union members

↳ user_define data types [structures_like]

But union members share same memory location [unlike_structures]

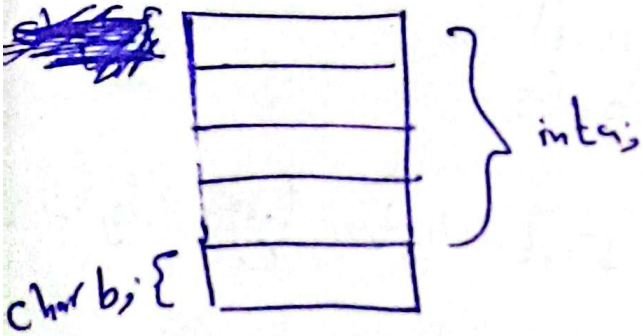
Ex: struct abc {
int a; → &a ≠ &b
char b; →
};

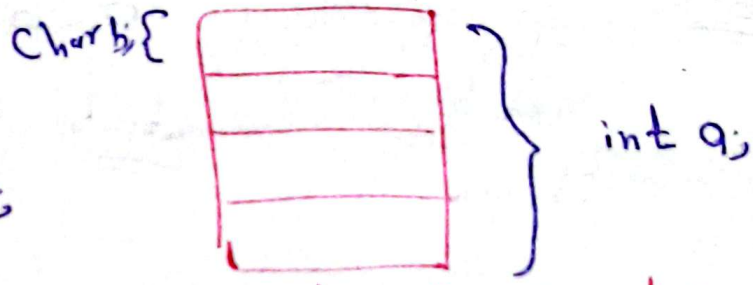
But union abc {
int a; →
char b; →
}; &a = &b

a and b share the same memory location

Ex Memory allocation:

struct abc





Both have the same base address

if you make a change on one member, the other members will be affected

Ex: union abc {
int a;
char b;
}vars;

Var.a = 65;

→ output: a: 65
b: A ← ASCII Code

* size(union) = size(largest member)

Ex:
```
union abc{
    char a;
    int b;
    float c;
};
```
→ size(abc) = 8 bytes

Like structures, you can access members of a union using pointer by (->) operator

Ex:
```
union {
    char a;
    int b;
} Var;
```

```
union abc *ptr = &Var;
P->a = 65;
```

output: a = A
b = 65

Example

Assume short: 2bytes, Float: 4 bytes, long = 8 bytes

ignoring alligument Consideration:

```
struct {
    short s[5];
    union {
        float y;
        long z;
    } u;
} t;
```

5x2 = 10 bytes

long z → 8 bytes

size(t) = ??

10 + 8 = 18 bytes

Applications of unions

① Assume a store selling Books, shirts

Saving memory

Books → title, author, num_Pages, Price

shirts → size, Color, Price

Common variables

First design: Struct

```
struct store {
8      double Price;
8x2    char* title, *author;
4      int num_Pages;
4      int Color;
4      int size;
};
```

→ 36 Bytes ignoring Padding

if you make an object for a:

book → waste Color, size: 8 bytes

shirt → waste: title, author, num: 20 bytes

struct store s;

To make an object of:

book: s.item.book.title = " ";

Well design:

```
struct store {
8      double Price; → Common
union {struct {                2
        char* title, *author;  16
        int num_Pages; } 4 book;
8 - struct { int Color, size; } shirt;
        } item; };
```

4 4

8 + 20 = 28 bytes

Application of unions ②.

To make a list [Array of different data types]

```
typedef union {
    int a;
    char b;
    double c;
} data;
```

```
data arr[10];
arr[0].a = 10;
arr[1].b = 'C';
arr[2].c = 7.777;
```

No structures → Waste memory

So unions also save memory here!

Enum,

↳ user-defined data type used to assign names to integral Constants

```
enum Bool {
    False, → 0
    True → 1
};
```

```
enum Bool Var = True;
Printf("%d", Var); → // 1
```

Why don't use #define?

enum

declared in Local Scope

Automatically assign integral constants to names of enum

Notes:

you Can assign numbers to the names by yourself and two or more names Can have Same Value

Ex:

```
enum Point {x=0, y=0, z=1};
Printf("%d %d %d", x,y,z); → // 0 0 1
```

* you Can use the names directly

* you also Can assign numbers in any order, the unassigned names get Value more than the Previous by +1

Ex

```
enum Point {x=2, y=1, z}
```

↳ z = 2

* only integral values are allowed

Ex: enum Point {X=34, y=2.5, Z=0};

Compile error

*** All enum names must be unique for the same scope

Ex: enum Point1 {X=34, y=2, Z=0};
enum Point2 {X=4, P=25, q=1};

error

---

sizeof enum is 4 → fixed

like integers

Bit Fields

sizeof ( ) : operator used to calculate size of its operands in units of bytes

operand

① data type

sizeof(int) = 4

sizeof(double) = 8

② expression

int a = 0; double d = 10.21;

sizeof((a+d)) = 8 bytes

int Promoted to double

uses : ① Calculating size of structs, arrays

② no. of elements of array * Length = $\frac{\text{sizeof(array)}}{\text{sizeof(arr[0])}}$

③ Dynamic memory Allocation [Heap]

int *ptr = malloc(10 * sizeof(int));

BitField : specify size of structure or unions members in Bits Not Bytes

memory efficient

Syntax: used with unsigned int, int only

→ unsigned int memberName : bitWidth;

Ex:

```
typedef struct {
  - unsigned int bit0 : 1;
  - unsigned int bit1 : 3;
}
```

Bit Name ↑ Bit width ↑

Note
No Pointers for Bit Fields
No Array of Bit fields

Note: BitWidth ∈ [1, 32]

Value of BitField: $[0, 2^{n}-1]$ → unsigned int (Bit width)

$[-2^{(n-1)}-1, 2^{n-1}]$ → int

Ex:

```
struct byteBits {
  unsigned a: 1;
  unsigned b: 2;
  unsigned c: 5;
} x;
```

x.a = 1; decimal

x.b = 0b10; Binary

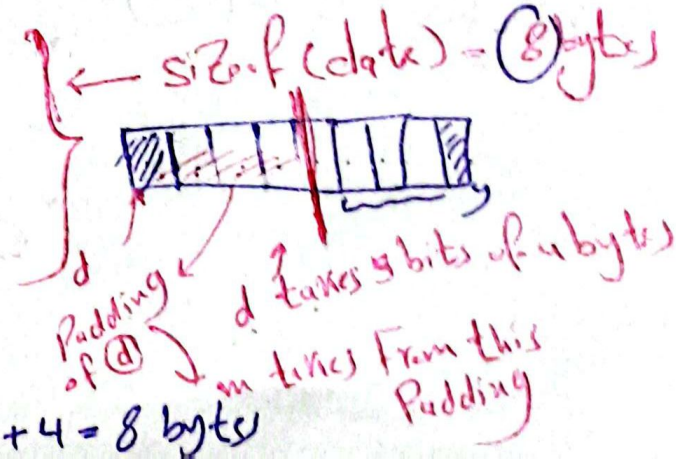
x.c = 0x0; Hexadecimal

Ex:

```
typedef struct {
  unsigned int d: 5;
  unsigned int m: 4;
  unsigned int y;
} date;
```

1→31

1→12 ~~/15~~



typedef

↳ define new types of existing data type(s)

→ Famous typedefs.

* typedef unsigned char uint8_t;

old Name

8 bits

New name

int

unsigned

typedef "Convension" For Readability

* typedef signed char int8_t;

* uint16_t → unsigned short
  int16_t → signed short
  uint32_t → unsigned int
  int32_t → signed int
  uint64_t → unsigned long long
  int64_t → signed long long

Note: You should pre-define them before using
They are not defined by default
or include <stdint.h> Library

Complex declaration

typedef old Name New name;

Keyword

Ex: typedef int INTEGER;

typedef unsigned char arr of five chars_t [5];

arr of Five chars_t x;

∴ x → array of Five chars

typedef struct Point {int x, y, z;} Point_t;