Structures, Unions, and Enumerations

Index

01
Structures

Structures

Introduction

A structure is a logical choice for storing a collection of related data items

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

Structures

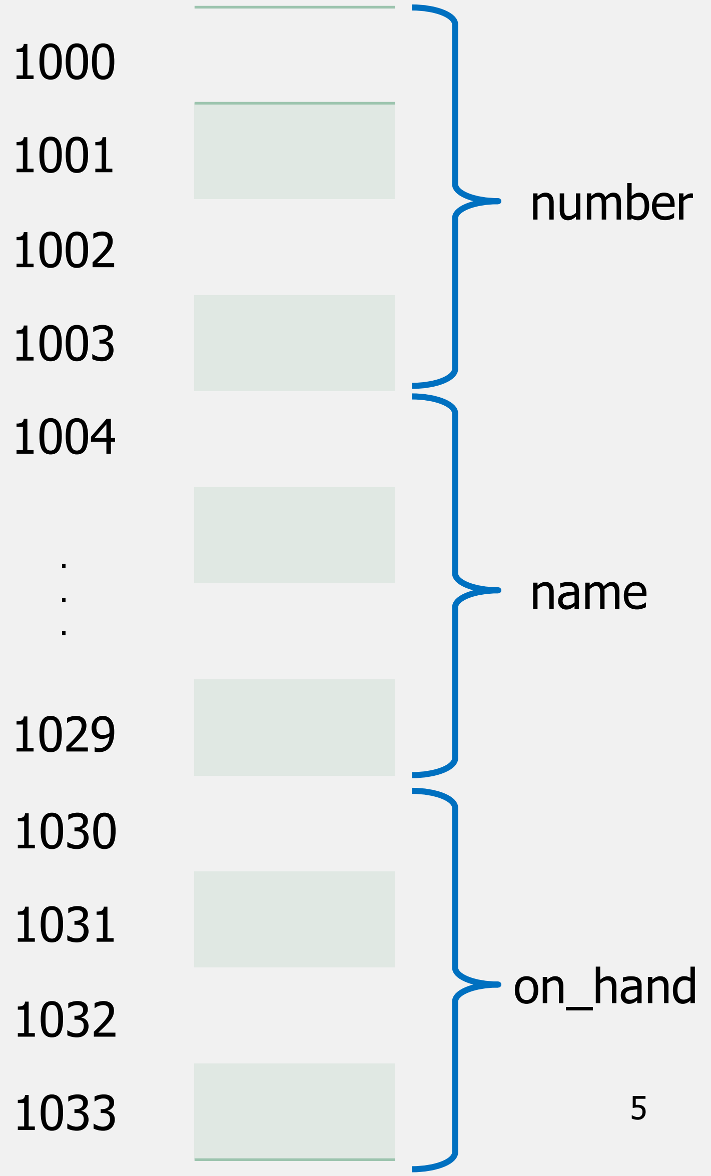
Declaration and Initialization

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;
```

The members of a structure are stored in memory in the order in which they're declared

- ➢ part1 is located at address 2000
- ➢ Integers occupy four bytes
- ➢ NAME_LEN has the value 25
- ➢ There are no gaps between the members



Structures

Declaration and Initialization

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

```
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;
```

Structures

Declaration and Initialization

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

```
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;
```

Structures
Declaration and Initialization

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};
```

part1

number	528
name	Disk drive
on_hand	10

part2

number	914
name	Printer cable
on_hand	5

Structures

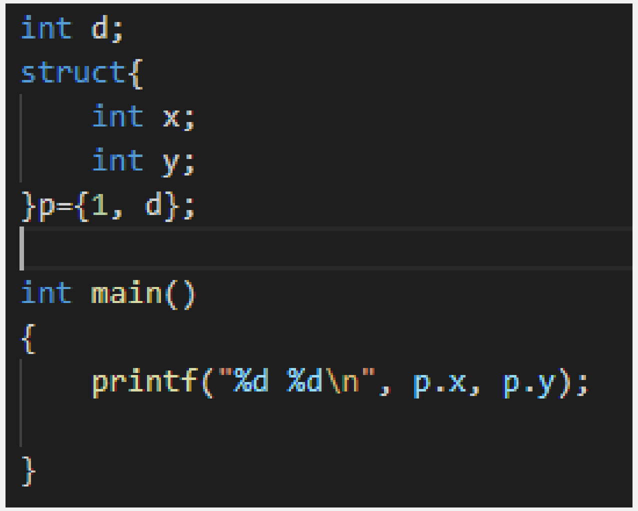
Declaration and Initialization

Structure initializers follow rules similar to those for array initializers

Expressions used in a structure initializer must be constant

An initializer can have fewer members than the structure it's initializing

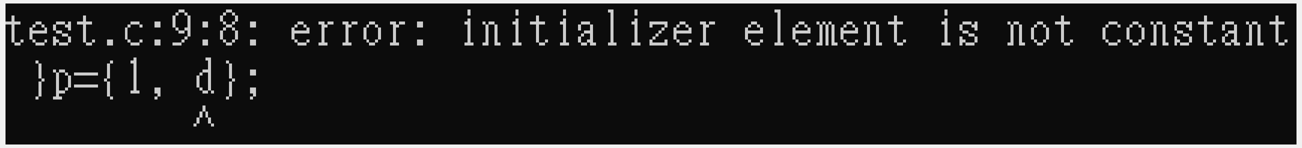
Any "leftover" members are given 0 as their initial value



```c
int d;
struct{
    int x;
    int y;
}p={1, d};

int main()
{
    printf("%d %d\n", p.x, p.y);

}
```



```
test.c:9:8: error: initializer element is not constant
 }p={1, d};
        ^
```

Structures

Declaration and Initialization

C99’s designated initializers can be used with structures

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

Structures

Declaration and Initialization

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

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

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

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

Structures
Operations

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

They can appear on the left side of an assignment or as the operand in an increment or decrement expression

The . operator takes precedence over nearly all other operators, including the &

	part1
number	528
name	Disk drive
on_hand	10

```
printf("Part number: %d\n", part1.number);
```

	part1
number	258
name	Disk drive
on_hand	10

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

```
scanf("%d", &part1.on_hand);
```

Structures

Operations

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

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 */
```

Structures

Operations

The = operator can be used only with structures of compatible types

Two structures declared at the same time (as part1 and part2 were) are compatible

Structures declared using the same "structure tag" or the same type name are also compatible

Other than assignment, C provides no operations on entire structures

In particular, the == and != operators can't be used with structures

Structures
Operations

Thinking

- ➢ Declare structure variables named a1, a2, and a3, each having members real and image of type double

```
struct {
	double real, image;
} a1, a2, a3;
```

- ➢ Modify the declaration in (1) to fit
  - a1's members initially have the values 0.0 and 1.0
  - a2's members are 1.0 and 0.0 initially

```
} a1={0.0, 1.0}, a2={1.0, 0.0}, a3;
```

- ➢ Write statement(s) that copy the members of a2 into a1

```
a1=a2;
```

- ➢ Write statements that add the corresponding members of a1 and a2, storing the result in a3

```
a3.real =a1.real + a2.real;
a3.image =a1. image + a2. image;
```

Structures

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

Structures

Types

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

Structures

Types

The part tag can be used to declare variables

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

We can't drop the word struct because part isn’t a type name; without the word struct, it is meaningless

```
part part1, part2;  //Wrong
```

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

Structures

Types

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 */
```

Structures

Types

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

Types

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
Types

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;
}
```

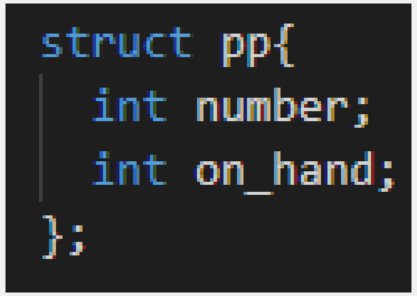
A call of build_part

```
part1 = build_part(528, "Disk drive", 10);
```

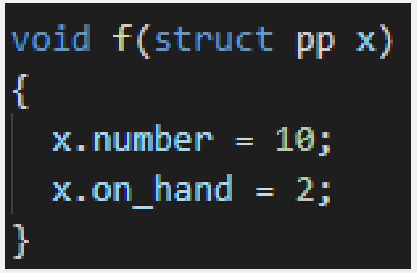
Structures

Types

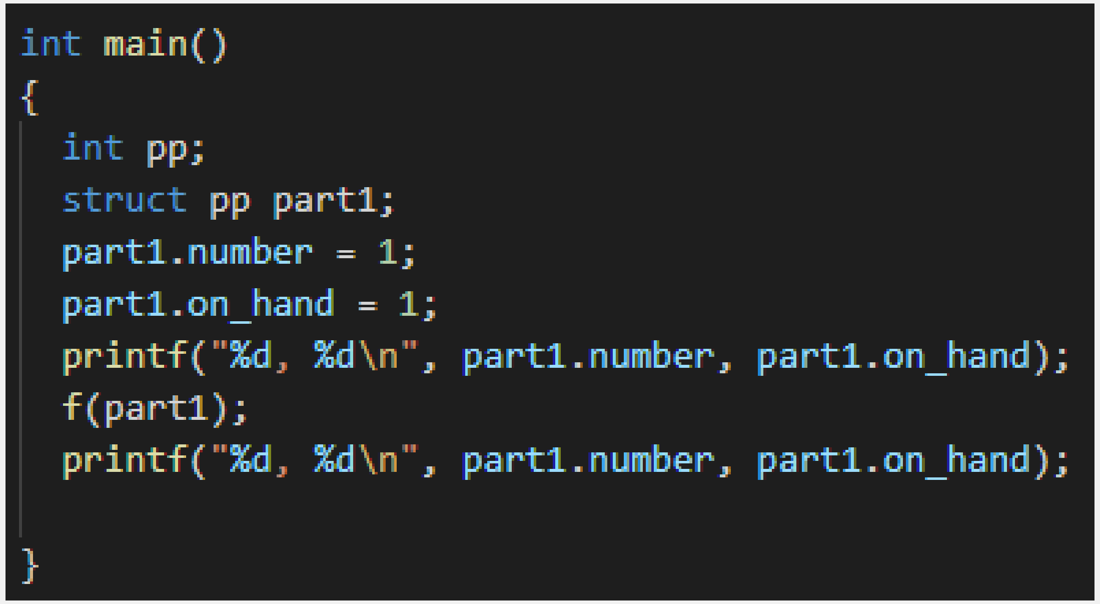
Passing a structure to a function and returning a structure from a function both require making a copy of all members in the structure



```
struct pp{
    int number;
    int on_hand;
};
```

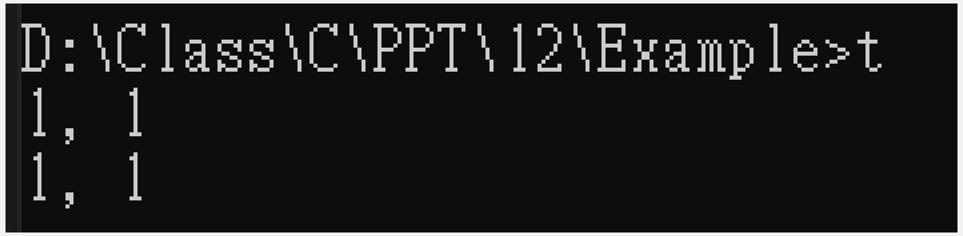


```
void f(struct pp x)
{
  x.number = 10;
  x.on_hand = 2;
}
```



```
int main()
{
  int pp;
  struct pp part1;
  part1.number = 1;
  part1.on_hand = 1;
  printf("%d, %d\n", part1.number, part1.on_hand);
  f(part1);
  printf("%d, %d\n", part1.number, part1.on_hand);

}
```



```
D:\Class\C\PPT\12\Example>t
1, 1
1, 1
```

Structures
Types

To avoid this overhead, it's sometimes advisable to pass a pointer to a structure or return a pointer to a structure

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

Types

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

Structures

Types

A compound literal can be used to create a structure "on the fly", without first storing it in a variable

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});
```

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

When a compound literal represents a structure, the type name can be a structure tag preceded by the word struct or a typedef name

Structures
Types

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

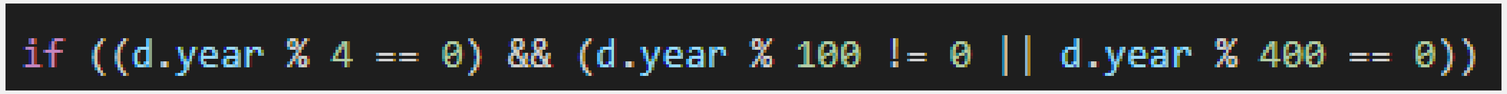
```
print_part((struct part) {.on_hand = 10,
                          .name = "Disk drive",
                          .number = 528});
```

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

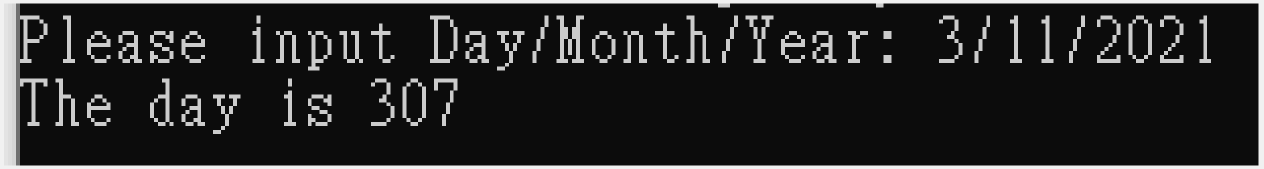
Structures
Types

Write two functions, assuming that the date structure contains three members: month, day, and year (all of type int)

- int day_of_year (struct date d);
  - Returns the day of the year (an integer between 1 and 366) that corresponds to the date d

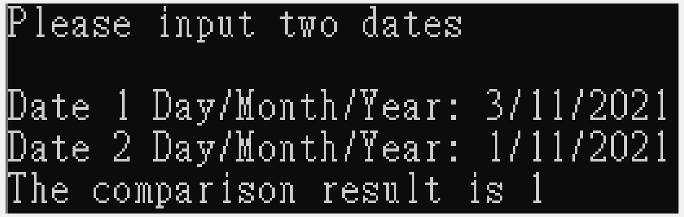


```
if ((d.year % 4 == 0) && (d.year % 100 != 0 || d.year % 400 == 0))
```



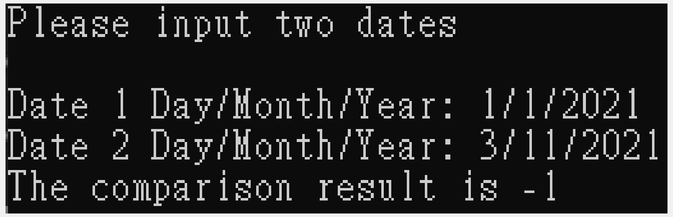
```
Please input Day/Month/Year: 3/11/2021
The day is 307
```

- int compare_dates (struct date d1, struct date d2);
  - Returns -1 if d1 is an earlier date than d2, +1 if d1 is a later date than d2, and 0 if d1 and d2 are the same



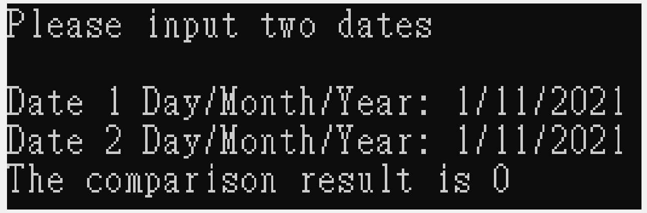
```
Please input two dates

Date 1 Day/Month/Year: 3/11/2021
Date 2 Day/Month/Year: 1/11/2021
The comparison result is 1
```



```
Please input two dates

Date 1 Day/Month/Year: 1/1/2021
Date 2 Day/Month/Year: 3/11/2021
The comparison result is -1
```



```
Please input two dates

Date 1 Day/Month/Year: 1/11/2021
Date 2 Day/Month/Year: 1/11/2021
The comparison result is 0
```

Structures

Nest and Array 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

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];
};
```

Structures

Nest and Array Structures

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");
```

Structures

Nest and Array Structures

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;
```

Structures

Nest and Array Structures

One of the most common combinations of arrays and structures is an array whose elements are structures

An array of part structures capable of storing information about 100 parts

```
struct part inventory[100];
```

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';
```

Structures

Nest and Array Structures

```
struct dialing_code {
      char *country;
      int code;
};
```

```
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}};
```

Structures

Nest and Array Structures

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'};
```

The first two items in the initializer use two designators; the last item uses three

Structures

Nest and Array Structures

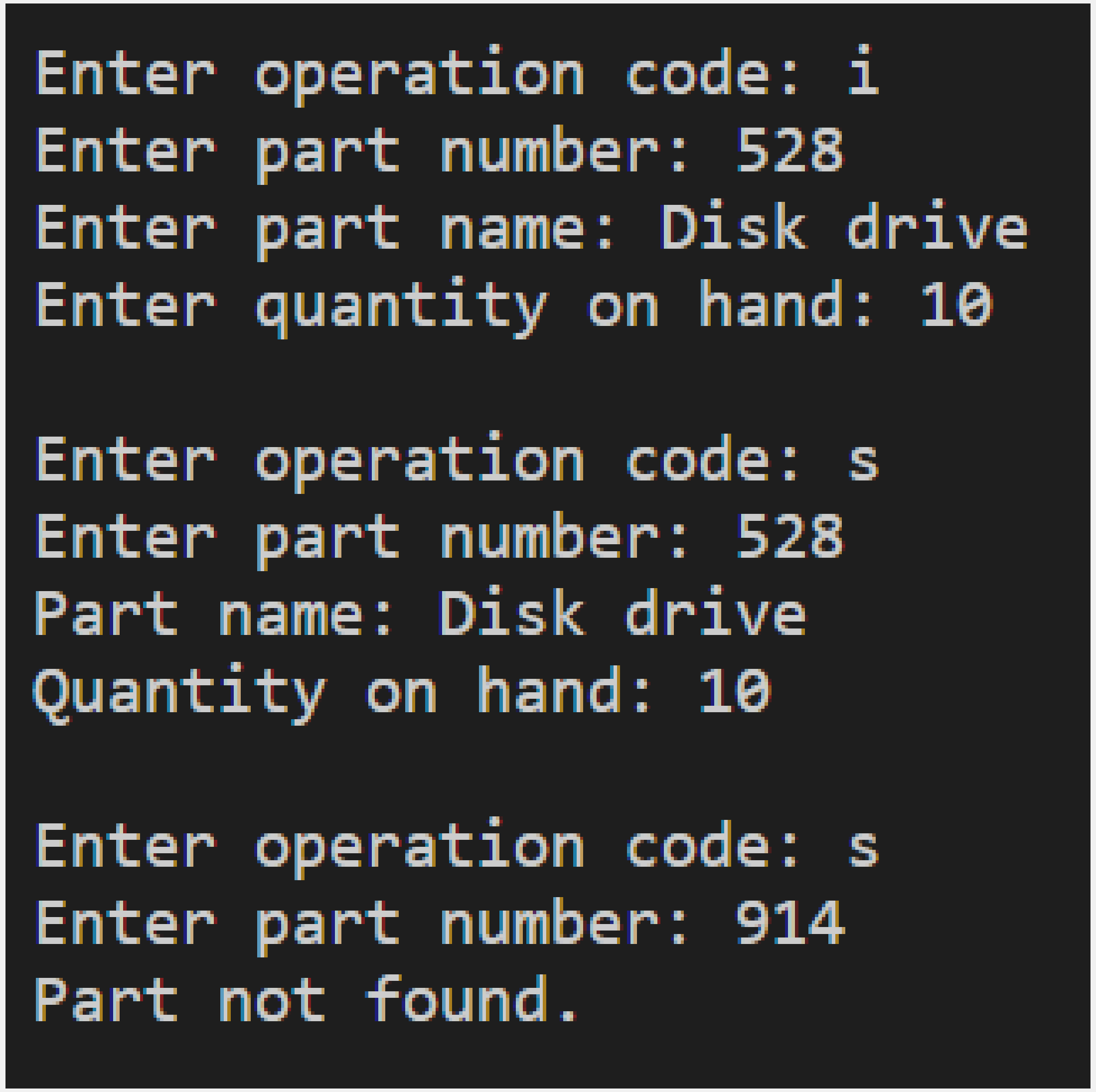
Write a program to maintain a parts database using the following structure

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

- i: Add a new part number, part name, and initial quantity on hand
- s: Given a part number, *s*, print the name of the part and the current quantity on hand
- u: Given a part number, *u*, change the quantity on hand
- p: Print a table showing all information in the database
- q: Terminate program execution

Structures

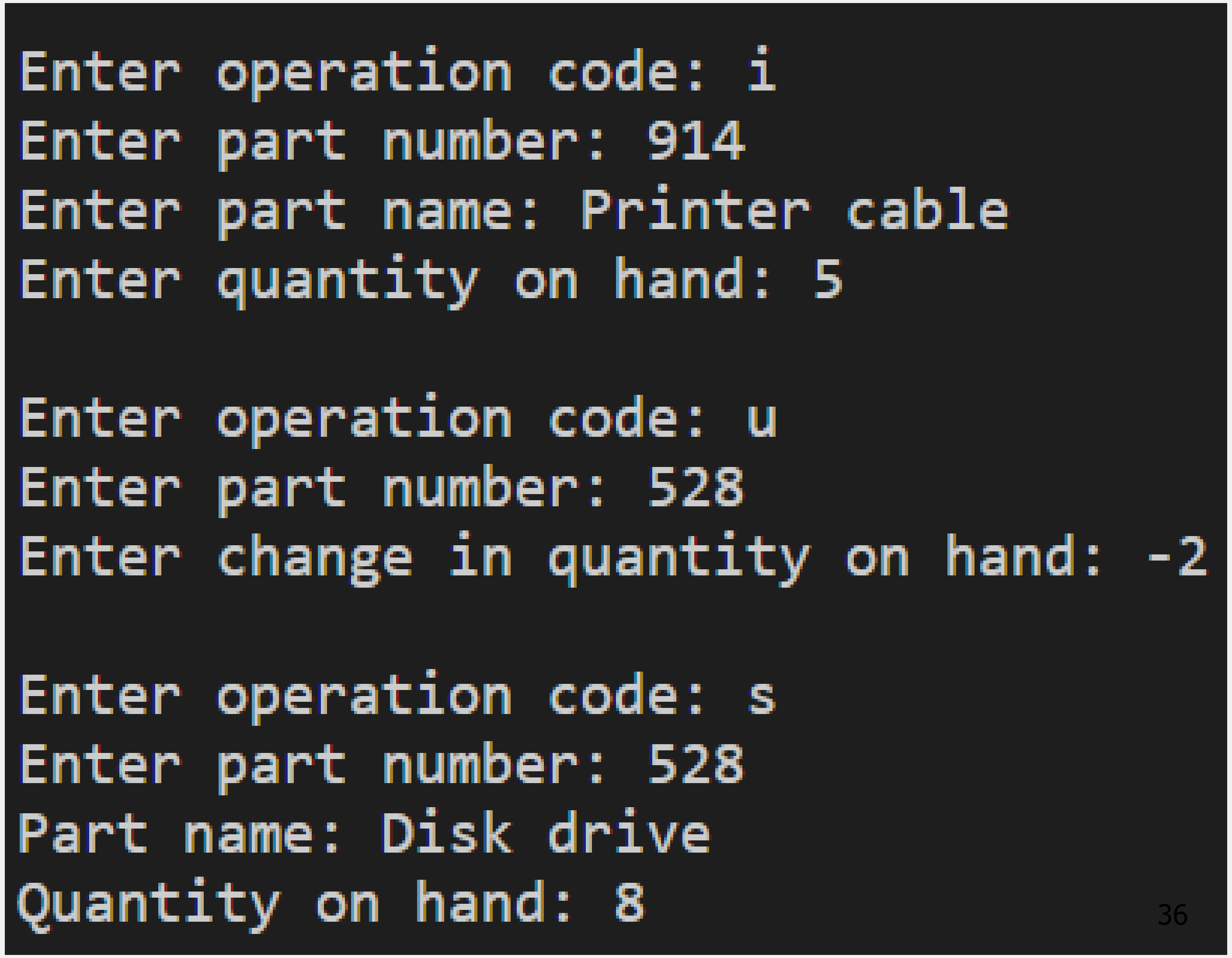
Nest and Array Structures



```
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

Enter operation code: s
Enter part number: 914
Part not found.
```



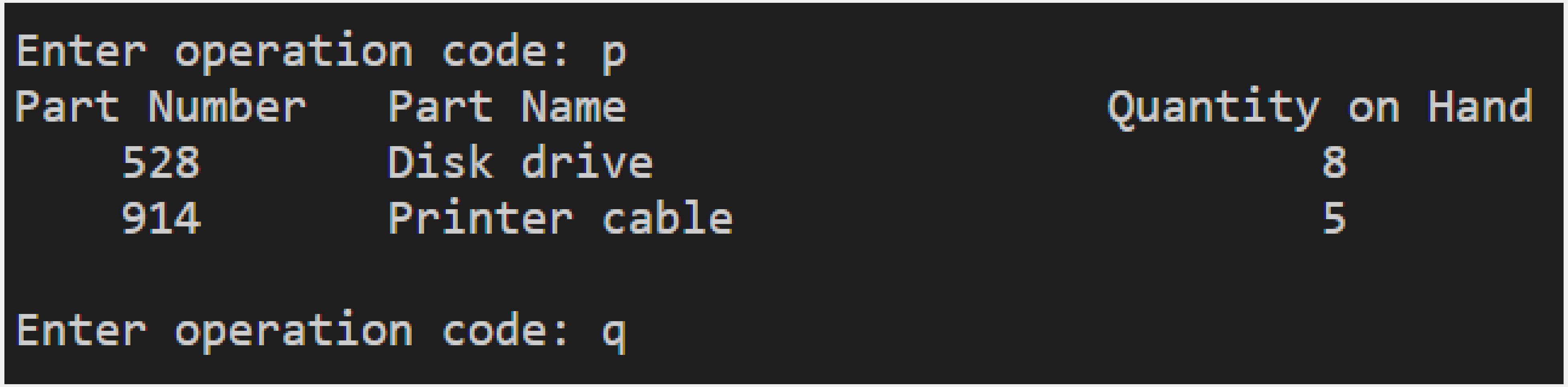
```
Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8
```

Structures

Nest and Array Structures



```
Enter operation code: p
Part Number   Part Name                    Quantity on Hand
    528       Disk drive                           8
    914       Printer cable                        5

Enter operation code: q
```

02
Unions

Unions
Introduction

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

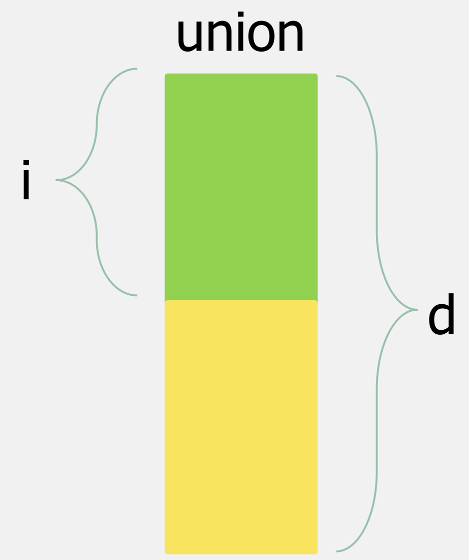
Assigning a new value to one member alters the values of the other members as well

Unions
Introduction

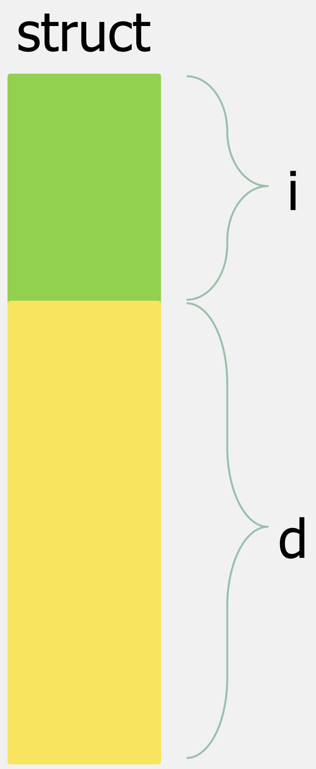
The structure s and the union u differ in just one way

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

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



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



Unions
Introduction

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

Only the first member of a union can be given an initial value

How to initialize the i member of u to 0

The expression inside the braces must be constant

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

Unions
Introduction

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

Applications for unions

- ➢ Saving space
- ➢ Building mixed data structures

Unions

Mixed Data Structure

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;
```

Unions

Mixed Data Structure

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;
```

Unions

Mixed Data Structure

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

Unions
Tag Field

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

Unions

Tag Field

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);
}
```

Unions
Tag Field

Suppose that u is the following union, how much space will a C compiler allocate for u? If char values occupy one byte, int values occupy four bytes, and double values occupy eight bytes

```
union {
    double a;
    struct {
        char b[4];
        double c;                    16
        int d;
    }e;
    char f[4];
} u;
```

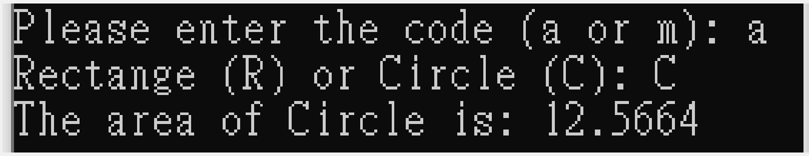
Unions
Tag Field

Let shape be the structure tag and write functions that perform the following operations on a shape structure s passed as an argument

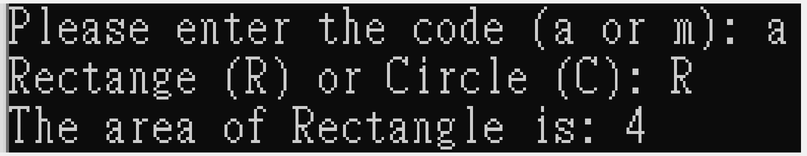
```
s.center.x = 0;
s.center.y = 0;
s.u.rectangle.height = 2;
s.u.rectangle.width = 2;
s.u.circle.radius = 2;
```

```
struct shape {
        int shape_kind;        //Rectangle or Circle
        struct{
                int x, y;           //Coordinates of center
        }center;
        union {
                struct {
                    int height, width;
                }rectangle;
                struct {
                    int radius;
                }circle;
        }u;
} s;
```

- Compute the area of s

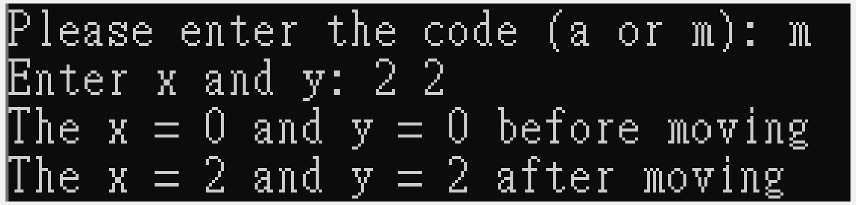


```
Please enter the code (a or m): a
Rectange (R) or Circle (C): C
The area of Circle is: 12.5664
```



```
Please enter the code (a or m): a
Rectange (R) or Circle (C): R
The area of Rectangle is: 4
```

- Move s by x units in the x direction and y units in the y direction and return modified version



```
Please enter the code (a or m): m
Enter x and y: 2 2
The x = 0 and y = 0 before moving
The x = 2 and y = 2 after moving
```

03

Enumerations

Enumerations

Introduction

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."

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
Introduction

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;
```

Enumerations

Introduction

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

An enumerated type is a type whose values are listed (“enumerated”) by the programmer

Each value must have a name (an enumeration constant)

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

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

Enumerations

Introduction

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

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;
```

Enumerations

Introduction

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

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

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

The programmer can choose different values for enumeration constants

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

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

Enumerations

Introduction

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

```
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

Introduction

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

Introduction

Suppose that b and i are declared as follows

```
enum {FALSE, TRUE} b;
int i;
```

Which of the following statements are legal? Which ones are "safe" (always yield a meaningful result)?

(a) b = FALSE;	Legal: all
(b) b = i;	
(c) b++;	Safe: (a), (d), (e)
(d) i = b;	
(e) i = 2 * b + 1;	Not safe: (b), (c)