The C - Assembly connection

Systems Software

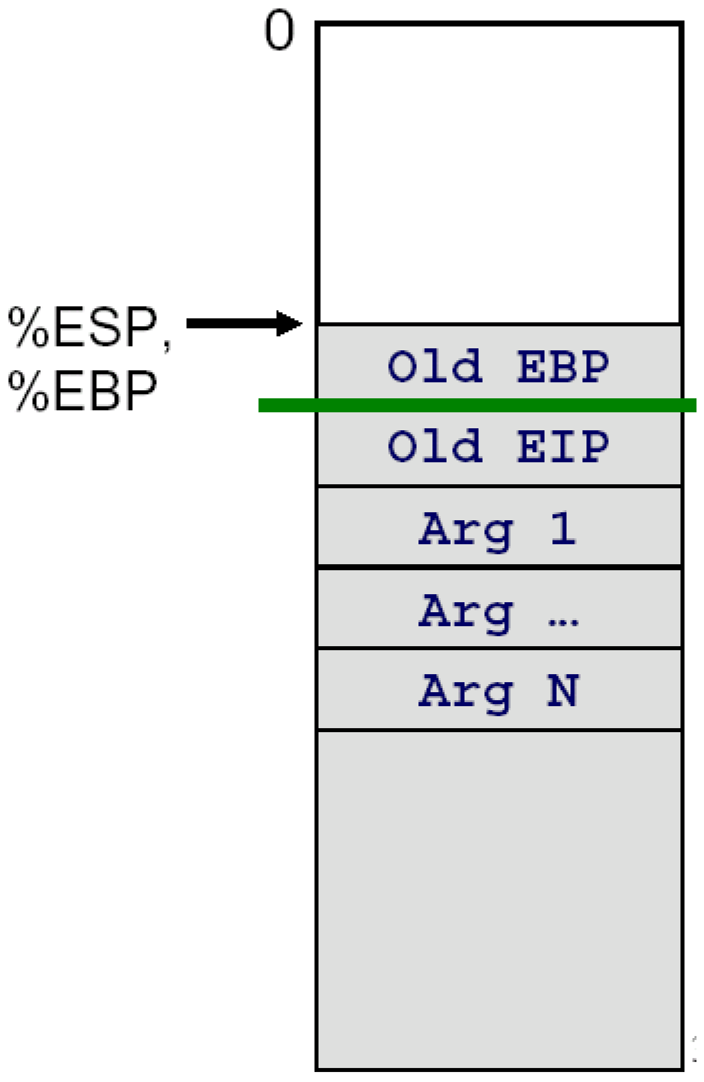
Displacement addressing

- **Used to access a memory address plus a constant displacement**
  - movl disp(reg1), reg2
    - reg1, reg2 are registers
    - disp is a 32-bit constant displacement value
    - Moves the contents of memory at address (reg1 + disp) to reg2

- **Example:** `movl 8(%ebp), %edx`
  - Moves value at memory address %ebp + 8 to %edx

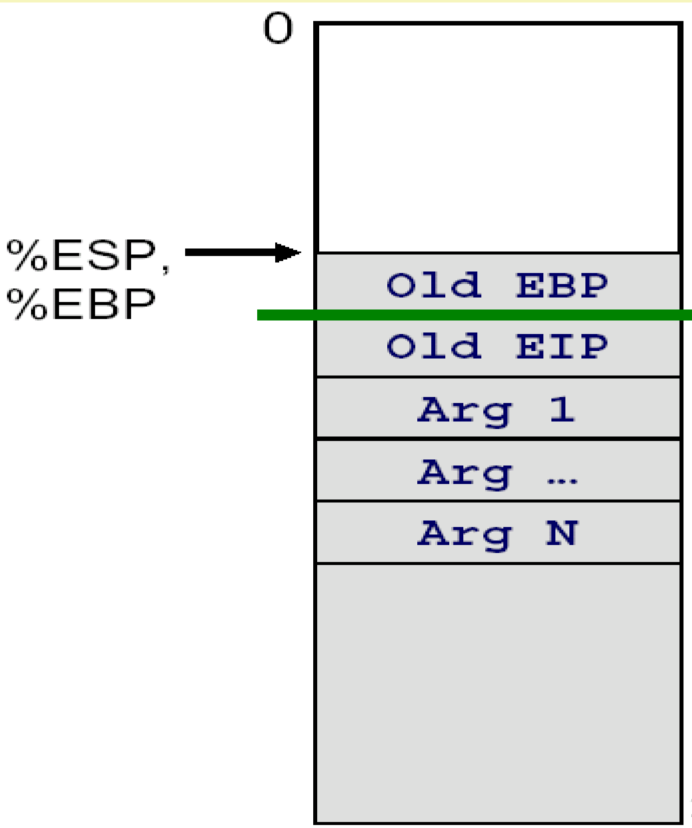
Base Pointer: EBP

- **Save caller's base pointer**
  - pushl %ebp
- **Set value of EBP for callee's use**
  - Current value of stack pointer used as callee's base pointer
  - movl %esp, %ebp



Using displacement addressing

```
/* sum first 3 elements of
array a and store result in
a[3]. */
void sumup(int *a) {
  int temp;
  temp = a[0] + a[1] + a[2];
  a[3] = temp;
}
```




```
sumup:
pushl %ebp
movl  %esp,%ebp
movl  8(%ebp), %edx
```

Set Up

```
movl 4(%edx), %eax
addl (%edx), %eax
addl 8(%edx), %eax
movl %eax, 12(%edx)
```

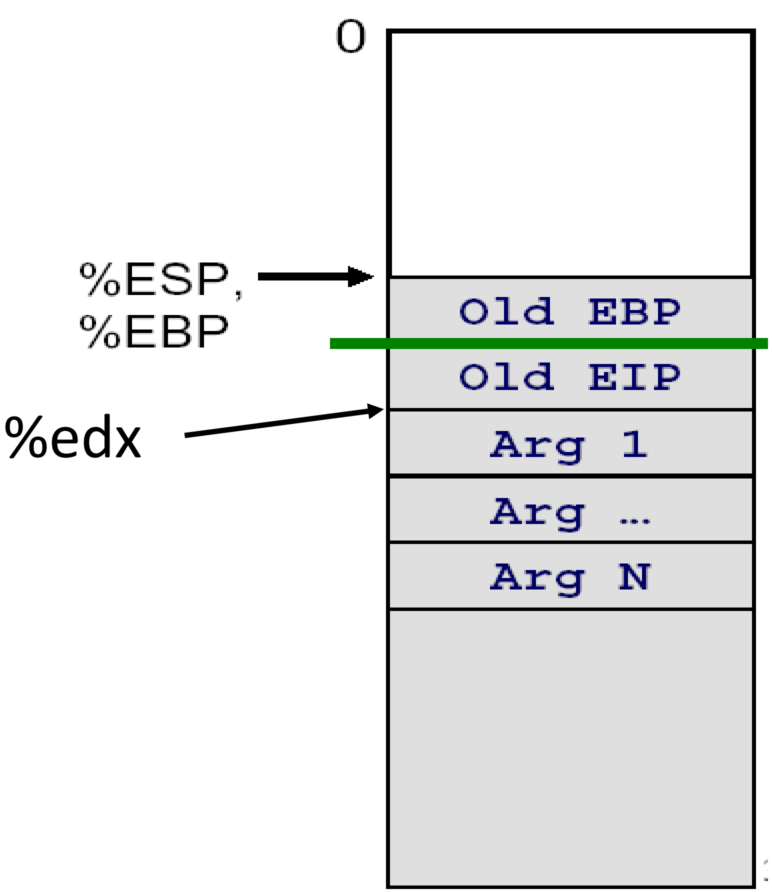
Body

```
popl %ebp
ret
```

Finish

Using displacement addressing

```
void sumup(int *a) {
  int tmp;
  temp = a[0] + a[1] + a[2];
  a[3] = tmp;
}
```




```
sumup:
pushl %ebp
movl  %esp,%ebp
movl  8(%ebp), %edx
```

Set Up

```
movl 4(%edx), %eax
addl (%edx), %eax
addl 8(%edx), %eax
movl %eax, 12(%edx)
```

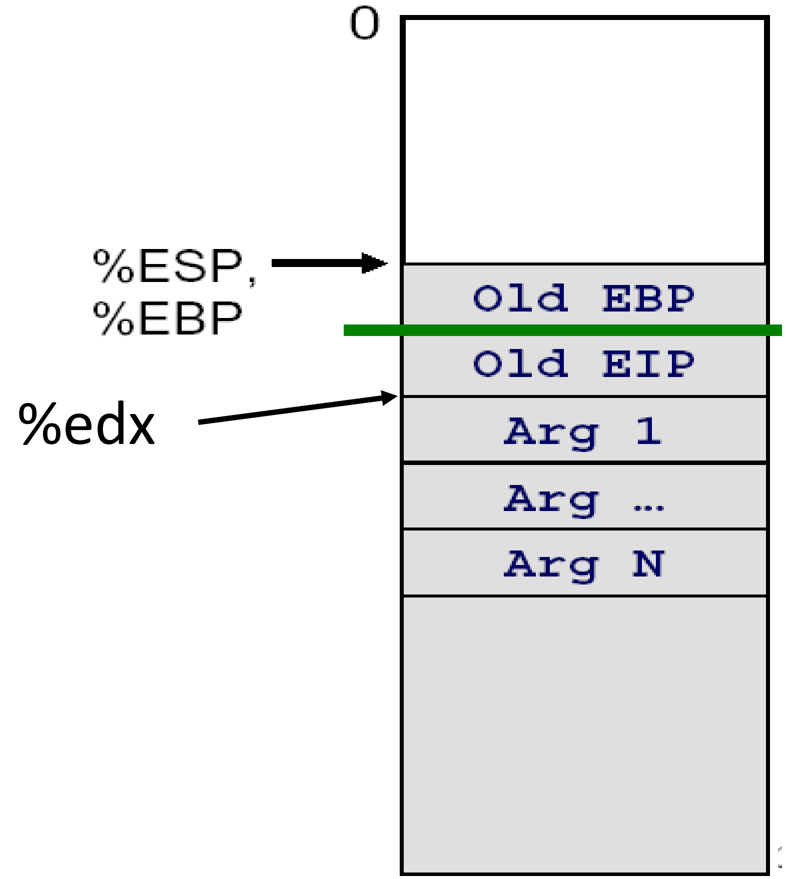
Body

```
popl %ebp
ret
```

Finish

Understanding sumup

```
/* sum first 3 elements
 * of array a and store result
 * in a[3].
*/
void sumup(int *a) {
  int tmp;
  temp = a[0] + a[1] + a[2];
  a[3] = temp;
}
```




```
Value of a (address of first array element) in %edx
movl 4(%edx), %eax      ; %eax = a[1]
addl (%edx), %eax       ; %eax += a[0]
addl 8(%edx), %eax      ; %eax += a[2]
movl %eax, 12(%edx)     ; a[3] = %eax
```

Indexed addressing

- **A very general purpose addressing mode**
  - **disp(reg1, reg2, scale)** accesses memory address: reg1 + (reg2 * scale) + disp
    - disp: constant displacement (32 bits)
    - reg1: Base register: Any of the 8 general-purpose registers
    - reg2: Index register: Any register except for %esp
    - scale: Must be 1, 2, 4, or 8

- **Example:** `movl 10(%eax, %ebx, 4), %ecx`
  - Accesses memory at address %eax + (%ebx * 4) + 10
  - Stores result in %ecx

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

Expression	Computation	Address
`0x8(%edx)`		
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

Expression	Computation	Address
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(,%edx,2)`		

Address Computation Examples

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)		
0x80(,%edx,2)		

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

Expression	Computation	Address
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%edx,2)`		

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

Expression	Computation	Address
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%edx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Computing addresses: `leal` instruction

- **Sometimes want to save the result of an address computation for later use**
  - Rather than reading or writing memory at the computed address
- **The `leal` instruction is used for this: "load effective address"**
- **Example: `leal 4(%eax, %ebx, 8), %ecx`**
  - Computes the address of the "source" argument, stores in the destination
  - Does not access the memory address itself: use **`movl`** for that!
- **Can also be used to compute arithmetic expressions!**
  - Anything of the form x + y*k, where k = 1, 2, 4, or 8

A more complex example

- **What does this code do?**

```
# Address of array in %ecx
movl 8(%ecx), %eax
movl 4(%ecx), %edx
leal (%edx, %eax, 2), %eax
movl (%ecx), %edx
leal (%edx, %eax, 2), %eax
movl  %eax, 12(%ecx)
```

A more complex example

- What does this code do?

```
# Address of array[] in %ecx
movl 8(%ecx), %eax              ; eax = array[2]
movl 4(%ecx), %edx              ; edx = array[1]
leal (%edx, %eax, 2), %eax      ; eax = array[1] + (2*array[2])
movl (%ecx), %edx               ; edx = array[0]
leal (%edx, %eax, 2), %eax      ; eax = array[0] + (2 * eax)
movl  %eax, 12(%ecx)            ; array[3] = eax
```

- Original source code

```
void dosomething(int *array) {
        int tmp;
        tmp = array[0] + (2 * array[1]) + (4 * array[2]);
        array[3] = tmp;
}
```

Structures

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Structures

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

Memory Layout

- **Characteristics**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structures

```
struct rec {
   int i;
   int a[3];
   int *p;
};
```

- **Accessing Structure Member**
  - Given an instance of the struct, we can use the `.` operator:
    - `struct rec r1;  r1.i = val;`
  - What if we have a *pointer* to a struct: `struct rec *r = &r1;`

Structures

```
struct rec {
   int i;
   int a[3];
   int *p;
};
```

- **Accessing Structure Member**
  - Given an instance of the struct, we can use the `.` operator, just like Java:
    - **`struct rec r1;   r1.i = val;`**
  - What if we have a *pointer* to a struct: **`struct rec *r = &r1;`**
    - Using `*` and `.` operators: `(*r).i = val;`
    - Or, use `->` operator for short: `r->i  = val;`
  - Pointer indicates first byte of structure; access members with offsets

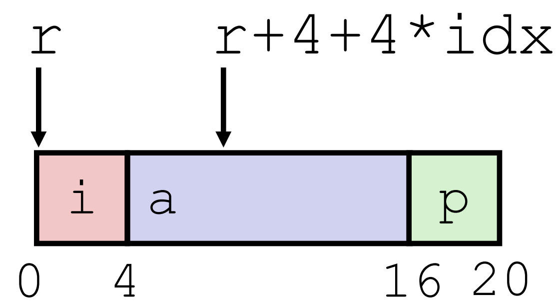
```
void
set_i(struct rec *r,
      int val)
{
     r->i = val;
}
```

IA32 Assembly

```
 # %eax = val
 # %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

Generating Pointer to Structure Member

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time

```
int *find_a
 (struct rec *r, int idx)
{
   return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Practice question

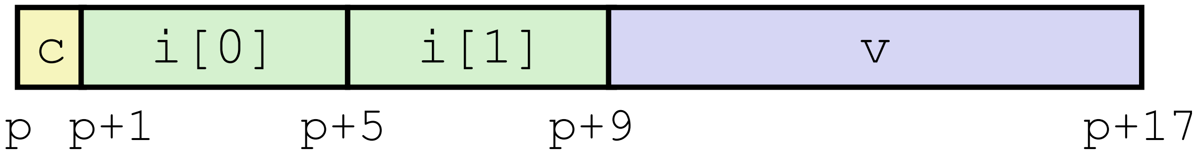
```
struct prob {
  int *p;
  struct {
     int x;
     int y;
  } s;
  struct prob *next;
} ;
```

```
void sp_init(struct prob *sp)
{
   sp->s.x  = __________;
   sp->p    = __________;
   sp->next = __________;
}
```

```
      movl 8(%ebp), %eax
      movl 8(%eax), %edx
      movl %edx, 4(%eax)
      leal 4(%eax), %edx
      movl %edx, (%eax)
      movl %eax, 12(%eax)
```

Structures & Alignment

- **Unaligned Data**

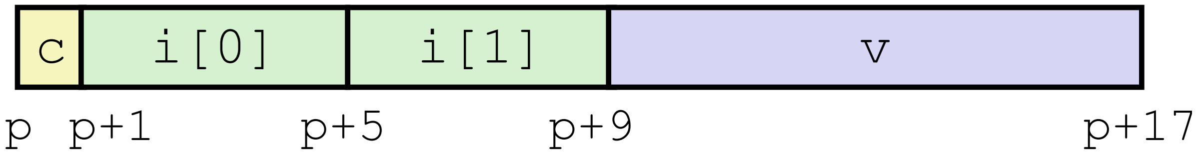



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- **How would it look like if data items were *aligned* *(address multiple of type size)*?**

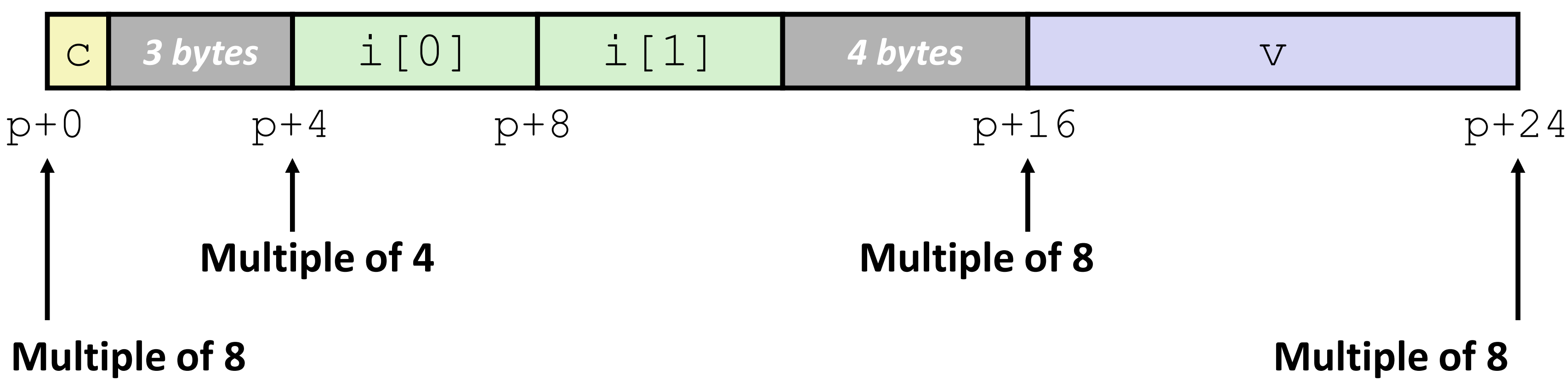
Structures & Alignment

- **Unaligned Data**




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



Alignment Principles

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K
- **Aligned data is required on some machines; it is *advised* on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **What is the motivation for alignment?**

Alignment Principles

- **Motivation for Aligning Data**
  - Physical memory is accessed by aligned chunks of 4 or 8 bytes (system-dependent)
    - Inefficient to load or store datum that crosses quad word boundaries

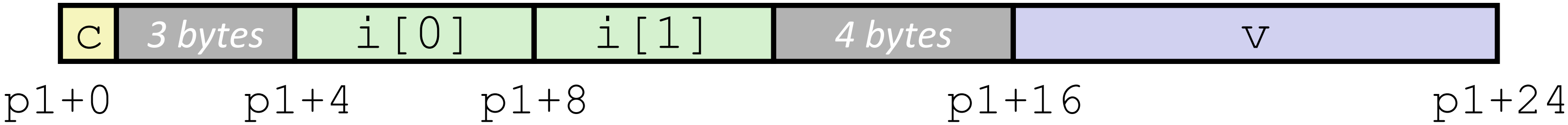
- **Compiler**
  - Inserts padding in structure to ensure correct alignment of fields
  - **`sizeof()`** should be used to get true size of structs

Specific Cases of Alignment (IA32)

- **1 byte: char, …**
  - no restrictions on address
- **2 bytes: short, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: int, float, char *, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: double, …**
  - Windows (and most other OSs & instruction sets): lowest 3 bits 000_2
  - Linux: lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type

Saving Space

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```



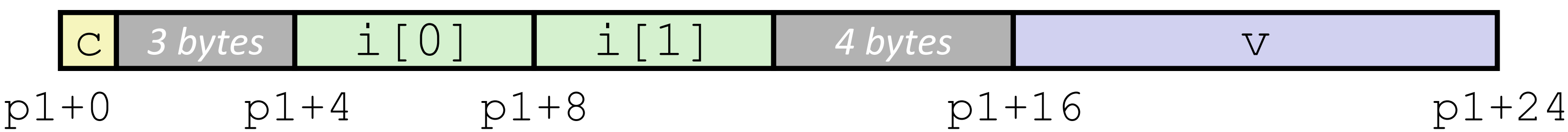
Saving Space

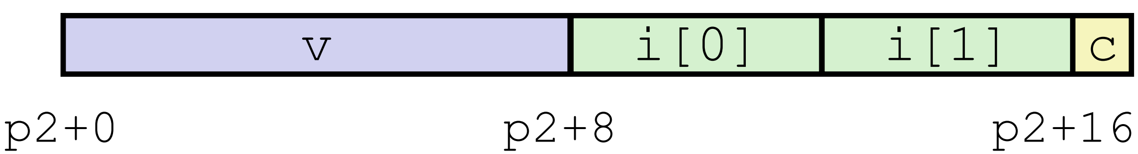
- **Put large data types first:**

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p2;
```

- **Effect (both have K=8)**





Unfortunately, doesn't satisfy requirement that struct's *total size* is a multiple of K

Arrays of Structures

- **Satisfy alignment requirement for every element**
- **How would accessing an element work?**

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```