Structures, Unions, and Enumerations (3)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Outline

- Union
- Eumerations

Unions

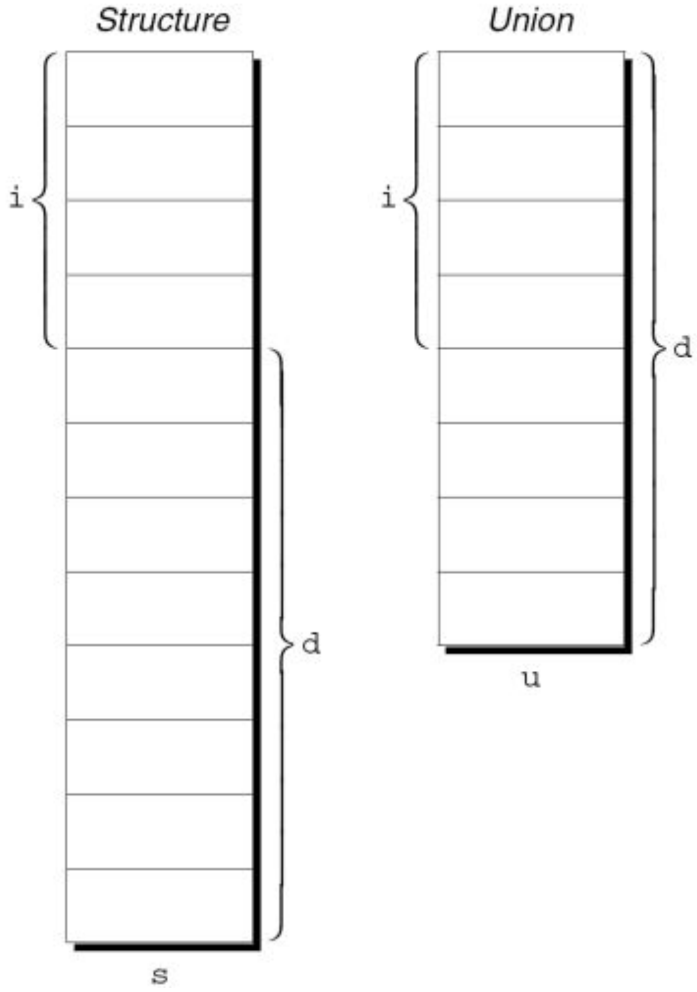
- A ***union,*** like a structure, consists of one or more members, possibly of different types.
- An example of a union variable, which closely resembles a structure declaration:

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

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

Unions

- 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.
- The members of structure `s` are stored at different addresses in memory.
- The members of union `u` are stored at the same address.



Union

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

```
u.i = 82;
u.d = 74.8; //value in u.i lost
```

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

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

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

Unions

- Like structures, **unions** can be **copied** using the = **operator**, passed to functions, and returned by functions.

```
union U {
     int i;
     double d;
} u1, u2;
…
u1 = u2;
```

Unions

- Union can also be initialized in a manner similar to structure
- However, only the first member of a union can be given an initial value.
- How to initialize the `i` member of `u1` to 0

```
union {
     int i;
     double d;
} u1 = {0};
```

Unions

- Designated initializers can also be used with unions.
- A designated initializer allows us to specify which member of a union should be initialized:
- Only one member can be initialized, but it doesn't have to be the first one.

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

Unions

- Why we need Union?
- There are some useful applications for unions:
  - Saving space
  - Building mixed data structures

Using Unions to Save Space

- Unions can be used to save space in structures.
- For example, suppose that we're designing a **structure** that will contain information about an item that's sold through a gift **catalog**.

Using Unions to Save Space

- Each item has a stock number and a price, as well as other information that depends on the type of the item
- Different types of prodcut have different type of information

Books: Title, author, number of pages

Mugs: Design

Shirts: Design, colors available, sizes available

Using Unions to Save Space

- A first attempt at designing the `catalog_item` structure
  - assume that the program only allow strings with length of 10
- The `item_type` member would have one of the values for `BOOK`, `MUG`, or `SHIRT`.

Books: Title, author, number of pages

Mugs: Design

Shirts: Design, colors available, sizes available

```
struct catalog_item {
     int stock_number;
     double price;
     int item_type;
     char title[10+1];
     char author[10+1];
     int num_pages;
     char design[10+1];
     int colors;
     int sizes;
};
```

Using Unions to Save Space

- The `colors` and `sizes` members would store encoded combinations of colors and sizes.
- This structure wastes space!
- since only part of the information in the structure is common to all items in the catalog.
- By putting a union inside, we can reduce the space required by the structure.

```
struct catalog_item {
     int stock_number;
     double price;
     int item_type;
     char title[10+1];
     char author[10+1];
     int num_pages;
     char design[10+1];
     int colors;
     int sizes;
};
```

```c
struct catalog_item {
     int stock_number;
     double price;
     int item_type;
     char title[10+1];
     char author[10+1];
     int num_pages;
     char design[10+1];
     int colors;
     int sizes;
};
```

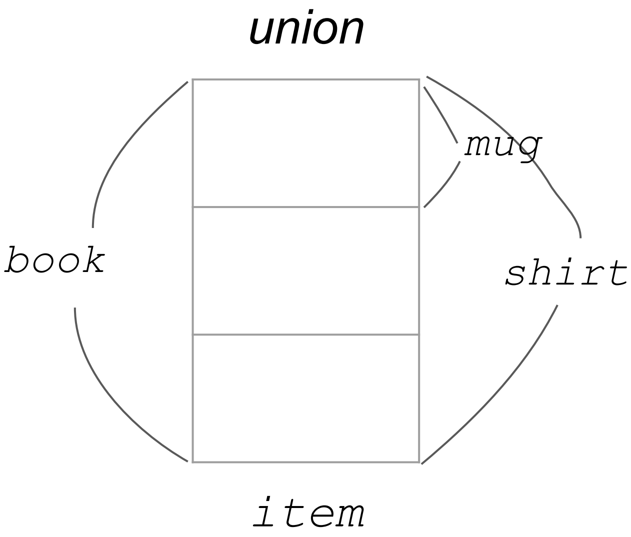
```c
struct catalog_item {
  …
  int item_type;
  union {
    struct {
      char title[10+1];
      char author[10+1];
      int num_pages;
    } book;
    struct {
      char design[10+1];
    } mug;
    struct {
      char design[10+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

Books: Title, author, number of pages

Mugs: Design

Shirts: Design, colors available, sizes available

```
struct catalog_item {
  …
  int item_type;
  union {
    struct {
      char title[10+1];
      char author[10+1];
      int num_pages;
    } book;
    struct {
      char design[10+1];
    } mug;
    struct {
      char design[10+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```




```
struct catalog_item {
  …
  int item_type;
  union {
    struct {
      char title[10+1];
      char author[10+1];
      int num_pages;
    } book;
    struct {
      char design[10+1];
    } mug;
    struct {
      char design[10+1];
      int colors;
      int sizes;
    } shirt;
  } item;
};
```

Using Unions to Build Mixed Data Structures

- The other application is that 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.
- Namely, we can store an `int` value in one element and `double` value in another in the same array.
- **How to do this with union?**

Using Unions to Build Mixed Data Structures

- First, we define a union type whose members represent the different kinds of data to be stored in the array
- Next, we create an array whose elements are `Number` values
- A `Number` union can store either an `int` value or a `double` value.

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

Number number_array[1000];
```

Using Unions to Build Mixed Data Structures

- This makes it possible to store a mixture of `int` and `double` values in `number_array`:
- For example

```
…
Number number_array[1000];
number_array[0].i = 5;
number_array[1].d = 8.395;
```

Adding a “Tag Field” to a Union

- Although these useful applications, unions suffer from a major problem.
- There’s no easy way to tell which member of a union was last changed and therefore contains a meaningful value.

Adding a “Tag Field” to a Union

- Consider the problem of writing a function that displays the value stored in a `Number` union
- There’s no way for `print_number` to determine whether `n` contains an integer or a floating-point number.

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

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

Adding a “Tag Field” to a Union

- In order to keep **track** of this information,
- we can **embed** the **union within** a **structure** that has one **other member**: a “tag field”
- 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.

```
struct catalog_item {
  …
  int item_type;
  union {
    struct {
      char title[10+1];
      char author[10+1];
      int num_pages;
    } book;
    struct {
```

Adding a “Tag Field” to a Union

- The `Number` type as a structure with an embedded union
- The value of `kind` will be either `INT_KIND` or `DOUBLE_KIND`.

```
#define INT_KIND 0
#define DOUBLE_KIND 1

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

Adding a “Tag Field” to a Union

- 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`

```
#define INT_KIND 0
#define DOUBLE_KIND 1

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

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

Adding a “Tag Field” to a Union

- 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.
- By using the modified `Number`, we can improve the function `print_number()`

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

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

Let’s Take a Break!

Enumerations

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

Enumerations

- A "suit" variable can be **declared** as an **integer**, with a set of codes that represent the possible values of the variable:
- **Problems** with this technique:
  - We **can't tell** that `s` has **only four** possible values.
  - The **meaning** of 2 isn't apparent.

```
int s;   /* s will store a suit */
…
s = 2;   /* 2 represents "hearts" */
```

Enumerations

- We may use macros to define a suit "type" and names for the various suits
- This version is more understandable than the previous version.
- However, there are still some problems…

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
…
SUIT s;
…
s = HEARTS;
```

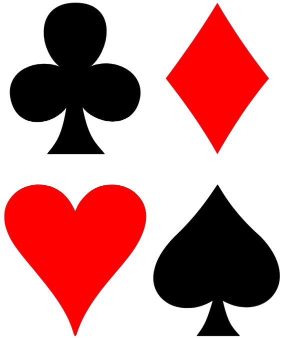
Enumerations

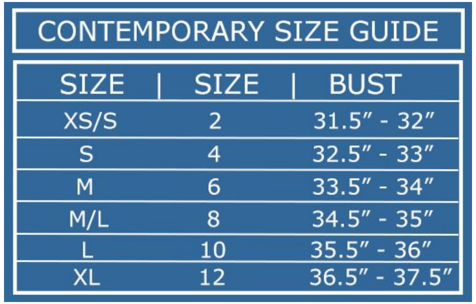
- There’s no indication to someone reading the program that the macros represent values of the **same** “**type**.”
- If the number of possible values is **more** than a few, defining a separate macro for each will be tedious.

```
#define SUIT     int
#define CLUBS    0
#define DIAMONDS 1
#define HEARTS   2
#define SPADES   3
…
SUIT s;
…
s = HEARTS;
```

Enumerations

- 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***).





CONTEMPORARY SIZE GUIDE

SIZE	SIZE	BUST
XS/S	2	31.5″ - 32″
S	4	32.5″ - 33″
M	6	33.5″ - 34″
M/L	8	34.5″ - 35″
L	10	35.5″ - 36″
XL	12	36.5″ - 37.5″

Enumerations

- Although enumerations have little in common with structures and unions, they're declared in a similar way:
- The names of enumeration constants must be different from other identifiers declared in the enclosing scope.

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

int CLUBS; //WRONG!
```

Enumerations

- 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.

```
void f(){
    enum {CLUBS, DIAMONDS, HEARTS, SPADES} s1, s2;
}
…
int CLUBS; //OK!
```

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:
- `suit` variables would be declared in the following way:

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

Enumeration Tags and Type Names

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

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

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.

```
                0       1         2        3
enum suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

Enumerations as Integers

- The programmer can choose different values for enumeration constants
- The values of enumeration constants may be arbitrary integers, listed in no particular order
- It’s even legal for two or more enumeration constants to have the same value.

```
enum suit {CLUBS = 1, DIAMONDS = 2, HEARTS = 3, SPADES = 4};
enum dept {RESEARCH = 20, PRODUCTION = 10, SALES = 25};
enum county {taipei = 1, chiayi = 1, kaohsiung = 1};
```

Enumerations as Integers

- 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.
- For example, `BLACK` has the value 0, `LT_GRAY` is 7, `DK_GRAY` is 8, and `WHITE` is 15.

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

Enumerations as Integers

- Enumeration values can be mixed with ordinary integers
- For example, `s` is treated as a variable of some integer type.

```
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             */
```

Enumerations as Integers

- 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`.

```
int i;
enum {CLUBS, DIAMONDS, HEARTS, SPADES} s;
s = 4; //WRONG!
```

Summary

- Union
    - Declaration and Initialization
    - Useful applications of Union
        - Save Space
        - Build Mixed Data Structures
- Eumerations
    - Applications
    - Declaration
    - Enumeration Tags and Type Names
    - Enumerations as Integers