Machine-Level Programming: Compound data types

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

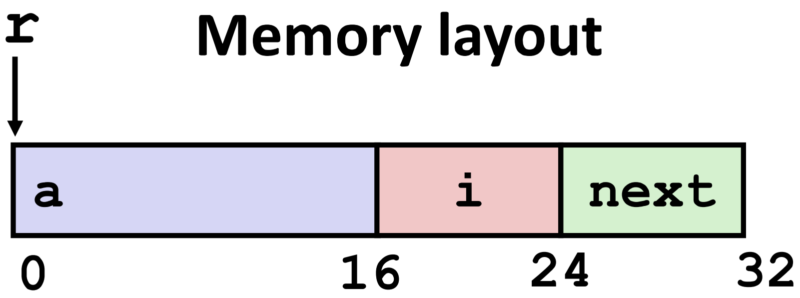
Today

- **Structures**
  - Allocation
  - Access
- **Data alignment**
- **Unions**

Structure allocation

```
struct rec {
  int a[4];
  long i;
  struct rec *next;
};
```

Memory layout



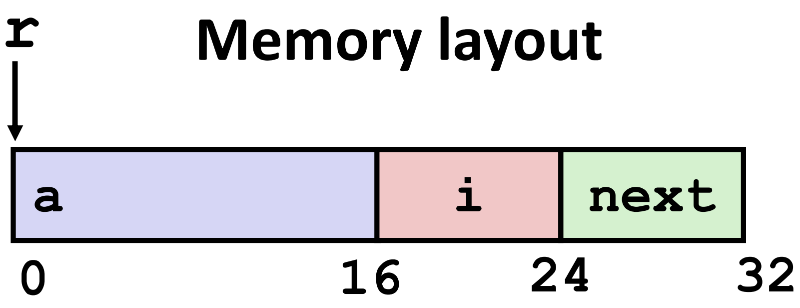
- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types
  - Compiler determines overall size and positions of fields
- **Fields ordered according to declaration**
  - Even if another ordering could yield a more compact representation

Declaring `struct` variables in C

- `struct rec r1, r2, r3;`
  - Declares and sets aside storage for three variables – `r1`, `r2`, and `r3` – each of type `struct rec`
- `struct rec *ptr;`
  - Declares a pointer to an object of type `struct rec`
- `struct rec vec[25];`
  - Declares a 25-element array of `struct rec`
  - Allocates 25 units of storage, each one big enough to hold the data of one *struct* rec

Structure access in C

```
struct rec {
  int a[4];
  long i;
  struct rec *next;
};
```

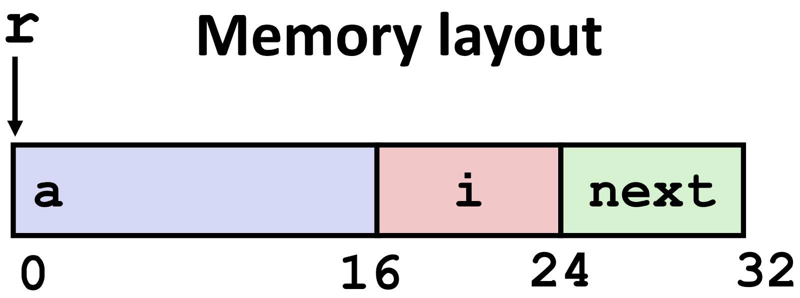


- **Given an instance of a structure:** `struct rec r1;`
  - `r1.i = val;`

- **Given a *pointer* to a structure:** `struct rec* r = &r1;`
  - Using * and . operators: `(*r).i = val;`
  - Or, use -> operator for short: `r->i = val;`

Structure access in Assembly

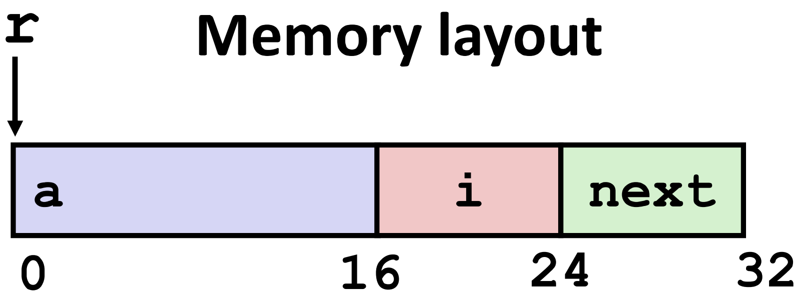
```
struct rec {
  int a[4];
  long i;
  struct rec *next;
};
```



- Machine-level program has no understanding of structures in source C code
- Address indicates first byte of structure
- Access elements with offsets to the address of the structure

Structure access in Assembly

```
struct rec {
  int a[4];
  long i;
  struct rec *next;
};
```



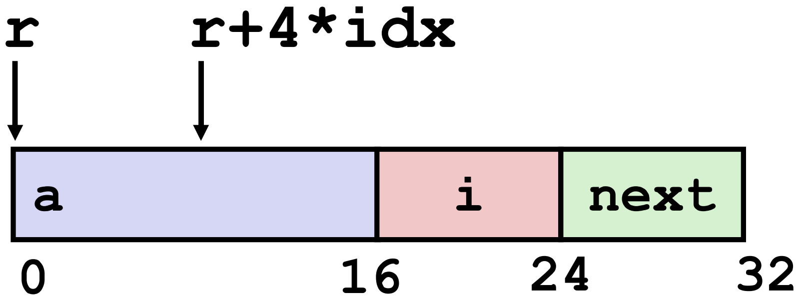

```
void set_i(struct rec *r, long val){
  r->i = val;
}
```

- **Access field with corresponding offset**

```
 # r in %rdi, val in %rsi
movq %rsi, 16(%rdi)            # Mem[r+16] = val
```

Address of structure member

```
struct rec {
  int a[4];
  long i;
  struct rec *next;
};
```




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

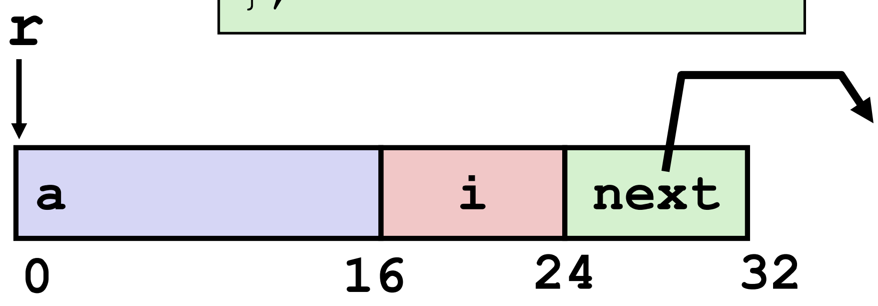
- **Add the field's offset to the structure's address**

```
  # r in %rdi, idx in %esi
  leaq  (%rdi,%rsi,4), %rax
```

Example: Following linked list

```
long length(struct rec*r){
     long len = 0;
     while (r) {
          len ++;
          r = r->next;
     }
     return len;
}
```

```
struct rec {
   int a[4];
   long i;
   struct rec *next;
};
```




```
# r in %rdi
.L11:                            # loop:
  addq     $1, %rax              #   len ++
  movq     24(%rdi), %rdi        #   r = Mem[r+24]
  testq    %rdi, %rdi            #   Test r
  jne      .L11                  #   If != 0, goto loop
```

Today

- Structures
  - Allocation
  - Access
- **Data alignment**
- Unions

Motivation

- **Many computer systems place restrictions on the allowable addresses for the primitive data types**
  - Require that the address for some type of object must be a multiple of some value K (typically 1, 2, 4, or 8)

- **The x86-64 hardware will work correctly regardless of the alignment of data**

- **However, Intel recommends that data be aligned**
  - Simplifies the design of the hardware forming the interface between the processor and the memory system
  - Improves memory system performance

Alignment principles

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

- **Align enforcement by compiler**
  - Places `.align` directives in the assembly code indicating the desired alignment for global data
  - Library routines that allocate memory, such as `malloc`, must be designed so that they return a pointer that satisfies the worst-case alignment restriction (typically 16 bytes on x86-84)
  - For code involving structures, **inserts gaps in structure** to ensure correct alignment of fields

- **Treated differently by IA32 Linux, x86-64 Linux, Windows and Mac OS!**

Specific cases of alignment (x86-64)

- **1 byte: `char`**
  - No restrictions on address
- **2 bytes: `short`**
  - Lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - Lowest 2 bits of address must be 00_2
- **8 bytes: `long`, `double`, `char *`, …**
  - Lowest 3 bits of address must be 000_2

Data alignment in structures

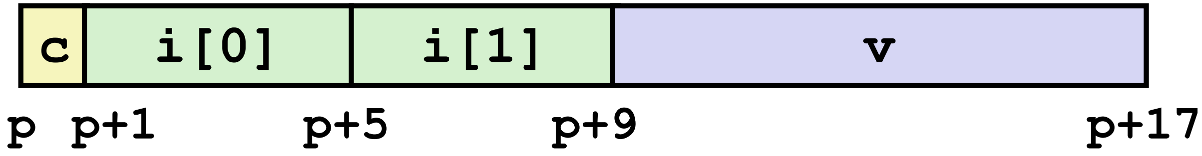
- **Inside the structure**
  - The **alignment** requirement of **each field** type must be satisfied

- **Placement of structure in memory**
  - Each structure has an alignment requirement of K, where K is the size of the largest member of the structure
  - The **starting address** of the structure must be a **multiple of K**
  - The **total size** of the structure must be a **multiple of K**

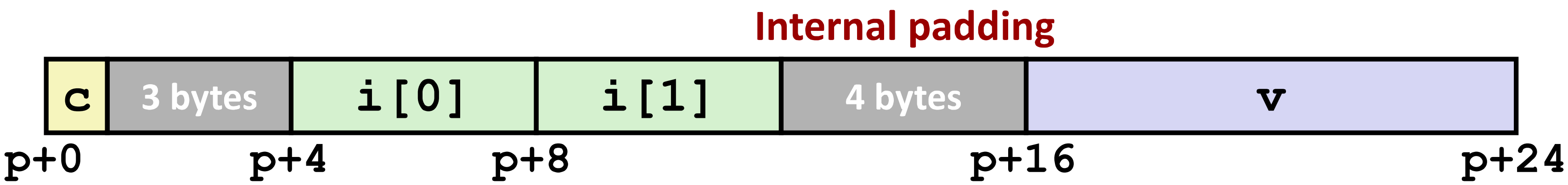
Alignment requirements of members

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

- **Unaligned data**



- **Aligned data under x86-64**

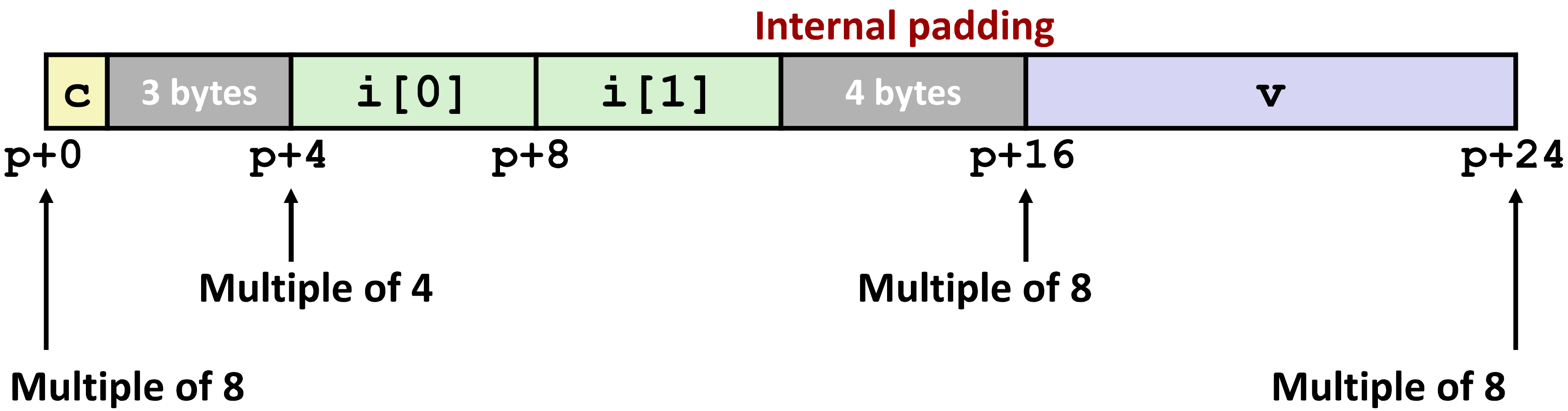


- **The compiler may need to add padding**
  - Between the fields (internal padding)
  - At the end of the structure (external padding)

Alignment requirements of members

- **For each member**
  - Satisfy alignment requirement
- **Overall structure placement**
  - Initial address and total size must be multiple of K
  - K = 8, largest alignment of any member, due to `long` element

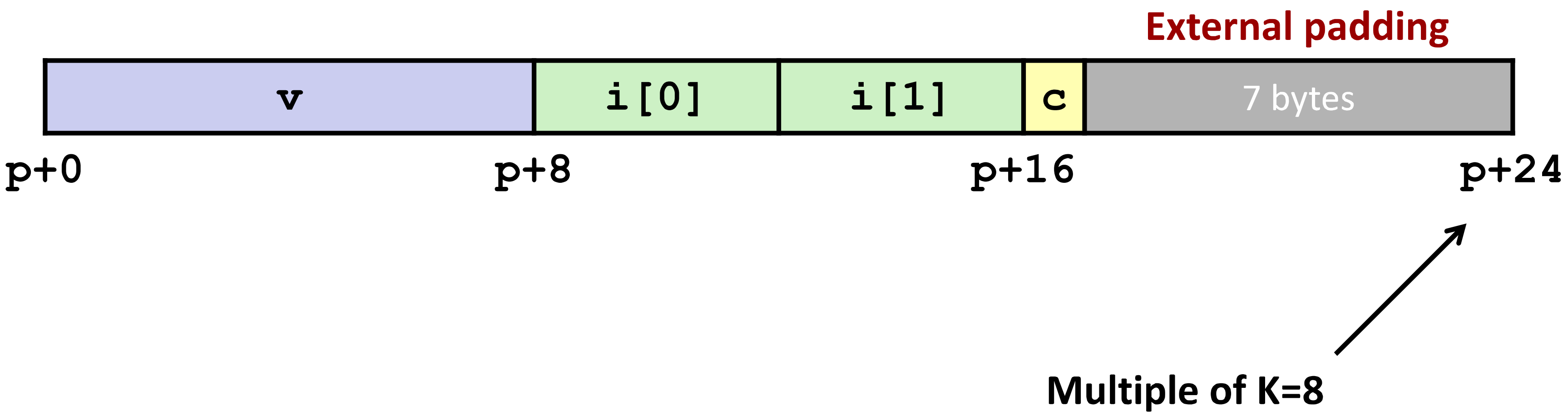
```
struct S1 {
  char c;
  int i[2];
  long v;
} *p;
```



Overall alignment requirement

- **Overall structure placement**
  - Initial address and total size must be multiple of K
  - K = 8, largest alignment of any member, due to `long` element

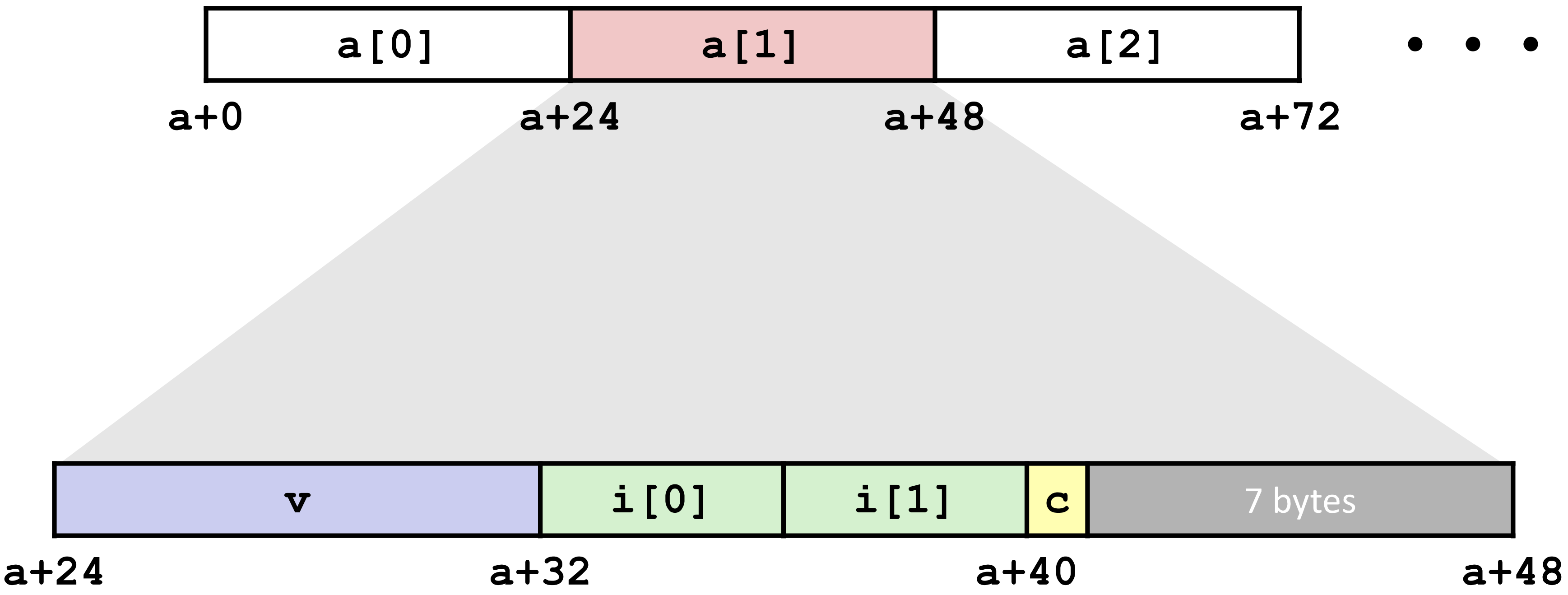
```
struct S2 {
  long v;
  int i[2];
  char c;
} *p;
```



Arrays of structures

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

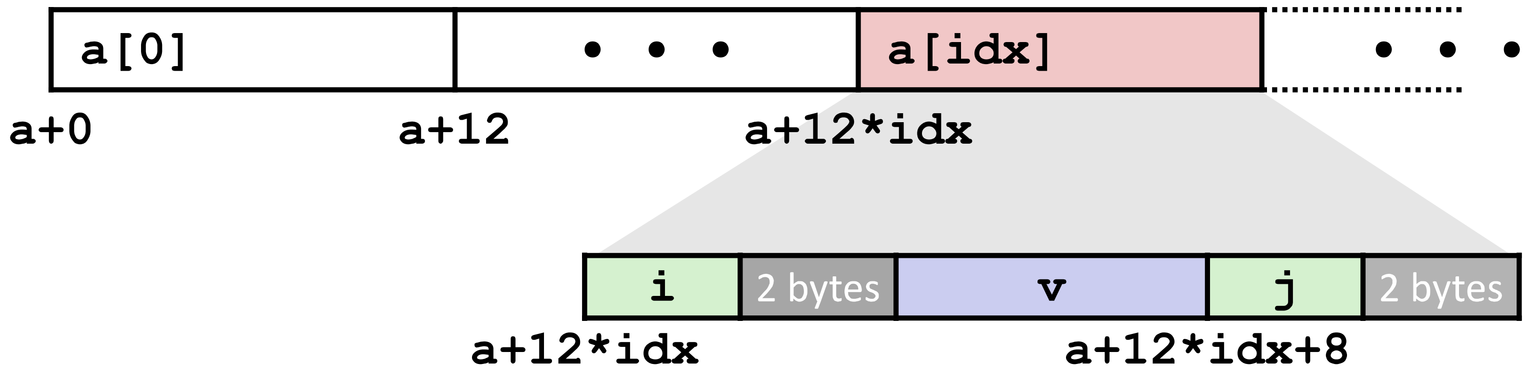
- **Contiguously allocated region of n * sizeof(struct x) bytes**
  - Overall structure length multiple of K
  - Satisfy alignment requirement for every element



Accessing array elements

```
struct S3 {
  short i;
  int v;
  short j;
} a[10];
```

- **Compute array offset 12 * idx**
  - `sizeof(S3) = 12`, including alignment padding
- **Element `j` is at offset 8 within structure**




```
short get_j(struct s3 *a,
       int idx){
  return a[idx].j;
}
```

```
# 3*idx
leaq (%rsi,%rsi,2),%rax
# a+12*idx+8
movw 8(%rdi,%rax,4),%ax
```

Example exam question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Example exam question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| a | X | X | X | X | X | X | X | b | b | b | b | b | b | b | b |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| c | c | c | c | d | d | d | X | e | e | e | e | e | e | e | e |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| f | f | f | f | f | f | f | f |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

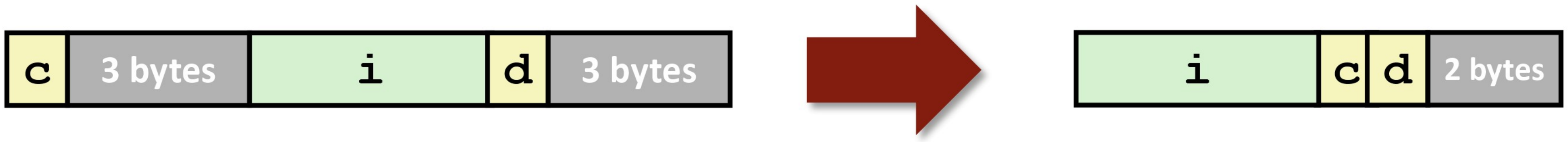
Saving space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect on x86-64 (K=4)



12 bytes

8 bytes

- This strategy can save space in certain structures

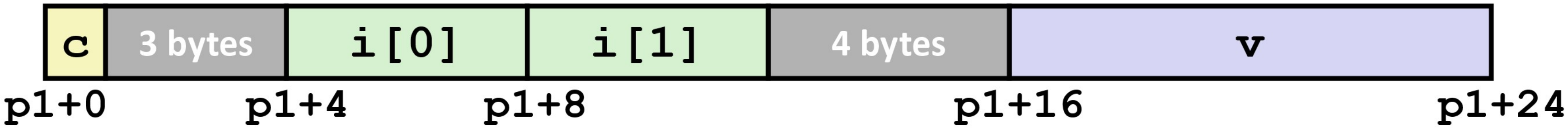
Saving space

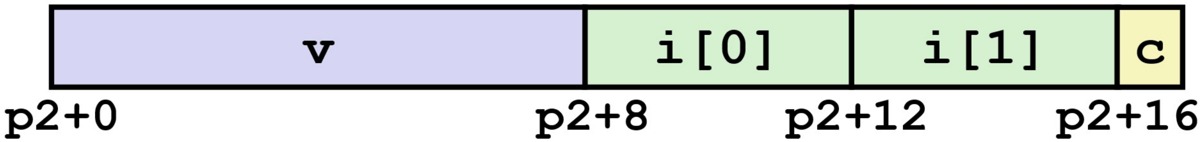
- Put large data types first

```
struct S6 {
  char c;
  int i[2];
  long v;
} *p1;
```

```
struct S7 {
  long v;
  int i[2];
  char c;
} *p2;
```

- Effect on x86-64 (K=8)





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

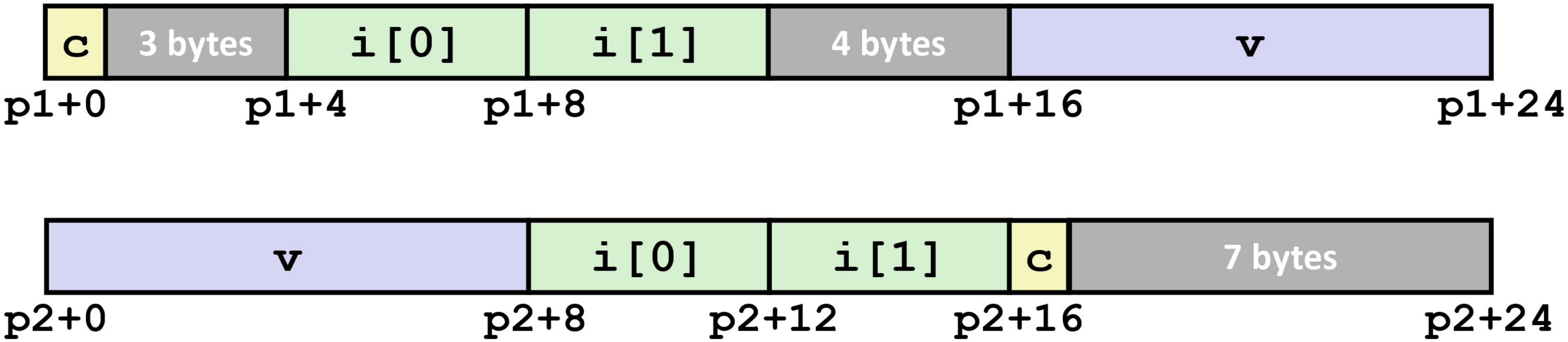
Saving space

- Put large data types first

```
struct S6 {
  char c;
  int i[2];
  long v;
} *p1;
```

```
struct S7 {
  long v;
  int i[2];
  char c;
} *p2;
```

- Effect on x86-64 (K=8)



Example exam question (Cont'd)

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Example exam question (Cont'd)

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```
typedef struct {
  char a;
  long b;
  float c;
  char d[3];
  int *e;
  short *f;
} foo;
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| a | d | d | d | c | c | c | c | b | b | b | b | b | b | b | b |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| e | e | e | e | e | e | e | e | f | f | f | f | f | f | f | f |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Today

- **Structures**
  - Allocation
  - Access
- **Data alignment**
- **Unions**

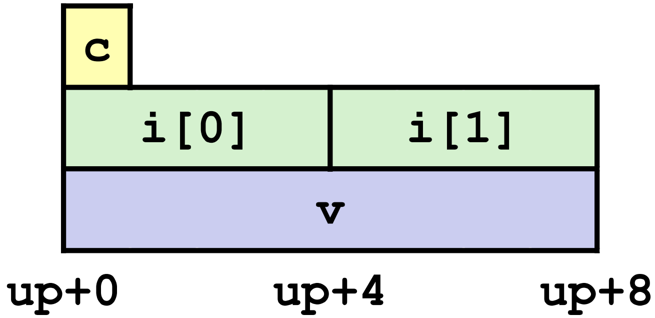
Unions

- **Provide a way to circumvent the type system of C**
  - Allow a single object to be referenced according to multiple types
- **The syntax of a union declaration is identical to that for structures, but its semantics are very different**
  - Rather than having the different fields reference different blocks of memory, they all reference the same block
- **The overall size of a union equals the maximum size of any of its fields**
  - Important when memory is scarcer, such as in embedded systems

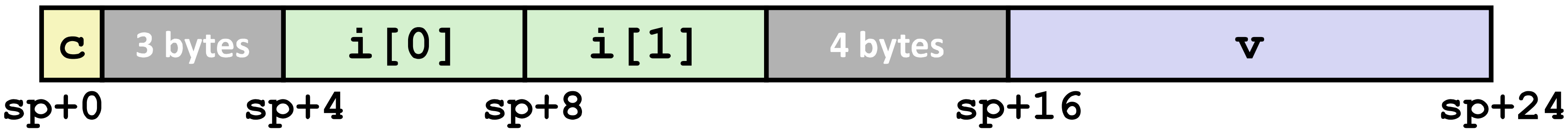
Union allocation

- **Allocated according to largest element**
- **Can only use one member at a time**

```
union U1 {
  char c;
  int i[2];
  long v;
} *up;
```




```
struct S1 {
  char c;
  int i[2];
  long v;
} *sp;
```



Accessing union members

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

data.i = 10;
printf("data.i   : %d\n", data.i);      /* 10 */

data.f = 220.5;
printf("data.f   : %f\n", data.f);      /* 220.5 */

strcpy(data.str,"C Programming");
printf("data.str : %s\n", data.str);    /* C Programming */
```

Accessing union members

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

data.i = 10;
data.f = 220.5;
strcpy(data.str,"C Programming");

printf("data.i   : %d\n", data.i);       /* 1917853763 */

printf("data.f   : %f\n", data.f);       /* 4122360580327.00 */

printf("data.str : %s\n", data.str);     /* C Programming */
```

Accessing union members

- Unions can be useful in several contexts, but they can also lead to nasty bugs
- No checking is done to make sure that the right sort of use is made of the members
- It's up to the programmer to keep track of whatever type is put into it

Common use

- **Combining them with *structs***
  - Produce variable types based on a common core, but then varying in parts where appropriate

```
typedef union {
   int units;
   float kgs;
} amount;

typedef struct {
    char selling[15];
    float unitprice;
    int unittype;
    amount howmuch;
  } product;
```

- **Still require a lot of discipline to ensure that only the active member is accessed at any time**

Summary

- **Structures**
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment

- **Combinations**
  - Can nest structure and array code arbitrarily

- **Unions**
  - Overlay declarations
  - Way to circumvent type system