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:

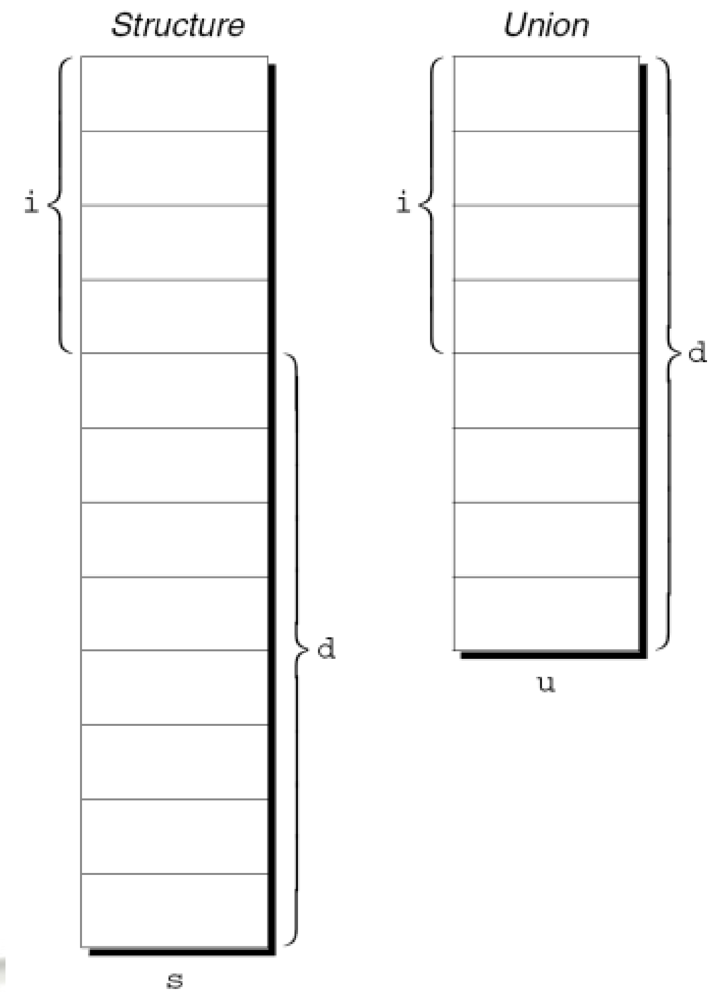
```
union {
  int i;
  double d;
} u;
```

- 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;
};
```

BOOK

MUG

SHIRT

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);
}
```

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

`n` is struct, `u` is union

`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.

IMSLab since 2010

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`