System Programming

08. Machine-Level Programming IV: Data (ch 3.8)

2019. Fall

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

These slides are based on your text book on the slides prepared by Dan Grossman, Ruth Anderson.

data science laboratory

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

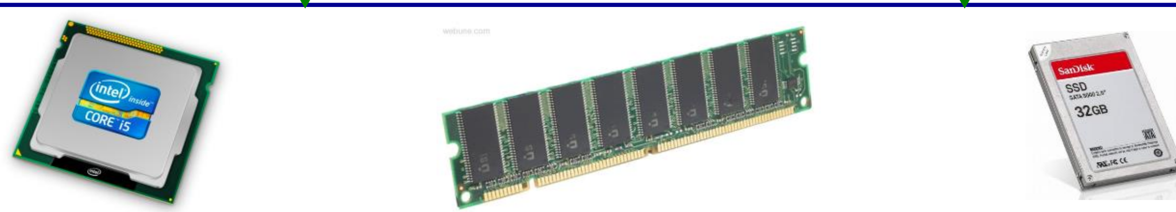
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

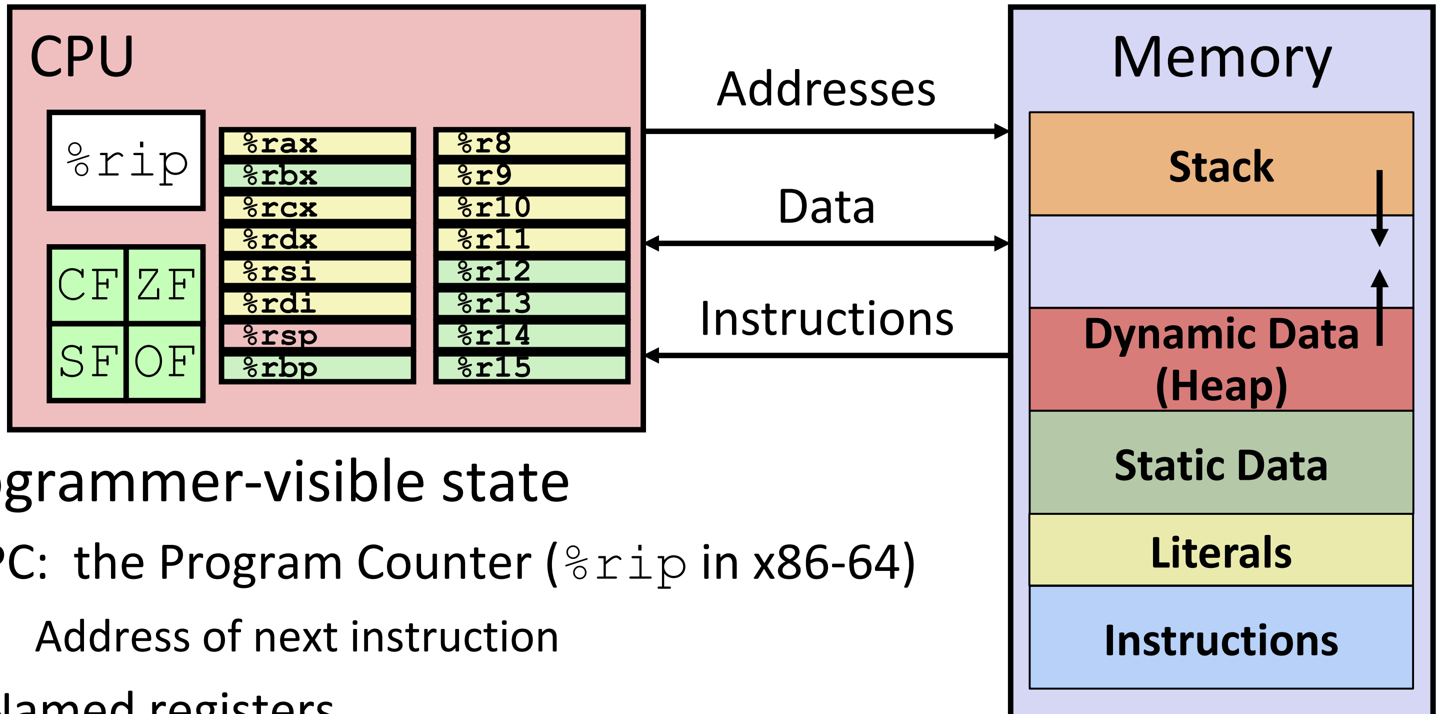
OS:

Computer system:



- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & **structs**
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Assembly Programmer's View

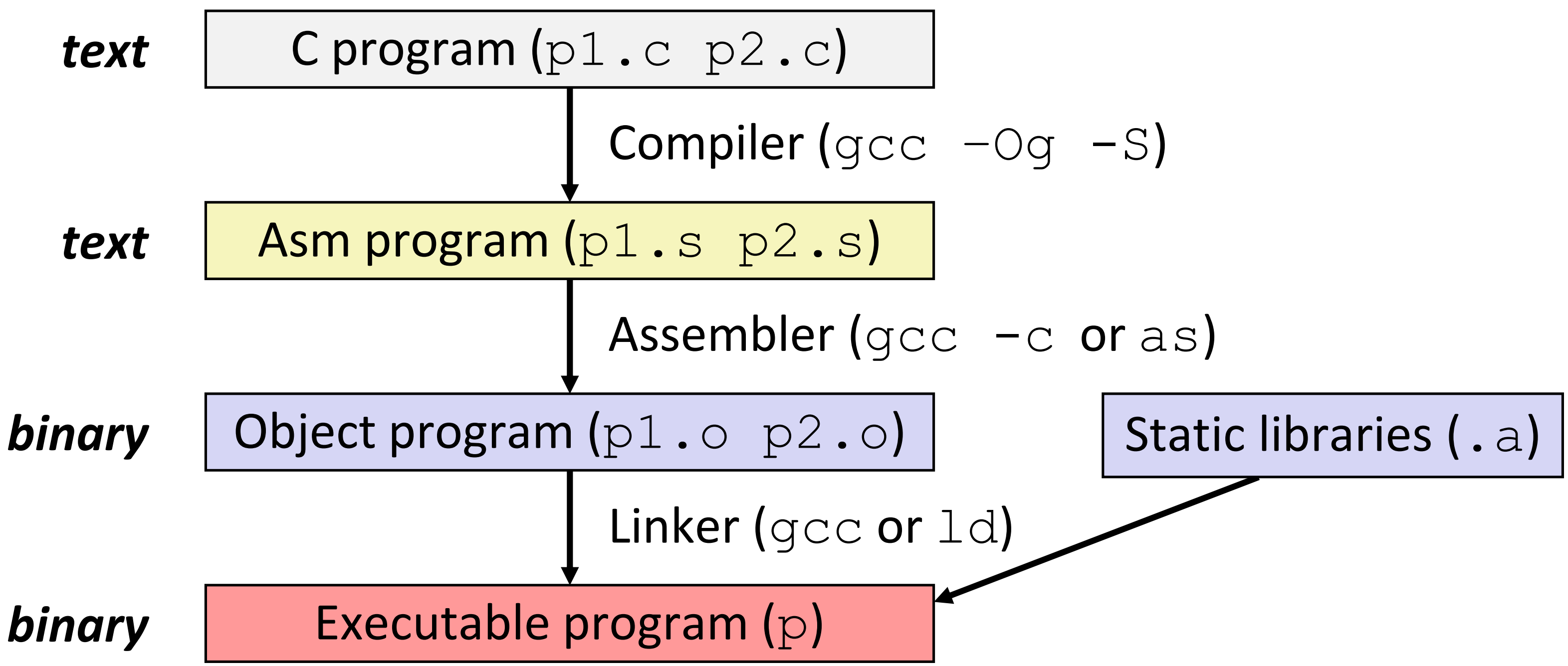


- Programmer-visible state
  - PC: the Program Counter (`%rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- Memory
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting machine code in file `p`



Assembling

- Executable has **addresses**

assembler

```
00000000004004f6 <pcount_r>:
  4004f6:   b8 00 00 00 00     mov    $0x0,%eax
  4004fb:   48 85 ff           test   %rdi,%rdi
  4004fe:   74 13              je     400513 <pcount_r+0x1d>
  400500:   53                 push   %rbx
  400501:   48 89 fb           mov    %rdi,%rbx
  400504:   48 d1 ef           shr    %rdi
  400507:   e8 ea ff ff ff     callq  4004f6 <pcount_r>
  40050c:   83 e3 01           and    $0x1,%ebx
  40050f:   48 01 d8           add    %rbx,%rax
  400512:   5b                 pop    %rbx
  400513:   f3 c3              rep ret
```

- `gcc -g pcount.c -o pcount`
- `objdump -d pcount`

Picture of Memory (64-bit view)

```
00000000004004f6 <pcount_r>:
  4004f6:  b8 00 00 00 00    mov    $0x0,%eax
  4004fb:  48 85 ff          test   %rdi,%rdi
  4004fe:  74 13             je     400513 <pcount_r+0x1d>
  400500:  53                push   %rbx
  400501:  48 89 fb          mov    %rdi,%rbx
  400504:  48 d1 ef          shr    %rdi
  400507:  e8 ea ff ff ff    callq  4004f6 <pcount_r>
  40050c:  83 e3 01          and    $0x1,%ebx
  40050f:  48 01 d8          add    %rbx,%rax
  400512:  5b                pop    %rbx
  400513:  f3 c3             rep ret
```

0|8	1|9	2|a	3|b	4|c	5|d	6|e	7|f	
								0x00
								0x08
								0x10
			...					...
						b8	00	0x4004f0
00	00	00	48	85	ff	74	13	0x4004f8
53	48	89	fb	48	d1	ef	e8	0x400500
ea	ff	ff	ff	83	e3	01	48	0x400508
01	d8	5b	f3	c3				0x400510

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Operating Systems

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

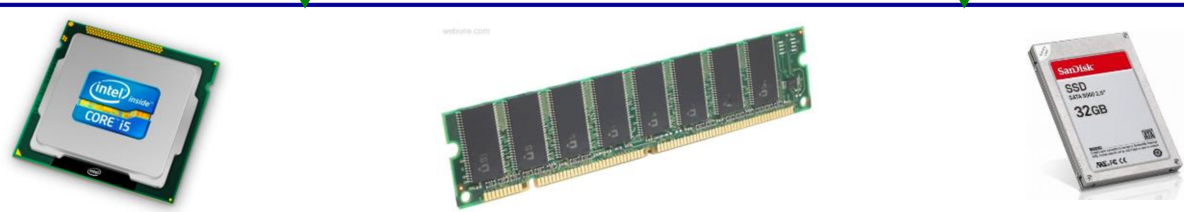
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:



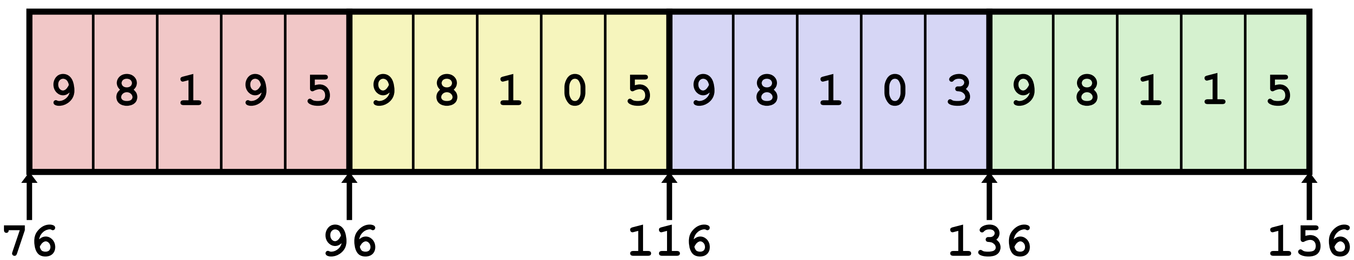
Data Structures in Assembly

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment
- Unions

Question

- Which of the following statements is <u>FALSE</u>?

`int sea[4][5];`



A. `sea[4][-2]` is a *valid* array reference

B. `sea[1][1]` makes *two* memory accesses

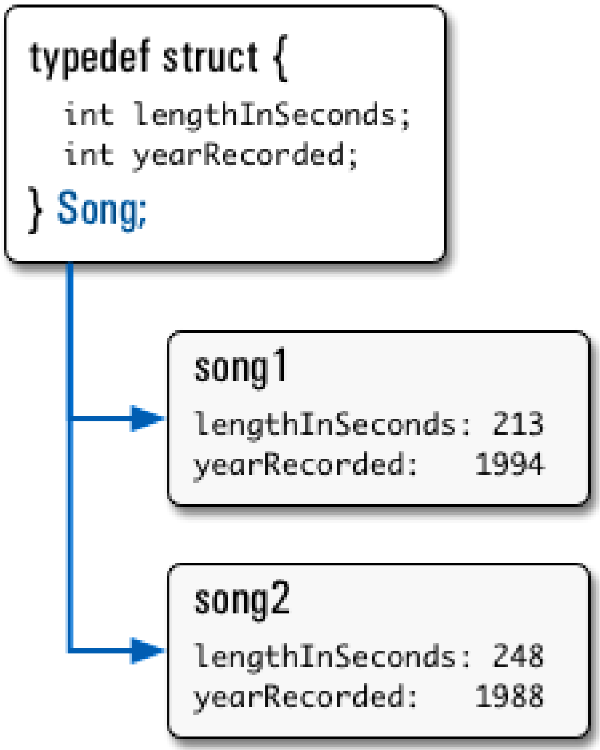
C. `sea[2][1]` will *always* be a higher address than `sea[1][2]`

D. `sea[2]` is calculated using *only* `lea`

Structs in C

- Way of defining compound data types
- A structured group of variables, possibly including other structs

```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

song1.lengthInSeconds =  213;
song1.yearRecorded    = 1994;

Song song2;

song2.lengthInSeconds =  248;
song2.yearRecorded    = 1988;
```



Review: Structs

```c
// Use typedef to create a type: FourInts
typedef struct {
   int a, b, c, d;
} FourInts;   // Name of type is "FourInts"

int main(int argc, char* argv[]) {
   FourInts f1;   // Allocates memory to hold a FourInts
                  // (16 bytes) on stack (local variable)
   f1.a = 0;      // Assign first field in f1 to be zero

   FourInts* f2; // Declare f2 as a pointer to FourInts

   // Allocate space for a FourInts on the heap,
   //   f2 is a "pointer to"/"address of" this space.
   f2 = (FourInts*) malloc(sizeof(FourInts));
   f2->b = 17;    // Assign the second field to be 17
   ...
}
```

Aside: Syntax for structs <u>without typedef</u>

```
struct rec {            // Declares the type "struct rec"
    int a[4];           // Total size = _______ bytes
    long i;
    struct rec *next;
};
struct rec r1;          // Allocates memory to hold a struct rec
                        // named r1, on stack or globally,
                        // depending on where this code appears

struct rec *r;          // Allocates memory for a pointer
r = &r1;                // Initializes r to "point to" r1
```

More Structs Syntax

Declaring a struct `struct rec`, then declaring a variable `r1`:

```
struct rec {          // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
};
struct rec r1;        // Declares r1 as a struct rec
```

Equivalent to:

```
struct rec {          // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
} r1;                 // Declares r1 as a struct rec
```

Declare type `struct rec` and variable `r1` at the same time!

Another Syntax Example

Declaring a struct `struct rec`, then declaring a variable `r`:

```
struct rec {            // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
};
struct rec *r;          // Declares r as pointer to a struct rec
```

Equivalent to:

```
struct rec {            // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
} *r;                   // Declares r as pointer to a struct rec
```

Declare type `struct rec` and variable `r` at the same time!

Struct Definitions

```
struct name {
    /* fields */
};
```

Easy to forget semicolon!

- Structure definition:
    - Does NOT declare a variable
- Variable definitions:
    - Variable type is "`struct name`"

pointer

```
struct name name1, *pn, name_ar[3];
```

array

- Joint struct definition and typedef

```
struct nm {
    /* fields */
};
typedef struct nm name;
name n1;
```

```
typedef struct {
    /* fields */
} name;
name n1;
```

Scope of Struct Definition

- Why is placement of struct definition important?
  - What actually happens when you declare a variable?
    - Creating space for it somewhere!
  - Without definition, program doesn't know how much space

4B×4
8B

```
struct data {
    int ar[4];
    long d;
};
```

Size = 24 bytes

```
struct rec {
    int a[4];
    long i;
8B  struct rec* next;
};
```

Size = 32 bytes

- Almost always define structs in global scope near the top of your C file
  - Struct definitions follow normal rules of scope

Accessing Structure Members

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

- Given a struct instance, access member using the . operator:

  ```
  struct rec r1;
  r1.i = val;
  ```

- Given a *pointer* to a struct:

  ```
  struct rec *r;
  r = &r1;  // or malloc space for r to point to
  ```

  We have two options:
  - Use `*` and `.` operators: `(*r).i = val;`
  - Use `->` operator for short: `r->i = val;`

- **In assembly:** register holds address of the first byte
  - Access members with offsets

Java side-note

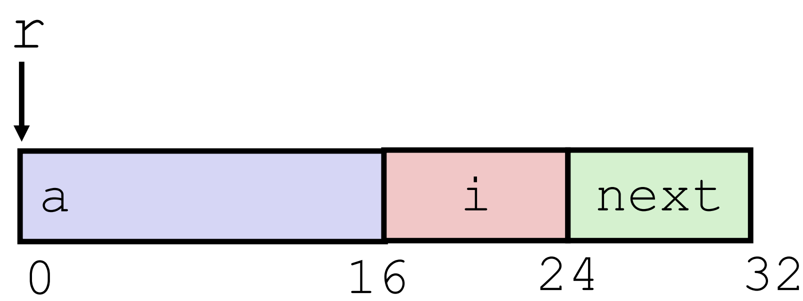
Java:

```
class Record { ... }
Record x = new Record();
```

- An instance of a class is like a *pointer to* a struct containing the fields
  - (Ignoring methods and subclassing for now)
  - So Java's `x.f` is like C's `x->f` or `(*x).f`
- In Java, almost everything is a pointer ("*reference*") to an object
  - Cannot declare variables or fields that <u>are</u> structs or arrays
  - Always a *pointer* to a struct or array
  - So every Java variable or field is ≤ 8 bytes (but can point to lots of data)

Structure Representation

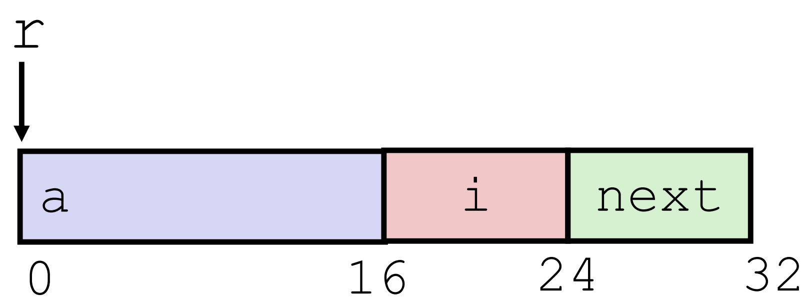
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



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

Structure Representation

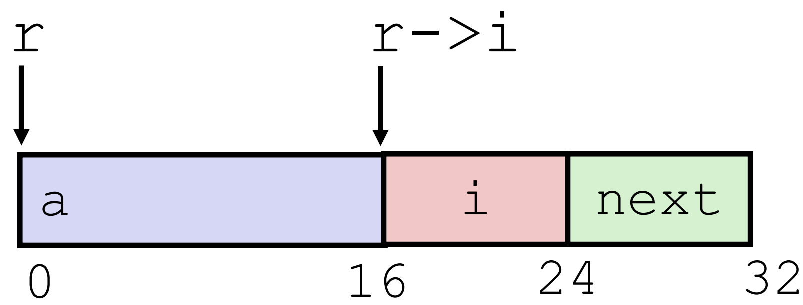
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Accessing a Structure Member

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



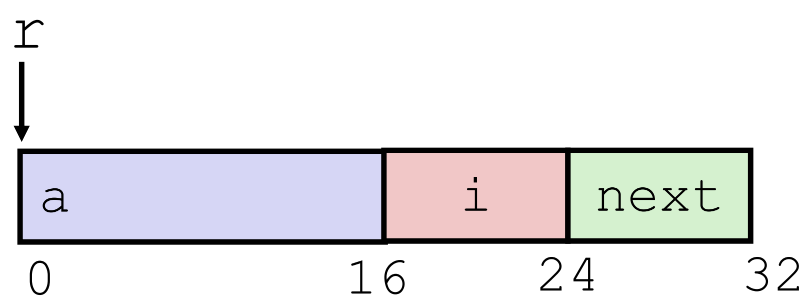

```
long get_i(struct rec *r)
{
  return r->i;
}
```

```
  # r in %rdi
  movq  16(%rdi), %rax
  ret
```

- Compiler knows the *offset* of each member within a struct.
  - Compute as `*(r+offset)`
  - Referring to absolute offset, so no pointer arithmetic

Exercise: Pointer to Structure Member

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




```
long* addr_of_i(struct rec *r)
{
  return &(r->i);
}
```

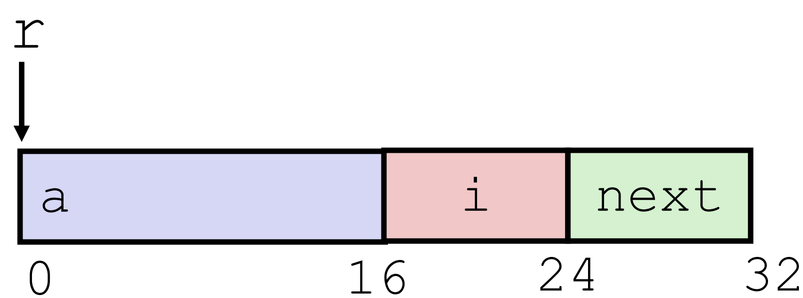
```
  # r in %rdi
  _______  _______,%rax
  ret
```

```
struct rec** addr_of_next(struct rec *r)
{
  return &(r->next);
}
```

```
  # r in %rdi
  _______  _______,%rax
  ret
```

Exercise: Pointer to Structure Member

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




```
long* addr_of_i(struct rec *r)
{
  return &(r->i);
}
```

```
# r in %rdi
leaq   16(%rdi), %rax
ret
```

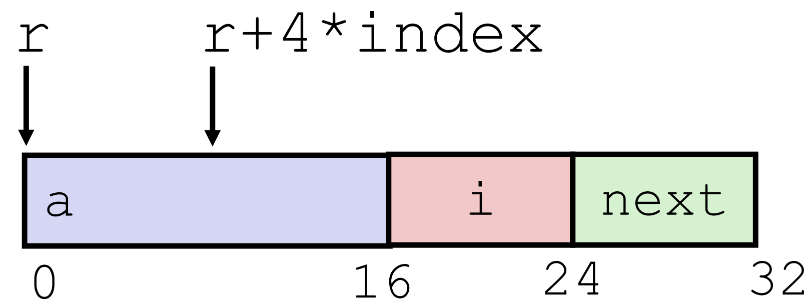
want address

```
struct rec** addr_of_next(struct rec *r)
{
  return &(r->next);
}
```

```
# r in %rdi
leaq   24(%rdi), %rax
ret
```

Generating Pointer to Array Element

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as: `r + 4*index`

```
int* find_addr_of_array_elem
   (struct rec *r, long index)
{
  return &r->a[index];
}
```

`&(r->a[index])`

```
  # r in %rdi, index in %rsi
  leaq   (%rdi,%rsi,4), %rax
  ret
```

Review: Memory Alignment in x86-64

- For good memory system performance, Intel recommends data be aligned
  - However the x86-64 hardware will work correctly regardless of alignment of data.
- *Aligned* means:
  - Any primitive object of K bytes must have an address that is a multiple of K.
- This means we could expect these types to have starting addresses that are the following multiples:

K	Type	Addresses
1	`char`	No restrictions
2	`short`	Lowest bit must be zero: $\ldots 0_2$
4	`int, float`	Lowest 2 bits zero: $\ldots 00_2$
8	`long, double, * (pointers)`	Lowest 3 bits zero: $\ldots 000_2$
16	`long double`	Lowest 4 bits zero: $\ldots 0000_2$

Alignment Principles

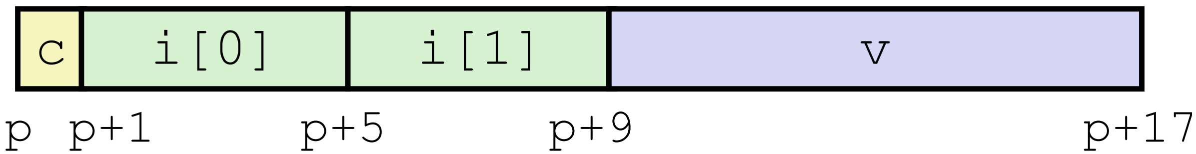
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64

- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store value that spans quad word boundaries
    - Virtual memory trickier when value spans 2 pages (more on this later)

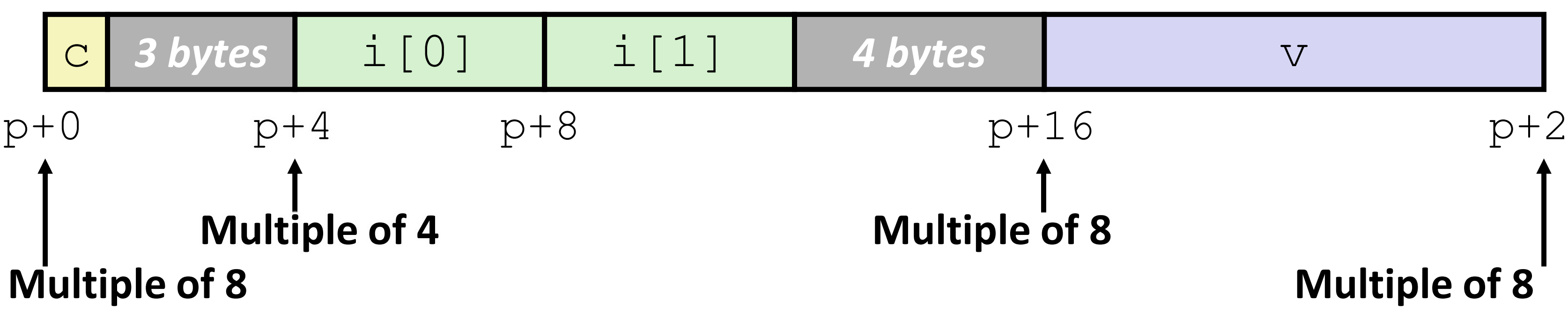
Structures & Alignment

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

- Unaligned Data



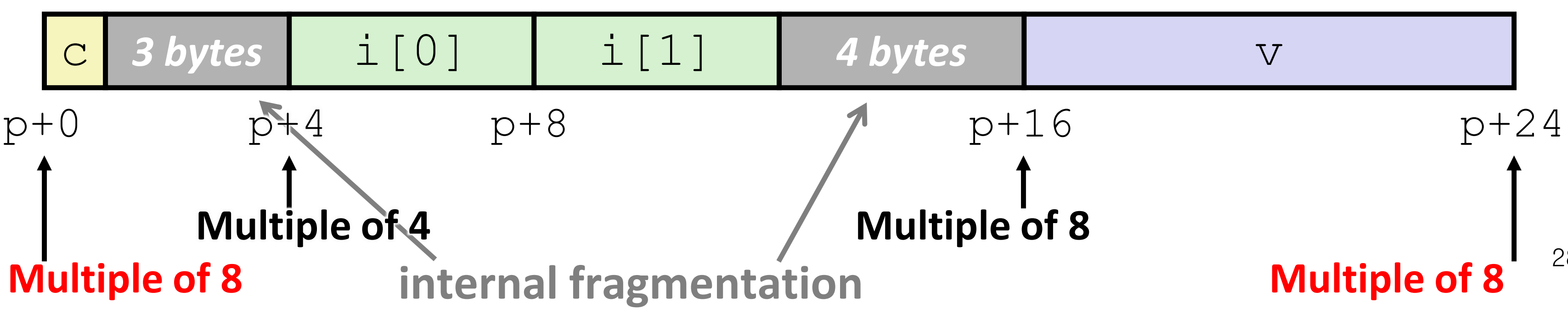
- Aligned Data
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



Satisfying Alignment with Structures (1)

- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement $K_{\max}$
    - $K_{\max}$ = Largest alignment of any element
    - Counts individual items in the array as elements (entire array is not an "element")
  - **Initial address of structure & structure length must be multiples of K**
- Example:
  - $K_{\max}$ = 8, due to `double` element

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

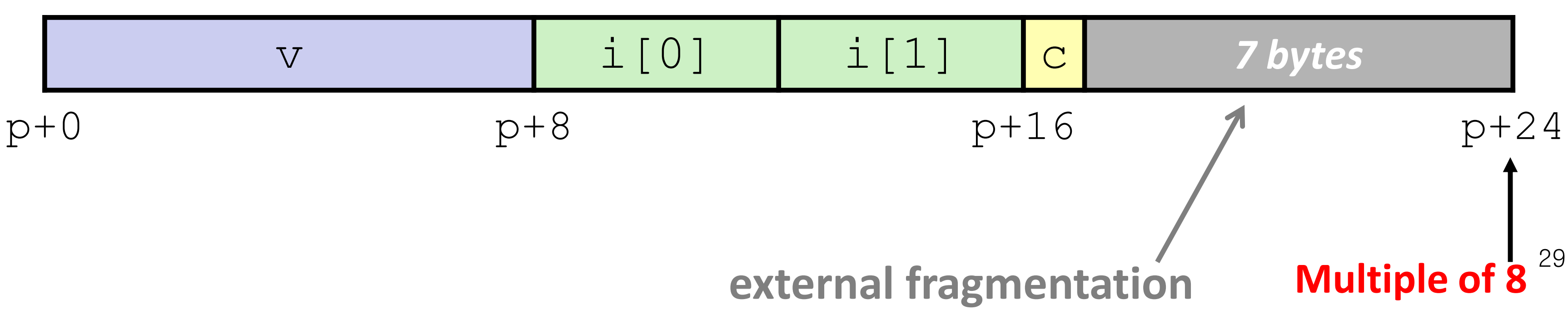


Satisfying Alignment Requirements:
Another Example

- Can find offset of individual fields using offsetof()
  - Need to #include <stddef.h>
  - <u>Example</u>: offsetof(struct S2,c) returns 16

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

- For largest alignment requirement $K_{\max}$, overall structure size must be multiple of $K_{\max}$
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



Alignment of Structs

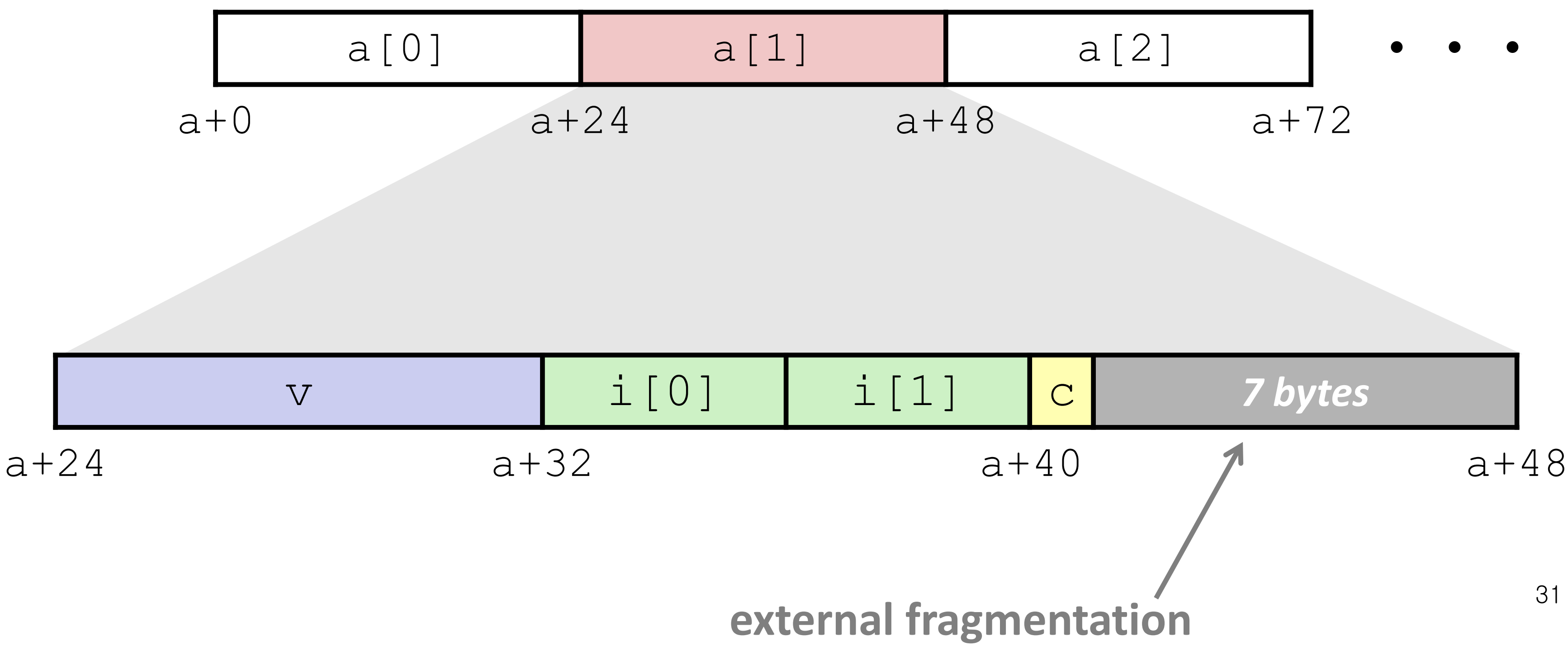
- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total struct ***size*** must be multiple of its alignment *(may insert padding)*
    - `sizeof` should be used to get true size of structs

Arrays of Structures

Create an array of ten S2 structs called "a"

- Overall structure length multiple of K_{max}
- Satisfy alignment requirement for every element in array

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

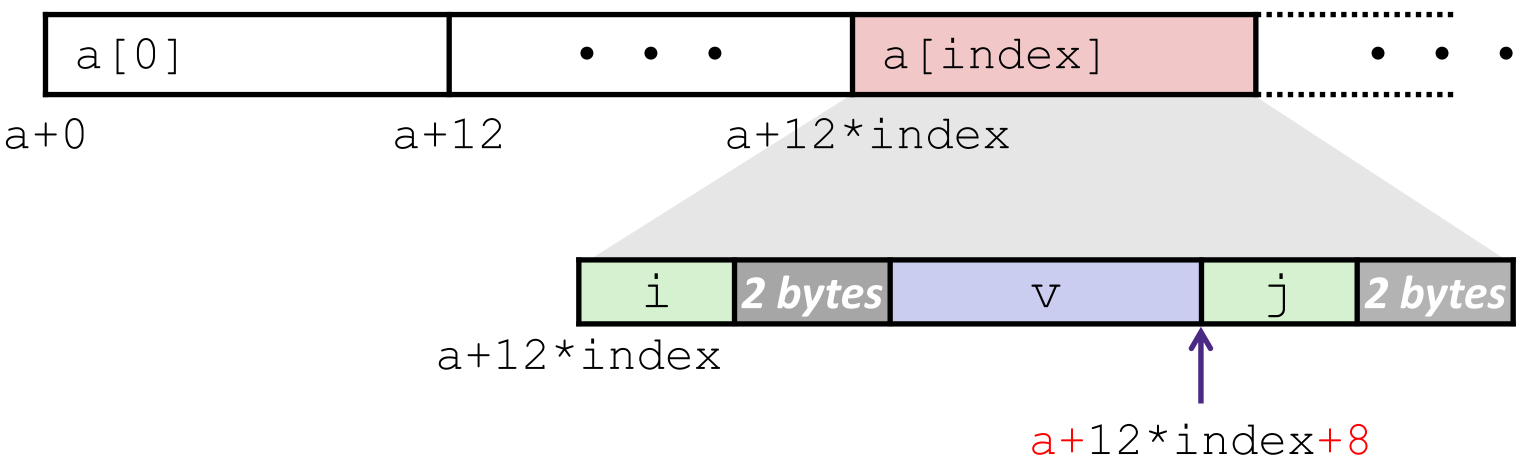


Accessing Array Elements

Create an array of ten S3 structs called “a”

- Compute start of array element as: `12*index`
  - `sizeof(S3) = 12`, including alignment padding
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8`

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




```
short get_j(int index)
{
  return a[index].j;
}
```

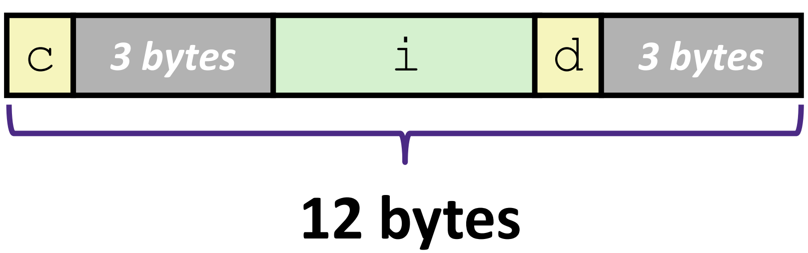
```
# %rdi = index
leaq (%rdi,%rdi,2),%rax   # 3*index
movzwl a+8(,%rax,4),%eax
```

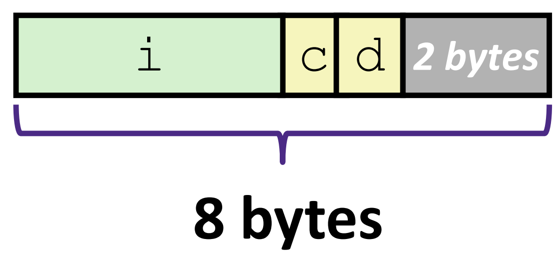
How the Programmer Can Save Space

- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

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

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





Question

- Minimize the size of the struct by re-ordering the vars

```
struct old {
  int i;

  short s[3];

  char *c;

  float f;
};
```

```
struct new {
  int     i;

  ______ ______;

  ______ ______;

  ______ ______;
};
```

- What are the old and new sizes of the struct?

  sizeof(struct old) = _____   sizeof(struct new) = _____

Answers

- Minimize the size of the struct by re-ordering the vars

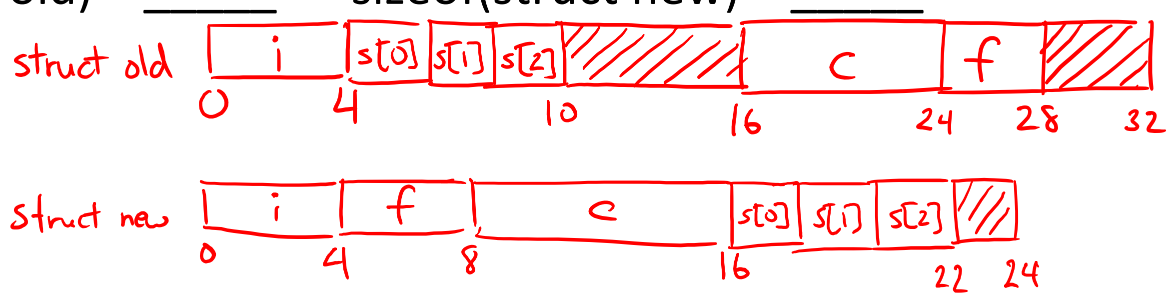
K
4
2
8
4
$K_{max} = 8$

```
struct old {
  int i;

  short s[3];

  char *c;

  float f;
};
```

```
struct new {
  int      i;

  float    f;

  char *   c;

  short    s[3];
};
```

could also switch these (internal vs. external frag)

- What are the old and new sizes of the struct?
  - sizeof(struct old) = 32 B sizeof(struct new) = 24 B



Data Structures in Assembly

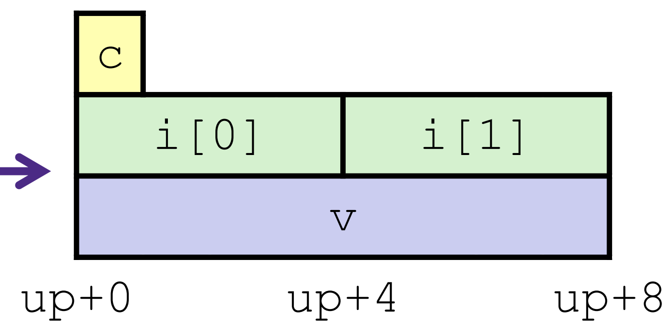
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment
- **Unions**

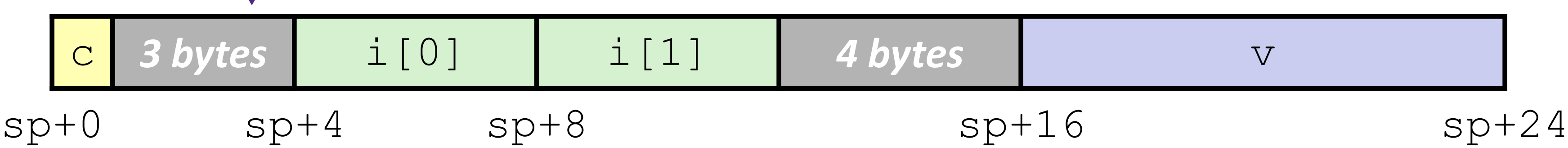
Unions

- Only allocates enough space for the largest element in union
- Can only use one member at a time

```
struct S {
  char c;
  int i[2];
  double v;
} *sp;
```

```
union U {
  char c;
  int i[2];
  double v;
} *up;
```





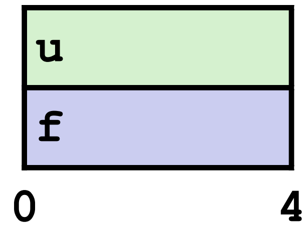
What Are Unions Good For?

- Unions allow the same region of memory to be referenced as different types
  - Different "views" of the same memory location
  - Can be used to circumvent C's type system (bad idea and technically not guaranteed to work)
- Better idea: use a struct inside a union to access some memory location either as a whole or by its parts
- But watch out for endianness at a small scale…
- Layout details are implementation/machine-specific

```
union int_or_bytes {
    int i;
    struct bytes {
      char b0, b1, b2, b3;
    }
}
```

Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(float) u` ?

Same as `(unsigned) f` ?

Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which byte is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- Big Endian
  - Most significant byte has lowest address
  - Sparc
- Little Endian
  - Least significant byte has lowest address
  - Intel x86, ARM Android and IOS
- Bi Endian
  - Can be configured either way
  - ARM

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

32-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

64-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

LSB MSB LSB MSB

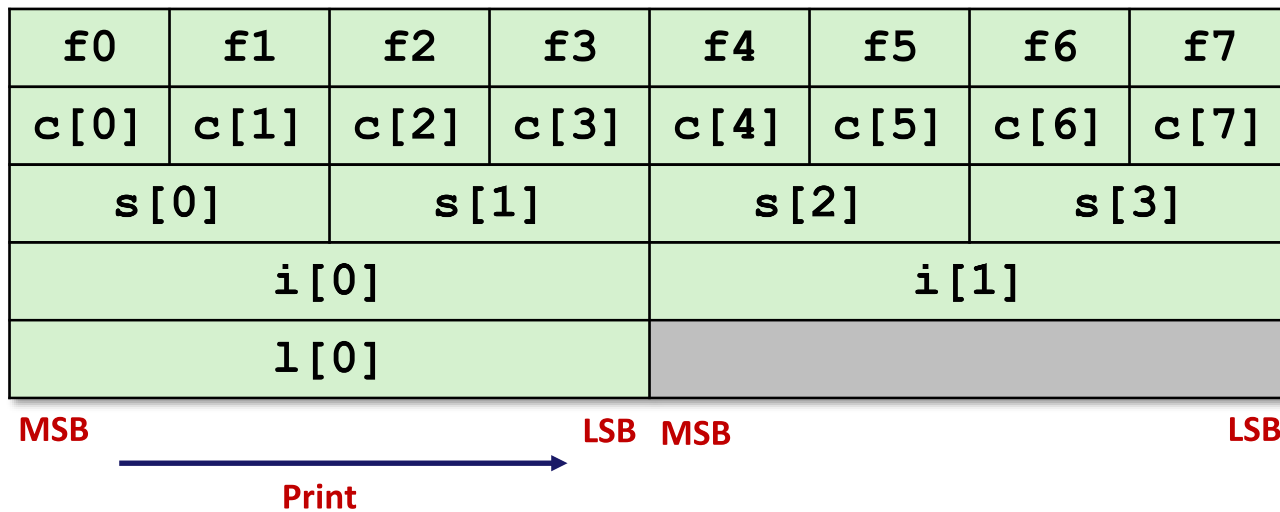
Print

Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian



Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

LSB ← Print — MSB

Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Unions For Embedded Programming

```
typedef union
{
   unsigned char byte;
   struct {
      unsigned char reserved:4;
      unsigned char b3:1;
      unsigned char b2:1;
      unsigned char b1:1;
      unsigned char b0:1;
   } bits;
} hw_register;

hw_register reg;
reg.byte = 0x3F;          // 00111111₂
reg.bits.b2 = 0;          // 00111011₂
reg.bits.b3 = 0;          // 00110011₂
unsigned short a = reg.byte;
printf("0x%X\n", a);      // output: 0x33
```

(Note: the placement of these fields and other parts of this example are implementation-dependent)

Summary

- Arrays in C
  - Aligned to satisfy every element's alignment requirement
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Provide different views of the same memory location

Q&A