STRUCTURES

Sevil ŞEN Hacettepe University December 2010

Introduction

Structures

- Collections of related variables (aggregates) under one name
 - Can contain variables of different data types
- · Commonly used to define records to be stored in files
- Combined with pointers, can create linked lists, stacks, queues, and trees

Structure Definitions

Example 1:

struct card {
 char *face;
 char *suit;
};

- struct introduces the definition for structure card
- card is the structure name and is used to declare variables of the structure type
- card contains two members of type char *
 - These members are face and suit

Structure Definitions

- A structure definition does not reserve space in memory
 - Instead creates a new data type used to define structure variables
- Variables can be defined as below: struct card {
 - char *face; char *suit; } oneCard, deck[52], *cPtr;
- Or defined like other variables:

 struct card {
 char *face;
 char *suit;
 ;
 struct card oneCard, deck[52], *cPtr;

Structure Definitions

Example 2:

```
struct point {
    int x;
    int y;
};
```

pt.x = 15; pt.y = 30; printf("%d, %d", pt.x, pt.y);

Structure Definitions

Valid Operations

- Assigning a structure to a structure of the same type
- Taking the address (&) of a structure
- Accessing the members of a structure
- Using the sizeof operator to determine the size of a structure

Structure Definitions

/* Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners. */

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

struct rect screen;

/* Print the pt1 field of screen */
printf("%d, %d",screen.pt1.x,screen.pt1.y);

/* Print the pt2 field of screen */
printf("%d, %d",screen.pt2.x,screen.pt2.y);

Initializing Structures

- Initializer lists
 - Example: card oneCard = { "Three", "Hearts" };
- Assignment statements
 - Example: card threeHearts = oneCard;
 - Could also define and initialize threeHearts as follows: card threeHearts; threeHearts.face = "Three";
 - threeHearts.suit = "Hearts";

Accessing Members of Structures

- Accessing structure members
 - Dot operator (.) used with structure variables card myCard; printf("%s", myCard.suit);
 - Arrow operator (->) used with pointers to structure variables card *myCardPtr = &myCard; printf("%s", myCardPtr->suit);
 - myCardPtr->suit is equivalent to (*myCardPtr).suit

#include <stdio.h>

```
/* card structure definition */
struct card {
    char *face; /* define pointer face */
    char *suit; /* define pointer suit */
}; /* end structure card */

Program Output:
Ace of Spades
Ace of Spades
Ace of Spades
```

int main() {

struct card a; /* define struct a */
struct card *aPtr; /* define a pointer to card */

a.face = "Ace"; a.suit = "Spades"; aPtr = &a; /* assign address of a to aPtr */

printf("%s of %s\n", a.face, a.suit);
printf("%s of %s\n", aPtr->face, aPtr->suit);
printf("%s of %s\n", (*aPtr).face, (*aPtr).suit);

return 0; /* indicates successful termination */
} /* end main */

typedef

typedef

- Creates synonyms (aliases) for previously defined data types
- Use typedef to create shorter type names

Example:

typedef struct point pixel;

• Defines a new type name **pixel** as a synonym for type **struct point**

typedef struct Card *CardPtr;

- Defines a new type name CardPtr as a synonym for type struct Card *
- typedef does not create a new data type
 - Only creates an alias

Using Structures With Functions

- Passing structures to functions
 - Pass entire structure
 - Or, pass individual members
 - Both pass call by value
- To pass structures call-by-reference
 - Pass its address
- Pass reference to it
- To pass arrays call-by-value
 - · Create a structure with the array as a member
 - · Pass the structure

```
Using Structures with Functions 1
#include<stdio.h> /* Demonstrates passing a structure to a
 function */
struct data{
   int amount:
   char fname[30]:
   char lname[30];
}rec:
void printRecord(struct data x){
  printf("\nDonor %s %s gave $%d". x.fname. x.lname. x.amount):
}
int main(void){
     printf("Enter the donor's first and last namesn");
     printf("separated by a space: ");
     scanf("%s %s",rec.fname, rec.lname);
     printf("Enter the donation amount: "):
     scanf("%d",&rec.amount);
     printRecord(rec);
     return 0:
}
 /* add two points */
 struct point addpoint (struct point p1, struct point p2)
      p1.x += p2.x;
      p1.y += p2.y;
      return p1;
 3
```

Both arguments and the return value are structures in the function addpoint.

Using Structures with Functions 2

/* Make a point from x and y components. */
struct point makepoint (int x, int y)
{
 struct point temp;

temp.x = x; temp.y = y; return (temp);

}

/* makepoint can now be used to initialize a structure */
struct rect screen;
struct point middle;

Structures and Pointers

p = &origin; printf("Origin is (%d, %d)\n", (*p).x, (*p).y);

- Parenthesis are necessary in (*p).x because the precedence of the structure member operator (dot) is higher than *.
- The expression *p.x = *(p.x) which is illegal because x is not a pointer.

Structures and Pointers

- Pointers to structures are so frequently used that an alternative is provided as a shorthand.
- If p is a pointer to a structure, then

p -> field_of_structure

refers to a particular field.

• We could write

```
printf("Origin is (%d %d)\n", p->x, p->y);
```

Assignments

struct student {
 char *last_name;
 int student_id;
 char grade;
};
struct student temp, *p = &temp;

temp.grade = 'A'; temp.last_name = "Casanova"; temp.student_id = 590017;

Expression temp.grade temp.last_name temp.student id

(*p).student id

Equiv. Expression p -> grade

p -> last name

p -> student id

p -> student id

<u>Value</u> A Casanova 590017 590017

Structures and Pointers

- Both . and -> associate from left to right
- Consider

```
struct rect r, *rp = &r;
```

• The following 4 expressions are equivalent.

```
r.ptl.x
rp -> ptl.x
(r.ptl).x
(rp->ptl).x
```

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Arrays of Structures

- Usually a program needs to work with more than one instance of data.
- For example, to maintain a list of phone #s in a program, you can define a structure to hold each person's name and number.

```
struct entry {
    char fname[10];
    char lname[12];
    char phone[8];
};
```

Arrays of Structures

- A phone list has to hold many entries, so a single instance of the entry structure isn't of much use. What we need is an array of structures of type entry.
- After the structure has been defined, you can define the array as follows:

struct entry list[1000];

```
struct entry list[1000]

list[0].fname

list[0].name

list[0].phone

list[1].fname

list[1].name

list[1].name

list[1].phone

list[999].fname[2]

list[999].fname

list[999].fname

list[999].name

list[990].name

list[990].name

list[990].name

list[990].name

list[990].name

list[990].name

list[990].name

list[990].name

list[900].name

list[900]
```

23

 To assign data in one element to another array element, you write

list[1] = list[5];

To move data between individual structure fields, you write

```
strcpy(list[1].phone, list[5].phone);
```

 To move data between individual elements of structure field arrays, you write

```
list[5].phone[1] = list[2].phone[3];
```

```
#define CLASS_SIZE 100
struct student {
       char *last_name;
       int student_id;
       char grade;
};
int main(void)
       struct student temp,
               class[CLASS_SIZE];
       ... /*Do some operation to fill class structure*/
       printf ("Number of A's in class: %d\n", countA(class));
}
int countA(struct student class[])
       int i, cnt = 0;
       for (i = 0; i < CLASS_SIZE; ++i)
               cnt += class[i].grade == 'A';
       return cnt;
}
```

• Arrays of structures can be very powerful programming tools, as can pointers to structures.

```
struct part {
    int number;
    char name [10];
};
```

```
struct part data[100];
struct part *p_part;
```

```
p_part = data;
printf("%d %s", p_part->number, p_part -> name);
```



• The above diagram shows an array named *x* that consists of 3 elements. The pointer *ptr* was initialized to point at x[0]. Each time *ptr* is incremented, it points at the next array element.

```
/* Array of structures */
#include <stdio.h>
#define MAX 4
```

struct part {
 int number;
 char name[10];

```
};
```

struct part data[MAX]= {1, "Smith", 2, "Jones", 3, "Adams", 4, "Will"};

int main (void)

```
{
```

}

```
struct part *p_part;
int count;
```

```
p_part = data;
for (count = 0; count < MAX; count++) {
    printf("\n %d %s", p_part -> number, p_part -> name);
    p_part++;
}
return 0;
```

```
Unions
```

• union

- · Memory that contains a variety of objects over time
- · Only contains one data member at a time
- Members of a union share space
- Conserves storage
- · Only the last data member defined can be accessed
- union definitions

Unions

- Valid union operations
 - Assignment to union of same type: =
 - Taking address: &
 - Accessing union members: .
 - Accessing members using pointers: ->

/* number union definition */ union number { int x; /* define int x */ double y; /* define double y */ }: /* end union number */

int main(){ union number value: /* define union value */

value.x = 100; /* put an integer into the union */

printf("Put a value in the integer member.\n");
printf(" int: %d\n double:%f\n\n", value.x, value.y);

value.y = 100.0; /* put a double into the same union */

printf("Put a value in the floating member.\n");
printf(" int: %d\n double:%f\n\n", value.x, value.y);

return 0; /* indicates successful termination */
} /* end main */

Put a value in the floating member. int: 0 double: 100.000000

Enumeration Constants

- Enumeration
 - Set of integer constants represented by identifiers
 - Enumeration constants are like symbolic constants whose values are automatically set
 - Values start at 0 and are incremented by 1
 - Values can be set explicitly with =
 - Need unique constant names
 - Example:
 - enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
 - Creates a new type enum Months in which the identifiers are set to the integers 1 to 12

#include <stdio.h>

int main()

£

enum months month; /* can contain any of the 12 months */

const char *monthName[] = { "", "January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December" };

for (month = JAN; month <= DEC; month++)
printf("%2d %11s\n", month, monthName[month]);</pre>

return 0; /* indicates successful termination */ $\}$ /* end main */

1 January 2 February 3 March 4 April 5 May 6 June 7 July 8 August 9 September 10 October 11 November 12 December

DATA STRUCTURES: LINKED LISTS

Introduction

- Dynamic data structures
 - Data structures that grow and shrink during execution
- Linked lists
 - · Allow insertions and removals anywhere
- Stacks
 - · Allow insertions and removals only at top of stack
- Queues
 - · Allow insertions at the back and removals from the front
- Binary trees
 - High-speed searching and sorting of data and efficient elimination of duplicate data items

```
Self-Referential Structures
```

- Self-referential structures
 - Structure that contains a pointer to a structure of the same type
 - Can be linked together to form useful data structures such as lists, queues, stacks and trees
 - Terminated with a NULL pointer (0)

```
struct node {
    int data;
    struct node *nextPtr;
}
```

```
    nextPtr
```

- · Points to an object of type node
- Referred to as a link
- Ties one node to another node

Dynamic Memory Allocation

- Dynamic memory allocation
 - Obtain and release memory during execution
- malloc
 - Takes number of bytes to allocate
 - Use sizeof to determine the size of an object
 - Returns pointer of type void *
 - A void * pointer may be assigned to any pointer
 - If no memory available, returns NULL
 - Example
 - newPtr = malloc(sizeof(struct node));
- free
 - Deallocates memory allocated by malloc
 - Takes a pointer as an argument
 - free (newPtr);

Linked Lists

- Linked list
 - · Linear collection of self-referential class objects, called nodes
 - · Connected by pointer links
 - · Accessed via a pointer to the first node of the list
 - Subsequent nodes are accessed via the link-pointer member of the current node
 - · Link pointer in the last node is set to NULL to mark the list's end
- Use a linked list instead of an array when
 - · You have an unpredictable number of data elements
 - Your list needs to be sorted quickly

Dynamic Memory Allocation

Two self-referential structures linked together



