

Lecture 16 - Structures, Unions and Enumerations

Meng-Hsun Tsai
CSIE, NCKU

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**.

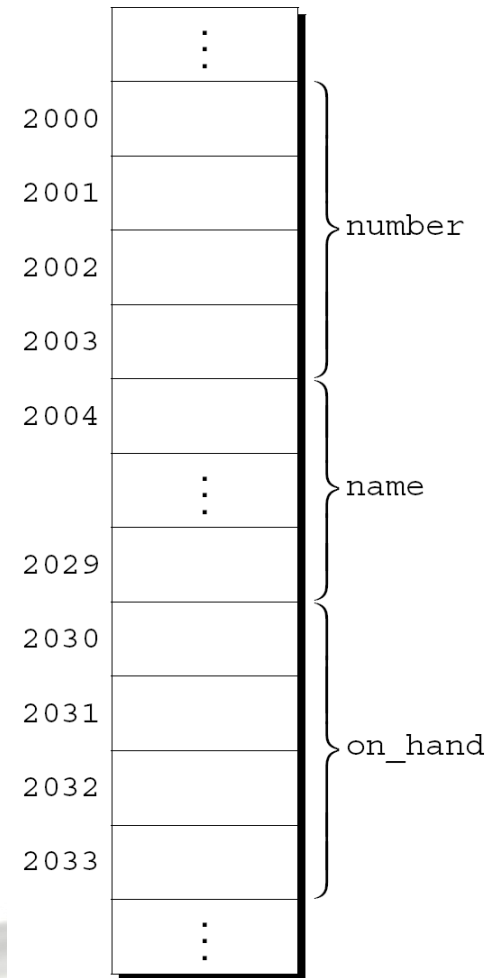
Declaring Structure Variables

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

Declaring Structure Variables (cont.)

- The members of a structure are stored in memory **in the order** in which **they're declared**.
- Appearance of `part1` \longrightarrow
- 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 (cont.)

- Each structure represents a new scope.
- Any names declared in that scope won't conflict with other names in a program.
- For example, the following two 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 {  
    int number;  
    char name[NAME_LEN+1];  
    int on_hand;  
} part1 = {528, "Disk drive", 10},  
   part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization:

number	528
name	Disk drive
on_hand	10

Initializing Structure Variables (cont.)

- Structure initializers follow **rules similar to** those for **array initializers**.
- **Expressions** used in a structure initializer **must be constant**. (This restriction is relaxed in C99.)
- An initializer **can have fewer members** than the structure it's initializing.
- Any **"leftover" members are given 0** as their initial value.

Designated Initializers (C99)

- The initializer for `part1` shown in the previous example:

```
{528, "Disk drive", 10}
```

- In a **designated initializer**, each value would be **labeled by the name of the member** that it initializes:

```
{.number = 528, .name = "Disk drive", .on_hand = 10}
```

- The combination of the **period and the member name** is called a ***designator***.

Designated Initializers (C99) (cont.)

- Designated initializers are **easier to read and check** for correctness.
- Also, values in a designated initializer **don't have to be placed in the same order** that the members are listed in the structure.
- The programmer **doesn't have to remember the order** in which the members were originally declared.
- The **order** of the members **can be changed in the future** without affecting designated initializers.

Designated Initializers (C99) (cont.)

- Not all values listed in a designated initializer need be prefixed by a designator.

- Example:

```
{.number = 528, "Disk drive", .on_hand = 10}
```

The compiler assumes that "Disk drive" initializes the member that follows `number` in the structure.

- Any members that the initializer fails to account for are set to zero.

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

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 (cont.)

- The members of a structure are **lvalues**.
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
```

```
/* changes part1's part number */
```

```
part1.on_hand++;
```

```
/* increments part1's quantity on hand */
```

Operations on Structures (cont.)

- 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 (cont.)

- 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 (cont.)

- 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 */
```

Structure Types

- Suppose that a program needs to declare several structure variables with identical members.
- We need a name that represents a type of structure, not a particular structure variable.
- Ways to name a structure:
 - Declare a "structure tag"
 - Use typedef to define a type name

Declaring a Structure Tag

- 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.

Declaring a Structure Tag (cont.)

- The `part` tag can be used to declare variables:

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

- We can't 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 (cont.)

- 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; the same type */
```

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
- A definition of a type named `Part`:

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

- `Part` can be used in the same way as the built-in types:

```
Part part1, part2;
```

Structures as Arguments and Return Values

- 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);
```

Structures as Arguments and Return Values (cont.)

- 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);
```

Structures as Arguments and Return Values (cont.)

- 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.
- Lecture 17 gives examples of functions that have a pointer to a structure as an argument and/or return a pointer to a structure.

Structures as Arguments and Return Values (cont.)

- There are other reasons to avoid copying structures.
- For example, the `<stdio.h>` header defines a type named `FILE`, which is typically a structure.
- Each `FILE` structure stores information about the state of an open file and therefore must be unique in a program.
- Every function in `<stdio.h>` that opens a file returns a pointer to a `FILE` structure.
- Every function that performs an operation on an open file requires a `FILE` pointer as an argument.

Structures as Arguments and Return Values (cont.)

- Within a function, the initializer for a structure variable can be another structure:

```
void f(struct part part1)
{
    struct part part2 = part1;
    ...
}
```

- The **structure being initialized** must have **automatic storage duration**.

Compound Literals (C99)

- A ***compound literal*** can be used to create a structure “on the fly,” without first storing it in a variable.
- The resulting structure can be passed as a parameter, returned by a function, or assigned to a variable.

Compound Literals (C99)

- A **compound literal** can be used to **create a structure** that will be **passed to a function**:

```
print_part((struct part) {528, "Disk drive", 10});
```

The compound literal is shown in **bold**.

- A compound literal can also **be assigned to a variable**:

```
part1 = (struct part) {528, "Disk drive", 10};
```

- A compound literal consists of a **type name within parentheses, followed by a set of values in braces**.

Compound Literals (C99) (cont.)

- A compound literal may contain designators, just like a designated initializer:

```
print_part((struct part) {.on_hand = 10,  
                          .name = "Disk drive",  
                          .number = 528});
```

- A compound literal may fail to provide full initialization, in which case any **uninitialized members default to zero**.

Nested Arrays and Structures

- Structures and arrays can be combined without restriction.
- Arrays may have structures as their elements, and structures may contain arrays and structures as members.

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 (cont.)

- 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");
```

Nested Structures (cont.)

- Having `name` be a structure **makes it easier to treat names as units of data.**
- A function that displays a name could be passed one `person_name` argument instead of three arguments:
`display_name(student1.name);`
- Copying the information from a `person_name` structure to the `name` member of a `student` structure would take **one assignment** instead of three:

```
struct person_name new_name;  
...  
student1.name = new_name;
```


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 (cont.)

- 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';
```

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.

Initializing an Array of Structures (cont.)

- 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;  
};
```

Initializing an Array of Structures (cont.)

```
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**.

Initializing an Array of Structures (cont.)

- C99's designated initializers **allow** an item to have **more than one designator**.
- A declaration of the `inventory` array that uses a designated initializer to create a single part:

```
struct part inventory[100] =  
    { [0].number = 528, [0].on_hand = 10,  
      [0].name[0] = '\0' };
```

Program: Maintaining a Parts Database

- The `inventory.c` program illustrates **how nested arrays and structures are used** in practice.
- The program **tracks parts stored in a warehouse.**
- Information about the parts is **stored in an array of structures.**

- **Contents of each structure:**

- Part number
- Name
- Quantity

`inventory[0]`

`inventory[1]`

number	name	on_hand
528	Disk drive	10
914	Printer cable	5
...

Program: Maintaining a Parts Database (cont.)

- Operations supported by the program:
 - Add a new part number, part name, and initial quantity on hand
 - Given a part number, print the name of the part and the current quantity on hand
 - Given a part number, change the quantity on hand
 - Print a table showing all information in the database
 - Terminate program execution

Program: Maintaining a Parts Database (cont.)

- The codes *i* (**insert**), *s* (**search**), *u* (**update**), *p* (**print**), and *q* (**quit**) will be used to represent these operations.
- A session with the program:

```
Enter operation code: i  
Enter part number: 528  
Enter part name: Disk drive  
Enter quantity on hand: 10
```

```
Enter operation code: s  
Enter part number: 528  
Part name: Disk drive  
Quantity on hand: 10
```

Program: Maintaining a Parts Database (cont.)

Enter operation code: s

Enter part number: 914

Part not found.

Enter operation code: i

Enter part number: 914

Enter part name: Printer cable

Enter quantity on hand: 5

Enter operation code: u

Enter part number: 528

Enter change in quantity on hand: -2

Program: Maintaining a Parts Database (cont.)

Enter operation code: s

Enter part number: 528

Part name: Disk drive

Quantity on hand: 8

Enter operation code: p

Part Number	Part Name	Quantity on Hand
528	Disk drive	8
914	Printer cable	5

Enter operation code: q

Program: Maintaining a Parts Database (cont.)

- The program will store information about each part in a structure.
- The structures will be **stored in an array** named `inventory`.
- A **variable** named `num_parts` will keep track of the **number of parts currently stored** in the array.

Program: Maintaining a Parts Database (cont.)

- An outline of the program's main loop:

```
for (;;) {  
    prompt user to enter operation code;  
    read code;  
    switch (code) {  
        case 'i': perform insert operation; break;  
        case 's': perform search operation; break;  
        case 'u': perform update operation; break;  
        case 'p': perform print operation; break;  
        case 'q': terminate program;  
        default: print error message;  
    }  
}
```

Program: Maintaining a Parts Database (cont.)

- Separate functions will perform the insert, search, update, and print operations.
- Since the functions will all need access to `inventory` and `num_parts`, these variables will be external.
- The program is split into three files:
 - `inventory.c` (the bulk of the program)
 - `readline.h` (contains the prototype for the `read_line` function)
 - `readline.c` (contains the definition of `read_line`)

Program: Maintaining a Parts Database (cont.)

`inventory.c`

```
#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

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

int num_parts = 0;    /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

Program: Maintaining a Parts Database (cont.)

```
int main(void)
{
    char code;
    for (;;) {
        printf("Enter operation code: ");
        scanf(" %c", &code);
        while (getchar() != '\n')    /* skips to end of line */
            ;
        switch (code) {
            case 'i': insert();
                       break;
            case 's': search();
                       break;
            case 'u': update();
                       break;
            case 'p': print();
                       break;
            case 'q': return 0;
            default: printf("Illegal code\n");
        }
        printf("\n");
    }
}
```


Program: Maintaining a Parts Database (cont.)

```
int find_part(int number)
{
    int i;

    for (i = 0; i < num_parts; i++)
        if (inventory[i].number == number)
            return i;
    return -1;
}

void insert(void)
{
    int part_number;

    if (num_parts == MAX_PARTS) {
        printf("Database is full; can't add more parts.\n");
        return;
    }
}
```

Program: Maintaining a Parts Database (cont.)

```
printf("Enter part number: ");
scanf("%d", &part_number);
if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
}
// insert at the end
inventory[num_parts].number = part_number;
printf("Enter part name: ");
read_line(inventory[num_parts].name, NAME_LEN);
printf("Enter quantity on hand: ");
scanf("%d", &inventory[num_parts].on_hand);
num_parts++;
}
```

Program: Maintaining a Parts Database (cont.)

```
void search(void)
{
    int i, number;

    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0) {
        printf("Part name: %s\n", inventory[i].name);
        printf("Quantity on hand: %d\n", inventory[i].on_hand);
    } else
        printf("Part not found.\n");
}
```

Program: Maintaining a Parts Database (cont.)

```
void update(void)
{
    int i, number, change;

    printf("Enter part number: ");
    scanf("%d", &number);
    i = find_part(number);
    if (i >= 0) {
        printf("Enter change in quantity on hand: ");
        scanf("%d", &change);
        inventory[i].on_hand += change;
    } else
        printf("Part not found.\n");
}
```

Program: Maintaining a Parts Database (cont.)

```
void print(void)
{
    int i;

    printf("Part Number    Part Name                "
           "Quantity on Hand\n");
    for (i = 0; i < num_parts; i++)
        printf("%7d          %-25s%11d\n", inventory[i].number,
               inventory[i].name, inventory[i].on_hand);
}
```

Enter operation code: p

Part Number	Part Name	Quantity on Hand
528	Disk drive	8
914	Printer cable	5

Program: Maintaining a Parts Database (cont.)

- The version of `read_line` in Lecture 13 won't work properly in the current program.
- Consider what happens when the user inserts a part:

Enter part number: 528

Enter part name: Disk drive

- The user presses the Enter key after entering the part number, leaving an invisible new-line character that the program must read.
- When `scanf` reads the part number, it consumes the 5, 2, and 8, but leaves the new-line character unread.

Program: Maintaining a Parts Database

- If we try to read the part name using the original `read_line` function, it will encounter the new-line character immediately and stop reading.
- This problem is common when numerical input is followed by character input.
- One solution is to write a version of `read_line` that skips white-space characters before it begins storing characters.
- This solves the new-line problem and also allows us to avoid storing blanks that precede the part name.

Program: Maintaining a Parts Database (cont.)

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/*****
 * read_line: Skips leading white-space characters, then
 *             reads the remainder of the input line and
 *             stores it in str. Truncates the line if its
 *             length exceeds n. Returns the number of
 *             characters stored.
 *****/
int read_line(char str[], int n);

#endif
```


Program: Maintaining a Parts Database (cont.)

```
                                readline.c
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
    int ch, i = 0;

    while (isspace(ch = getchar()))
        ;
    while (ch != '\n' && ch != EOF) {
        if (i < n)
            str[i++] = ch;
        ch = getchar();
    }
    str[i] = '\0';
    return i;
}
```

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 (cont.)

- An example of a union variable:

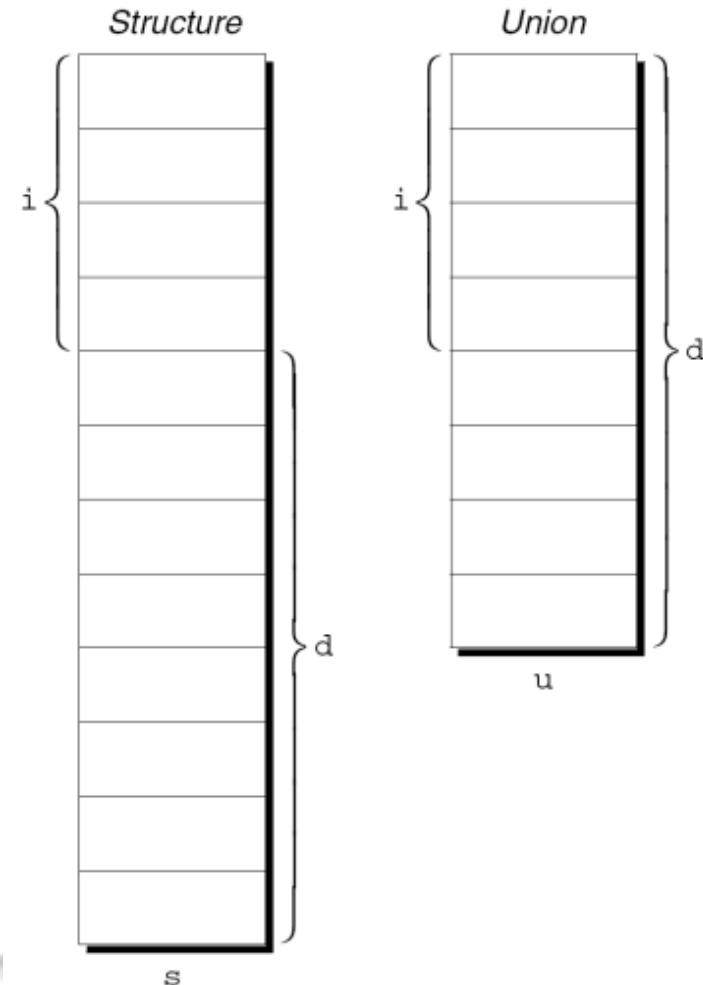
```
union {  
    int i;  
    double d;  
} u;
```

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

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

Unions (cont.)

- 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 (cont.)

- Members of a union are **accessed in the same way** as members of a structure:

`u.i = 82;`

`u.d = 74.8;`

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

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

Unions (cont.)

- 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 (cont.)

- Only the first member of a union can be given an initial value.
- How to initialize the `i` member of `u` to 0:

```
union {  
    int i;  
    double d;  
} u = {0};
```
- The expression inside the braces **must be constant**.
(The rules are slightly different in C99.)

Unions (cont.)

- Designated initializers can also be used with unions.
- A designated initializer allows us to specify which member of a union should be initialized:

```
union {  
    int i;  
    double d;  
} u = {.d = 10.0};
```

- Only one member can be initialized, but it doesn't have to be the first one.

Unions (cont.)

- Applications for unions:
 - Saving space
 - Building mixed data structures
 - Viewing storage in different ways

Using Unions to Save Space

- Unions can be used to save space in structures.
- Suppose that we're designing a structure that will contain information about an item that's sold through a gift catalog.
- Each item has a **stock number** and a **price**, as well as **other information** that **depends on the type** of the item:

Books: Title, author, number of pages

Mugs: Design

Shirts: Design, colors available, sizes available

Using Unions to Save Space (cont.)

- The `item_type` member would have one of the values `BOOK`, `MUG`, or `SHIRT`.
- This structure **wastes space**, since **only part of the information** in the structure **is common** to all items in the catalog.

```
struct catalog_item {  
    int stock_number;  
    double price;  
    int item_type; //BOOK, MUG or SHIRT  
    char title[TITLE_LEN+1];  
    char author[AUTHOR_LEN+1];  
    int num_pages;  
    char design[DESIGN_LEN+1];  
    int colors;  
    int sizes;  
};
```

Diagram illustrating the structure layout and item types:

- Blue arrows point from the `title`, `author`, and `num_pages` fields to the label **BOOK**.
- Blue arrows point from the `design`, `colors`, and `sizes` fields to the label **SHIRT**.
- A red arrow points from the `design` field to the label **MUG**.

Using Unions to Save Space (cont.)

- By putting a union inside the `catalog_item` structure, we can reduce the space required by the structure.

```
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];  
        } mug;  
        struct {  
            char design[DESIGN_LEN+1];  
            int colors;  
            int sizes;  
        } shirt;  
    } item;  
};
```

Using Unions to Save Space (cont.)

- If `c` is a `catalog_item` structure that represents a book, we can print the book's title in the following way:

```
printf("%s",  
c.item.book.title);
```

```
struct catalog_item {  
    ...  
    union {  
        struct {  
            char title[TITLE_LEN+1];  
            char author[AUTHOR_LEN+1];  
            int num_pages;  
        } book;  
        struct {  
            ...  
        } mug;  
        struct {  
            ...  
        } shirt;  
    } item;  
} c;
```

Using Unions to Save Space (cont.)

- The **union** embedded in the `catalog_item` structure contains three structures as members.
- Two of these (`mug` and `shirt`) begin with a matching member (`design`).
- Now, suppose that we assign a value to one of the `design` members:

```
strcpy(c.item.mug.design, "Cats");
```

- The `design` member in the other structure will be defined and have the same value:

```
printf("%s", c.item.shirt.design); // prints "Cats"
```

Using Unions to Build Mixed Data Structures

- Unions can be used to create **data structures** that **contain a mixture** of data of **different types**.
- Suppose that we need **an array** whose **elements** are a **mixture of `int` and `double` values**.
- First, we define a union type whose members represent the different kinds of data to be stored in the array:

```
typedef union {  
    int i;  
    double d;  
} Number;
```

Using Unions to Build Mixed Data Structures (cont.)

- Next, we create **an array** whose elements are `Number` values:

```
Number number_array[1000];
```

- A `Number union` can store either an `int` value or a `double` value.
- This makes it possible to store a mixture of `int` and `double` values in `number_array`:

```
number_array[0].i = 5;  
number_array[1].d = 8.395;
```


Adding a “Tag Field” to a Union

- There's **no easy way to tell which member** of a union **was last changed** and therefore contains a meaningful value.
- Consider the problem of writing **a function** that **displays the value** stored in a `Number` union:

```
void print_number(Number n)
{
    if (n contains an integer)
        printf("%d", n.i);
    else
        printf("%g", n.d);
}
```

There's **no way** for `print_number` to determine whether `n` contains an integer or a floating-point number.

Adding a “Tag Field” to a Union (cont.)

- In order to keep track of this information, we can embed the union within a structure that has one other member: a “tag field” or “discriminant.”
- The purpose of a tag field is to remind us what’s currently stored in the union.
- `item_type` served this purpose in the `catalog_item` structure.

Adding a “Tag Field” to a Union (cont.)

- The `Number` type as a structure with an embedded union:

```
#define INT_KIND 0
#define DOUBLE_KIND 1

typedef struct {
    int kind;    /* tag field */
    union {
        int i;
        double d;
    } u;
} Number;
```

- The **value of** `kind` will be either `INT_KIND` **or** `DOUBLE_KIND`.

Adding a “Tag Field” to a Union (cont.)

- Each time we assign a value to a member of `u`, we'll also change `kind` to remind us which member of `u` we modified.
- An example that assigns a value to the `i` member of `u`:

```
n.kind = INT_KIND;  
n.u.i = 82;
```

`n` is assumed to be a `Number variable`.

Adding a “Tag Field” to a Union (cont.)

- When the number stored in a `Number` variable is retrieved, **kind will tell us which member** of the union **was the last to be assigned** a value.
- A function that takes advantage of this capability:

```
void print_number(Number n)
{
    if (n.kind == INT_KIND)
        printf("%d", n.u.i);
    else
        printf("%g", n.u.d);
}
```

n is struct, u is union

```
void print_number(Number n)
{
    if (n contains an integer)
        printf("%d", n.i);
    else
        printf("%g", n.d);
}
```

n is union

Enumerations

- In many programs, we'll need variables that **have only a small set of meaningful values**.
- A variable that stores the **suit** of a playing card should **have only four potential values**: "**clubs**," "**diamonds**," "**hearts**," and "**spades**."

Enumerations (cont.)

- A “suit” variable can be declared as an integer, with a set of codes that represent the possible values of the variable:

```
int s;    /* s will store a suit */
```

```
...
```

```
s = 2;    /* 2 represents "hearts" */
```

- Problems with this technique:
 - We can't tell that `s` has only four possible values.
 - The significance of 2 isn't apparent.

Enumerations (cont.)

- Using macros to define a suit “type” and names for the various suits is a step in the right direction:

```
#define SUIT      int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS    2
#define SPADES    3
```

- An updated version of the previous example:

```
SUIT s;
...
s = HEARTS;
```

```
int s;
...
s = 2;
```


Enumerations (cont.)

- **Problems** with this technique:
 - There's **no indication** to someone reading the program that **the macros represent values of the same "type."**
 - If the **number of possible values** is **more** than a few, defining a separate macro for each will be **tedious**.
 - The names **CLUBS, DIAMONDS, HEARTS, and SPADES** **will be removed by the preprocessor**, so they **won't be available during debugging**.

Enumerations (cont.)

- C provides a special kind of type designed specifically for variables that **have a small number of possible values**.
- An **enumerated type** is a type whose **values are listed** ("enumerated") by the programmer.
- Each value must have a name (an **enumeration constant**).

Enumerations (cont.)

- Although enumerations have little in common with structures and unions, they're declared in a similar way:

```
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
```

- The **names of enumeration constants must be different from other identifiers** declared in the enclosing scope.

Enumerations (cont.)

- 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, there are **two ways to name an enumeration**: by **declaring a tag** or by **using typedef** to create a genuine type name.
- Enumeration tags resemble structure and union tags:

```
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```
- `suit` variables would be declared in the following way:

```
enum suit s1, s2;
```

Enumeration Tags and Type Names (cont.)

- As an alternative, we could use `typedef` to make `Suit` a type name:

```
typedef enum {CLUBS, DIAMONDS, HEARTS, SPADES} Suit;  
Suit s1, s2;
```

- In C89, **using** `typedef` to name an enumeration is an excellent way **to create a Boolean type**:

```
typedef enum {FALSE, TRUE} Bool;
```

Enumerations as Integers

- Behind the scenes, **C** treats enumeration variables and constants **as integers**.
- By default, the **compiler assigns the integers 0, 1, 2, ...** to the constants in a particular enumeration.
- In the `suit` enumeration, **CLUBS, DIAMONDS, HEARTS, and SPADES represent 0, 1, 2, and 3, respectively.**

Enumerations as Integers (cont.)

- The programmer **can choose different values** for enumeration constants:

```
enum suit {CLUBS = 1, DIAMONDS = 2,  
           HEARTS = 3, SPADES = 4};
```

- The values of enumeration constants may be **arbitrary integers**, listed in **no particular order**:

```
enum dept {RESEARCH = 20,  
           PRODUCTION = 10, SALES = 25};
```

- It's even **legal** for **two or more** enumeration constants to **have the same value**.

Enumerations as Integers (cont.)

- When no value is specified for an enumeration constant, its value is one greater than the value of the previous constant.
- The first enumeration constant has the value 0 by default.
- Example:

```
enum EGA_colors {BLACK, LT_GRAY = 7,  
                 DK_GRAY, WHITE = 15};
```

BLACK has the value 0, LT_GRAY is 7, DK_GRAY is 8, and WHITE is 15.

Enumerations as Integers (cont.)

- 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 (cont.)

- Although it's convenient to be able to use an enumeration value as an integer, 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`.

```
s = 4;
```

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;
```

```
#define INT_KIND 0  
#define DOUBLE_KIND 1  
  
typedef struct {  
    int kind;    /* tag field */  
    union ...  
} 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:
 - Does away with the `INT_KIND` and `DOUBLE_KIND` macros
 - Makes it obvious that `kind` has only two possible values: `INT_KIND` and `DOUBLE_KIND`