C Structures and Enumerations

Deitel 8th Edition, Chapter 10 (and some from 5)

TOPICS

Enumerated Data Types

- Some of this is from chapter 5 – Functions

Using enums

Structures

Accessing the Components of a Structure

Using Structures

Using Structures in Arrays

Creating Data Types

Sometimes **primitive data types** (int, double, char, etc.) aren’t enough for what we want to do

We’re going to talk about two approaches to **creating our own data types**

Both use the keyword **typedef**

typedef means **type definition**, or **def**ining a data **type**

Two Ways to Create Data Types

Enumerated data type

- Really just a way to give more meaningful names to **num**bers, like 0 or 1 or 2…

Structure

- A way to combine different data types into a group that we can access with one name

ENUMERATED DATA TYPES

REVIEW – VARIABLES, DATA TYPES, AND VALUES

A **variable** is a place to store data

The type of data that can be stored in a variable is defined we declare it using a **data type** like int, double, char, etc.

The **values** that can be stored in a variable depend on its data type:

- Integer data types can store whole numbers
- Double data types can store numbers with decimals
- Char data types can store single characters
- Etc.

Certain Numbers Have a Meaning

It can be useful to **give names to certain integer values**

For example, what could these numbers refer to?

1 – 12		
1 – 7		
1 – 4		
1 – 13		

Certain Numbers Have a Meaning – Examples

It can be useful to **give names to certain integer values**

For example, what could these numbers refer to?

1 – 12	Months of the year	January, February, March, April, May, June, July, August, September, October, November, December
1 – 7	Days of the week	Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
1 – 4	Years in college	Freshman, Sophomore, Junior, Senior
1 – 13	Cards in a deck	Ace, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Knight, Queen, King

Naming These Numbers Can Make the Meaning Clear

Which is more readable, and more understandable?

“The current card is a 13”

—or—

“The current card is a king”

```
currentCard = 13;
```

—or—

```
currentCard = king;
```

Enumerated Data Type Definition

A way to assign names to integers

We could also use constant variables or symbolic constants for this purpose

But enums can keep things more organized than tediously creating a lot of constants

And enums can be used to allocate memory

- Unlike symbolic constants

Steps to Creating and Using an Enumerated Type

1. Create a type with a list of constants

2. Create a variable of that type

CREATING AN ENUMERATED TYPE

Remember, an enumerated data type – or enum – is defined by the programmer, as a new data type

So we need to define the name for this new data type, and a list of its possible values

This information goes in a **type declaration**

A type declaration statement goes **ABOVE MAIN**

Creating an Enumerated Type – Type Declaration Statement

General format:

typedef enum {

enum_constant, enum_constant, etc.

} ***name_of_enum*****;**

Creating an Enumerated Type – Example

```
typedef  enum {

     rock, paper, scissors                         ← enumeration constants

}  rpsPlay_t;                                      ← name of the enum
```

rpsPlay_t is the **name** of the enum and also a new **data type**

Common to use **_t** to indicate it was created using a **typedef**

Note the **;** at the end

Enumeration Constants

enum constants can be used in place of integers

The first enum constant in the list can be used in place of a 0, the second constant in the list can be used for a 1, etc.

rock,	paper,	scissors
0	1	2

Enumerated Type – Creating a Variable

We can then create variables of this type:

```
rpsPlay_t    playerPlay;
```

This type of statement goes in main or other functions

- Just like all variable declarations

This variable can be assigned any of the enum constants from the typedef statement, or any integer.

enums.c

PRINTING ENUM CONSTANTS

Cannot print the names of enum constants

Just like we can’t print the names of symbolic constants

To print the names of enum constants, must use selection to identify the integer value and print a string constant

printEnumConstantNames.c

USING ENUMS

CAN USE ENUMERATED SUBSCRIPTS WITH ARRAYS

```
typedef enum {
    mon, tue, wed, thu, fri
} weekday_t;


int  sales[5];
sales[mon] = 900;
sales[1] = 10000;
```

enumMonths1.c

Can Specify an Enum Constant Starting Value Other Than 0

```
typedef enum {
    mon = 1, tue, wed, thu, fri           ←Note the initialization of mon
} weekday_t;
```

This list of enums starts at 1 instead of 0

This may affect other code, so be careful

enumMonths2.c (Note array initialization and printf statement)

Can Leave Out the typedef Keyword

```
enum weekday {
    mon = 1, tue, wed, thu, fri
};
```

The name of the enumerated type must be moved next to keyword enum

Anywhere that type is used, the keyword enum must be used

enumMonths3.c

STRUCTURES

What is a Structure?

A way to store multiple pieces of information using one name

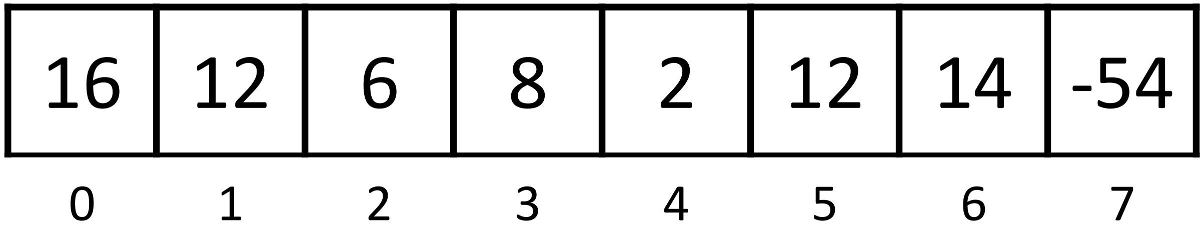
An **array** can store multiple items of the **same data type** only

A **structure** can store multiple items of **different data types**

The data stored is related to each other

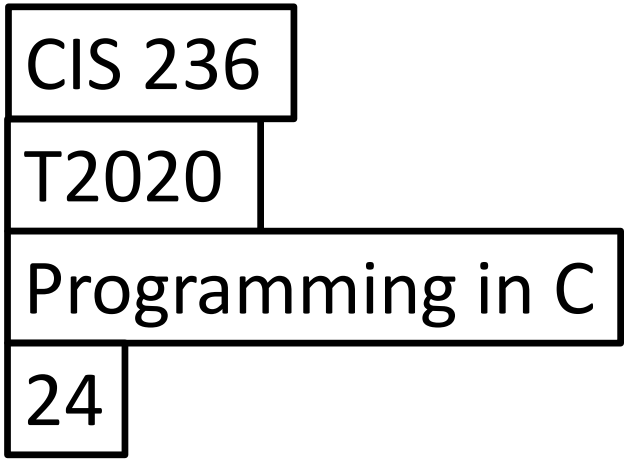
DIFFERENCES BETWEEN ARRAY & STRUCTURE 1

Array – collection of multiple memory cells that are all the same type of data



16	12	6	8	2	12	14	-54
0	1	2	3	4	5	6	7

Structure – collection of multiple memory cells that may be different type of data



CIS 236

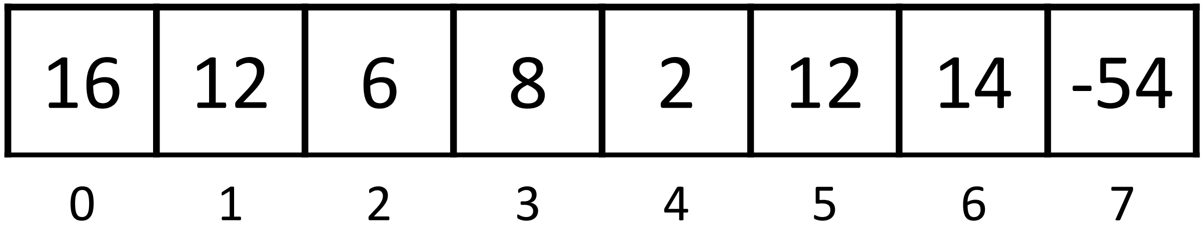
T2020

Programming in C

24

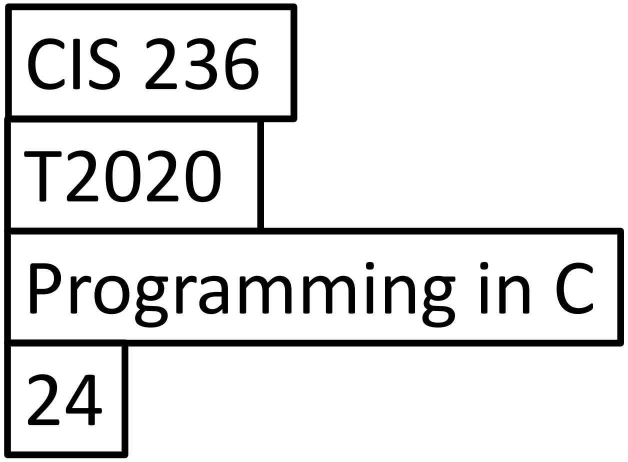
DIFFERENCES BETWEEN ARRAY & STRUCTURE 2

Cells in an array are **contiguous** in memory



16	12	6	8	2	12	14	-54
0	1	2	3	4	5	6	7

Cells in a structure may or may not be contiguous in memory, but programmer doesn't have to care



CIS 236

T2020

Programming in C

24

Structure Example – Database

Database – collection of information, stored in an organized way

Record – a subdivision of a database that contains specific info about one entity

In C we can use a structure type to define a record

BEFORE WE CAN USE A STRUCTURE:

We must perform two steps:

1. **Define** the structure type.
2. **Declare** a variable of that type.

Step 1 – Define the Structure Type

Must define the organization of the structure before we can use it

General Format:

```
typedef  struct  {
      datatype  variableName;
      datatype  variableName;
}  structureTypeName;
```

Step 1 – Define the Structure Type – Example 1

Example 1:

```
typedef  struct  {
        char   name[20];
        double   diameter;
        int  moons;
}  planet_t;
```

STEP 1 – DEFINE THE STRUCTURE TYPE – EXAMPLE 2

Example 2:

```
typedef  struct  {
      char   name[20];
      int  level;
      int  health;
      int  number;
} player_t;
```

Important Note!

TYPEDEF DOES NOT ALLOCATE MEMORY!

It acts like a blueprint for creating real items later.

STEP 2 – DECLARE A VARIABLE OF THAT STRUCTURE TYPE

General format:

***structureTypeNameFromTypedef   variableName*;**

Examples:

```
planet_t   current_planet;

planet_t   blank_planet = {“ ”, 0, 0};

player_t   current_player;

player_t   new_player = {“Jack”, 2, 10, 3};
```

Example – A Car Structure

Create a structure named **car_t** to use for storing information about a car

Include components named **price** (double), **horsepower** (integer) and **bodyType** (string)

Create a variable called **carInfo** of the **car_t** type, to be used in a program to store car information.

EXAMPLE – A CAR STRUCTURE 2

Create a structure named car_t

Components: price (double), horsepower (integer), bodyType (string)

Create variable carInfo of the car_t type

```
typedef struct {
    double price;
    int horsepower;
    char bodyType[10];
} car_t;
```

Example – A Car Structure 3

Create a structure named car_t

Components: price (double), horsepower (integer), bodyType (string)

Create variable carInfo of the car_t type

```
typedef struct {
    double price;
    int horsepower;
    char bodyType[10];
} car_t;
```

EXAMPLE – A CAR STRUCTURE 4

Create a structure named car_t

Components: price (double), horsepower (integer), bodyType (string)

Create variable carInfo of the car_t type

```
typedef struct {
      double price;
      int horsepower;
      char bodyType[10];
} car_t;

car_t carInfo;
```

Example – A Car Structure 5

Remember that the typedef struct statement goes above main.

The variable declaration goes in main, or another function.

```
typedef struct {
    double price;
    int horsepower;
    char bodyType[10];
} car_t;

car_t carInfo;
```

HIERARCHICAL STRUCTURES

Structure components can also be structures!

Example:

```
typedef struct {
     char  name[10];
     car_t   carsForSale[100];
}  dealership_t;
```

Manipulating Whole Structures

An entire structure can be referred to using its variable name

Assignment can be used with **whole structure variables**

```
pamsNewCar = newCar;

junkCar = pamsOldCar;
```

Accessing the Components of a Structure

To Access the Components of a Structure, We Need an Operator

To reference one component of a structure, use the **direct component selection operator**, which is the period: **•**

Components of a structure are usually just regular ints, double, chars, etc.

General Format:

structureVariableName.componentName

Examples

```
strcpy(currentCar.bodyType, “sedan”);

currentCar.price += 200;

toupper(dealership.name[0]);

printf(“%.2f”, usedCar.price);

scanf(“%d”, &newCar.horsepower);
```

Structures and Operator Precedence

Component operator has highest precedence, equal to subscripting and function calls

We'll see an example of this later.

USING STRUCTURES

USING STRUCTURES WITH FUNCTIONS

Structures may be passed to functions by:

- Passing individual structure components separately (pass by value)
- Passing pointers to the structure components (pass by reference)
- Passing the entire structure (pass by value)
- Passing a pointer to the structure (pass by reference)

Performance Tip

- **Passing structures by reference is more efficient** than passing structures by value (which requires the entire structure to be copied).

Passing a Structure to a Function

Can pass entire structures to a function

Passed by value, just like all other types of data

All components are COPIED into the components of the function’s parameter.

printPlayer.c

USING A STRUCTURE AS A RETURN VALUE

A function can return a structure as a **return value**

Returns structure by value (as a copy), just like the primitive data types char, int, double

returnPlayer.c

COMPARING STRUCTURES

Cannot use **relational or equality operators** with structure as a unit

Can compare the individual **components of a structure**

- Except for strings, or components that are structures themselves

We can write a function to compare structures and compare each component separately in the function

comparePlayers.c

Using a Pointer to a Structure

Can use a pointer to a structure, just like pointers to any other data type

```
player_t  playerInfo;

player_t * playerPtr;

playerPtr = &playerInfo;
```

pointer_to_player.c

But there is a problem…

REFERENCING STRUCTURE COMPONENTS W/A POINTER

How can we access the structure components when we have a pointer to the structure?

Referencing Structure Components w/a Pointer – Use the Pointer Directly

Attempt 1:

```
printf(“%d”, playerPtr.number);
```

The compiler tries to find a component called “**number**” of what kind of data?

Referencing Structure Components w/a Pointer – Use the Pointer Directly 2

Attempt 1:

```
printf(“%d”, playerPtr.number);
```

The compiler tries to find a component called “**number**” of what kind of data?

An address – which doesn’t have components

So this doesn’t work.

REFERENCING STRUCTURE COMPONENTS BY DEREFERENCING THE POINTER

Attempt 2: dereference the pointer…

```
printf(“%d”, *playerPtr.number);
```

But there are two operators in this statement: the ***** and the **.**

The dot operator has higher precedence than the *****

REFERENCING STRUCTURE COMPONENTS BY DEREFERENCING THE POINTER 2

Attempt 2:

printf("%d", *playerPtr.number);

So the compiler tries to find a component called "**number**" of an address, and then tries to dereference it, which doesn't work either

REFERENCING STRUCTURE COMPONENTS BY DEREFERENCING THE POINTER 3

Attempt 3:

```
printf(“%d”, *playerPtr.number);
```

Need to dereference the pointer FIRST, and THEN find the component

What can be added to force the ***** to be evaluated first?

Referencing Structure Components by Dereferencing the Pointer 4

Parentheses!

```
printf(“%d”, (*playerPtr).number);
```

This forces ***playerPtr** to be dereferenced first, so the structure it’s pointing to can be located.

Then the **number component** of that structure is accessed

But this syntax is kind of ugly…

STRUCTURE POINTER OPERATOR —>

Replaces parentheses, asterisk and dot operator

Can make code easier to understand

(*playerPtr).number can be written as **playerPtr—>number**

&(*playerPtr).number can be written as **&playerPtr—>number**

USING STRUCTURE COMPONENTS W/SCANF – GENERAL STEPS

1. **Find the structure first** using parentheses around * and structure pointer
2. **Find the component within the structure** using the dot operator and name of component
3. **Find the component's address** by adding the AddressOf operator to the front.

Combine steps 1 and 2 using the structure pointer operator **—>**

pointer_to_player_scanf.c

Using Structures in Arrays

Can Store Structures in Arrays

Create an array of player structures.

Steps:

1. Create the structure using a typedef

2. Create an array of that type

Step 1 – Create the Structure

Create a structure named player_t

Create an array of that type

```
typedef  struct  {

        char name[20];
        int   level;
        int   health;
        int   number;

} player_t;
```

Step 2 – Create the Array

Create a structure named player_t

Create an array of that type

This is when memory is allocated!

```
typedef  struct  {

        char name[20];
        int   level;
        int   health;
        int   number;

} player_t;


player_t player_list[3];
```

Referring to Components of Structures in Arrays

To refer to one player:

player_list[x]

To refer to one player’s level or health:

player_list[x].level

player_list[x].health

array_of_players.c