Assembly IV:
Complex Data Types

Jo, Heeseung

Question

```
#define tri(x...) { \
        printf("[%d:%s:%d] %s = ", getpid(), __func__, __LINE__, #x); \
        printf("%d\n", x); }

int main()
{
        int array[5]={2,4,8,10,12};

        tri(&array);
        tri(array);

        tri(*array);
        tri(*array+1);
        tri(*(array+1));
}
```

Basic Data Types

Integer

- 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

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

GAS: GNU Assembler Syntax

Complex Data Types

Complex data types in C

- Pointers
- Arrays
- Structures
- Unions
- ...

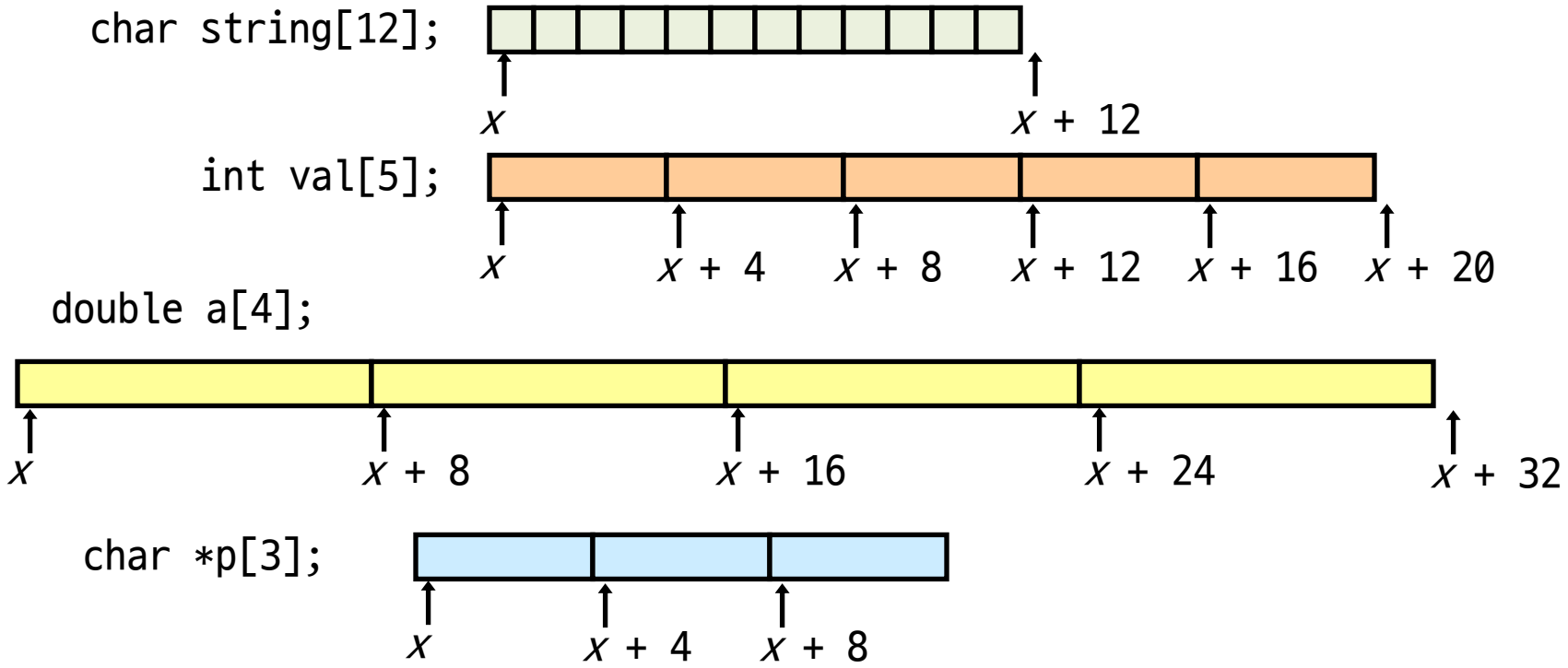
Can be combined

- Pointer to pointer, pointer to array, …
- Array of array, array of structure, array of pointer, …
- Structure in structure, pointer in structure, array in structure, …

Array Allocation

Basic principle: **T A[L];**

- Array of data type T and length L
- Contiguously allocated region of L * sizeof(T) bytes

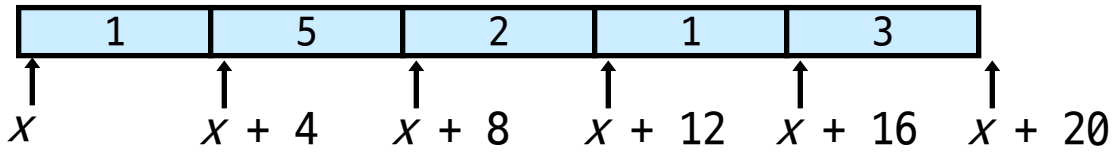


Array Access

Basic principle: **T A[L];**

- Array of data type T and length L
- Identifier A can be used as a pointer to element 0

int val[5];

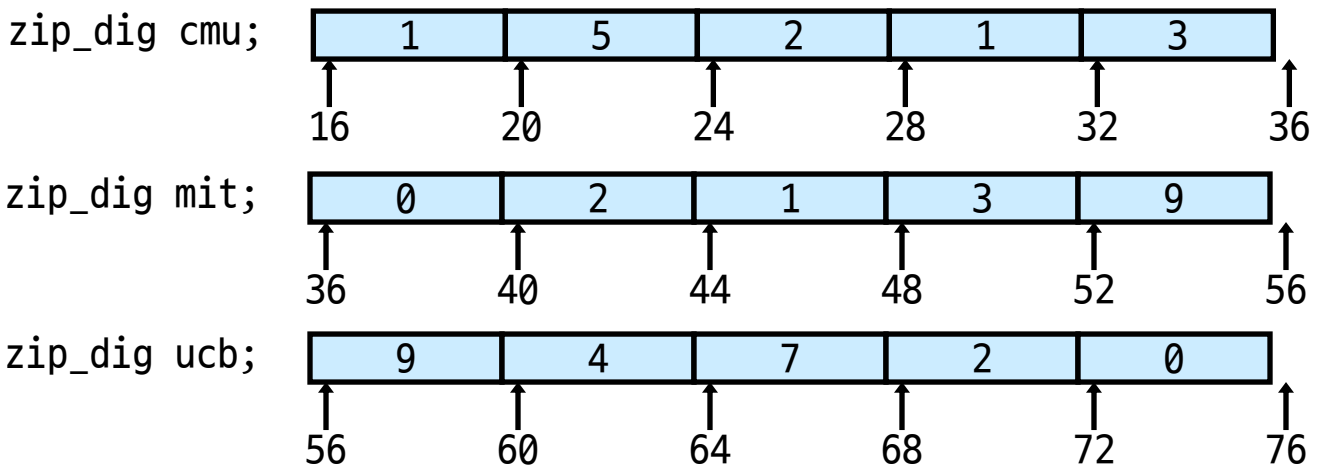


Reference	Type	Value
val[4]	int	
val		
val + 1		
&val[2]		
val[5]		
*(val+1)		
val + i		

Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



Notes

- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example (1)

zip_dig z;



Computation

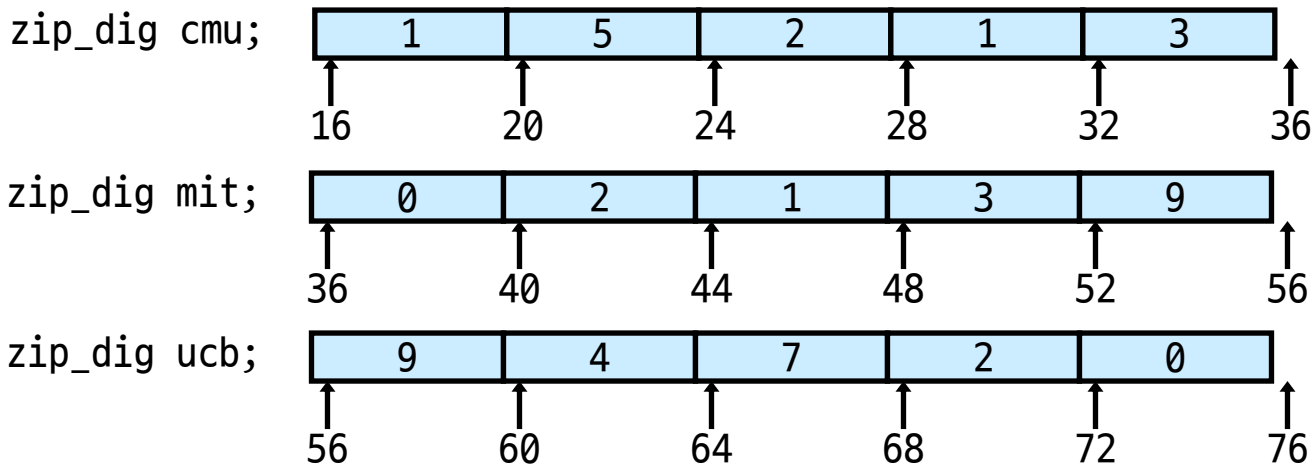
- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4 * %eax + %edx
- Use memory reference: (%edx, %eax, 4)

```
int get_digit (zip_dig z, int dig)
{
  return z[dig];
}
```

Memory Reference Code

```
# %edx = z
# %eax = dig
movl (%edx, %eax, 4), %eax # z[dig]
```

Array Accessing Example (2)

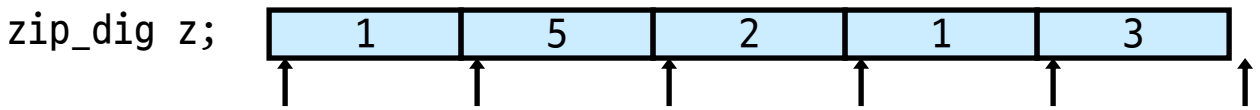


Code does not do any bounds checking!

Reference	Address	Value	Guaranteed?
mit[3]			
mit[5]			
mit[-1]			
cmu[15]			

- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Array Loop Example (1)



Original source

```
int zd2int(zip_dig z){
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++)
    zi = 10 * zi + z[i];
  return zi;
}
```

Transformed version

- As generated by GCC
- Eliminate loop variable i
- Convert array code to pointer code
- Express in do-while form
  - No need to test at entrance
- If z=0x100, then zend= ?

```
int zd2int(zip_dig z){
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

Array Loop Example (2)

Registers

- %ecx z
- %eax zi
- %ebx zend

```
int zd2int(zip_dig z){
  int zi = 0;
  int *zend = z + 4;
  do {
    zi = 10 * zi + *z;
    z++;
  } while(z <= zend);
  return zi;
}
```

z++ increments by 4

10 * zi + *z
= *z + 2*(zi+4*zi)

```
                                # %ecx = z
   xorl %eax, %eax              # zi = 0
   leal 16(%ecx), %ebx          # zend = z + 4
.L59:
   leal (%eax, %eax, 4), %edx   # 5*zi
   movl (%ecx), %eax            # *z
   addl $4, %ecx                # z++
   leal (%eax, %edx, 2), %eax   # zi = *z + 2*(5*zi)
   cmpl %ebx, %ecx              # z : zend
   jle .L59                     # if <= goto loop
```

Question

```
#define tri(x...) { \
          printf("[%d:%s:%d] %s = ", getpid(), \
          __func__, __LINE__, #x); \
          printf("%d\n", x); }

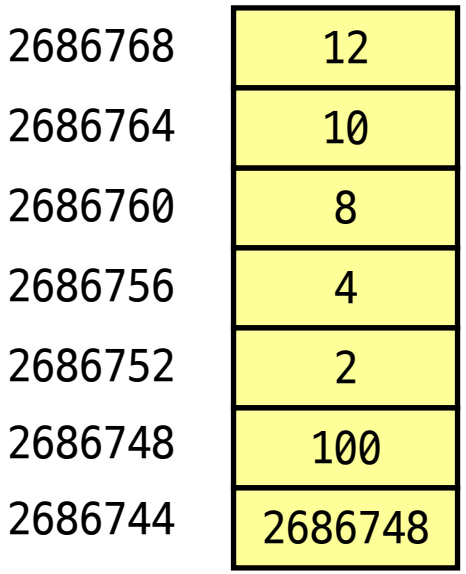
int main()
{
        int array[5]={2,4,8,10,12};
        int num=100;
        int *p=&num;

        tri(&num);
        tri(&num+1);
        tri(*(&num+1));

        tri(&p);
        tri(p);

        tri(&array);
        tri(array);

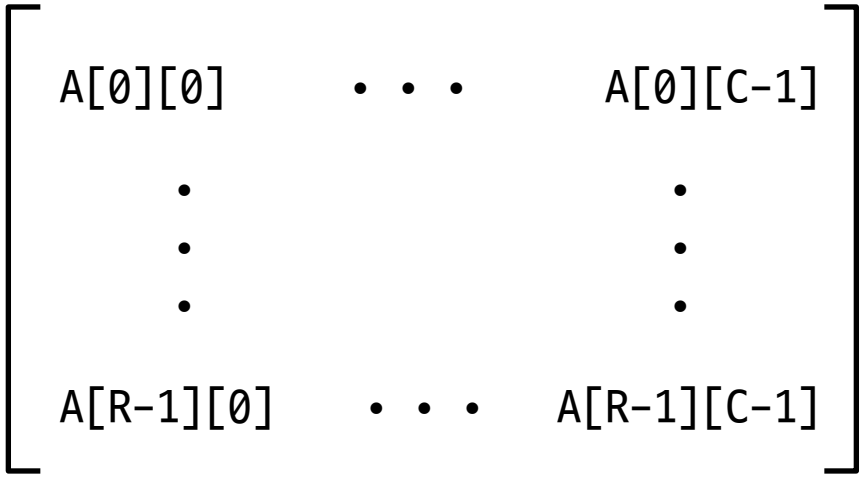
        tri(*array);
        tri(*array+1);
        tri(*(array+1));
}
```



Nested Array (1)

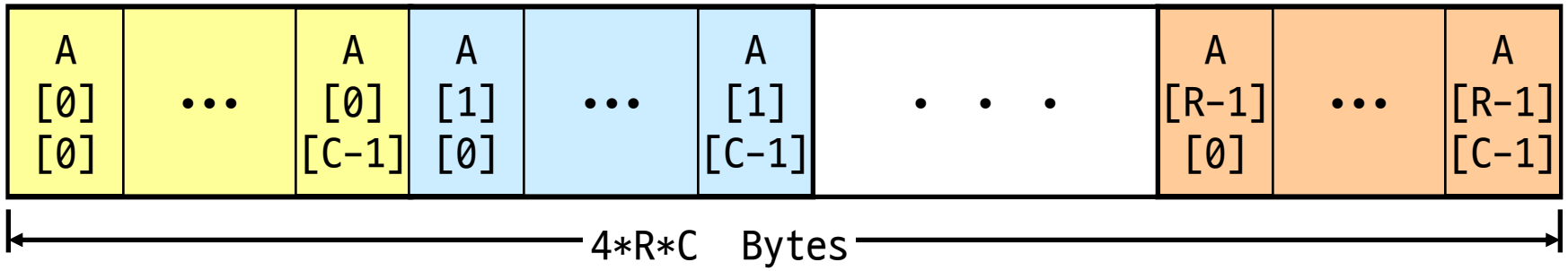
Declaration: **T A[R][C];**

- 2D array of data type T
- R rows, C columns
- Array size =
  R * C * sizeof(T)



Arrangement

- Row-major ordering



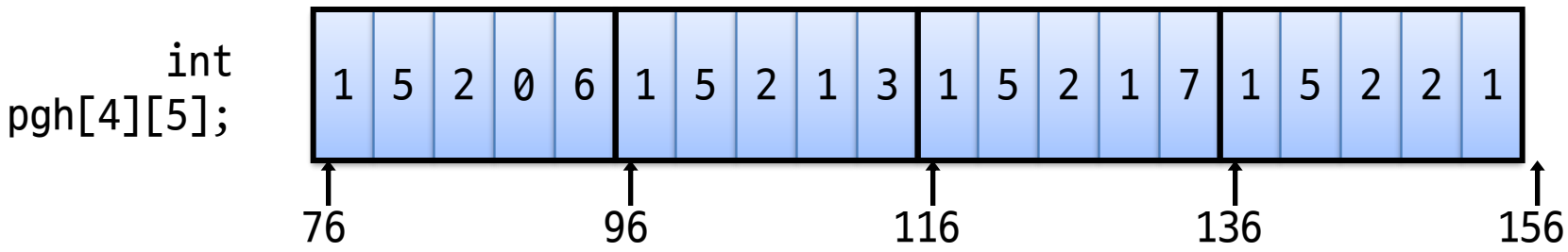
Nested Array (2)

C code

- Variable pgh denotes array of 4 elements
  - Allocated contiguously
- Each element is an array of 5 int's
  - Allocated contiguously
- pgh[1][1]? pgh[1]? pgh[8]?

```
int pgh[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

Row-major ordering of all elements guaranteed

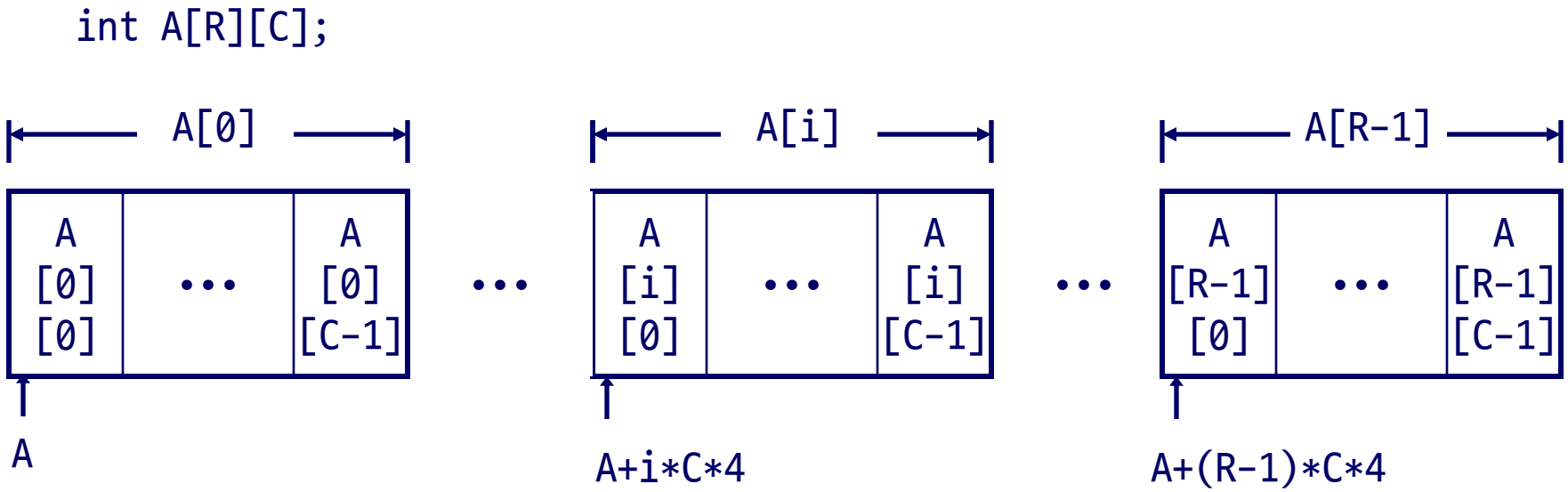


Nested Array Access (1)

Row vectors

- A[i] is array of C elements
- Each element of type T requires K bytes
- Starting address A + i * (C * K)

```
int pgh[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



Nested Array Access (2)

Row vectors

- pgh[index] is array of 5 int's
- Starting address **pgh + 20 * index**

A + index * (C * K)

```
????? get_pgh_zip(int index)
{
     return pgh[index];
}
```

Code

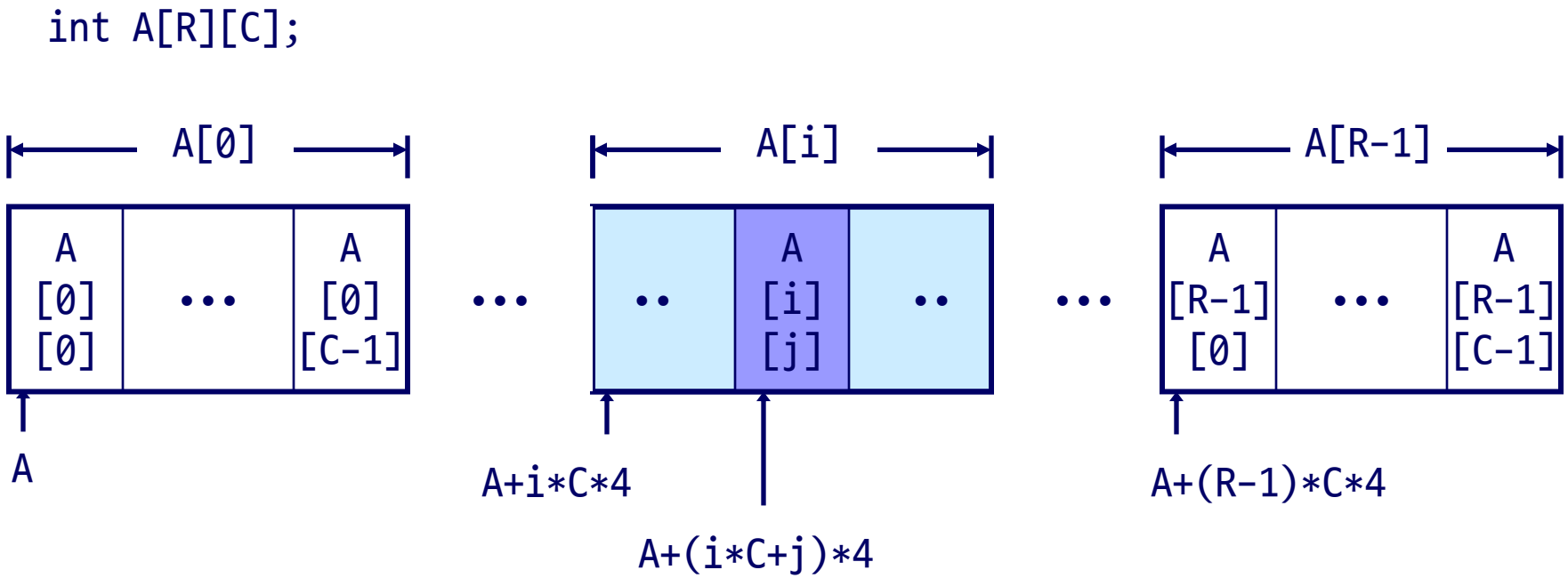
- Computes and returns address
- Compute as **pgh + 4 * (index + 4 * index)**

```
# %eax = index
 leal (%eax,%eax,4),%eax      # 5 * index
 leal pgh(,%eax,4),%eax       # pgh + (20 * index)
```

Nested Array Access (3)

Array elements

- A[i][j] is element of type T
- Address A + i * (C * K) + j * K = A + (i * C + j) * K



Nested Array Access (4)

Array elements

- pgh[index][dig] is int
- Address:
  pgh + 20 * index + 4 * dig

```
??? get_pgh_digit (int index, int dig)
{
   return pgh[index][dig];
}
```

Code

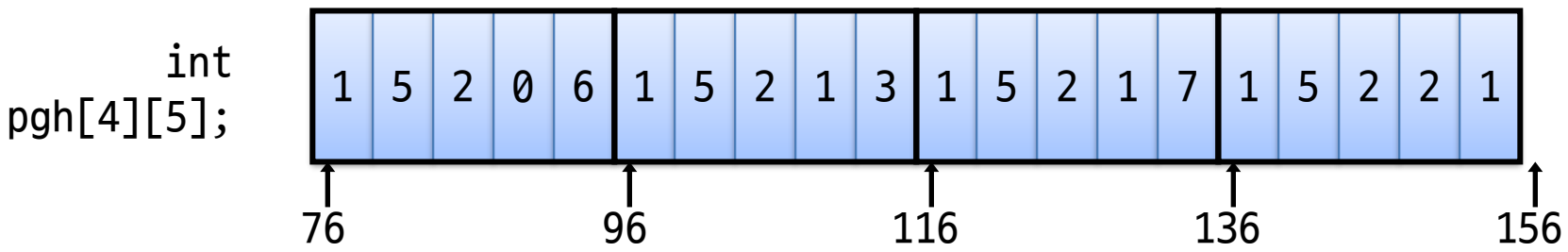
- Computes address **pgh + 4*dig + 4*(index + 4*index)**
- **movl** performs memory reference

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4), %edx            # 4*dig
leal (%eax,%eax,4), %eax         # 5*index
movl pgh(%edx,%eax,4), %eax      # *(pgh + 4*dig + 20*index)
```

Nested Array Access (5)

Strange referencing examples

- Code does not do any bounds checking
- Ordering of elements within array guaranteed



Reference	Address	Value	Guaranteed?
pgh[3][3]	76 + 20*3 + 4*3 = 148	2	Yes
pgh[2][5]			
pgh[2][-1]			
pgh[4][-1]			
pgh[0][19]			
pgh[0][-1]			

Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler optimizations

- Compiler often turns array code into pointer code
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Exercise

```
  0   -1   -2   -3   -4
-10  -11  -12  -13  -14
-20  -21  -22  -23  -24
-30  -31  -32  -33  -34
```

```
2686688
2686688

2686708
2687088

-10
-10

-9
-11

2687088
74          // garbage

-11
-20
0           // garbage

-20
-30
```

```
int pgh[4][5];
int *x=(int *)pgh;

printf("%d\n", pgh);
printf("%d\n", pgh[0]);
printf("\n");

printf("%d\n", pgh[1]);
printf("%d\n", pgh[20]);
printf("\n");

printf("%d\n", pgh[1][0]);
printf("%d\n", *pgh[1]);
printf("\n");

printf("%d\n", *pgh[1]+1 );
printf("%d\n", *(pgh[1]+1) );
printf("\n");

printf("%d\n", pgh[20]);
printf("%d\n", *pgh[20]);
printf("\n");

printf("%d\n", *( *(pgh+1) +1 ) );
printf("%d\n", *( *pgh+10) );
printf("%d\n", **(pgh+10) );
printf("\n");

printf("%d\n", *(x+10) );
printf("%d\n", *(x+15) );
```

Structures

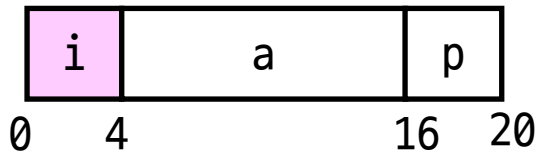
Concept

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

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

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

Memory Layout



Assembly

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

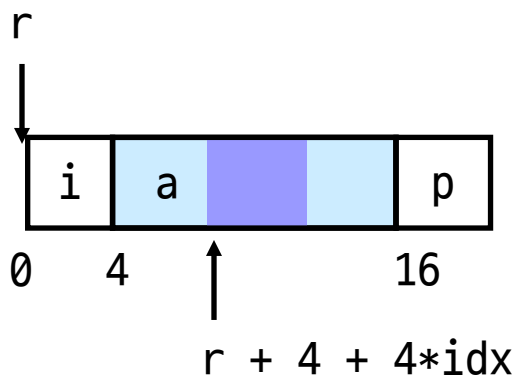
Structure Referencing (1)

Generating pointer to structure member

- Order and offset of each member determined at compile time

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

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



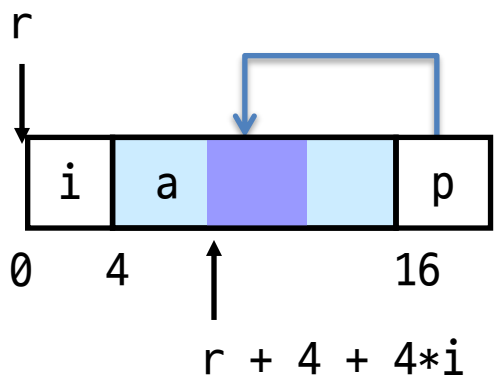

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax        # 4*idx
leal 4(%eax,%edx),%eax      # r+4*idx+4
```

Structure Referencing (2)

Generating pointer to member (cont'd)

```
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(%eax,%edx),%eax        # r+4+4*(r->i)
movl %eax,16(%edx)            # update r->p
```

Alignment (1)

Aligned data

- Primitive data type requires **K** bytes
- Address must be multiple of **K**
- Required on some machines; advised on IA-32
  - Treated differently by Linux and Windows

Motivation for aligning data

- Memory hardware accessed by (aligned) double or quad-words
  - Inefficient to load or store datum that spans quad word boundaries
- OS virtual memory system
  - Very tricky when datum spans 2 pages

Compiler

- Inserts gaps (or "pads") in structure to ensure correct alignment of fields

Alignment (2)

According to the 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)

- Windows, Linux: lowest 2 bits of address must be 00_2 (i.e., treated the same as a 4-byte primitive data type)

Alignment (3)

Offsets within structure

- Must satisfy element's alignment requirement

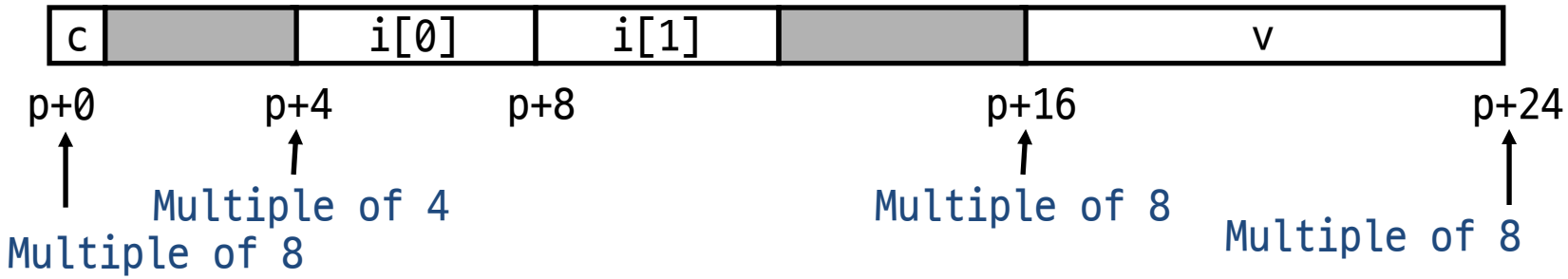
Overall structure placement

- 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

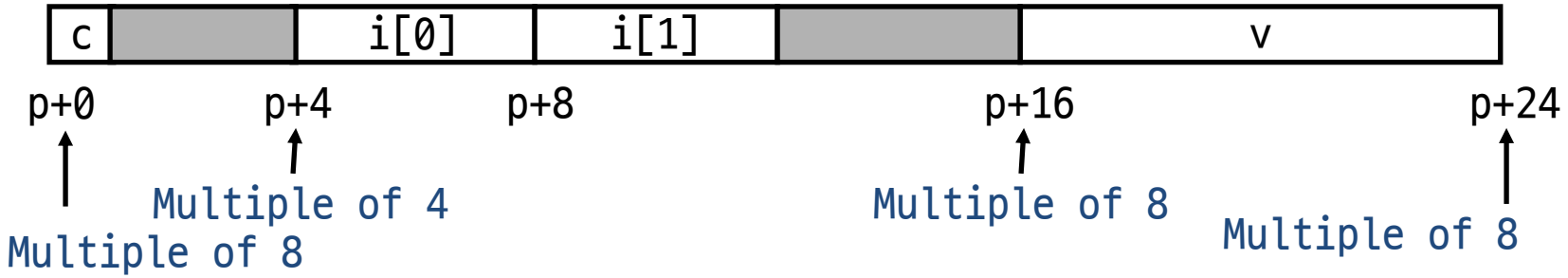


Alignment (4)

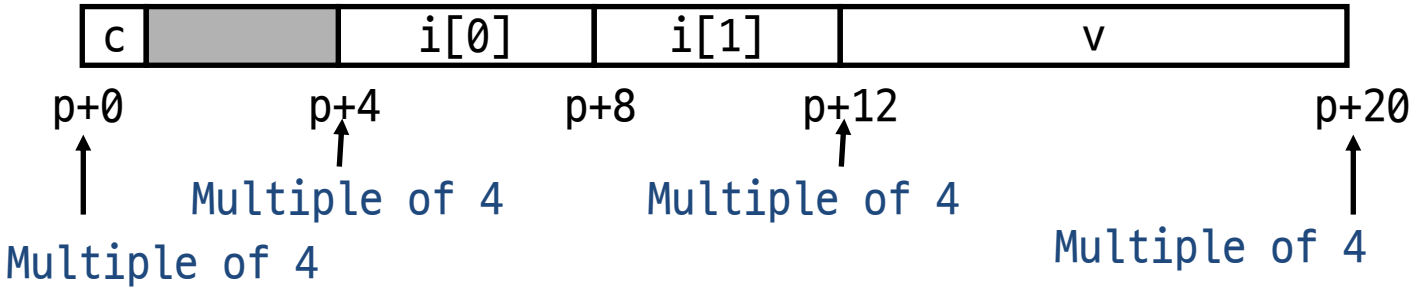
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Linux vs. Windows

- Windows (including Cygwin): K = 8



- Linux: K = 4

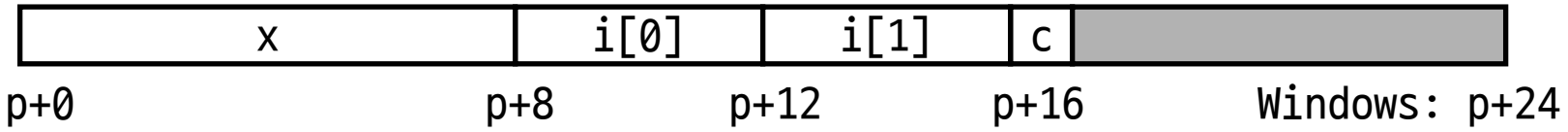


Alignment (5)

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 (all cases)



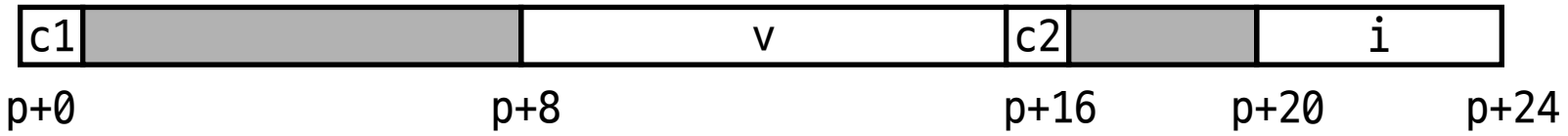
p+0 p+4 p+8 p+12 p+16 p+20

Alignment (6)

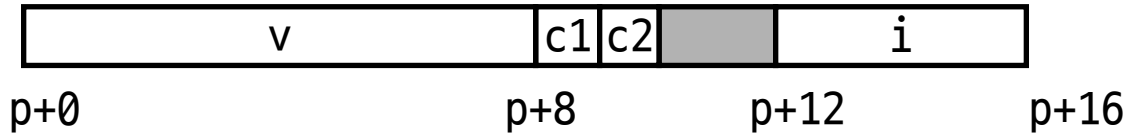
Ordering elements within structure

```
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;
```



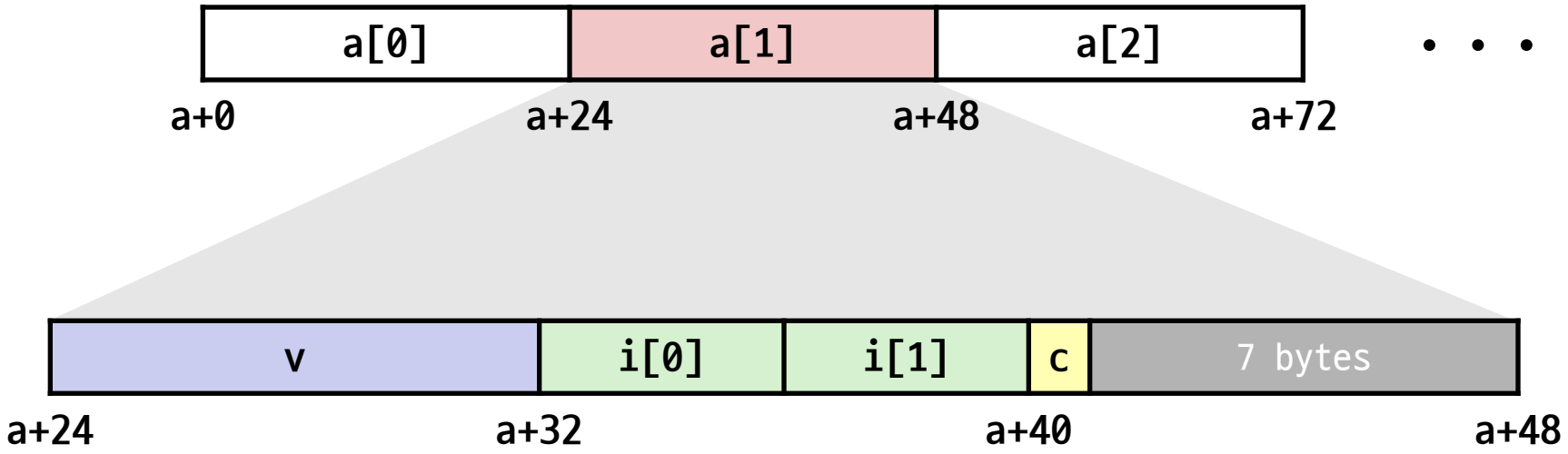
2 bytes wasted space

Arrays of Structures

Overall structure length multiple of K

Satisfy alignment requirement
for every element

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



Accessing Array Elements

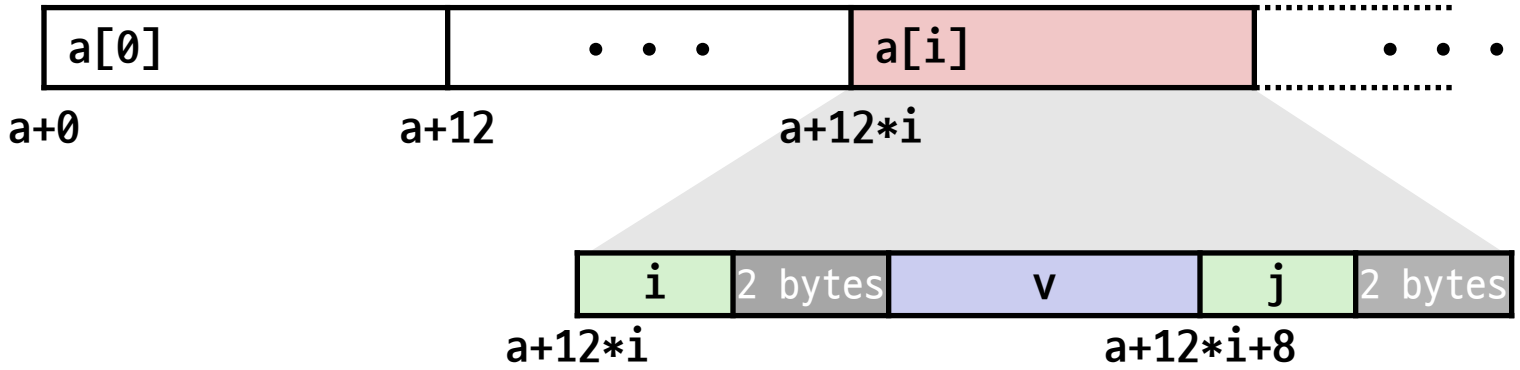
Compute array offset 12*i

- sizeof(S3), including alignment spacers

Element j is at offset 8 within structure

Assembler gives offset a+12*i + 8

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




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

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

Saving Space

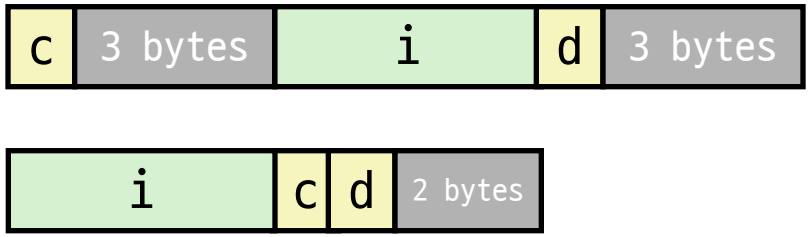
Put large data types first

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



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

Effect (K=4)



Double Alignment in C Structures on macOS:

- **Default Alignment:** A `double` data type typically requires 8-byte alignment on 64-bit systems, which includes modern macOS systems. This means that a `double` variable in memory should ideally start at an address that is a multiple of 8.

Example:

Consider the following structure:

```c
struct MyStruct {
    char c;
    double d;
    int i;
};
```

- `char c` (1 byte)
- `double d` (8 bytes, requires 8-byte alignment)
- `int i` (4 bytes, requires 4-byte alignment)

On a 64-bit macOS system:

- `c` is placed at an offset of 0.
- To align `d` to an 8-byte boundary, 7 padding bytes will be inserted after `c`.
- `d` is then placed at an offset of 8.
- `i` is placed at an offset of 16.
- The total size of `MyStruct` would be 1 (char) + 7 (padding) + 8 (double) + 4 (int) = 20 bytes.
- The overall alignment of `MyStruct` would be 8 bytes, due to the `double` member.

4.4. Field Packing

When a class has multiple fields, the JVM may distribute those fields in such a way as to minimize padding waste. For example, consider the *FieldsArrangement* class:

```
public class FieldsArrangement {
    private boolean first;
    private char second;
    private double third;
    private int fourth;
    private boolean fifth;
}
```

The field declaration order and their order in memory layout are different:

```
OFFSET  SIZE      TYPE DESCRIPTION                               VALUE
      0    12           (object header)                           N/A
     12     4       int FieldsArrangement.fourth                  N/A
     16     8    double FieldsArrangement.third                   N/A
     24     2      char FieldsArrangement.second                  N/A
     26     1   boolean FieldsArrangement.first                   N/A
     27     1   boolean FieldsArrangement.fifth                   N/A
     28     4           (loss due to the next object alignment)
```

The main motivation behind this is to minimize padding waste.

```python
import sys

class MyClass:
    class_var = 10 # 클래스 변수
    c="a"
    d=3.14
    def __init__(self, instance_var):
        self.instance_var = instance_var # 인스턴스 변수

obj = MyClass(1)

print(f"객체 obj의 크기: {sys.getsizeof(obj)} bytes")
print(f"클래스 MyClass의 크기: {sys.getsizeof(MyClass)} bytes")
print(f"인스턴스 변수 obj.instance_var의 크기: {sys.getsizeof(obj.instance_var)} bytes")
```

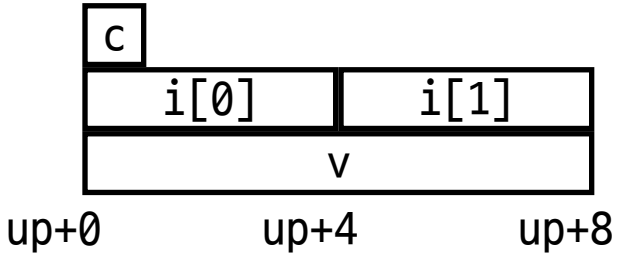
```
객체 obj의 크기: 56 bytes
클래스 MyClass의 크기: 1056 bytes
인스턴스 변수 obj.instance_var의 크기: 28 bytes
```

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;
```



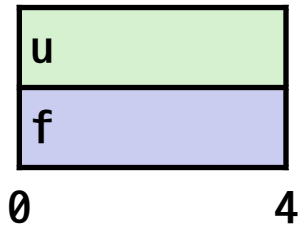

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```

(Windows alignment)



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;
}
```

Same as (float) u ?

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

Same as (unsigned) f ?

Byte Ordering Revisited

Idea

- Short/long/quad words stored in memory as 2/4/8 consecutive bytes
- Which is most (least) significant?
- Can cause problems when exchanging binary data between machines

Big Endian

- Most significant byte has lowest address
- Sparc

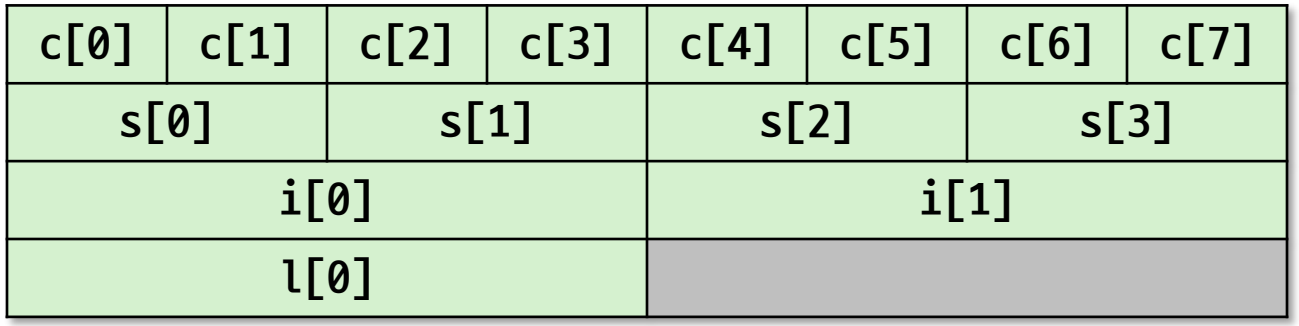
Little Endian

- Least significant byte has lowest address
- Intel x86

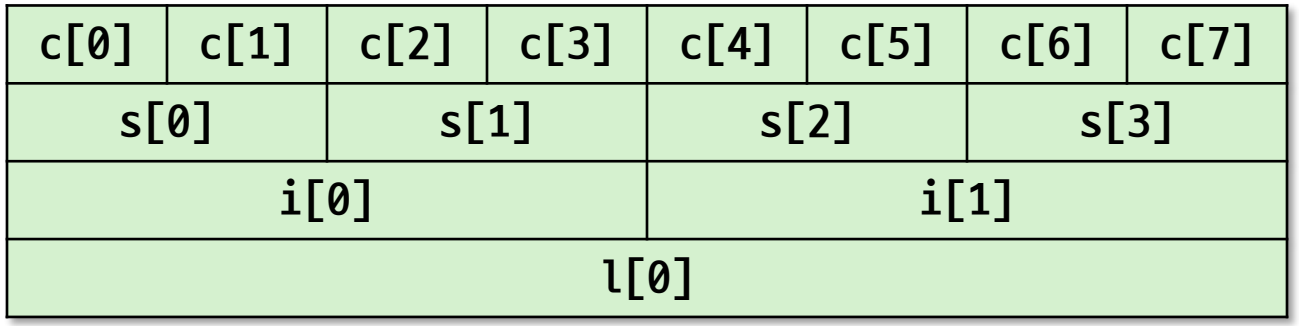
Byte Ordering Example

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

32-bit



64-bit



Byte Ordering Example (Cont).

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

```
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]);

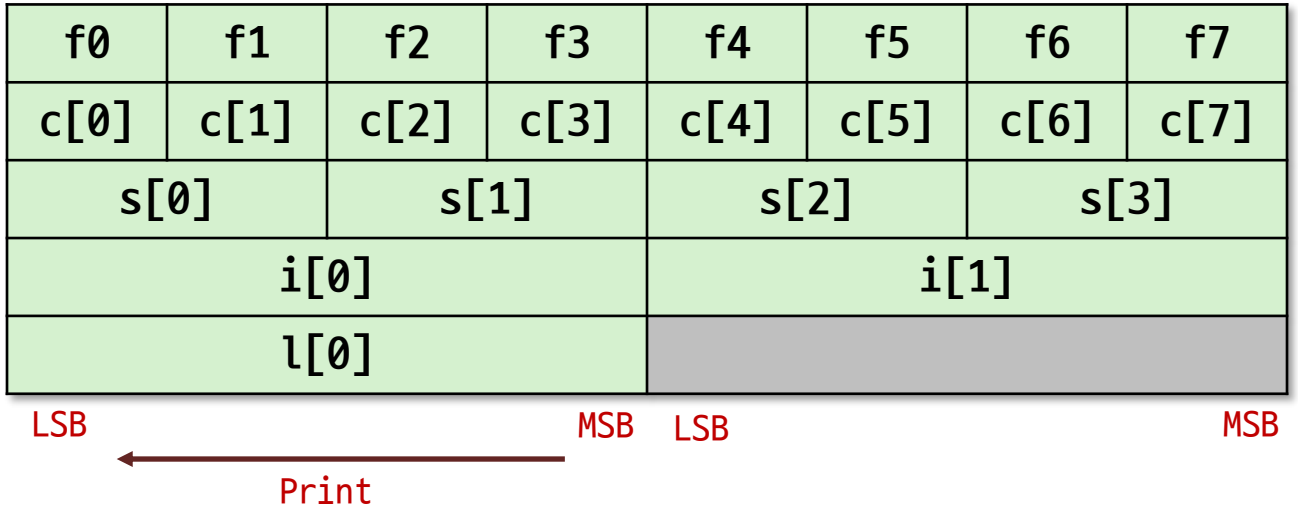
printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

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

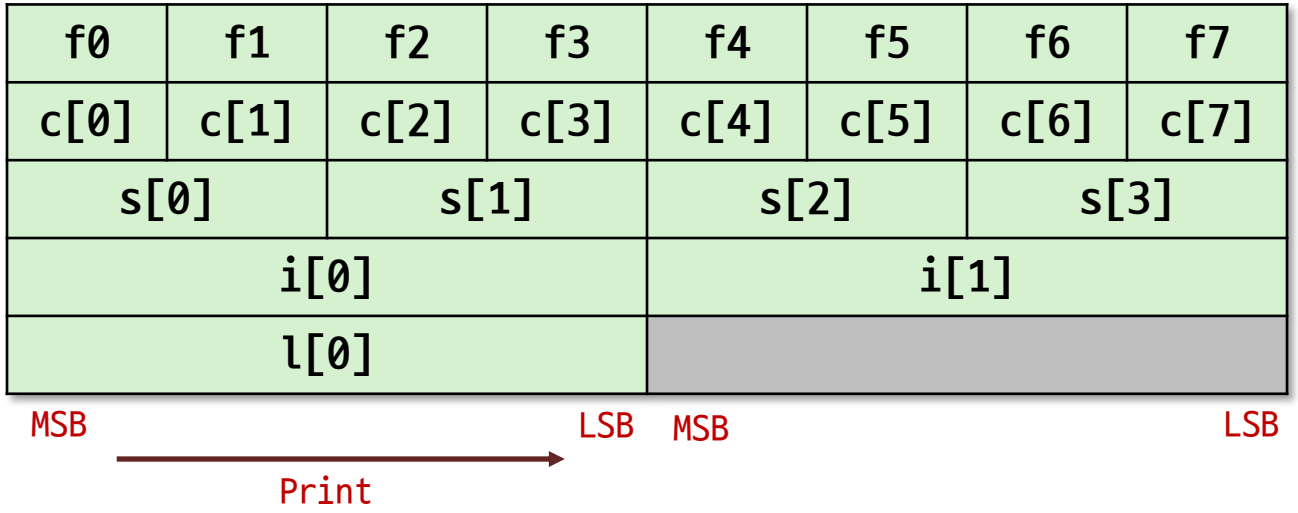


Output:

```
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

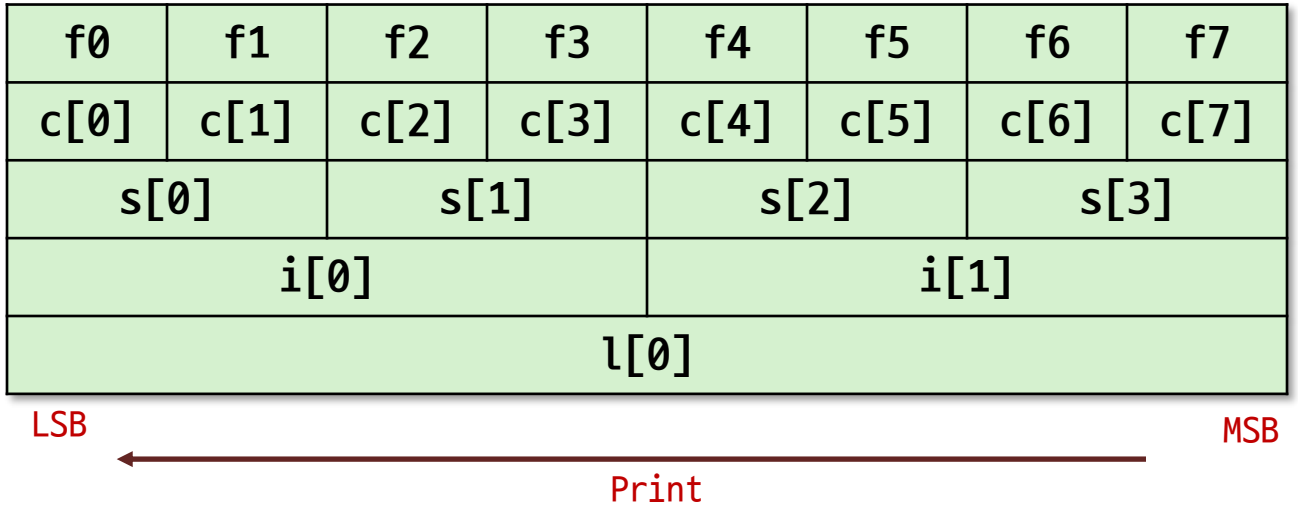


Output on Sun:

```
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



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]
```

Summary

Structures

- Allocate bytes in order declared
- To reduce memory consumption, consider allocation order
- Pad in middle and at end to satisfy alignment

Unions

- Overlay declarations