C Structures

Structures

- **Structures** – sometimes referred to as aggregates in the C standard – are collections of related variables under one name.
- Structures may contain variables of many different data types – in contrast to arrays, which contain only elements of the same data type.
- Structures are commonly used to define records to be stored in files.
- Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees.

Structures

- Structures are derived data types – they’re constructed using objects of other types.
- Structure Definition:

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

  - Keyword **struct** introduces a structure definition.
  - **The identifier card** is the structure tag, which names the structure definition and is used with struct to declare variables of the structure type.
  - Each structure definition must end with a semicolon.

Structures

- Variables declared within the braces of the structure definition are the **structure’s members**.
- Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict.
- The definition of struct card contains members face and suit, each of type char*.
- Structure members can be variables of the primitive data types (e.g., int, float, etc.), or aggregates, such as arrays and other structures.
- As we know, each element of an array must be of the same type. Structure members, however, can be of different types.

Structures

- The struct "employee" contains:
  - An employee’s first and last names
  - An unsigned int member for the employee’s age
  - A char member that would contain 'M' or 'F' for the employee’s gender
  - A double member for the employee’s hourly salary

```
struct employee {
    char firstName[20];
    char lastName[20];
    unsigned int age;
    char gender;
    double hourlySalary;
};
```

Structures

- A variable of a struct type cannot be declared in the definition of that same struct type.
- A pointer to that struct type, however, may be included.
- A structure containing a member that's a pointer to the same structure type is referred to as a **self-referential structure**.
- Self-referential structures are used for building linked data structures.

```
struct employee {
        char firstName[20];
        char lastName[20];
        unsigned int age;
        char gender;
        double hourlySalary;
        // ERROR
        struct employee teamLeader;
        struct employee *teamLeaderPtr;
};
```

Structures

- **Structure definitions do not reserve any space in memory.**
- Each definition creates a new data type that’s used to define variables – like a blueprint of how to build instances of that struct.
- Structure variables are defined like variables of other types.

```
struct employee emp1, emp2;
struct employee emps[5];
struct employee *empPtr;
```

Structures

- Structures can be initialized using initializer lists as with arrays.

  ```
  struct employee emp3 = { "Baris", "Manco", 23, 'M', 110.5, NULL };
  ```

  - If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to 0 (or NULL if the member is a pointer).
  - Structure variables defined outside a function definition (i.e., externally) are initialized to 0 or NULL if they’re not explicitly initialized in the external definition.

Structures

- Variables of a given structure type may also be declared by placing a comma separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition.

```
struct card {
    char *suit;
    char *face;
} card1, deck[52], *cardPtr;
```

  - The structure tag name (for this example, card) is optional.
  - If a structure definition does not contain a structure tag name, variables of the structure type may be declared only in the structure definition, not in a separate declaration.
  - Always provide a structure tag name when creating a structure type.

Structure Operations

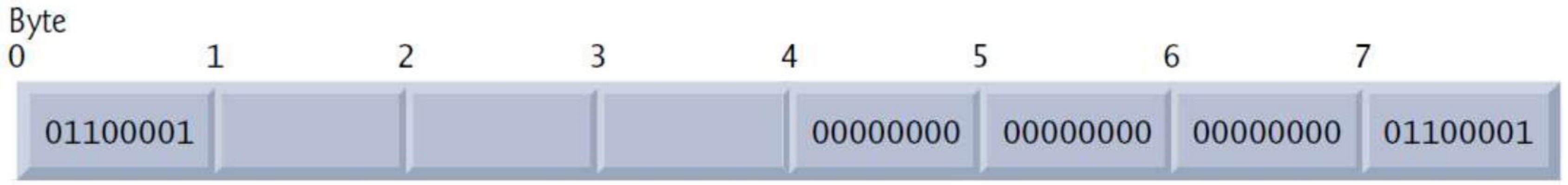
- The only valid operations that may be performed on structures are:
  - Using the **sizeof** operator to determine the size of a struct variable
  - **Taking the address (&)** of a struct variable
  - **Assigning struct variables** to struct variables of the same type – for a pointer member, this copies only the address stored in the pointer
  - **Accessing the members** of a struct variable
    - Using the structure member operator
    - Using the structure pointer operator

Data Alignment in Memory

- Structures may not be compared using operators == and !=, because **structure members are not necessarily stored in consecutive bytes of memory.**
- Sometimes there are “holes” in a structure, because **computers may store specific data types only on certain memory boundaries such as half-word, word or double-word boundaries.**
- A **word** is a memory unit used to store data in a computer – usually 4 bytes or 8 bytes.
- Always know and respect your system’s **memory alignment** requirements. It is practically impossible to master and apply memory bandwidth optimizations without respecting these requirements.
- Because the size of data items of a particular type is machine dependent and because storage alignment considerations are machine dependent, so is the representation of a structure.

Structure Member Alignment

```
struct example {
        char c;
        int x;
} sample1, sample2;
```



- A computer with 4-byte words might require that each member of struct example be aligned on a word boundary, i.e., at the beginning of a word – this is machine dependent.
- Figure shows a sample storage alignment for a variable of type struct example that has been assigned the character 'a' and the integer 97.
- If the members are stored beginning at word boundaries, there's a three-byte hole (bytes 1–3 in the figure) in the storage for variables of type struct example.
- **The value in the three-byte hole is undefined.**
- Even if the member values of sample1 and sample2 are in fact equal, the structures are not necessarily equal, because the undefined three-byte holes are not likely to contain identical values.

Accessing Structure Members

- **The structure member operator (.)**—also called the dot operator
  - accesses a structure member via the structure variable name.
- **The structure pointer operator (->)**—also called the arrow operator
  - accesses a structure member via a pointer to the structure.
  - consisting of a minus **(-)** sign and a greater than **(>)** sign **with no spaces**

```
struct card {
    char *suit;
    char *face;
} card1 = { "Hearts", "Ace" }, *cardPtr=&card1;
printf("%s\n", card1.suit);
printf("%s\n", cardPtr->face);
```

Accessing Structure Members

- The expression **cardPtr->face** is equivalent to **(*cardPtr).face**, which dereferences the pointer and accesses the member face using the structure member operator.
  - **The parentheses are needed** here because the structure member operator (.) has a higher precedence than the pointer dereferencing operator (*).
- The structure pointer operator and structure member operator, along with parentheses (for calling functions) and brackets ([]) used for array indexing, have the highest operator precedence and associate from left to right.

Passing Structures to Functions

- Structures may be passed to functions by
  - passing individual structure members
  - passing an entire structure
  - passing a pointer to a structure
- When structures or individual structure members are passed to a function, they're **passed by value**. Therefore, the members of a caller's structure cannot be modified by the called function.
- To pass a structure by reference, pass the address of the structure variable.
- Arrays of structures—like all other arrays—are automatically passed by reference.

Passing Structures to Functions

- You can use a structure to pass an array by value.
  - To do so, create a structure with the array as a member.
  - Structures are passed by value, so the array is passed by value.
- Passing structures by reference is more efficient than passing structures by value (which requires the entire structure to be copied).

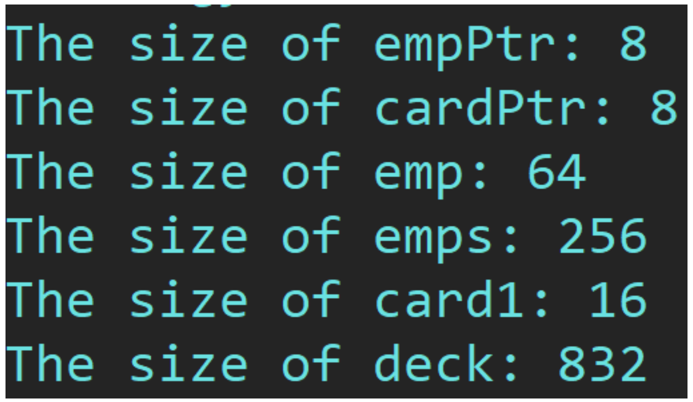
Structure Example

```
#include <stdio.h>
#include <string.h>
struct employee {
        char firstName[20];
        char lastName[20];
        unsigned int age;
        char gender;
        double hourlySalary;
        struct employee *teamLeaderPtr; };
void display1(struct employee);
void display2(struct employee*);
void assignTeamLeader1(struct employee*, struct employee*);
void assignTeamLeader2(struct employee*, struct employee);
```

```
char* suits[4] = {"Clubs", "Diamonds", "Hearts", "Spades"};
char* faces[13] = {"Two", "Three", "Four", "Five", "Six",
                "Seven", "Eight", "Nine", "Ten", "Jack",
                "Queen", "King", "Ace", };
struct {
        char* suit;
        char* face;
} deck[52], card1 = {"Hearts", "Ace"}, *cardPtr = &card1;
```

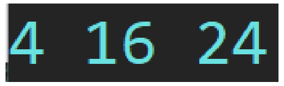
Structure Example

```
void main() {
        struct employee emp = {"Emel", "Kangal", 30, 'F', 275.5, NULL};
        struct employee *empPtr = &emp;
        struct employee emps[4] = {{"Ali", "Kartal", 35, 'M', 180.0, NULL},
                                   {"Tuna", "Kaplan", 25, 'F', 218.3, NULL},
                                   {"Onur", "Aslan", 33, 'M', 194.9, NULL}};
        printf("The size of empPtr: %lu\n", sizeof(empPtr));
        printf("The size of cardPtr: %lu\n", sizeof(cardPtr));
        printf("The size of emp: %lu\n", sizeof(emp));
        printf("The size of emps: %lu\n", sizeof(emps));
        printf("The size of card1: %lu\n", sizeof(card1));
        printf("The size of deck: %lu\n\n", sizeof(deck));
```



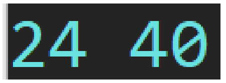
```
The size of empPtr: 8
The size of cardPtr: 8
The size of emp: 64
The size of emps: 256
The size of card1: 16
The size of deck: 832
```

Structure Example



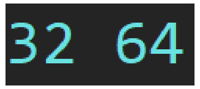
```
4 16 24
```

```
struct {
        char c;
        short x;
} align1;
printf("%lu ", sizeof(align1));
struct {
        double x;
        char c;
} align2;
printf("%lu ", sizeof(align2));
struct {
        short x;
        double y;
        char c;
} align3;
printf("%lu\n", sizeof(align3));
```



```
24 40
```

```
struct {
        short x;
        char c;
        double a;
        short y, z;
        int b;
} align4;
printf("%lu ", sizeof(align4));
struct {
        short a;
        double x;
        short b;
        double y;
        int z;
} align5;
printf("%lu\n", sizeof(align5));
```

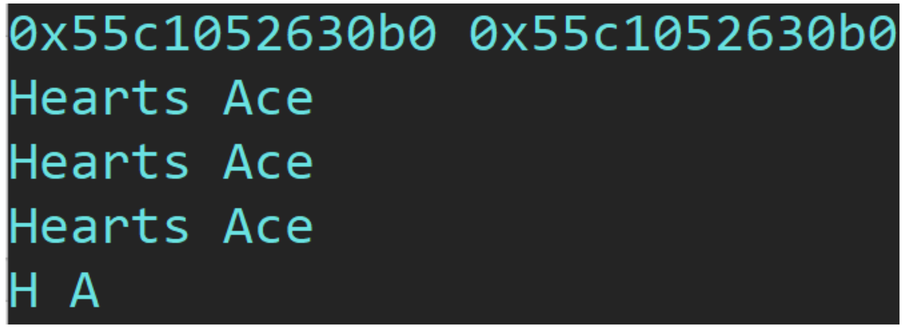


```
32 64
```

```
struct {
        short x;
        double a;
        char c;
        int b;
        short y;
} align6;
printf("%lu ", sizeof(align6));
struct {
        char s[21];
        int x;
        short a;
        char s2[19];
        double y;
} align7;
printf("%lu\n\n", sizeof(align7));
```

Structure Example

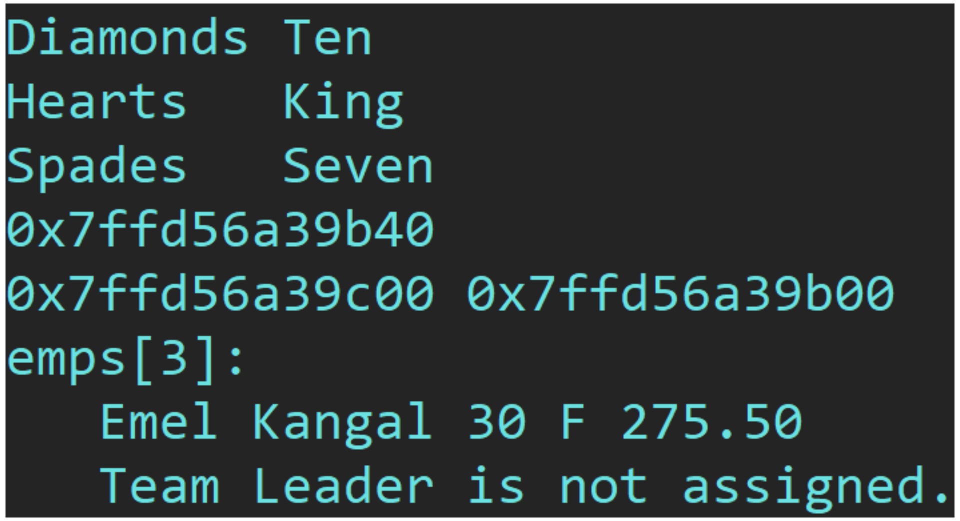
```
// printf("%p\n", card1);        // error
printf("%p %p\n", &card1, cardPtr);
printf("%s %s\n", card1.suit, card1.face);
printf("%s %s\n", cardPtr->suit, cardPtr->face);
printf("%s %s\n", (*cardPtr).suit, (*cardPtr).face);
printf("%c %c\n\n", *card1.suit, *card1.face);
for (int i=0; i<52; i++){
        deck[i].suit = suits[i/13];
        deck[i].face = faces[i%13];
}
/*
for (int i=0; i<52; i++){
        printf("%2d %-8s %s\n", i+1, deck[i].suit, deck[i].face);
}*/
```



```
0x55c1052630b0 0x55c1052630b0
Hearts Ace
Hearts Ace
Hearts Ace
H A
```

Structure Example

```
card1.suit = deck[8+13].suit;
cardPtr->face = deck[8+13].face;
printf("%-8s %s\n", card1.suit, card1.face);
card1 = deck[11+13*2];
printf("%-8s %s\n", card1.suit, card1.face);
*cardPtr = deck[5+13*3];
printf("%-8s %s\n", card1.suit, card1.face);
emps[3] = emp;
printf("%p\n%p %p\n", emps, &emps[3], &emp);
printf("emps[3]:\n");              display1(emps[3]);
printf("\n");
```



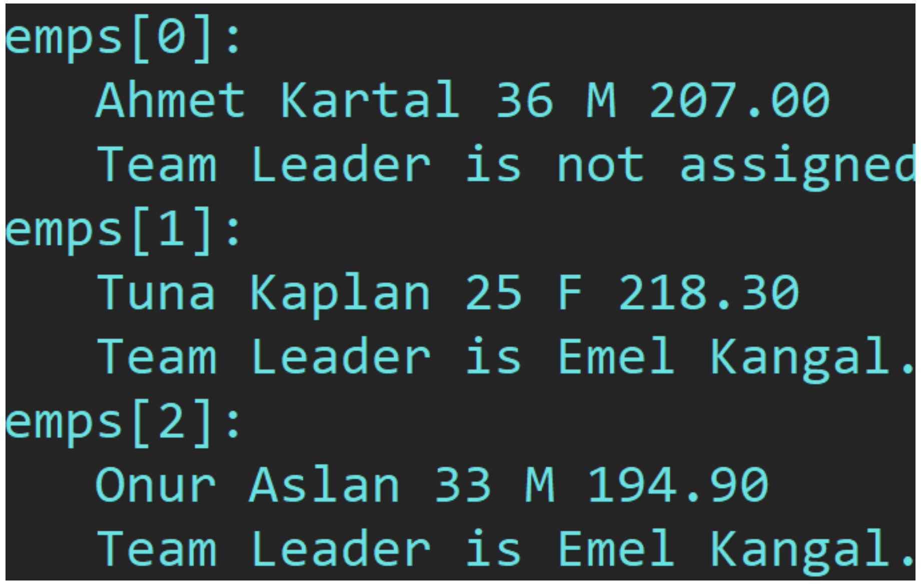
```
Diamonds Ten
Hearts   King
Spades   Seven
0x7ffd56a39b40
0x7ffd56a39c00 0x7ffd56a39b00
emps[3]:
   Emel Kangal 30 F 275.50
   Team Leader is not assigned.
```

Structure Example

```
void display1(struct employee emp){
        printf("   %s %s %d %c %.2f\n", emp.firstName, emp.lastName,
                        emp.age, emp.gender, emp.hourlySalary);
        if (emp.teamLeaderPtr == NULL)
                printf("   Team Leader is not assigned.\n");
        else    printf("   Team Leader is %s %s.\n",
        emp.teamLeaderPtr->firstName, emp.teamLeaderPtr->lastName);    }
void display2(struct employee *p){
        printf("   %s %s %d %c %.2f\n", p->firstName, p->lastName,
                        p->age, p->gender, p->hourlySalary);
        if (p->teamLeaderPtr == NULL)
                printf("   Team Leader is not assigned.\n");
        else    printf("   Team Leader is %s %s.\n",
        p->teamLeaderPtr->firstName, p->teamLeaderPtr->lastName);
}
```

Structure Example

```
emps[0].age++;
emps[0].hourlySalary *= 1.15;
// emps[0].firstName = "Ahmet";          // error
strcpy(emps[0].firstName, "Ahmet");
printf("emps[0]:\n");             display2(emps);
assignTeamLeader1(emps+1, emps+3);
printf("emps[1]:\n");             display2(emps+1);
assignTeamLeader2(emps+2, emps[1]);
printf("emps[2]:\n");             display2(emps+2);                 }
void assignTeamLeader1(struct employee *p1, struct employee *p2){
            p1->teamLeaderPtr = p2;
}
void assignTeamLeader2(struct employee *p, struct employee emp){
            p->teamLeaderPtr = emp.teamLeaderPtr;
}
```



```
emps[0]:
   Ahmet Kartal 36 M 207.00
   Team Leader is not assigned
emps[1]:
   Tuna Kaplan 25 F 218.30
   Team Leader is Emel Kangal.
emps[2]:
   Onur Aslan 33 M 194.90
   Team Leader is Emel Kangal.
```

Preprocessor Directives

- **Preprocessor directives**, such as #define, are typically used to make source programs easy to change and easy to compile in different execution environments.
- Directives in the source file tell the preprocessor to take specific actions.
- For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text.
- The number sign (#) must be the first nonwhite-space character on the line containing the directive.
- White-space characters can appear between the number sign and the first letter of the directive.

Preprocessor Directives

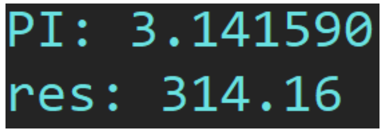
- #include
- #define
- #if expression
- #elif expression
- #else
- #endif
- #ifdef identifier
  - same as #if defined identifier
- #ifndef identifier
  - same as #if !defined identifier

Preprocessor Directives

```
#include <stdio.h>
#define mul(x1, x2) (x1 * x2)
#define NUM 100

#ifndef NUM
#define NUM 10000
#endif

#ifndef PRECISION
#define PRECISION 5
#endif
```

```
#ifndef PI
#if PRECISION==2
#define PI 3.14
#elif PRECISION==3
#define PI 3.141
#elif PRECISION==4
#define PI 3.1415
#else
#define PI 3.14159
#endif
#endif
```

```
void main() {
        printf("PI: %.6f\n", PI);
        printf("res: %.2f\n", mul(PI, NUM));
}
```



```
PI: 3.141590
res: 314.16
```

typedef

- The keyword **typedef** provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- typedef does not create a new type; typedef simply creates a new type name, which may be used as an alias for an existing type name.
- typedef is handled by the compiler – NOT the preprocessor.
- Names for structure types are often defined with typedef to create shorter type names.

```
typedef unsigned int uint;
uint x=10;
```

  - Defines the new type name **uint** as a synonym for **unsigned int**.

```
typedef struct card Card;
Card card1;
```

  - Define the new type name **Card** as a synonym for type **struct card**.

typedef

- C programmers often use typedef to define a structure type, so a structure tag is not required.
- Creates the structure type Card **without the need for a separate typedef statement**.

```
typedef struct {
      char *suit;
      char *face;
} Card;
Card deck[52];
```

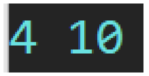
- A meaningful name helps make the program self-documenting. For example, when we read the previous declaration, we know “deck is an array of 52 Cards.”

typedef

- Often, typedef is used to create synonyms for the basic data types.
- For example, a program requiring four-byte integers may use type int on one system and type long on another.
- Programs designed for portability often use typedef to create an alias for four-byte integers, such as myUint32.
- The alias myUint32 can be changed once in the program to make the program work on both systems.
- **Use typedef to help make a program more portable.**

typedef

```
void main() {
        uint32_t x = 10;
        printf("%lu %d\n", sizeof(uint32_t), x);
}
```



```
4 10
```

```
#include <stdio.h>
#include <limits.h>
#if SHRT_MAX == 2147483647
typedef unsigned short uint32_t;
#elif INT_MAX == 2147483647
typedef unsigned int uint32_t;
#elif LONG_MAX == 2147483647
typedef unsigned long uint32_t ;
#elif LLONG_MAX == 2147483647
typedef unsigned long long uint32_t;
#else
#error "Cannot find integer of 4 bytes."
#endif
```

Unions

- **Like a structure, a union also is a derived data type, but with members that share the same storage space.**
- For different situations in a program, some variables may not be relevant, but other variables are—so a union shares the space instead of wasting storage on variables that are not being used.
- The members of a union can be of any data type.
- The amount of storage required to store a union is implementation dependent but will always be at least as large as the largest member of the union.

Unions

- The union definition is normally placed in a header and included in all source files that use the union type.
- A union definition has the same format as a structure definition.

```
union number {
    int x;
    double y;
};
```

- In a declaration, a union may be initialized with **a value of the same type as the first union member**.

```
union number num1 = { 10.9 };
```

  - truncates the initializer value's floating-point part, then assigns to num1.x

Unions

- In most cases, unions contain two or more data types.
- **Only one member, and thus one data type, can be referenced at a time.**
- It’s your responsibility to ensure that the data in a union is referenced with the proper data type.
- Referencing data in a union with a variable of the wrong type is a logic error.
- **If data is stored in a union as one type and referenced as another type, the results are implementation dependent.**

Union Operations

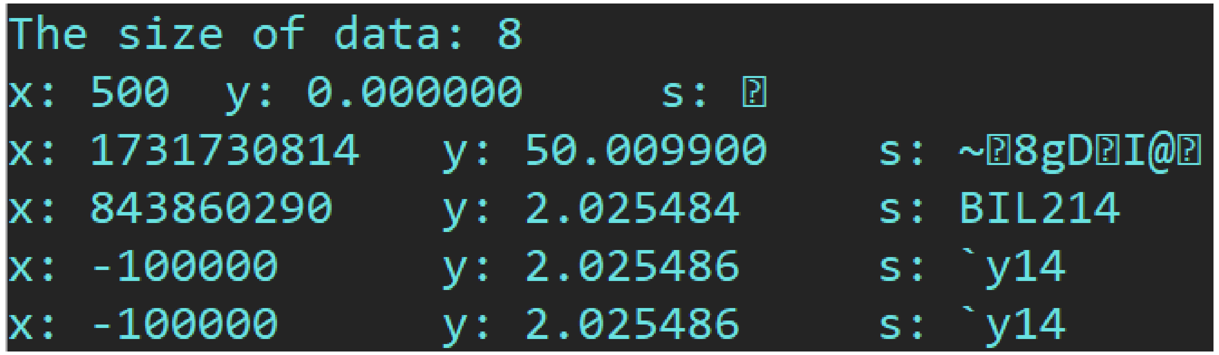
- The operations that can be performed on a union are:
  - Using the **sizeof** operator to determine the size of a union variable
  - **Taking the address (&)** of a union variable
  - **Assigning a union** to another union of the same type
  - **Accessing union members**
    - Using the structure member operator
    - Using the structure pointer operator
- Unions may not be compared using operators == and != for the same reasons that structures cannot be compared.

Anonymous Structures and Unions

- C11 supports anonymous structs and unions that can be nested in named structs and unions.
- The members in a nested anonymous struct or union are considered to be members of the enclosing struct or union and can be accessed directly through an object of the enclosing type.

```
struct MyStruct {
        int x;
        struct {
                int y;
                float z;
        };
} myStruct;
myStruct.y = 10;
```

Union Example 1



```
The size of data: 8
x: 500   y: 0.000000      s: �
x: 1731730814    y: 50.009900     s: ~�8gD�I@�
x: 843860290     y: 2.025484      s: BIL214
x: -100000       y: 2.025486      s: `y14
x: -100000       y: 2.025486      s: `y14
```

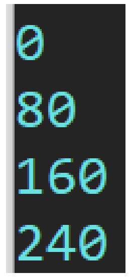
```
#include <stdio.h>
#include <string.h>

typedef union {
        int x;
        double y;
        char s[7];
} data;

void main() {
        data data1, data2 = { 500.9999 };
        printf("The size of data: %lu\n", sizeof(data));
        printf("x: %d\ty: %f\ts: %s\n", data2.x, data2.y, data2.s);
        data1.y = 50.0099;
        printf("x: %d\ty: %f\ts: %s\n", data1.x, data1.y, data1.s);
        strcpy(data1.s, "BIL214");
        printf("x: %d\ty: %f\ts: %s\n", data1.x, data1.y, data1.s);
        data1.x = -100000;
        printf("x: %d\ty: %f\ts: %s\n", data1.x, data1.y, data1.s);
        data2 = data1;
        printf("x: %d\ty: %f\ts: %s\n", data2.x, data2.y, data2.s);
}
```

Union Example 2

- Unions are particularly useful in Embedded programming or in situations where direct access to the hardware/memory is needed.

```
#include <stdio.h>
typedef union {
	struct {
		unsigned char byte1;
		unsigned char byte2;
		unsigned char byte3;
		unsigned char byte4;
	};
	unsigned int word;
} Register;
```

```
void main() {
		Register reg1;
		reg1.word = 0xF0A05000;
		printf("%d\n", reg1.byte1);
		printf("%d\n", reg1.byte2);
		printf("%d\n", reg1.byte3);
		printf("%d\n", reg1.byte4);
}
```



```
0
80
160
240
```

Enumerations

- An enumeration introduced by the keyword **enum**, is a **set of integer enumeration constants** represented by identifiers.
- Values in an enum start with 0, unless specified otherwise, and are incremented by 1.
- For example, the following enumeration creates a new type, enum months, in which the identifiers are set to the integers 0 to 11, respectively.

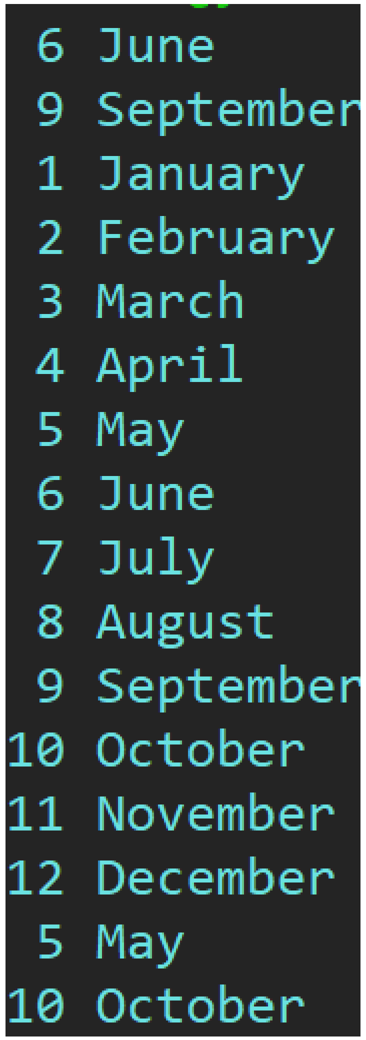
```
enum months { JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

- To number the months 1 to 12, use:

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

Enumeration Example 1

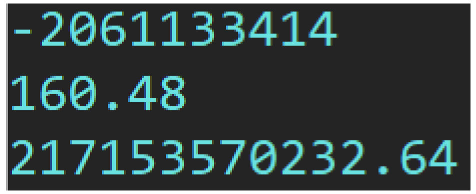
```
#include <stdio.h>
const char *monthNames[] = {"", "January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October", "November", "December"};
enum months1 { JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
void main() {
        enum months1 m1 = JUN;
        printf("%2d %s\n", m1, monthNames[m1]);
        m1 += 3;
        printf("%2d %s\n", m1, monthNames[m1]);
        for (enum months1 m = JAN; m<=DEC; m++)
                printf("%2d %s\n", m, monthNames[m]);
        enum months2 { JAN=1, FEB, MAY=5, JUN, JUL, OCT=JUL+3, NOV, DEC };
        enum months2 m2 = MAY;
        printf("%2d %s\n", m2, monthNames[m2]);
        m2 = OCT;
        printf("%2d %s\n", m2, monthNames[m2]);
}
```



```
 6 June
 9 September
 1 January
 2 February
 3 March
 4 April
 5 May
 6 June
 7 July
 8 August
 9 September
10 October
11 November
12 December
 5 May
10 October
```

Pseudo-Polymorphism Implementation

```
#include <stdio.h>
enum numberType { INT, FLOAT, DOUBLE };
typedef struct {
        enum numberType type;
        union {
                int nints[2];
                float nfloats[2];
                double ndouble;
        };
} number;
```



```
-2061133414
160.48
217153570232.64
```

```
void operate(number *p) {
        switch(p->type) {
        case INT: printf("%d\n", p->nints[0]+p->nints[1]); break;
        case FLOAT: printf("%.2f\n", p->nfloats[0]+p->nfloats[1]);
                break;
        case DOUBLE: printf("%.2f\n", p->ndouble); break;
        default: break;     }          }
void main() {
        number num1;
        num1.nfloats[0] = 110.16;
        num1.nfloats[1] = 50.32;
        for (enum numberType t=INT; t<=DOUBLE; t++){
                num1.type = t;
                operate(&num1);
        }
}
```