Structures, Unions, and Enumerations

Based on slides from K. N. King and Dianna Xu

Bryn Mawr College
CS246 Programming Paradigm

Structures

- Structures group multiple (heterogeneous) variables
  - 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.

Structure Operations

- Structure type declaration
- Structure variable declaration
- Member assignment/reference
- Structure initialization
- Structure assignment

Structure Type (Structure Tag)

- Suppose that a program needs to declare several structure variables with identical members.
- A structure tag is a name used to identify a particular kind of structure.
- The declaration of a structure tag named part:

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

- Note that a semicolon must follow the right brace.

Structure Variables

- The part tag can be used to declare variables:

```
struct part part1, part2;
```

- We cannot drop the word struct:

```
part part1, part2;  /*** WRONG ***/
```

part isn’t a type name; without the word struct, it is meaningless.

- Since structure tags aren’t recognized unless preceded by the word struct, they don’t conflict with other names used in a program.

Declaring a Structure Tag

- The declaration of a structure tag can be combined with the declaration of structure variables:

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

- All structures declared to have type struct part are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;
part2 = part1;  /* legal; both parts have the same type */
```
Structure Representation

- Abstract representations of a structure:
  - Appearance of part 1
    - Assumptions:
      - part 1 is located at address 2000.
      - Integers occupy four bytes.
      - NAME_LEN has the value 25.
      - There are no gaps between the members.

Type Definition

- The #define directive can be used to create a “Boolean type” macro:
  ```
  #define BOOL int
  ```
- A better way to define a synonym for existing (complicated) types is to use type definition:
  ```
  typedef int Bool;
  typedef int* IntPtr;
  ```
- Array and pointer types cannot be defined as macros.
- typedef names are subject to the same scope rules as variables.

typedef and Structures

- Instead of
  ```
  struct part part1;
  ```
  use
  ```
  typedef struct part Part;
  ```
  then
  ```
  Part part1;
  ```
- Part is a new user-defined type and can be used in the same way as the built-in types.
- typedefed type names by convention have the first letter in uppercase.

Structure Variable Declaration

```c
struct part {  
  int number;  
  char name[NAME_LEN+1];  
  int on_hand;  
} part1, part2;  
```  
```c
typedef struct part {  
  int number;  
  char name[NAME_LEN+1];  
  int on_hand;  
} Part;
```  
- When it comes time to name a structure, we can usually choose either to declare a structure tag or to use typedef.

Scope of 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.

Initializing Structure Variables

- A structure declaration may include an initializer:
  ```
  struct part {  
    int number;  
    char name[NAME_LEN+1];  
    int on_hand;  
  } part1, part2;
  ```
  ```
  struct employee {  
    char name[NAME_LEN+1];  
    int number;  
    char sex;  
  } employee1, employee2;
  ```
- Appearance of part 1 after initialization:
  ```
  part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
  ```
Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
- An initializer can have fewer members than the structure it’s initializing.
- Any "leftover" members are given 0 as their initial value.
- Like array initializations, this only works at the time of declaration.
- Afterwards you must assign/initialize each member one by one.

Member Reference/Assignment

- To access a member within a structure, we write `structVar.memberName`
  ```c
  printf("Part number: %d\n", part1.number);
  printf("Part name: %s\n", part1.name);
  printf("Quantity on hand: %d\n", part1.on_hand);
  ```
- The members of a structure are lvalues.
  ```c
  part1.number = 258; /* changes part1's part number */
  part1.on_hand++ /* increments part1's quantity on hand */
  ```

Operator

- The period used to access a structure member is actually a C operator.
- It takes precedence over nearly all other operators.
- Example:
  ```c
  scanf("%d", &part1.on_hand);
  ```
The . operator takes precedence over the & operator, so & computes the address of part1.on_hand.

Structure Assignment

- The other major structure operation is assignment:
  ```c
  part2 = part1;
  ```
- The effect of this statement is to copy part1.number into part2.number, part1.name into part2.name, and so on.
- Each member’s value will be copied.
- Arrays can’t be copied using the = operator, but an array embedded within a structure is copied when the enclosing structure is copied.

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

• A function that returns a part structure:
  ```c
  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:
  ```c
  part1 = build_part(528, "Disk drive", 10);
  ```

Pointer to Structure

• Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure.
  ```c
  void updateNumOnHand(Part *b) {
    b->on_hand += 10; /* same as *(b).on_hand */
  }
  ```
  
  • To modify the original value, pass the pointer to a structure
  ```c
  int main() {
    Part a = initialization;
    updateNumOnHand (&a);
    return 0;
  }
  ```

Nested Arrays and Structures

• Structures and arrays can be combined without restriction.
  ```c
  struct person_name{
    char first[FIRST_NAME_LEN+1];
    char middle_initial;
    char last[LAST_NAME_LEN+1];
  };
  ```

  • Arrays may have structures as their elements, and structures may contain arrays and structures as members.
  ```c
  struct student {
    struct person_name name;
    int id, age;
    char sex;
  } student1, student2;
  ```

Nested Structures

• Nesting one structure inside another is often useful.
  ```c
  struct person_name {
    char first[FIRST_NAME_LEN+1];
    char middle_initial;
    char last[LAST_NAME_LEN+1];
  };
  ```

  • Accessing student1’s first name:
  ```c
  strcpy(student1.name.first, "Fred");
  ```

  • Copying the information from a person_name structure to the name member of a student structure would take one assignment instead of three:
  ```c
  struct person_name new_name;
  student1.name = new_name;
  ```
Arrays of Structures

- An array of structures can serve as a simple database.
- An array of part structures:
  ```c
  struct part inventory[100];
  ```
- Accessing a part in the array:
  ```c
  print_part(inventory[i]);
  ```
- Accessing a member within a part structure:
  ```c
  inventory[i].number = 883;
  ```
- Accessing a single character in a part name:
  ```c
  inventory[i].name[0] = '\0';
  ```

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.
- Example: an array that contains country codes used when making international telephone calls.
  ```c
  const struct dialing_code country_codes[] = {
    {"Argentina", 54}, {"Bangladesh", 880},
    {"Brazil", 55}, {"Burma [Myanmar]", 95},
    {"China", 860}, {"Colombia", 57},
    {"Ethiopia", 251}, {"France", 33},
    {"Germany", 49}, {"India", 91},
    {"Indonesia", 62}, {"Iran", 98},
    {"Italy", 39}, {"Japan", 81},
    {"Kenya", 254}, {"Liberia", 231},
    {"Malaysia", 60}, {"Morocco", 212},
    {"Mozambique", 358}, {"Nepal", 977},
    {"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.

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 – Member Access

- Members of a union are accessed in the same way as members of a structure:
  ```c
  u.i = 82;
  u.d = 74.8;
  ```
- Changing one member of a union alters any value previously stored in any of the other members.
  o Storing a value in u.d causes any value previously stored in u.i to be lost.
  o Changing u.i corrupts u.d.

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.
Initializing Unions

• Only the first member of a union can be given an initial value.
• How to initialize the \( i \) member of \( u \) to 0:

```c
union {
  int i;
  double d;
} u = {0};
```

Using Unions to Save Space

• If \( c \) is a `catalog_item` structure that represents a book, we can print the book’s title in the following way:

```c
printf("%s", c.item.book.title);
```
• As this example shows, accessing a union that’s nested inside a structure can be awkward.

Using Unions to Save Space

```c
struct catalog_item {
  int stock_number;
  double price;
  int item_type;
  union {
    struct {
      char title[TITLE_LEN+1];
      char author[AUTHOR_LEN+1];
      int num_pages;
    } book;
    struct {
      char design[DESIGN_LEN+1];
      int sizes;
    } mug;
    struct {
      char design[DESIGN_LEN+1];
      int colors;
    } shirt;
  } item;
};
```

Enumerations

• A special type in C whose values are enumerated by the programmer
• A way to group a set of related `defines`

```c
#define SUIT int
#define CLUB 0
#define DIAMOND 1
#define HEART 2
#define SPADE 3
```

```c
typedef enum {CLUB, DIAMOND, HEART, SPADE} Suit;
```
• If unspecified, enums by default start from 0 and increment by 1

```c
enum REDSUIT {HEART=10, DIAMOND=1};
enum EGA {BLACK, LTGRAY=7, DKGRAY, WHITE=15};
```

```c
void print(Number n){
  switch(n.type) {
    case(INTEGER):
      printf("%d", n.u.i);
    case(FLOAT):
      printf("%f", n.u.f);
  }
}
```

Enumerations

• All enums are integers.
• More flexible `enum`
  ○ Specify values:
  ```c
  enum REDSUITS {HEART=10, DIAMOND=1};
  ```
  ○ If no value specified, value is 1 greater than the previous constant (first constant is by default 0):
  ```c
  enum SUITS [BLACK, LIGHTGRAY=7, DARKGRAY, WHITE=15];
  ```
  ○ C allows mixing `enum` and `int`
  ```c
  enum {CLUB, DIAMOND, HEART, SPADE} a;
  int i = DIAMOND; // i is 1
  a = 2; // a is HEART
  i++; // i is HEART
  ```
Enumerations

• The names of enumeration constants must be different from other identifiers declared in the enclosing scope.
• Enumeration constants are similar to constants created with the #define directive, but they’re not equivalent.
• If an enumeration is declared inside a function, its constants won’t be visible outside the function.

Enumeration Tags and Type Names

• As with structures and unions, to name an enumeration:
  o by declaring a tag
  o by using typedef to create a genuine type name.
• Enumeration tags:
  enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
  enum suit s1, s2;
• Use typedef to make Suit a type name:
  typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;
  Suit s1, s2;
  typedef enum {FALSE, TRUE} Bool;

Enumerations as Integers

• Enumeration values can be mixed with ordinary integers:
  int i;
  enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;
  i = DIAMONDS; /* i is now 1 */
  s = 0; /* s is now 0 (CLUBS) */
  s++; /* s is now 1 (DIAMONDS) */
  i = s + 2; /* i is now 3 */
• s is treated as a variable of some integer type.
• CLUBS, DIAMONDS, HEARTS, and SPADES are names for the integers 0, 1, 2, and 3.

Enumerations as Integers

• It’s dangerous to use an integer as an enumeration value.
• For example, we might accidentally store the number 4—which doesn’t correspond to any suit—into s.

Using Enumerations to Declare “Tag Fields”

• Enumerations are perfect for determining which member of a union was the last to be assigned a value.
• In the Number structure, we can make the kind member an enumeration instead of an int:
  typedef struct {
    enum {INT_KIND, DOUBLE_KIND} kind;
    union {
      int i;
      double d;
    } u;
  } Number;

Using Enumerations to Declare “Tag Fields”

• The new structure is used in exactly the same way as the old one.
• Advantages of the new structure:
  o Does away with the INT_KIND and DOUBLE_KIND macros
  o Makes it obvious that kind has only two possible values: INT_KIND and DOUBLE_KIND