C FOR EMBEDDED SYSTEMS (CONT. 2)

LECTURE III
TDDI11 Embedded Software

DEPT. COMPUTER AND INFORMATION SCIENCE (IDA)

LINKÖPINGS UNIVERSITET

OUTLINE

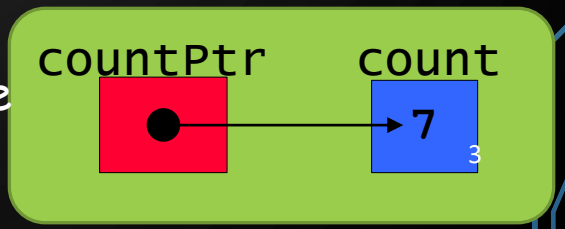
- Pointers
- Structures
- Unions
- Endianness
- Bitfield

POINTER VARIABLE DEFINITIONS AND INITIALIZATION

Pointer variables

- Contain memory addresses as their values
- Normal variables contain a specific value (direct reference)

- Pointers contain address of a variable that has a specific value (indirect reference)
- Indirection – referencing a pointer value



POINTER VARIABLE DEFINITIONS AND INITIALIZATION (CONT.)

- * used with pointer variables:

```
int *myPtr;
```

- Defines a pointer to an `int` (pointer of type `int *`)
- Multiple pointers require using a * before each variable definition

```
int *myPtr1, *myPtr2;
```

- Can define pointers to any data type
- Initialize pointers to 0, `NULL`, or an address
  - 0 or `NULL` – points to nothing

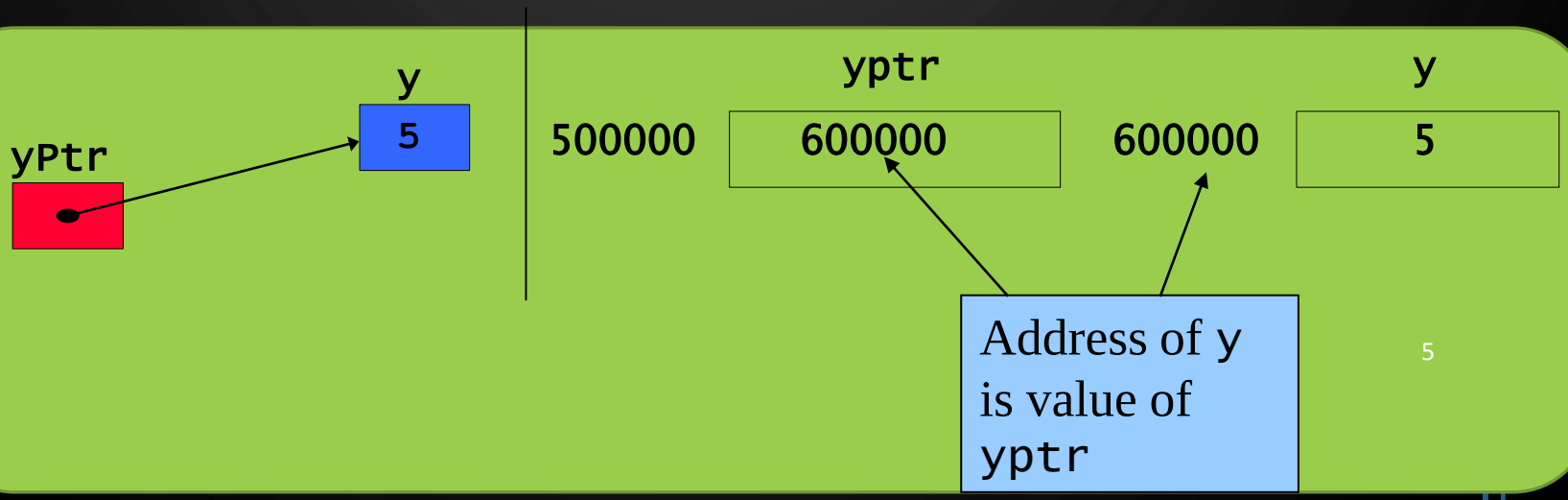
POINTER OPERATORS

& (address operator)

- Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;  /* yPtr gets address of y */
```

- yPtr "points to" y



POINTER OPERATORS, CONT.

- * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - *yptr returns y (because yptr points to y)
  - * returns an lvalue that can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  /* changes y to 7 */
```

- * and & are inverses
  - They cancel each other out

DISPLAY EXAMPLE

- Alphanumeric color display
  - Memory mapped at address 0xB8000
  - 80 characters on each line
  - Each character consists of two bytes
    - First byte: ASCII code
    - Second byte: foreground + background colors

```
char *p = (char *)(0xB8000 + 2 * (80 * row + col));

*p = 65; /* e.g., display 'A' on screen */
```

CALLING FUNCTIONS BY REFERENCE

- Call by reference with pointer arguments
  - Pass address of argument using & operator
  - Allows you to change actual location in memory
  - Arrays are not passed with & because the array name is already a pointer
- * operator
  - Used as alias/nickname for variable inside of function
  - *number used as nickname for the variable passed

```
void double(int *number)
 {
    *number = 2 * ( *number );
 }
```

POINTER EXPRESSIONS AND POINTER ARITHMETIC

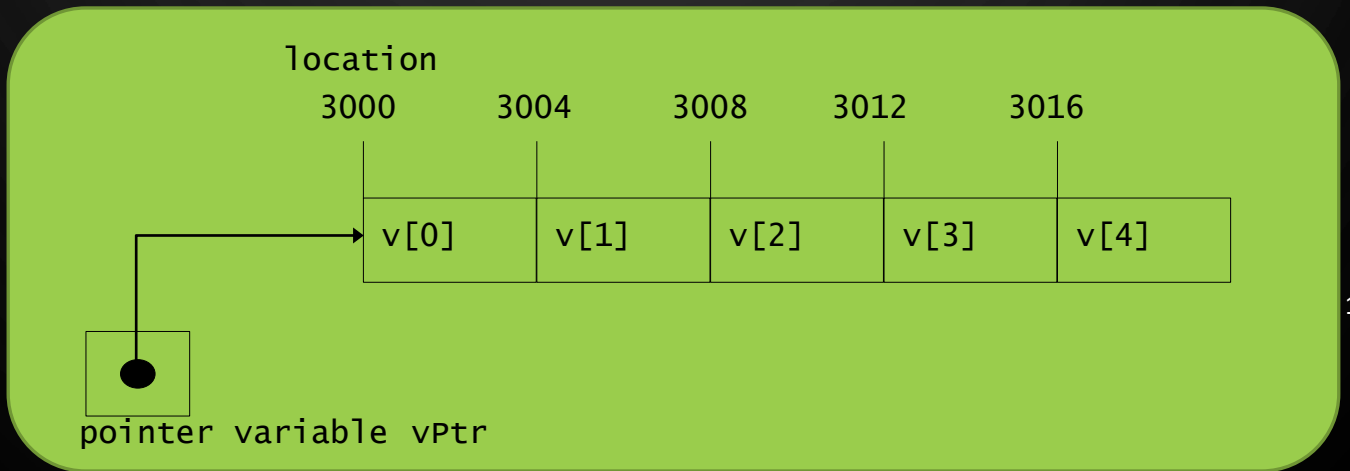
Arithmetic operations can be performed on pointers

- Increment/decrement pointer (++ or --)
- Add an integer to a pointer(+ or += , - or -=)
- Pointers may be subtracted from each other
- Operations meaningless unless performed on an array

POINTER EXPRESSIONS AND ARITHMETIC, CONT.

5 element `int` array on machine with 4-bytes `int`s

- `vPtr` points to first element `v[ 0 ]`
  - at location 3000 (`vPtr = 3000`)
- `vPtr += 2;` sets `vPtr` to 3008
  - `vPtr` points to `v[ 2 ]` (incremented by 2), but the machine has 4 byte `int`s, so it points to address 3008



POINTER EXPRESSIONS AND ARITHMETIC, CONT.

- Subtracting pointers
  - Returns number of elements from one to the other. If

```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
```

  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to 0

POINTER EXPRESSIONS AND POINTER ARITHMETIC, CONT.

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to `void` (type `void *`)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to `void` pointer
    - `void` pointers cannot be dereferenced

THE RELATIONSHIP BETWEEN POINTERS AND ARRAYS

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Define an array `b[ 5 ]` and a pointer `bPtr`
  - To set them equal to one another use:

    ```
    bPtr = b;
    ```

  - The array name (b) is actually the address of first element of the array `b[ 5 ]`

    ```
    bPtr = &b[ 0 ];
    ```

  - Explicitly assigns `bPtr` to address of first element of b

THE RELATIONSHIP BETWEEN POINTERS AND ARRAYS, CONT.

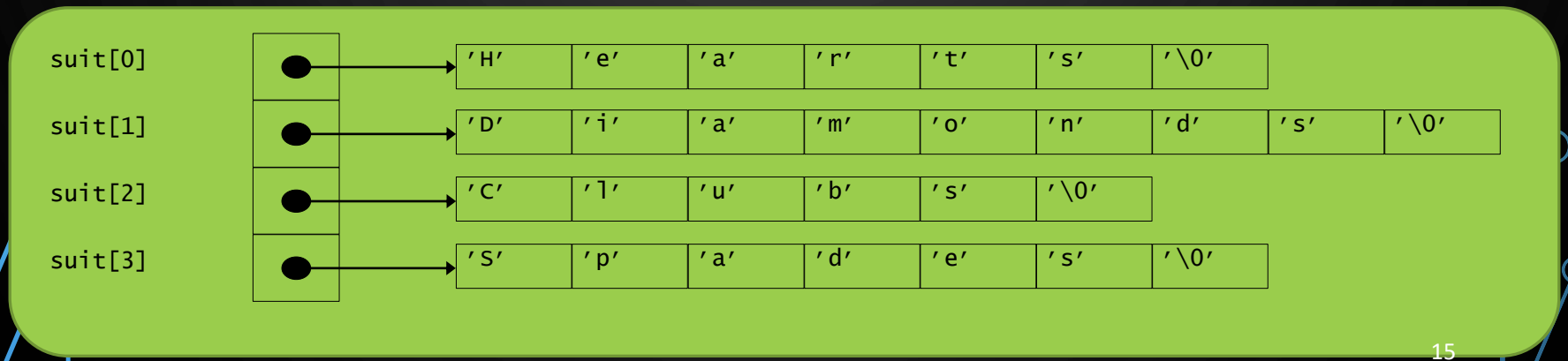
- Element `b[ 3 ]`
  - Can be accessed by `*( bPtr + 3 )`
    - Where `n` is the offset. Called pointer offset notation
  - Can be accessed by `bptr[ 3 ]`
    - Called pointer subscript notation
    - `bPtr[ 3 ]` same as `b[ 3 ]`
  - Can be accessed by performing pointer arithmetic on the array itself `*( b + 3 )`

ARRAYS OF POINTERS

- Arrays can contain pointers. For example: an array of strings

```
char * suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

  - Strings are pointers to the first character
  - `char *` - each element of `suit` is a pointer to a `char`
  - The strings are not actually stored in the array `suit`, only pointers to the strings are stored



  - `suit` array has a fixed size, but strings can be of any size

POINTERS TO FUNCTIONS (3)

- Most frequent use: generic functions
- Example: Generic sort routine

```
void bubble_sort(int arr[],int asize,int (*cmp)(int,int)){
        int i, j, tmp;
        for (i = 0;i < asize-1; i++)
                for (j = i+1;j < asize; j++)
                        if (cmp(arr[i] , arr[j])) {
                                tmp=arr[i];
                                arr[i]=arr[j];
                                arr[j]=tmp;
                        }
}

int gt(int a, int b)  { if (a > b) return 1; else return 0; }
bubble_sort(somearray, 100, gt);
bubble_sort(otherarray, 200, lt);
```

POINTERS TO FUNCTIONS

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function
- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

POINTERS TO FUNCTIONS, CONT.

- Example: bubblesort
  - Function `bubble` takes a function pointer
    - `bubble` calls this helper function
    - this determines ascending or descending sorting
  - The argument in `bubblesort` for the function pointer:

    ```
    int ( *cmp )( int a, int b )
    ```

    tells `bubblesort` to expect a pointer to a function that takes two `int`s and returns an `int`
  - If the parentheses were left out:

    ```
    int *cmp( int a, int b )
    ```

    - Defines a function that receives two integers and returns a pointer to a `int`

OUTLINE

- Pointers
- Structures
- Unions
- Endianness
- Bitfield

STRUCTURES

- Collections of related variables (aggregates) under one name
  - Can contain variables of different data types
- Commonly used to define records to be stored in files
- Combined with pointers, can create linked lists, stacks, queues, and trees

- Can hold the data associated to a hardware device

STRUCTURE DEFINITIONS

- Example

```
struct card {
    char *face;
    char *suit;
};
```

  - struct introduces the definition for structure card
  - card is the structure name and is used to declare variables of the structure type
  - card contains two members of type char *
    - These members are face and suit

STRUCTURE DEFINITIONS, CONT.

- A struct cannot contain an instance of itself
- Can contain a member that is a pointer to the same structure type
- A structure definition does not reserve space in memory. Instead creates a new data type used to define structure variables

Definitions

- Defined like other variables:

```
Struct card oneCard, deck[ 52 ], *cPtr;
```

- Can use a comma separated list:

```
struct card {
   char *face;
   char *suit;
} oneCard, deck[ 52 ], *cPtr;
```

STRUCTURE DEFINITIONS, CONT

- Valid operations
  - Assigning a structure to a structure of the same type
  - Taking the address (&) of a structure
  - Accessing the members of a structure
  - Using the sizeof operator to determine the size of a structure

INITIALIZING STRUCTURES

- Initializer lists

```
struct card oneCard = { "Three", "Hearts" };
```

- Assignment statements

```
struct card threeHearts = oneCard;
```

- Could also define and initialize threeHearts as follows:

```
struct card threeHearts;
threeHearts.face = “Three”;
threeHearts.suit = “Hearts”;
```

ACCESSING MEMBERS OF STRUCTURES

- Accessing structure members
  - Dot operator (.) used with structure variables

```
struct card myCard;
printf( "%s", myCard.suit );
```

  - Arrow operator (->) used with pointers to structure variables

```
struct card *myCardPtr = &myCard;
printf( "%s", myCardPtr->suit );
```

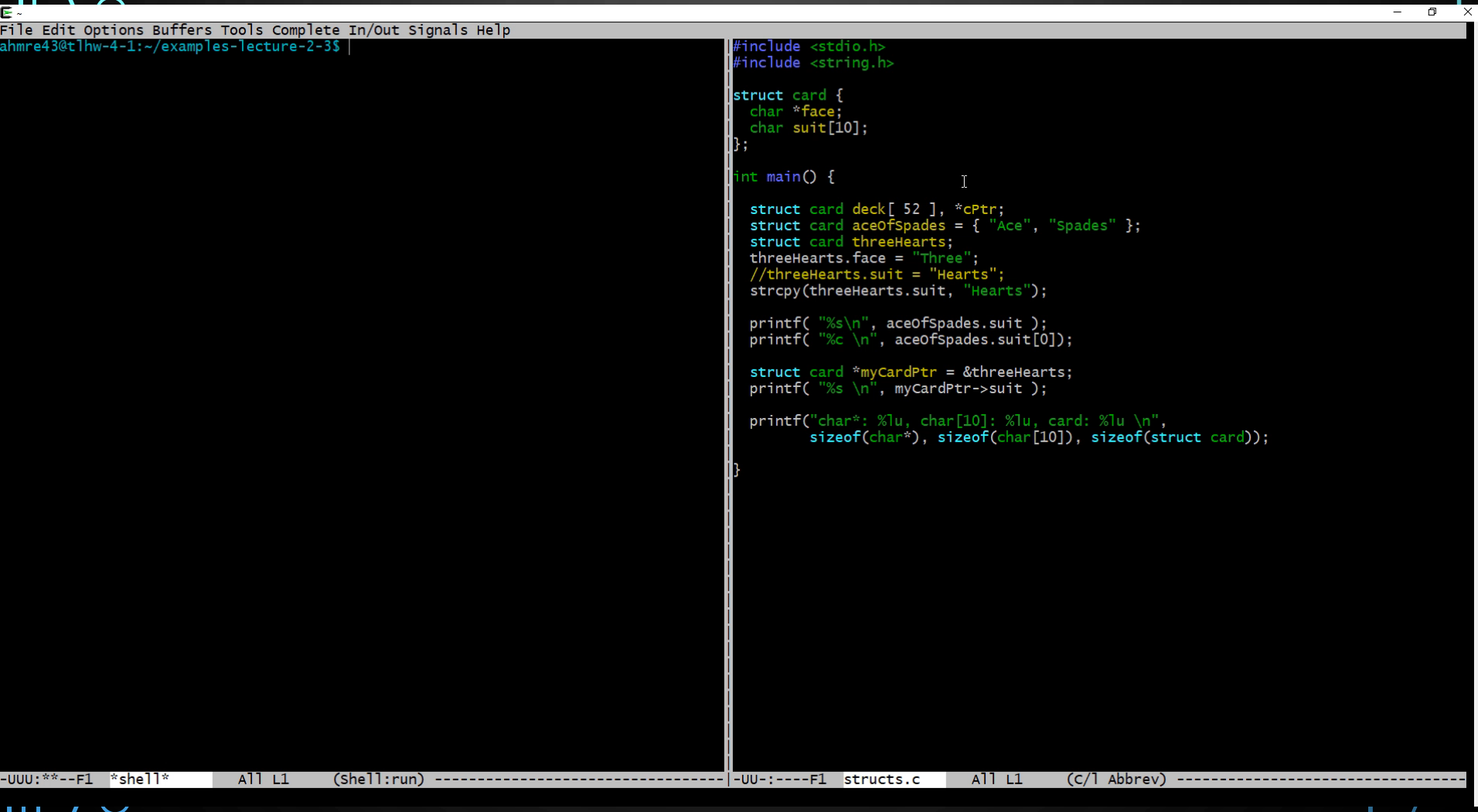
  - myCardPtr->suit is equivalent to

```
(*myCardPtr).suit
```

USING STRUCTURES WITH FUNCTIONS

- Passing structures to functions
  - Pass entire structure or pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

LET'S CODE!


File Edit Options Buffers Tools Complete In/Out Signals Help
ahmre43@tlhw-4-1:~/examples-lecture-2-3$
-UUU:**--F1 *shell* All L1 (Shell:run)
#include <stdio.h>
#include <string.h>
struct card {
char *face;
char suit[10];
};
int main() {
struct card deck[52], *cPtr;
struct card aceOfSpades = { "Ace", "Spades" };
struct card threeHearts;
threeHearts.face = "Three";
//threeHearts.suit = "Hearts";
strcpy(threeHearts.suit, "Hearts");
printf("%s\n", aceOfSpades.suit);
printf("%c \n", aceOfSpades.suit[0]);
struct card *myCardPtr = &threeHearts;
printf("%s \n", myCardPtr->suit);
printf("char*: %lu, char[10]: %lu, card: %lu \n",
sizeof(char*), sizeof(char[10]), sizeof(struct card));
}
-UU-:----F1 structs.c All L1 (C/l Abbrev)

TYPEDEF

- `typedef`
  - Creates synonyms (aliases) for previously defined data types
  - Use `typedef` to create shorter type names
  - Example:

```
typedef struct Card *CardPtr;
```

  - Defines a new type name `CardPtr` as a synonym for type `struct Card *`
  - `typedef` does not create a new data type
    - Only creates an alias

TYPEDEFS

```
unsigned long int count ;
```

versus

```
typedef unsigned long int DWORD32 ;
DWORD32 count ;
```

TYPEDEFS AND #DEFINES

```
typedef unsigned char          BYTE8;
typedef unsigned short int     WORD16;
typedef unsigned long int      DWORD32;

typedef int                    BOOL;
#define FALSE                  0
#define TRUE                   1
```

OUTLINE

- Structures
- **Unions**
- Endianness
- Bitfield
- Bit manipulation

UNIONS

- Memory that contains a variety of objects over time
- Only contains one data member at a time
- Members of a union share space
- Conserves storage
- Only the last data member assigned can be accessed
- union definitions same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```

UNIONS

```
union Data
{
    int i;
    float f;
    char str[20];
} data;
```

The memory occupied by a union will be large enough to hold the largest member of the union.

For example, in above example the type data will occupy 20 bytes.

UNIONS

- Valid union operations
  - Assignment to union of same type: =
  - Taking address: &
  - Accessing union members: .
  - Accessing members using pointers: ->

VARIANT ACCESS WITH POINTERS, CASTS, & SUBSCRIPTING

- Given an address, we can cast it as a pointer to data of the desired type, then dereference the pointer by subscripting.
- Without knowing the data type used in its declaration, we can read or write various parts of an object named *operand* using:

```
((BYTE8 *) &operand)[k]
```

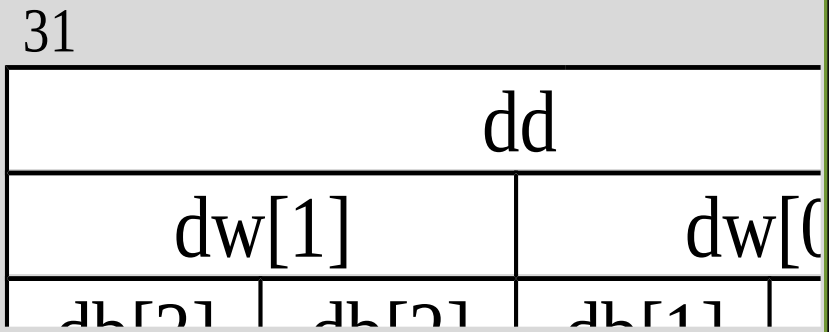
VARIANT ACCESS WITH POINTERS, CASTS, & SUBSCRIPTING, CONT.

```
typedef struct KYBD_INFO
{
  BYTE8     lo ;
  BYTE8     hi ;
  WORD16  both ;
} KYBD_INFO ;
```

```
BOOL Kybd_Flags_Changed(KYBD_INFO *kybd)
{
  // ...

  kybd->both = ((WORD16  *) &new_flags)[0] ;
  kybd->lo     = ((BYTE8  *) & new_flags)[0] ;
  kybd->hi     = ((BYTE8  *) &new_flags)[1] ;

  if (kybd->both == old_flags) return FALSE ;
  old_flags = kybd->both ;

 return TRUE ;
}
```

VARIANT ACCESS WITH UNIONS

```
union {
   unsigned long   dd ;
   unsigned short  dw[2] ;
   unsigned char   db[4] ;
```



VARIANT ACCESS WITH UNIONS, CONT.

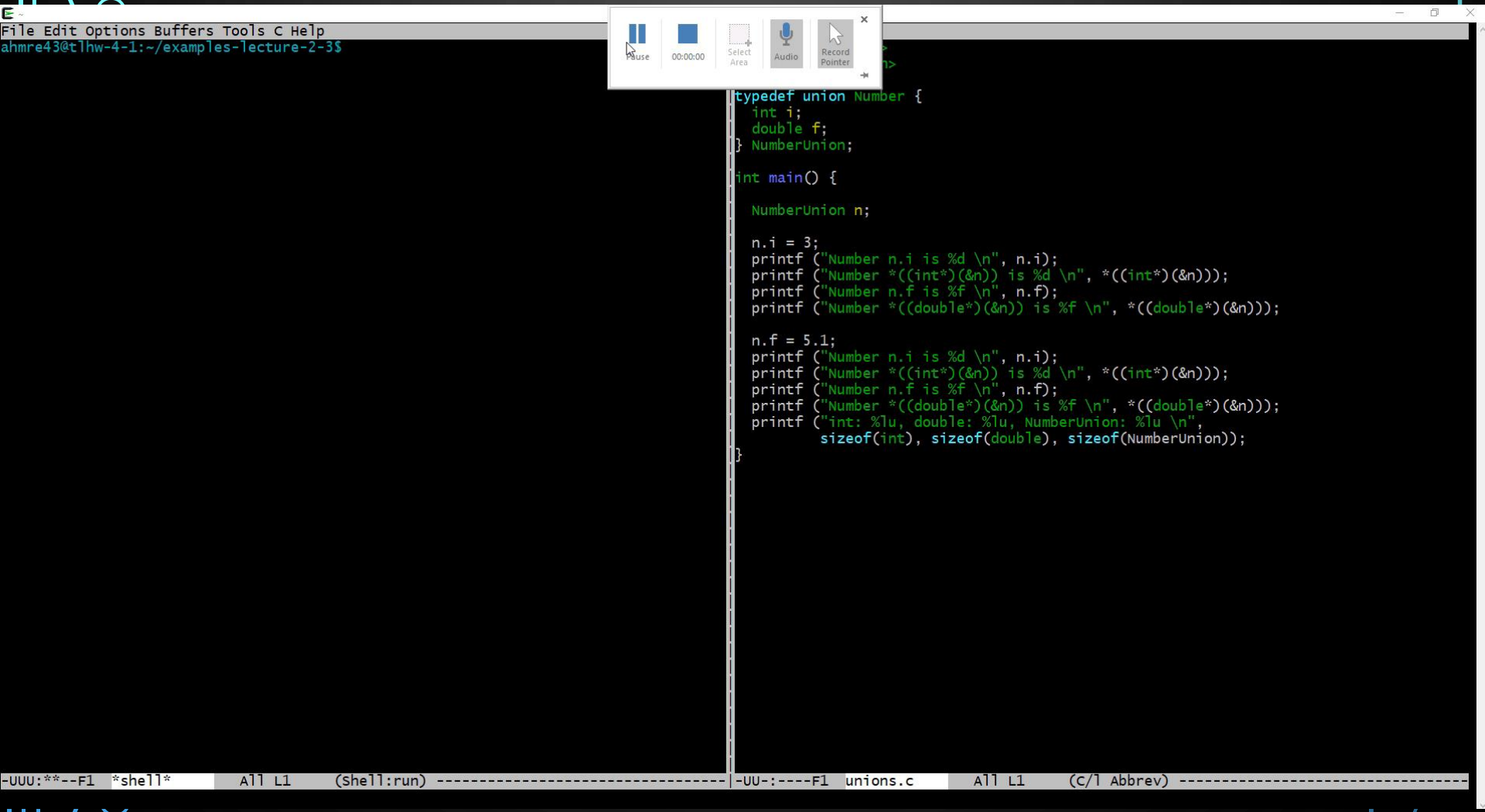
```
typedef union VARIANT {
 BYTE8      b[2] ;
 WORD16     w ;
} VARIANT ;
```

```
BOOL Kybd_Flags_Changed(KYBD_INFO *kybd)
{
      static WORD16 old_flags = 0xFFFF;
      VARIANT *flags = (VARIANT *) malloc(sizeof(VARIANT));
      dosmemget(0x417, sizeof(VARIANT), (void *) flags);

      kybd>both = flags->w;
      kybd->lo    = flags->b[0];
      kybd->hi    = flags->b[1];
      free(flags);

      if (kybd->both == old_flags) return FALSE;
      old_flags = kybd->both;
      return TRUE;
}
```

```
typedef struct KYBD_INFO
{
  BYTE8      lo ;
  BYTE8      hi ;
  WORD16  both ;
} KYBD_INFO ;
```

LET'S CODE!

union.c

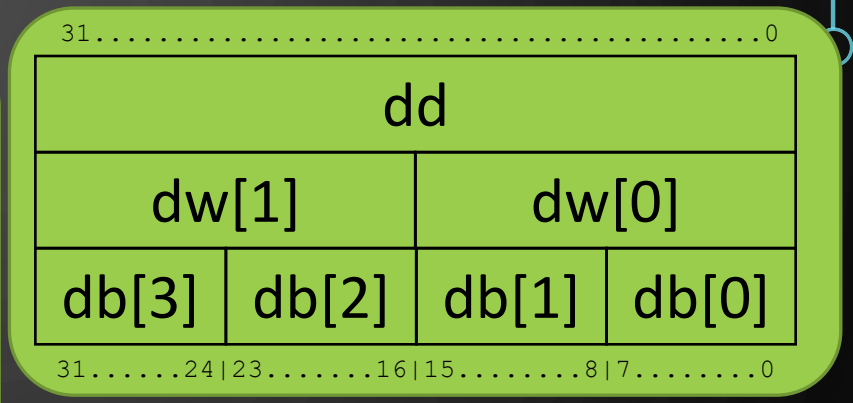

File Edit Options Buffers Tools C Help
ahmre43@tlhw-4-1:~/examples-lecture-2-3$
Pause
00:00:00
Select Area
Audio
Record Pointer
typedef union Number {
int i;
double f;
} NumberUnion;
int main() {
NumberUnion n;
n.i = 3;
printf ("Number n.i is %d \n", n.i);
printf ("Number *((int*)(&n)) is %d \n", *((int*)(&n)));
printf ("Number n.f is %f \n", n.f);
printf ("Number *((double*)(&n)) is %f \n", *((double*)(&n)));
n.f = 5.1;
printf ("Number n.i is %d \n", n.i);
printf ("Number *((int*)(&n)) is %d \n", *((int*)(&n)));
printf ("Number n.f is %f \n", n.f);
printf ("Number *((double*)(&n)) is %f \n", *((double*)(&n)));
printf ("int: %lu, double: %lu, NumberUnion: %lu \n",
sizeof(int), sizeof(double), sizeof(NumberUnion));
}
-UUU:**--F1 *shell* All L1 (Shell:run)
-UU-:----F1 unions.c All L1 (C/l Abbrev)

OUTLINE

- Pointers
- Structures
- Unions
- Endianness
- Bitfield

ENDIANNESS

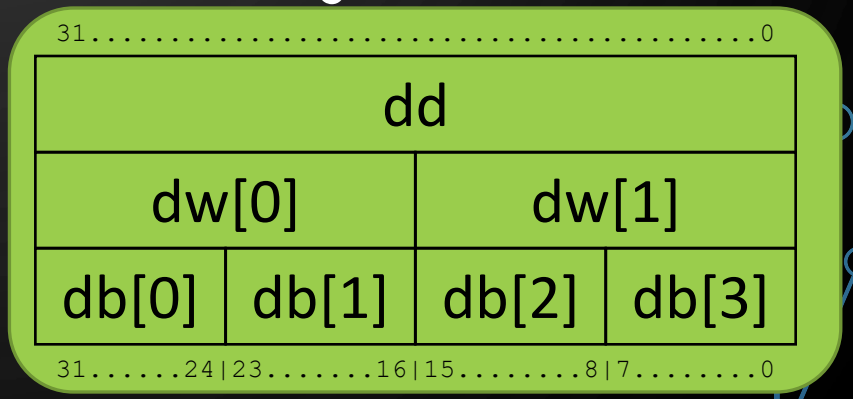
```
union
{
    unsigned long dd;
    unsigned short dw[2];
    unsigned char db[4];
};
```



Little endian

vs

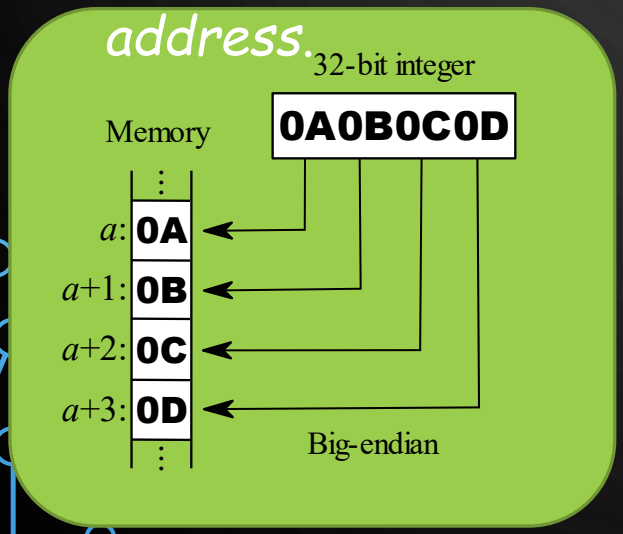
Big endian

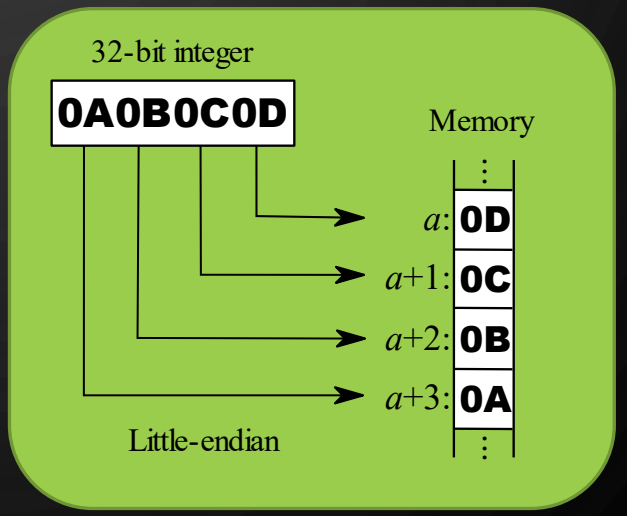


Endianness differ depending on architecture.
X86: little
Motorola, sparc: big

ENDIANNESS

- **Big-endian** systems are systems in which the *most significant byte* of the word is stored in the *smallest address* given and the least significant byte is stored in the largest. In contrast, **little endian** systems are those in which the *least significant byte* is stored in the *smallest address*.





WHY IS ENDIANNESS IMPORTANT FOR EMBEDDED SOFTWARE DEVELOPERS?

- Think about communication between two machines that have different Endianness
- One machine writes integers to a file and another machine with opposite Endianness reads it.
- Sending numbers over network between two machines with different Endianness. Think about serial communication when we split the data into multiple chunks!!

OUTLINE

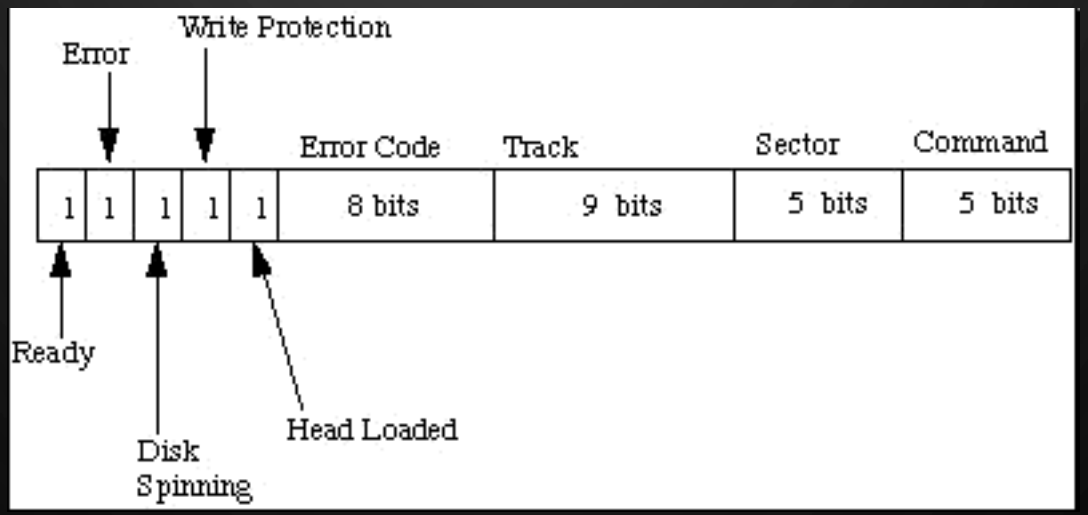
- Pointers
- Structures
- Unions
- Endianness
- Bitfield

BIT FIELD

- In embedded systems, storage is at a premium
- It may be necessary to pack several objects into one word
- Bit fields allow single bit objects
- They must be part of a structure

BITFIELD EXAMPLE

- Embedded systems must communicate with peripherals at low-level.
- A register of a disk controller, for example, has several fields.



- How can we represent this memory compactly?

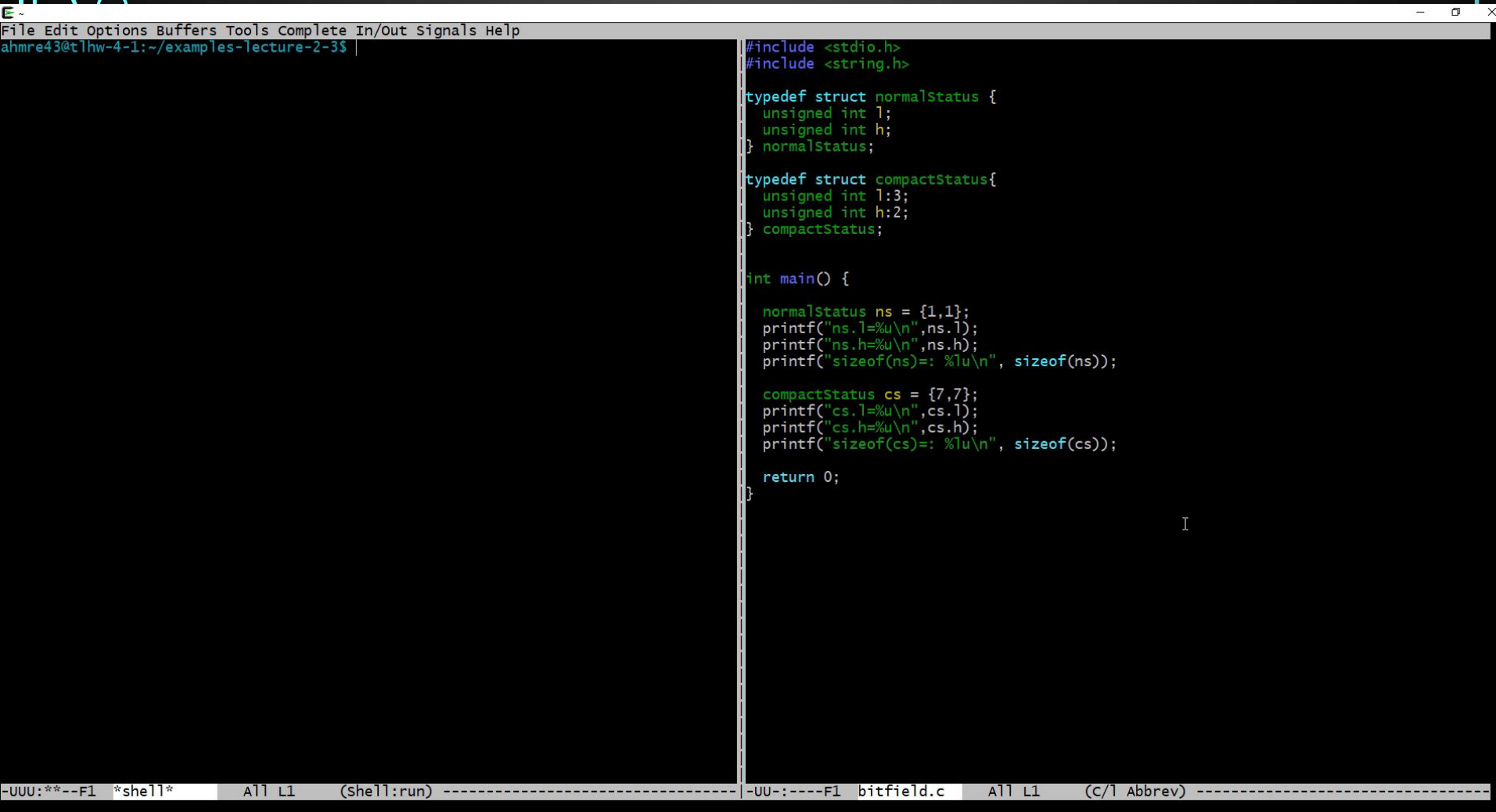
BIT FIELD EXAMPLE

```
struct DISK_REGISTER {
    unsigned int ready:1;
    unsigned int error_occured:1;
    unsigned int disk_spinning:1;
    unsigned int write_protect:1;
    unsigned int head_loaded:1;
    unsigned int error_code:8;
    unsigned int track:9;
    unsigned int sector:5;
    unsigned int command:5;
    };
```

- Bit fields must be part of a structure/union – stipulated by the C standard

LET'S CODE!

bitfield.c


File Edit Options Buffers Tools Complete In/Out Signals Help
ahmre43@tlhw-4-1:~/examples-lecture-2-3$
-UUU:**--F1 *shell* All L1 (Shell:run)
#include <stdio.h>
#include <string.h>
typedef struct normalStatus {
unsigned int l;
unsigned int h;
} normalStatus;
typedef struct compactStatus{
unsigned int l:3;
unsigned int h:2;
} compactStatus;
int main() {
normalStatus ns = {1,1};
printf("ns.l=%u\n",ns.l);
printf("ns.h=%u\n",ns.h);
printf("sizeof(ns)=: %lu\n", sizeof(ns));
compactStatus cs = {7,7};
printf("cs.l=%u\n",cs.l);
printf("cs.h=%u\n",cs.h);
printf("sizeof(cs)=: %lu\n", sizeof(cs));
return 0;
}
-UU-:----F1 bitfield.c All L1 (C/l Abbrev)