structure* -> *member_name

  ⇔ (pointer_to_structure*).*member_name***

  ****pointer_to_structure.member_name***

  ⇔ *(*pointer_to_structure.member_name*)

Accessing Members of a Structure

[complex.h]

```
struct complex{
    double re;
    double im;
};
typedef struct complex complex;
```

[2_add.c]

```
#include “complex.h”
void add(complex *a, complex *b, complex *c)
{
    a->re = b->re + c->re;
    a->im = b->im + c->im;
}
```

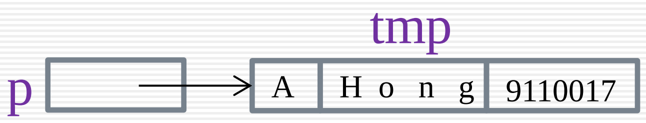
Operator Precedence and Associativity

Operator	Associativity
() [] . -> ++ *(postfix)* -- *(postfix)*	left to right
++ (*prefix*) -- (*prefix*) | ~ sizeof (*type*) + (*unary*) - (*unary*) & (*address*) * (*dereference*)	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= etc.	right to left
, (*comma operator*)	left to right

Accessing Members of a Structure

Declarations and Initializations

struct student{
 char *last_name;
 int   student_id;
 char grade;
};
struct student tmp, *p = &tmp;
tmp.grade = 'A';
tmp.last_name = "Hong";
tmp.student_id = 910017;



Expression	Equivalent expression	Value
tmp.grade	p->grade	A
tmp.last_name	p->last_name	Hong
(*p).student_id	tmp.student_id	910017
*p->last_name - 1	(*(p->last_name)) - 1	G
*(p->last_name + 2)	(p->last_name)[2]	n

Using Structures with Functions

- When a structure is passed as an argument to a function, it is passed by **value**
  - A local copy is made for use in the body of the function.
  - If a structure member is an array, the array gets copied as well.
  - **relatively inefficient !!**

```
struct dept {
        char        dept_name[25];
        int         dept_no;
}
typedef struct {
        char        name[25];
        int         employee_id;
        struct dept   department;
        struct home_address   *a_ptr;
        double      salary;
        ….
} employee_data;
```

Using Structures with Functions

```
employee_data update(employee_data r)
{
        ….
        printf(“Input the department number: “);
        scanf(“%d”, &n);
        r.department.dept_no = n;
              ⇔ (r.department).dept_no = n;
        ….
        return r;
}


employee_data e;

e = update(e);
```

```
void update(employee_data *p)
{
   ….
   printf(“Input the department number: “);
   scanf(“%d”, &n);
   p->department.dept_no = n;
              ⇔ (p->department).dept_no = n;
   ….
}


employee_data e;

update(&e);
```

Initialization of Structures

-
  ```
  struct card {
          int pips;
          char suit;
  };
  typedef  struct card  card;

  card c = {13, ‘h’};
  ```

-
  ```
  typedef struct{
          float  re;
          float  im;
  } complex;

  complex a[3][3] = {
          {{1.0, -0.1}, {2.0, 0.2}, {3.0, 0.3}},
          {{4.0, -0.4}, {5.0, 0.5}, {6.0, 0.6}}
  };      /*    a[2][] is assigned zeros     */
  ```

Unions (1/2)

- **union**
  - a derived type, following the same syntax as the structures
  - have members that share storage
  - defines a set of alternative values that may be stored in a shared portion of memory
  - The compiler allocates a piece of storage that can accommodate the largest of members.

```
union int_or_float {
      int     i;
      float   f;
 }
```

```
union int_or_float   a,  b,  c;
```

Unions (2/2)

- Bit Fields
  - An int or unsigned member of a structure or union can be declared to consist of a specified number of bits, i.e., a bit field member.
  - Width (# of bits) is specified by a nonnegative constant integral expression following a colon (:).

```
#include <stdio.h>
typedef struct {
      unsigned     b0:8, b1:8, b2:8, b3:8
}  word_bytes;

typedef struct {
      unsigned     b0:1, b1:1, b2:1, b3:1,
                    b4:1, b5:1, b6:1, b7:1,
                    b8:1, b9:1, b10:1, b11:1,
                    b12:1, b13:1, b14:1, b15:1,
                    b16:1, b17:1, b18:1, b19:1,
                    b20:1, b21:1, b22:1, b23:1,
                    b24:1, b25:1, b26:1, b27:1,
                    b28:1, b29:1, b30:1, b31:1
}  word_bits;
```

```
 typedef union {
        int   i;
        word_bits     bit;
        word_bytes  byte;
 } word;

int main(void)
{
        word      w = {0};

         w.bits.b8=1;
         w.byte.b0=‘a’;
         printf(“%d\n”, w.i);
         return 0;
}
```

00000000 00000000 00000001 01100001 (353)