

15-213

“The course that gives CMU its Zip!”

Structured Data II Heterogenous Data February 13, 2001

Topics

- Structure Allocation
- Alignment
- Unions
- Byte Ordering
- Byte Operations
- IA32/Linux Memory Organization
- Understanding C declarations

Basic Data Types

Integral

- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int, long, char *
quad word		8	[unsigned] long long (in gcc)

Floating Point

- Stored & operated on in floating point registers

Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12	long double

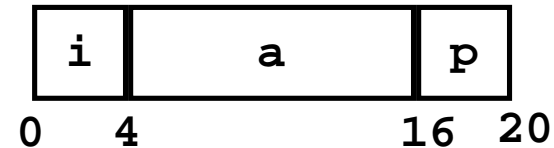
Structures

Concept

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

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

Memory Layout



Accessing Structure Member

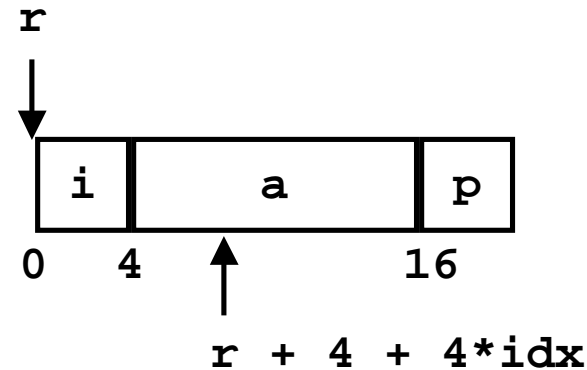
```
void  
set_i(struct rec *r,  
      int val)  
{  
    r->i = val;  
}
```

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];  
}
```

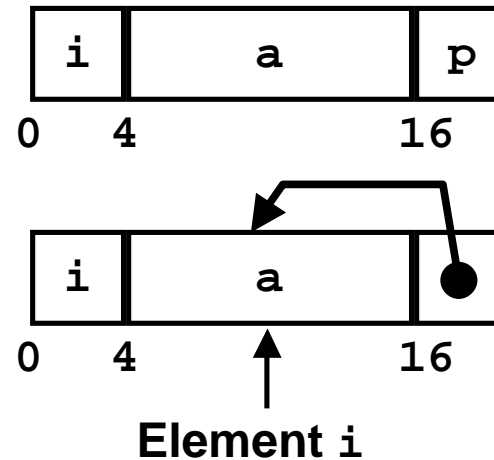
```
# %ecx = idx  
# %edx = r  
leal 0(,%ecx,4),%eax # 4*idx  
leal 4(%eax,%edx),%eax # r+4*idx+4
```

Structure Referencing (Cont.)

C Code

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

```
void  
set_p(struct rec *r)  
{  
    r->p =  
        &r->a[r->i];  
}
```



```
# %edx = r  
movl (%edx),%ecx      # r->i  
leal 0(,%ecx,4),%eax  # 4*(r->i)  
leal 4(%edx,%eax),%eax # r+4+4*(r->i)  
movl %eax,16(%edx)    # Update r->p
```

Alignment

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
 - treated differently by Linux and Windows!

Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words
 - Inefficient to load or store datum that spans quad word boundaries
 - Virtual memory very tricky when datum spans 2 pages

Compiler

- Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment

Size of Primitive Data Type:

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

Satisfying Alignment with Structures

Offsets Within Structure

- Must satisfy element's alignment requirement

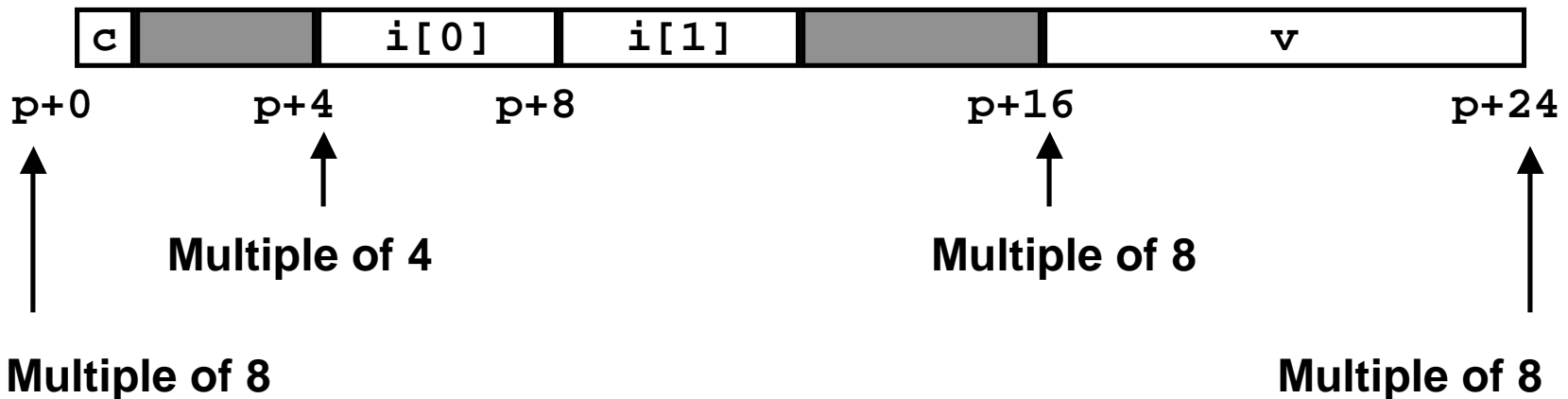
Overall Structure Placement

- Each structure has alignment requirement K
 - Largest alignment of any element
- Initial address & structure length must be multiples of K

```
struct s1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

Example (under Windows):

- $K = 8$, due to double element

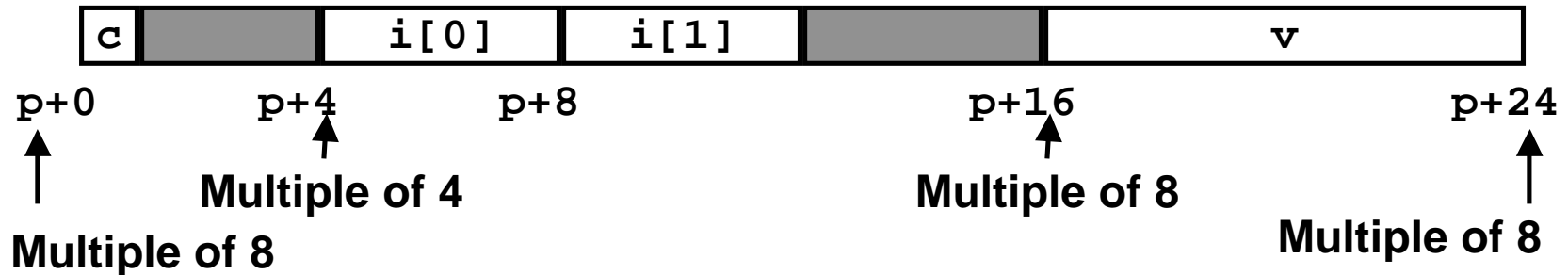


Linux vs. Windows

Windows (including Cygwin):

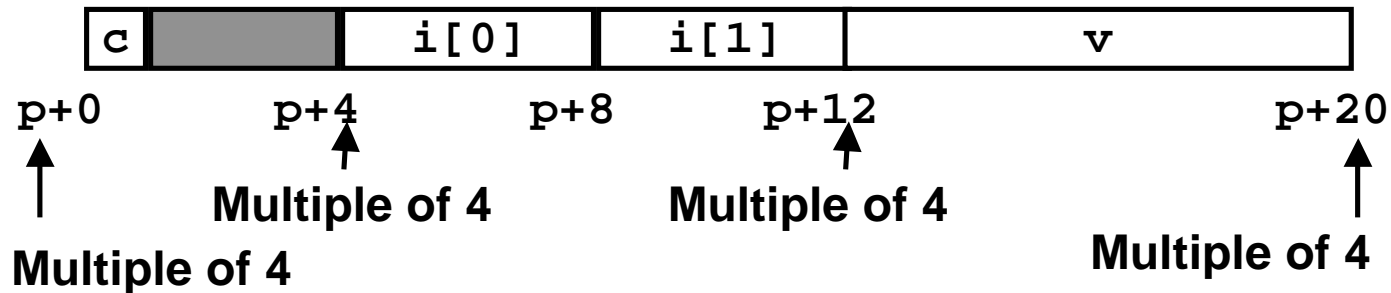
- `K = 8`, due to `double` element

```
struct s1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```



Linux:

- `K = 4`; `double` treated like a 4-byte data type



Effect of Overall Alignment Requirement

```
struct s2 {  
    double x;  
    int i[2];  
    char c;  
} *p;
```

p must be multiple of:
8 for Windows
4 for Linux



p+0

p+8

p+12

p+16

Windows: p+24

Linux: p+20

```
struct s3 {  
    float x[2];  
    int i[2];  
    char c;  
} *p;
```

p must be multiple of 4 (in either OS)



p+0

p+4

p+8

p+12

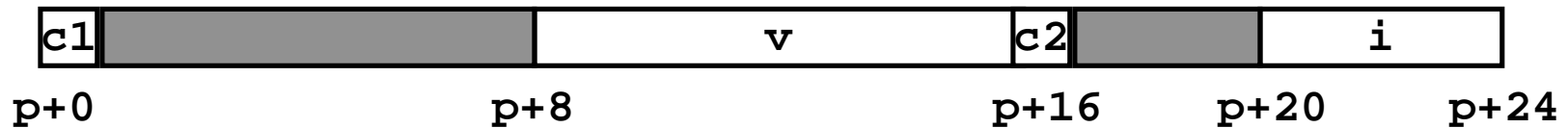
p+16

p+20

Ordering Elements Within Structure

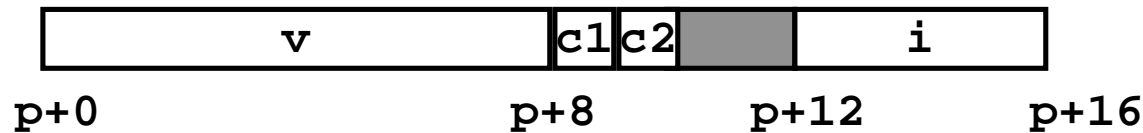
```
struct s4 {  
    char c1;  
    double v;  
    char c2;  
    int i;  
} *p;
```

10 bytes wasted space in Windows



```
struct s5 {  
    double v;  
    char c1;  
    char c2;  
    int i;  
} *p;
```

2 bytes wasted space

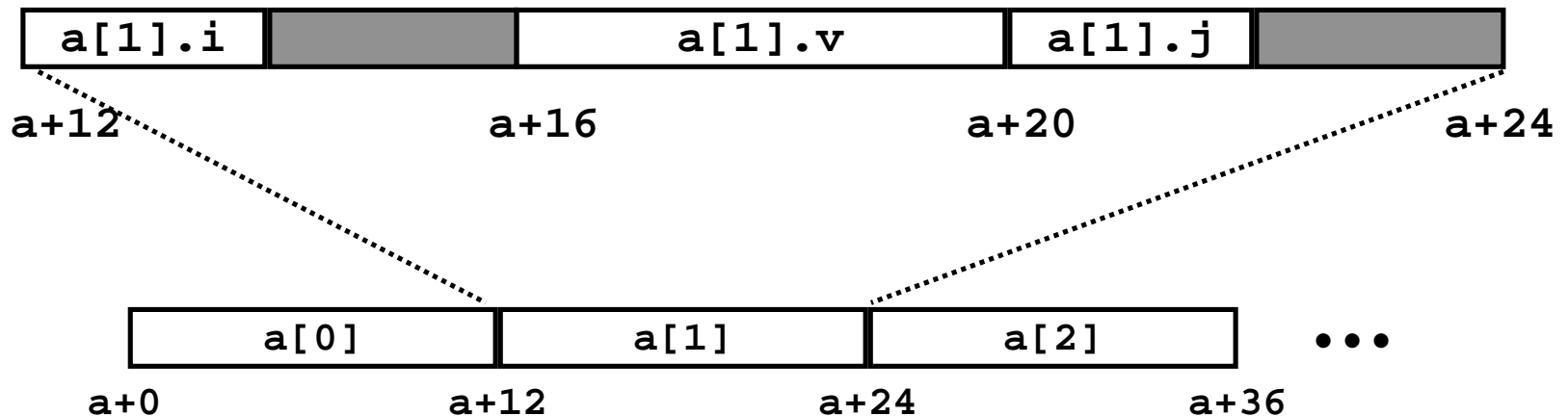


Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct s6 {  
    short i;  
    float v;  
    short j;  
} a[10];
```



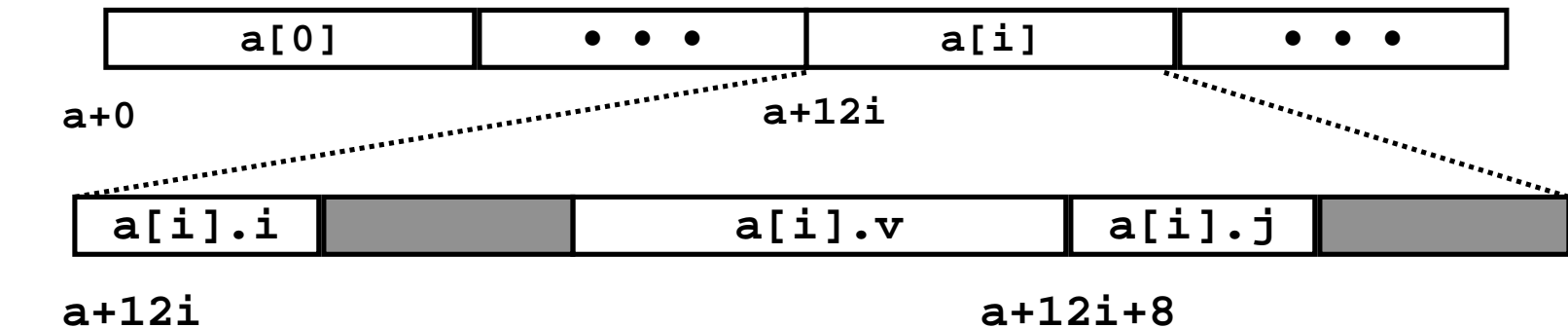
Accessing Element within Array

- Compute offset to start of structure
 - Compute $12*i$ as $4*(i+2i)$
- Access element according to its offset within structure
 - Offset by 8
 - Assembler gives displacement as $a + 8$
 - » Linker must set actual value

```
struct s6 {  
    short i;  
    float v;  
    short j;  
} a[10];
```

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

```
# %eax = idx  
leal (%eax,%eax,2),%eax # 3*idx  
movswl a+8(,%eax,4),%eax
```

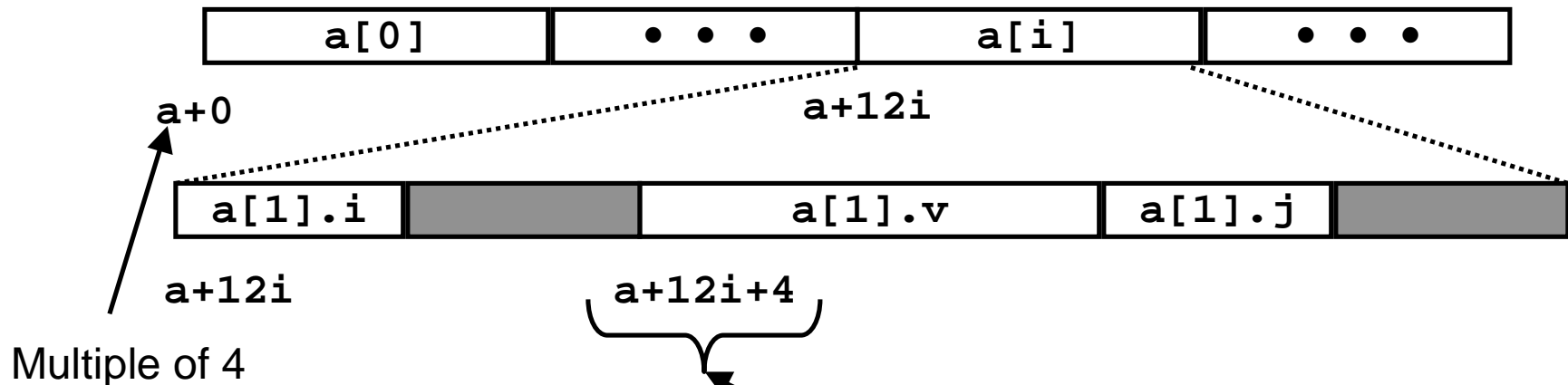


Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
 - a must be multiple of 4
- Offset of element within structure must be multiple of element's alignment requirement
 - v 's offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
 - Structure padded with unused space to be 12 bytes

```
struct s6 {  
    short i;  
    float v;  
    short j;  
} a[10];
```

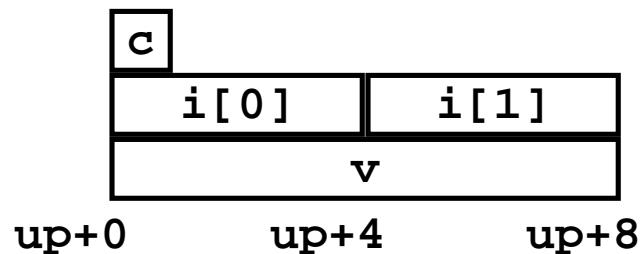


Union Allocation

Principles

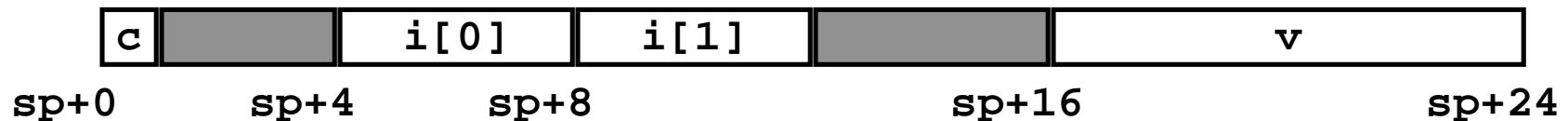
- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

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



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

(Windows alignment)



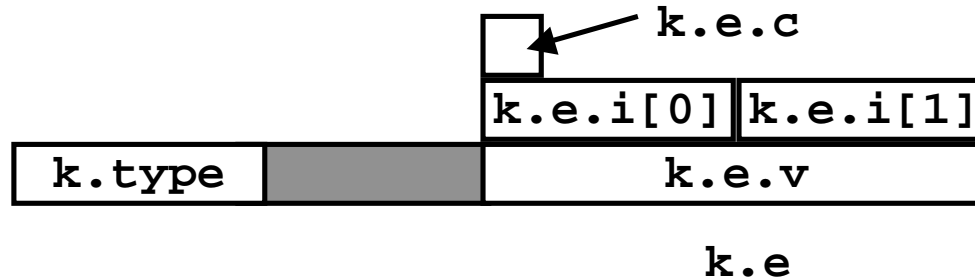
Implementing “Tagged” Union

- Structure can hold 3 kinds of data
- Only one form at any given time
- Identify particular kind with flag `type`

```
typedef enum { CHAR, INT, DBL }
    utype;

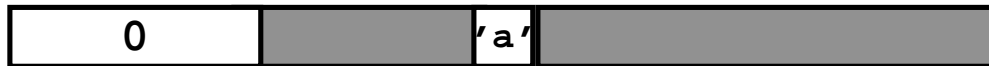
typedef struct {
    utype type;
    union {
        char c;
        int i[2];
        double v;
    } e;
} store_ele, *store_ptr;

store_ele k;
```



Using “Tagged” Union

```
store_ele k1;  
k1.type = CHAR;  
k1.e.c = 'a';
```



```
store_ele k2;  
k2.type = INT;  
k2.e.i[0] = 17;  
k2.e.i[1] = 47;
```

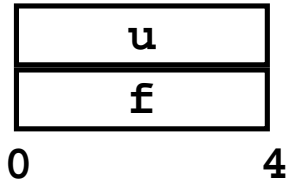


```
store_ele k3;  
k3.type = DBL;  
k1.e.v =  
    3.14159265358979323846;
```



Using Union to Access Bit Patterns

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



- **Get direct access to bit representation of float**
- **bit2float generates float with given bit pattern**
 - NOT the same as `(float) u`
- **float2bit generates bit pattern from float**
 - NOT the same as `(unsigned) f`

```
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;  
}
```

Byte Ordering

Idea

- Long/quad words stored in memory as 4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian

- Most significant byte has lowest address
- IBM 360/370, Motorola 68K, Sparc

Little Endian

- Least significant byte has lowest address
- Intel x86, Digital VAX

Byte Ordering Example

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

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

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]);

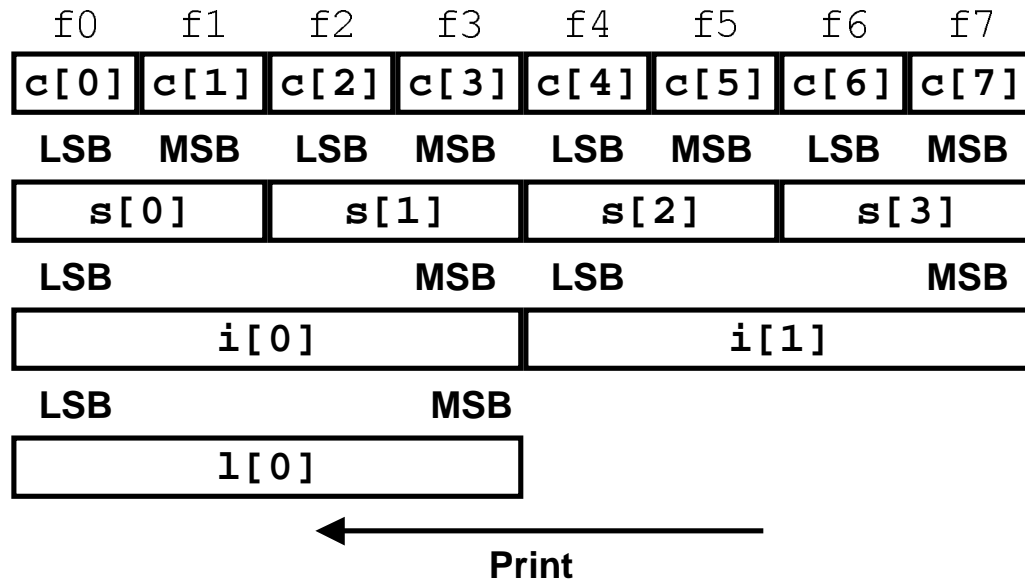
printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
      dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

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 x86

Little Endian

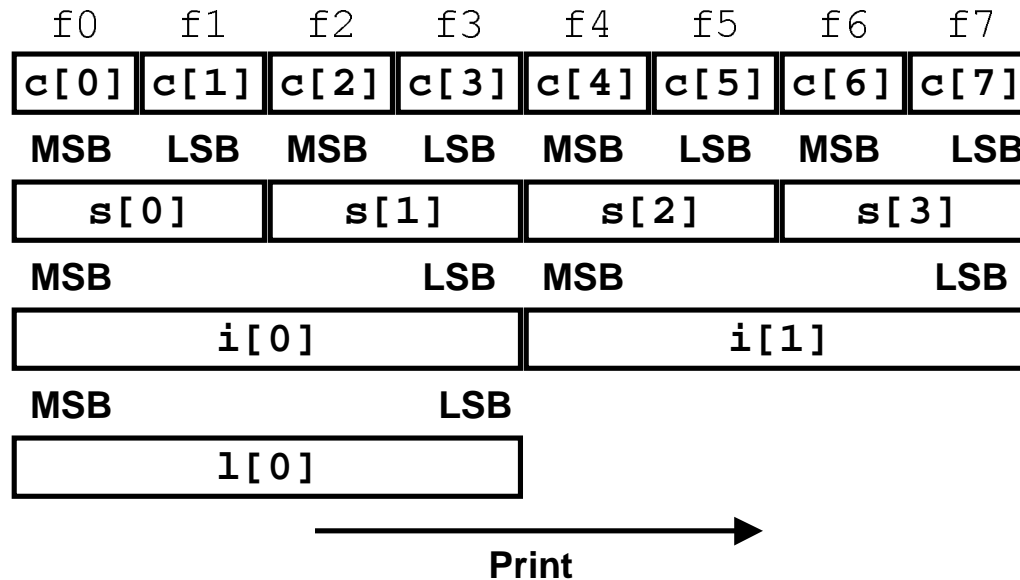


Output on Pentium:

```
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    == [f3f2f1f0]
```

Byte Ordering on Sun

Big Endian

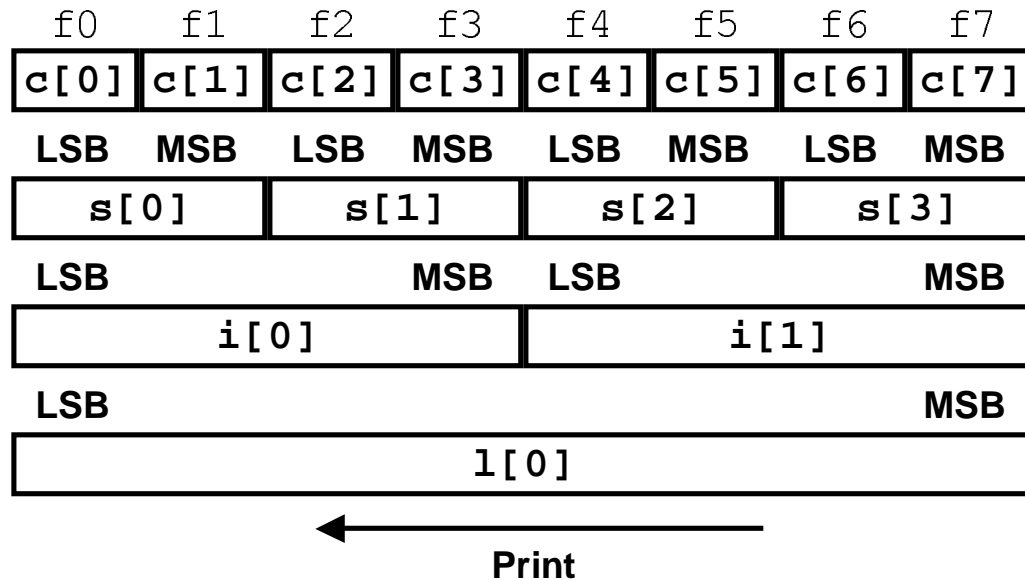


Output on Sun:

```
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 Alpha

Little Endian



Output on Alpha:

```
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]
```


Byte-Level Operations

IA32 Support

- Arithmetic and data movement operations have byte-level version
`movb`, `addb`, `testb`, etc.
- Some registers partially byte-addressable
- Can perform single byte memory references

Compiler

- Parameters and return values of type `char` passed as `int`'s
- Use `movsbl` to sign-extend byte to `int`

<code>%eax</code>	<code>%ah</code>	<code>%al</code>
<code>%edx</code>	<code>%dh</code>	<code>%dl</code>
<code>%ecx</code>	<code>%ch</code>	<code>%cl</code>
<code>%ebx</code>	<code>%bh</code>	<code>%bl</code>

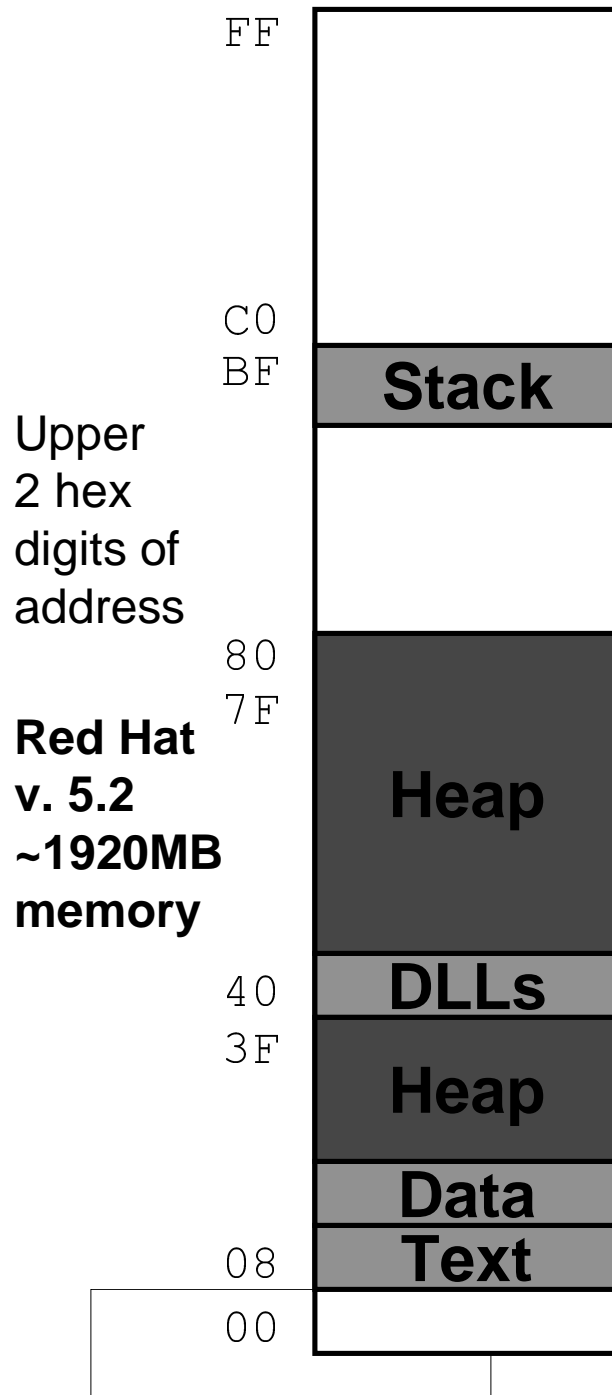
Byte-Level Operation Example

- Compute Xor of characters in string

```
char string_xor(char *s)
{
    char result = 0;
    char c;
    do {
        c = *s++;
        result ^= c;
    } while (c);
    return result;
}
```

```
# %edx = s, %cl = result
movb $0,%cl      # result = 0
L2:              # loop:
movb (%edx),%al  # *s
incl %edx        # s++
xorb %al,%cl     # result ^= c
testb %al,%al   # al
jne L2           # If != 0, goto loop
movsbl %cl,%eax # Sign extend to int
```

Linux Memory Layout



Stack

- Runtime stack (8MB limit)

Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

DLLs

- Dynamically Linked Libraries
- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed

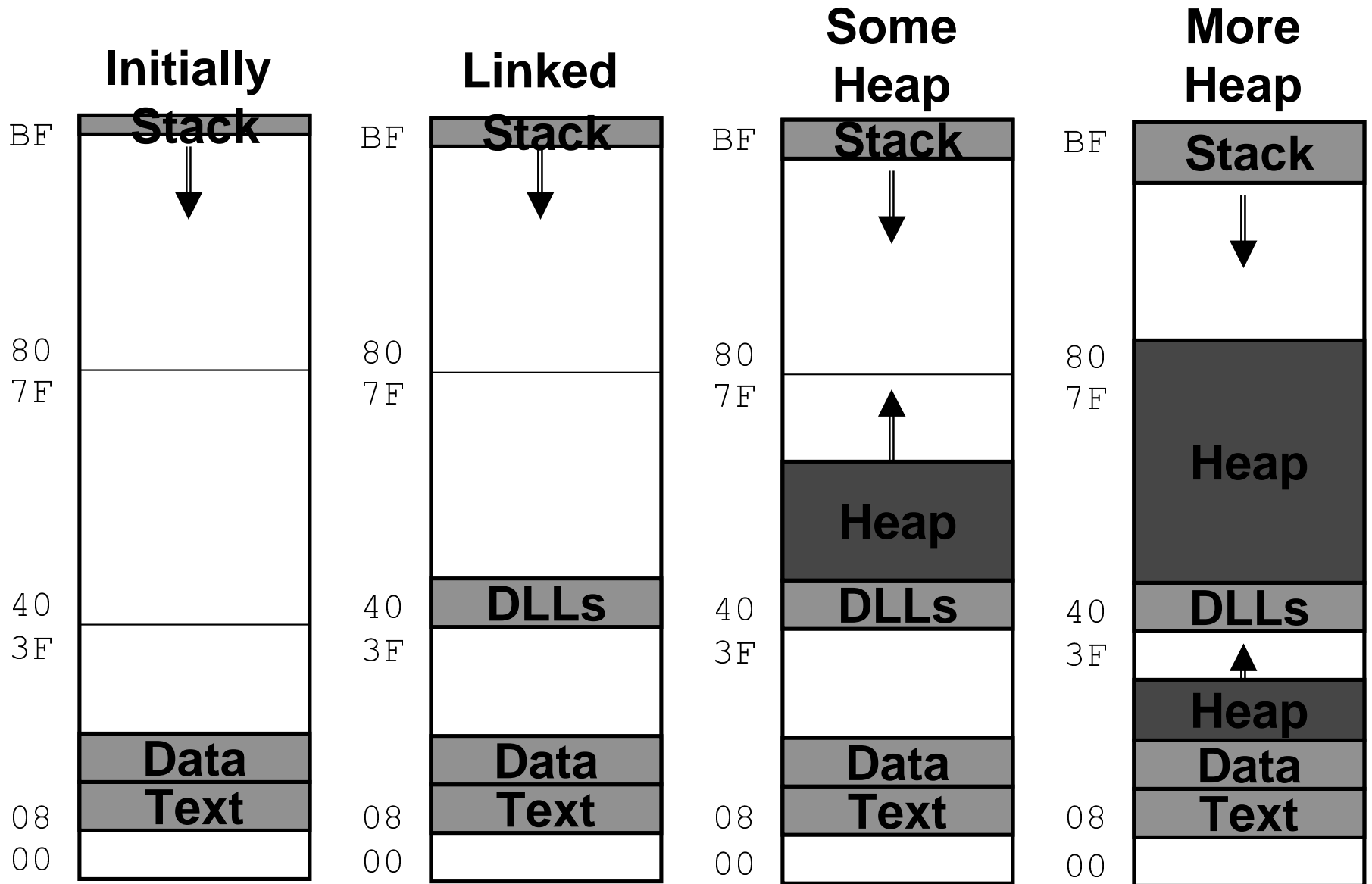
Data

- Statically allocated data
- E.g., arrays & strings declared in code

Text

- Executable machine instructions
- Read-only

Linux Memory Allocation



Memory Allocation Example

```
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */
int beyond;
char *p1, *p2, *p3, *p4;
int useless() { return 0; }

int main()
{
    p1 = malloc(1 <<28); /* 256 MB */
    p2 = malloc(1 << 8); /* 256 B */
    p3 = malloc(1 <<28); /* 256 MB */
    p4 = malloc(1 << 8); /* 256 B */
    /* Some print statements ... */
}
```

Dynamic Linking Example

```
(gdb) print malloc
$1 = {<text variable, no debug info>
      0x8048454 <malloc>}
(gdb) run
Program exited normally.
(gdb) print malloc
$2 = {void *(unsigned int)}
      0x40006240 <malloc>
```

Initially

- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x08048454

Final

- Code in DLL region

Breakpointing Example

```
(gdb) break main
(gdb) run
  Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
$3 = (void *) 0xbffffc78
```

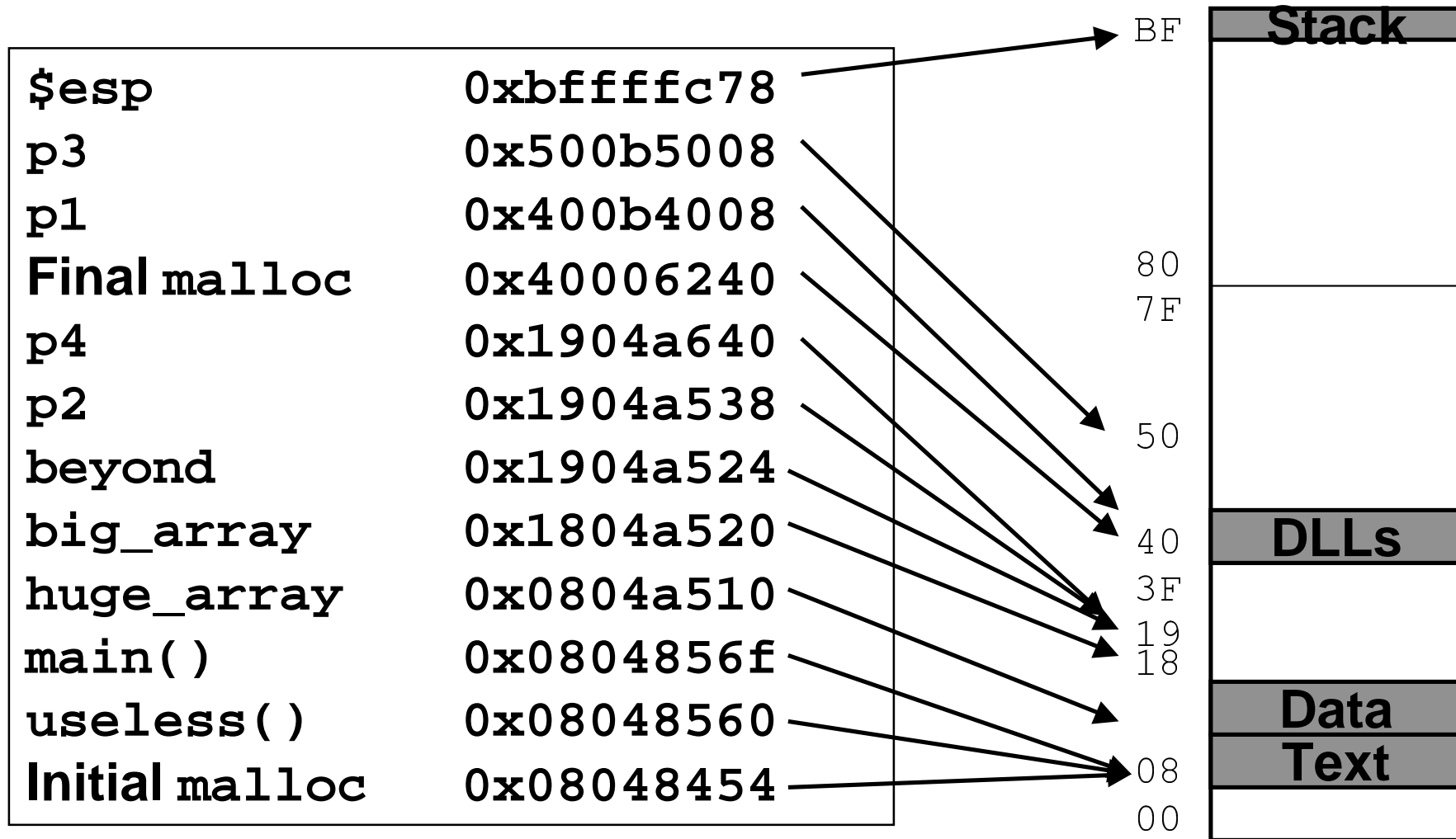
Main

- Address 0x804856f should be read 0x0804856f

Stack

- Address 0xbffffc78

Example Addresses



C operators

Operators

`() [] -> .`
`! ~ ++ -- + - * & (type) sizeof`
`* / %`
`+ -`
`<< >>`
`< <= > >=`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= += -= *= /= %= &= ^= != <<= >>=`
`,`

Associativity

left to right
right to left
left to right
left to right
left to right
left to right
left to right
left to right
left to right
left to right
right to left
right to left
left to right

Note: Unary +, -, and * have higher precedence than binary forms

C pointer declarations

<code>int *p</code>	p is a pointer to int
<code>int *p[13]</code>	p is an array[13] of pointer to int
<code>int *(p[13])</code>	p is an array[13] of pointer to int
<code>int **p</code>	p is a pointer to a pointer to an int
<code>int (*p)[13]</code>	p is a pointer to an array[13] of int
<code>int *f()</code>	f is a function returning a pointer to int
<code>int (*f)()</code>	f is a pointer to a function returning int
<code>int (*(*f()) [13]) ()</code>	f is a function returning ptr to an array[13] of pointers to functions returning int
<code>int (*(*x[3]) ()) [5]</code>	x is an array[3] of pointers to functions returning pointers to array[5] of ints