[Chap.3-6] Machine-level Representation of Programs

Young Ik Eom (yieom@skku.edu, 031-290-7120)
Distributing Computing Laboratory
Sungkyunkwan University
http://dclab.skku.ac.kr

Contents

- …
- Procedures
- Compound data structures
- Pointers
- GDB debugger
- Buffer overflow
- Floating-point codes

Compound Data Structures

- **Compound data structures**
  - Arrays
  - Structures
  - Unions

  - Data alignment

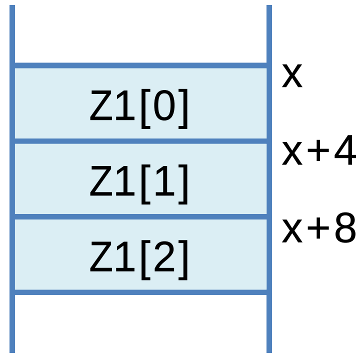
Arrays

- **Review: arrays and pointers in C**
  - Row-major order memory allocation

```
int z1[3];
```

Z1[0]	Z1[1]	Z1[2]

Programmer view

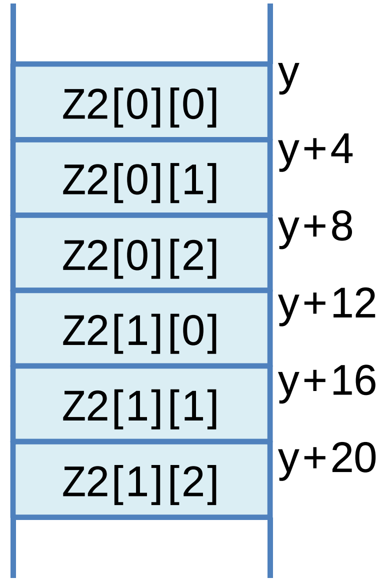


In memory

```
int z2[2][3];
```

Z2[0][0]	Z2[0][1]	Z2[0][2]
Z2[1][0]	Z2[1][1]	Z2[1][2]

Programmer view



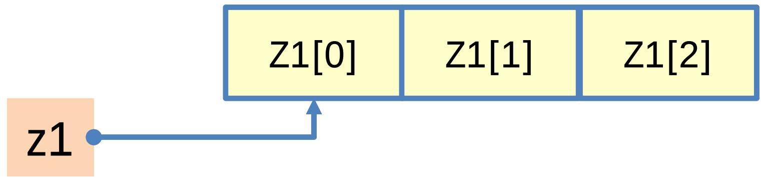
In memory

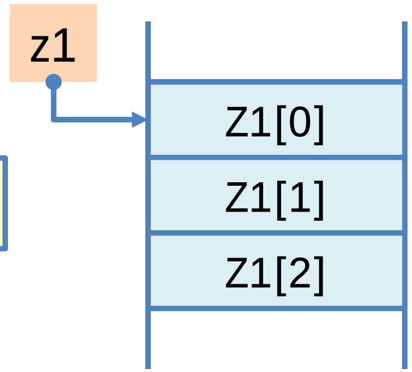
Arrays

Review: arrays and pointers in C

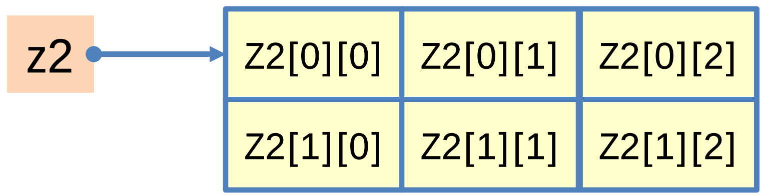
- Array name points to the 1st element of the array

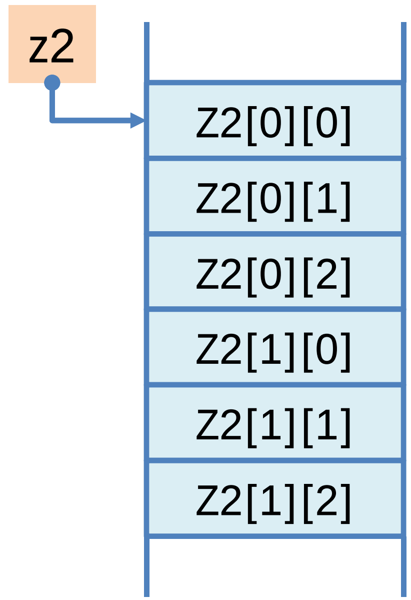
```
int z1[3];
```






```
int z2[2][3];
```

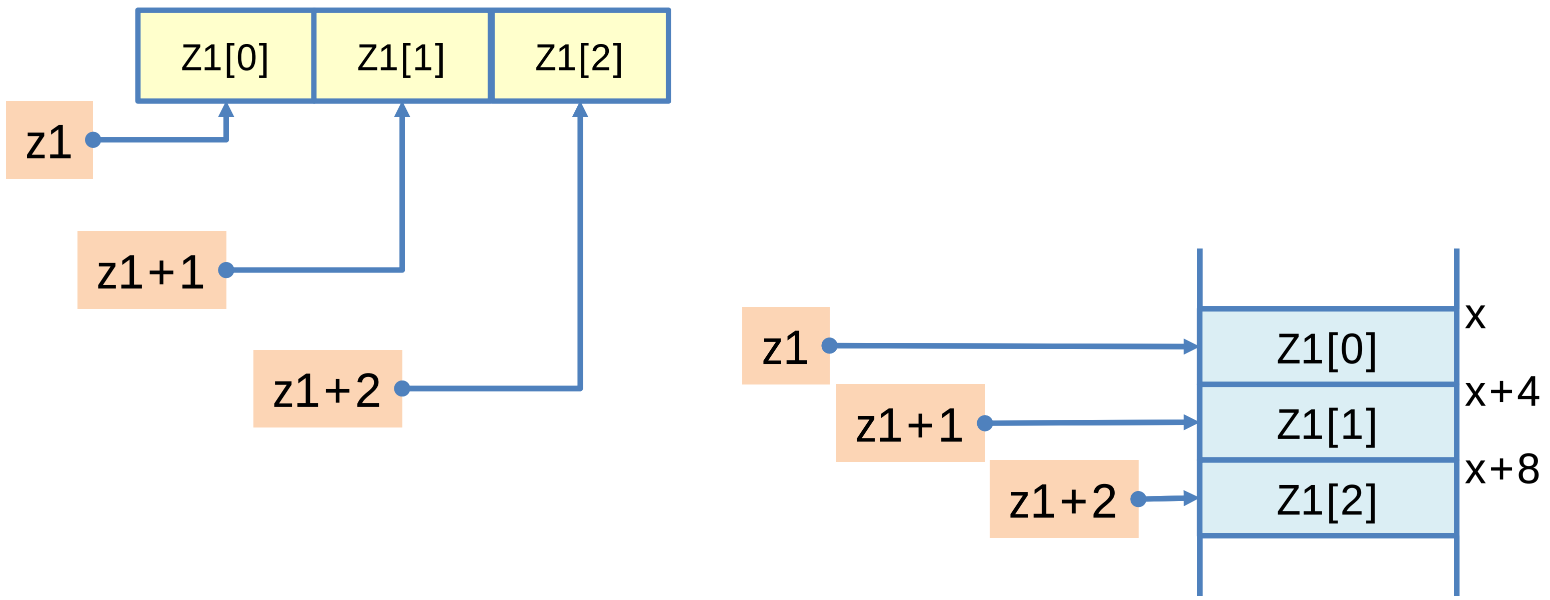




Arrays

- **Review: arrays and pointers in C**
  - Pointer arithmetic

```
int z1[3];
```

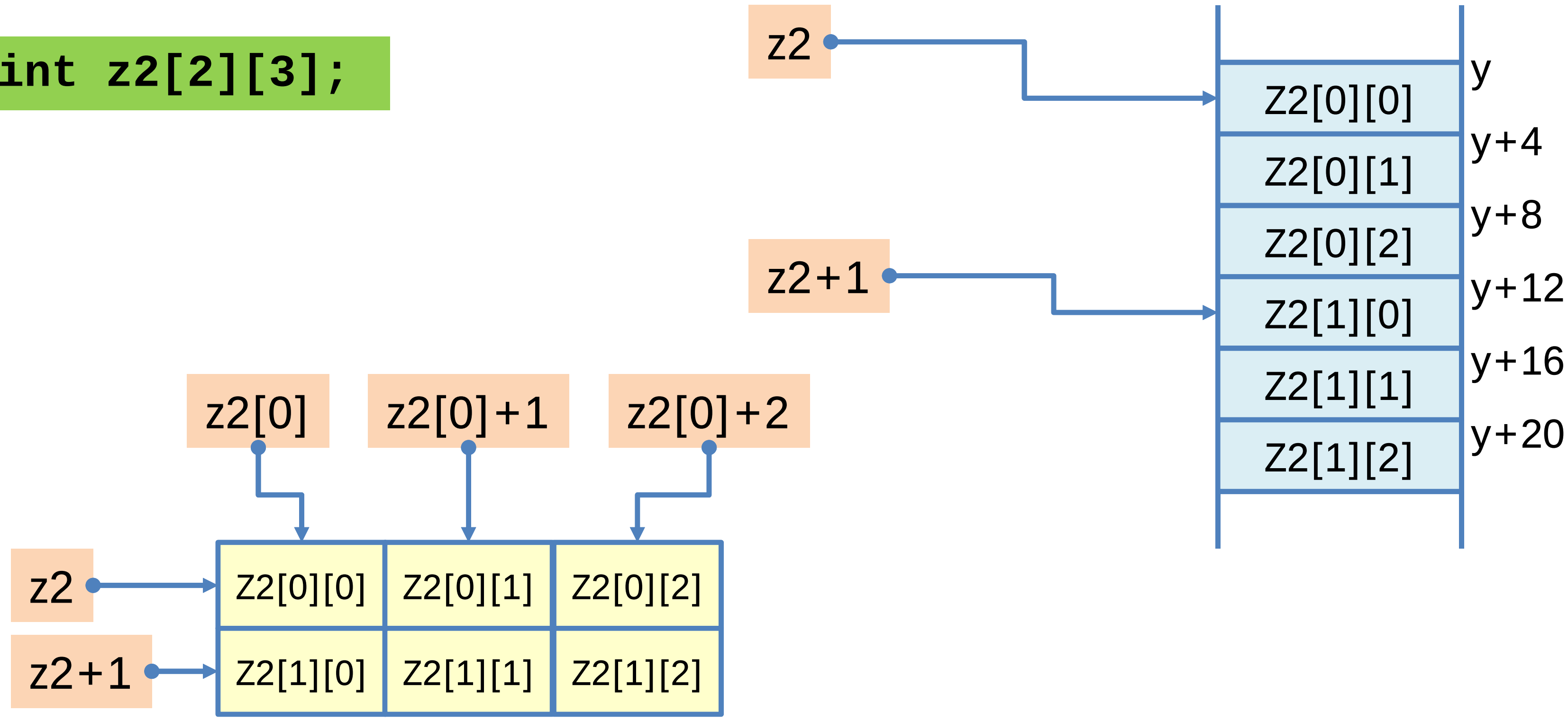


Arrays

Review: arrays and pointers in C

- Pointer arithmetic

```
int z2[2][3];
```

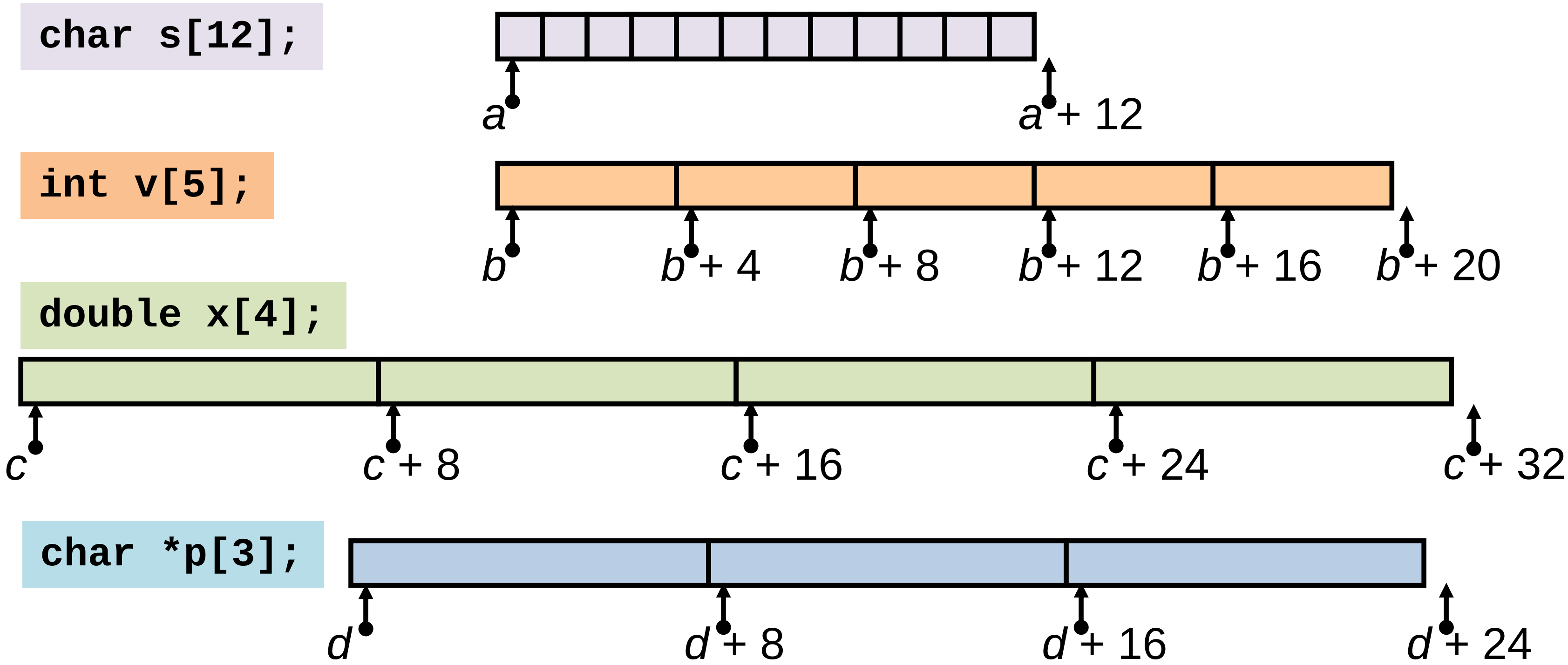


Arrays

char (*p) [3]

Basic principles `T A[N];`

- Array of **N** elements of data type **T**
- Allocates a contiguous region of **N**·sizeof(**T**) bytes
- Array element **i** will be stored at address ($\mathbf{x_A}$ + **i**·sizeof(**T**))



Arrays

Basic principles

- Memory referencing instructions of x86-64 are designed to simplify array access
- Example) **int z[10];**
  - The start address of the array **z** (**x_Z**) in **%rdx**
  - Index **i** in **%rcx**
  - Then to copy **z[i]** into **%eax**

```
movl (%rdx,%rcx,4),%eax
```

$x_Z + 4 \cdot i$

Arrays

Pointer arithmetic

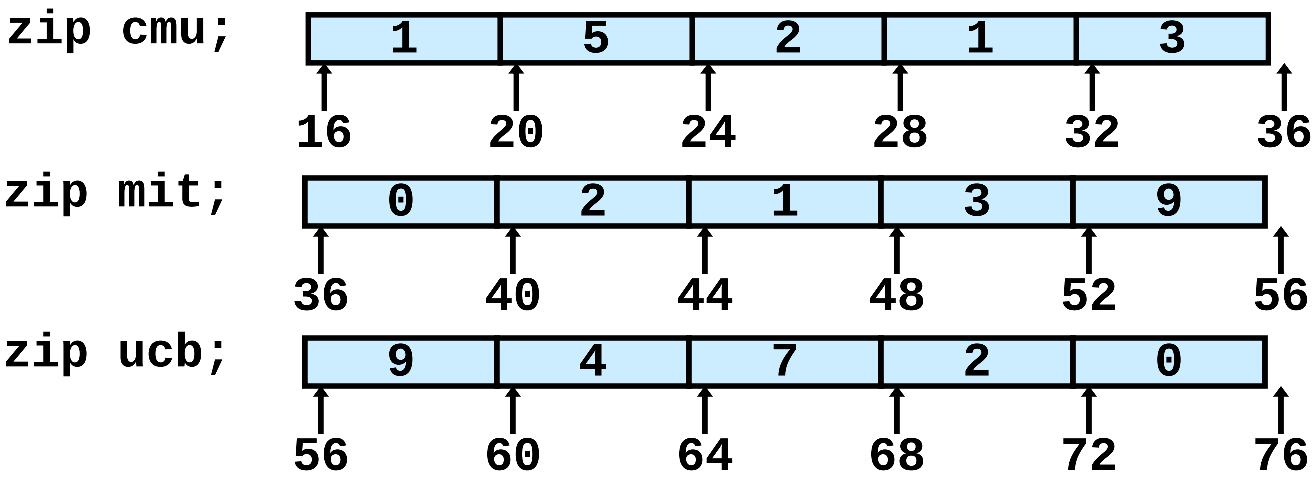
- The expression **p + i** has value $\mathbf{x_p + i \cdot sizeof(T)}$
  - **p** is a pointer to data of type **T**
  - The value of **p** is $\mathbf{x_p}$
- Example) Array **E** of integer
  - Starting address in **%rdx** and index **i** in **%rcx**
  - Destination **%eax** or **%rax**

Expression	Type	Value	Assembly code
E	int *	x_E	movq %rdx,%rax
E[0]	int	$M[x_E]$	movl (%rdx),%eax
E[i]	int	$M[x_E + 4i]$	movl (%rdx,%rcx,4),%eax
&E[2]	int *	$x_E + 8$	leaq 8(%rdx),%rax
E + i - 1	int *	$x_E + 4i - 4$	leaq -4(%rdx,%rcx,4),%rax
*(E + i – 3)	int	$M[x_E + 4i – 12]$	movl -12(%rdx,%rcx,4),%eax
&E[i] - E	long	i	movq %rcx,%rax

Arrays

- **Example)**

```
typedef int zip[5];

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



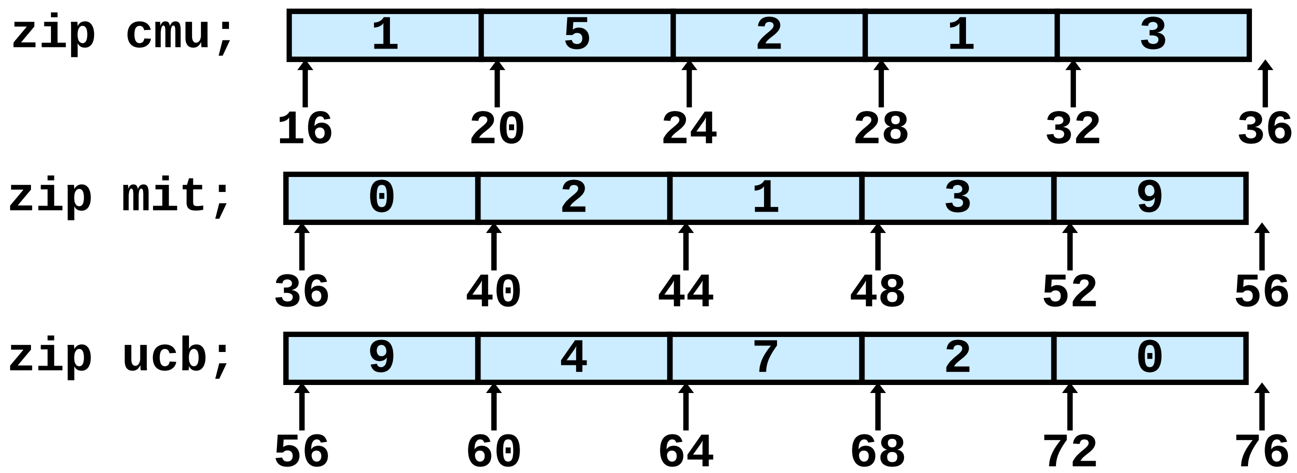
- Notes)
  - Example arrays were allocated in successive 20 byte blocks
    - ✓ Not guaranteed to happen in general

Arrays

Example) (Cont'd)

- No bounds checking in arrays!

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



Arrays

Example) Array & loop

```
typedef long zip[5];
...
long zip2int(zip z){
   int i;
   long zr = 0;
   for (i = 0; i < 5; i++)
       zr = 10 * zr + z[i];
   return zr;
}
```

Original source

$zr = z[0] \cdot 10^4 + z[1] \cdot 10^3 + z[2] \cdot 10^2$
$z[3] \cdot 10^1 + z[4]$

- Array code to pointer code
- Eliminate loop variable i
- Use do-while loop structure

Transformed version

```
long zip2int(zip z){
   long zr = 0;
   long *zend = z + 4;
   do {
       zr = 10 * zr + *z;
       z++;
   } while(z <= zend);
   return zr;
}
```

Arrays

Example2) Array & loop

```
long zip2int(zip z){
   long zr = 0;
   long *zend = z + 4;
   do {
      zr = 10 * zr + *z;
      z++;
   } while(z <= zend);
   return zr;
}
```

[Registers]

%rdi	z
%rax	zr
%rbx	zend

```
    z in %rdi

    xorq %rax,%rax              zr = 0
    leaq 32(%rdi),%rbx          zend = z + 4
.L59:
    leaq (%rax,%rax,4),%rdx     Compute 5*zr
    movq (%rdi),%rax            Access *z
    addq $8,%rdi                Compute z++
    leaq (%rax,%rdx,2),%rax     zr = *z + 2*(5*zr)
    cmpq %rbx,%rdi              Compare z : zend
    jle .L59                    if <=, goto loop
    ret
```

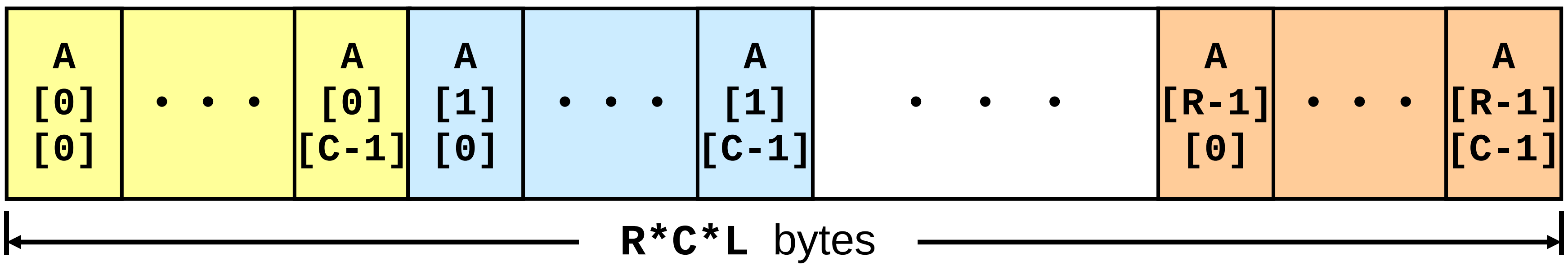
z++ increments by 8

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

Arrays

- **Nested arrays** `T A[R][C];`
  - 2D array of type **T**
    - sizeof(**T**) = **L**
  - **R** rows, **C** columns
  - Array size
    - **R·C·sizeof(T) = R·C·L**
  - Ordering
    - Row-major order

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

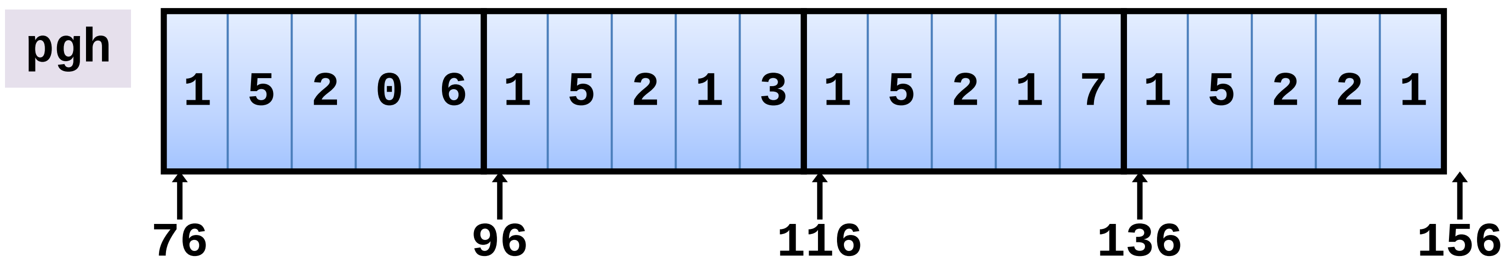


Arrays

- **Nested arrays** `T A[R][C];`
  - Variable A denotes an array of R elements, where each element is an array of C elements
  - Example)

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

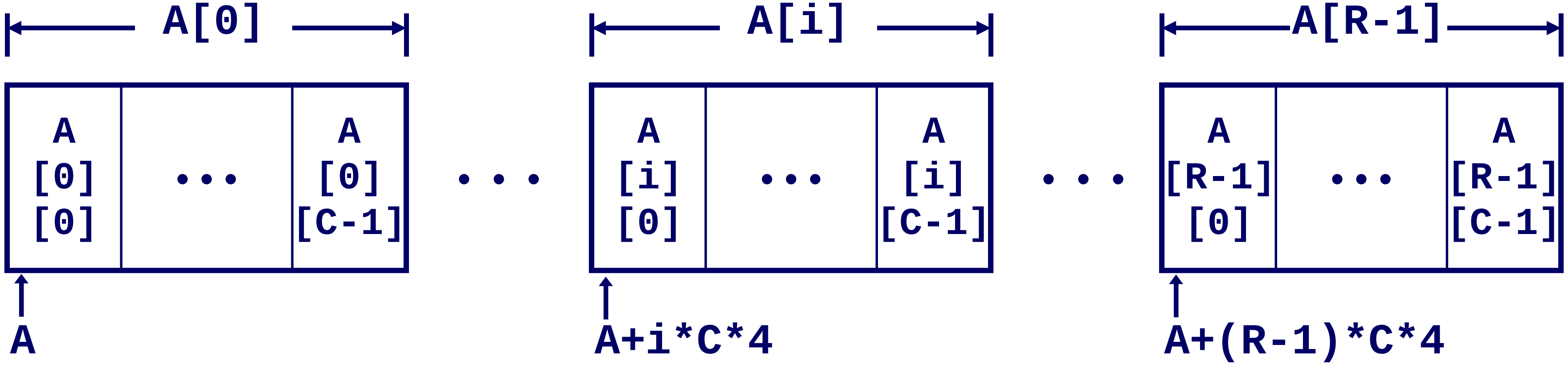
    - Variable **pgh** denotes an array of 4 elements (allocated contiguously)
    - Each element is an array of 5 integers (allocated contiguously)



Arrays

- **Nested arrays: Row vectors** `T A[R][C];`
  - **A[i]** is an array of **C** elements
  - Starting address of **A[i]** at **A + i·(C·L)**

`int A[R][C];`



Arrays

Nested arrays: Row vectors

- Example) `int pgh[4][5];`
  - **pgh[i]** is an array of 5 integers
  - Getting the starting address of **pgh[i]**
    - ✓ **pgh + 20·i**

```
int *get_pgh_zip(int i)
{
     return pgh[i];
}
```

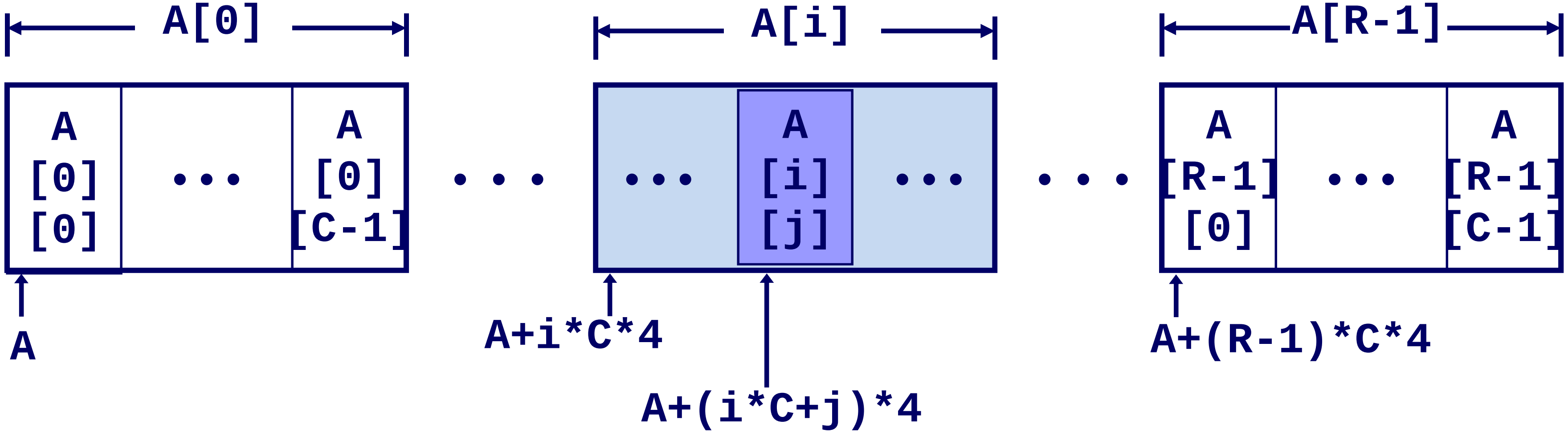
  - Compute as **pgh + 4 * (i + 4 * i)**

```
i in %rdi

leaq (%rdi,%rdi,4),%rax    Compute 5*i
leaq pgh(,%rax,4),%rax     Compute pgh+(20*i)
ret
```

Arrays

- **Nested arrays: Array elements** `T A[R][C];`
  - **A[i][j]** is an element of type **T**
    - Address at **A + i·(C·L) + j·L = A + (i·C + j)·L**

```
int A[R][C];
```



Arrays

Nested arrays: Array elements

- Example) `int pgh[4][5];`
  - **pgh[i][j]** is an integer
  - Getting the address of **pgh[i][j]**
    - ✓ **pgh + 20·i + 4·j**

```
int get_pgh_digit(int i, int j)
{
   return pgh[i][j];
}
```

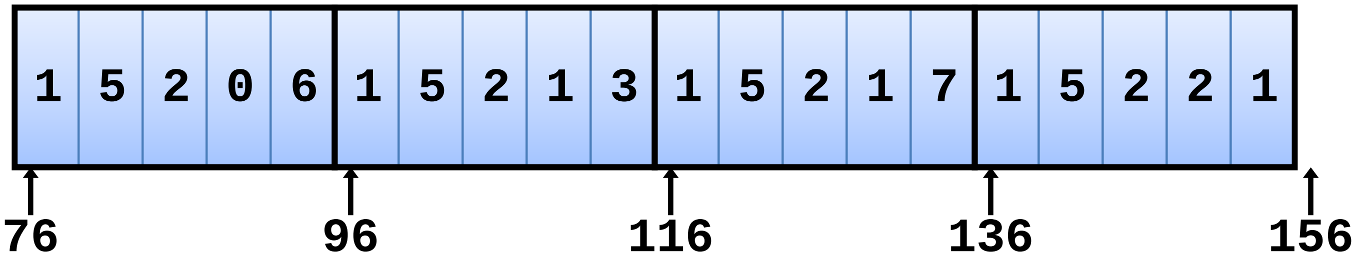
  - Compute as **pgh + 4 * j + 4 * (i + 4 * i)**

```
i in rdi, j in rsi

leaq 0(,%rsi,4),%rdx          Compute 4*j
leaq (%rdi,%rdi,4),%rax       Compute 5*i
movl pgh(%rdx,%rax,4),%eax    Access *(pgh+4*j+20*i)
ret
```

Arrays

- **Nested arrays: Strange referencing examples**

```
int pgh[4][5];
```



Reference	Address	Value	Guaranteed?
pgh[3][3]			
pgh[2][5]			
pgh[2][-1]			
pgh[4][-1]			
pgh[0][19]			
pgh[0][-1]			

Arrays

Fixed-size arrays

- Optimizations by gcc
  - Translation of array expressions into pointer expressions

- Example)
  - Compute element *i,k* of the product of matrices A and B
  - $\sum_{j=0}^{n-1} a_{i,j} \cdot b_{j,k}$

```
#define N 16
typedef int f_mat[N][N];
```

Arrays

- **Fixed-size arrays**
  - Example)

[Original C code]

```
/* Compute element i,k of fixed matrix product */

int fm_pr_el(f_mat A, f_mat B, long i, long k)
{
   long j;
   int res = 0;
   for (j = 0; j < N; j++)
      res += A[i][j]*B[j][k];
   return res;
}
```

Arrays

Fixed-size arrays

- Example)

[Optimized C code]

```
/* Compute element i,k of fixed matrix product */

int fm_pr_el_opt(f_mat A, f_mat B, long i, long k)
{
   int *Aptr = &A[i][0];
   int *Bptr = &B[0][k];
   int *Bend = &B[N][k];
   int res = 0;
   do {
      res += *Aptr * *Bptr;
      Aptr++;
      Bptr += N;
   } while (Bptr != Bend);
   return res;
}
```

Arrays

Fixed-size arrays

- Example)

[Assembly code]

```
int fm_pr_el(f_mat A, f_mat B, long i, long k)
A in %rdi, B in %rsi, i in %rdx, k in %rcx
fm_pr_el:
   salq $6,%rdx                Compute 64*i
   addq %rdx,%rdi              Compute Aptr = A+64·i = &A[i][0]
   leaq (%rsi,%rcx,4),%rcx     Compute Bptr = B+4k = &B[0][k]
   leaq 1024(%rcx),%rsi        Compute Bend = B+4k+1024 = &B[N][k]
   movl $0,%eax                Set res = 0
.L7:                        loop:
   movl (%rdi),%edx            Read *Aptr
   imull (%rcx),%edx           Multiply by *Bptr
   addl %edx,%eax              Add to res
   addq $4,%rdi                Increment Aptr++
   addq $64,%rcx               Increment Bptr += N
   cmpq %rsi,%rcx              Compare Bptr:Bend
   jne .L7                     If !=, goto loop
   rep; ret                    Return
```

Arrays

- **Variable-size arrays**
  - Historically, C only supported multi-dimensional arrays where the sizes could be determined at compile time
    - With the possible exception of the first dimension
  - ISO C99 introduced the capability of having array dimension expressions that are computed as the array is being allocated
  - Declaration of variable-size array
    - Either as a local variable or function argument

```
int A[expr1][expr2];
```

    - The dimensions of the array are determined by evaluating the expressions **`expr1`** and **`expr2`** at the time the declaration is encountered

Arrays

Variable-size arrays

- Example)

```
int vm_el(long n, int A[n][n], long i, long j)
{
   return A[i][j];
}
```

```
int vm_el(long n, int A[n][n], long i, long j)
n in %rdi, A in %rsi, i in %rdx, j in %rcx

vm_el:
   imulq %rdx,%rdi            Compute n*i
   leaq (%rsi,%rdi,4),%rax    Compute A + 4*n*i
   movl (%rax,%rcx,4),%eax    Read from M[A + 4*n*i + 4*j]
   ret                        Return
```

Arrays

Summary

- Arrays in C
  - Contiguous allocation of memory
  - Same type of all elements
  - Array name
    - ✓ Pointer (constant) to the 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

Structures

Structure

- Aggregates multiple objects of different types into a single unit
  - Contiguous allocation of memory
  - Members may be of different types
  - Refer to members by name (in C)
  - Refer to members by offset (in machine code)
- Uses keyword **struct** in C
  - No structure in machine code

Structures

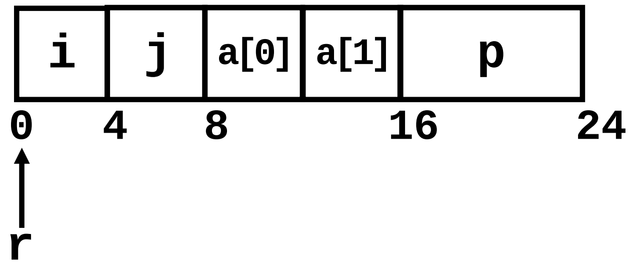
Structure

- Member access
  - By name in C and by offset in assembly/machine code

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};
```

```
void set(struct rec *r)
{
  r->j = r->i;
}
```

Memory Layout



Assembly code

```
  r in %rdi

set:
  movl (%rdi),%eax     Get r->i
  movl %eax,4(%rdi)    Store in r->j
  ret
```

Structures

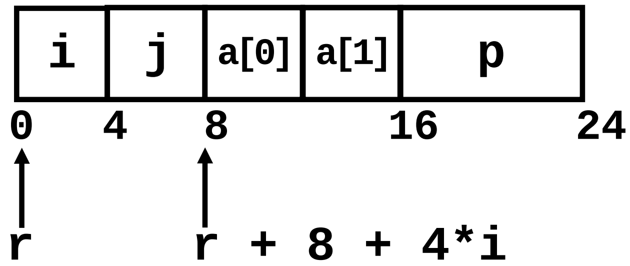
Structure

- Getting pointer to a member
  - Offset of each member determined at compile time

```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};
```

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

Memory Layout



Assembly code

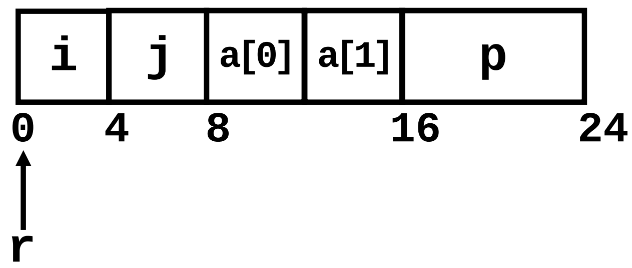
```
   r in %rdi, i in %rsi

get:
   leaq 8(%rdi,%rsi,4),%rax  &(r->a[i])
   ret
```

Structures

Structure

- Getting pointer to a member
  - Offset of each member determined at compile time

Memory Layout




```
struct rec {
  int i;
  int j;
  int a[2];
  int *p;
};
```

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

Assembly code

```
   r in %rdi
put:
   movl 4(%rdi),%eax
   addl (%rdi),%eax
   cltq
   leaq 8(%rdi,%rax,4),%rax
   movq %rax,16(%rdi)
```

Data Alignment

- **Alignment restrictions**
  - Restrictions on the allowable addresses for the primitive data types
    - Primitive data type that requires **K** bytes
      - ✓ Its address must be multiple of **K**
    - Required on many machines
    - Recommended on x86-64

    - Simplify the design of hardware (Interface between the processor and the memory)
    - Improve the memory system performance

Data Alignment

- **Alignment restrictions**
  - 1 byte (char)
    - No restrictions on address
  - 2 bytes (short)
    - Must have address that is a multiple of 2
  - 4 bytes (int, float)
    - Must have address that is a multiple of 4
  - 8 bytes (long, double, char *, …)
    - Must have address that is a multiple of 8

Data Alignment

Compiler directives for alignment

- The compiler places directives in the assembly code indicating the desired alignment for data
- Example)

```
.align 8
```

- Align address to multiple of 8

```
.section .rodata
  .align 8
.L4:
  .quad .L3          Case 100: l_A
  .quad .L8          Case 101: l_def
  .quad .L5          Case 102: l_B
  .quad .L6          Case 103: l_C
  .quad .L7          Case 104: l_D
  .quad .L8          Case 105: l_def
  .quad .L7          Case 106: l_D
```

Data Alignment

Alignments for structures

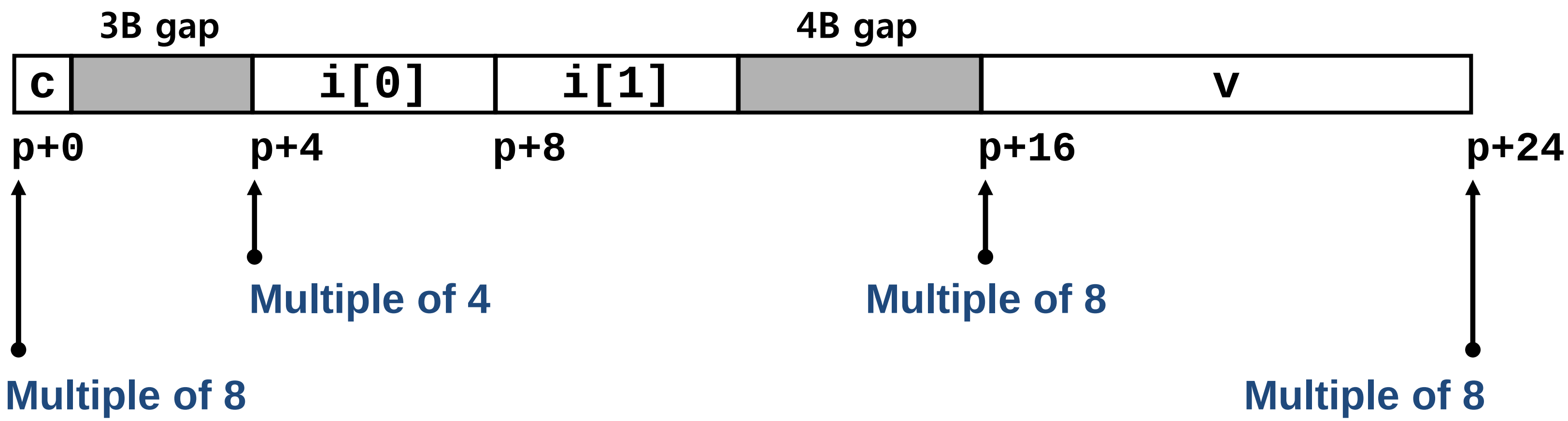
- Member alignments for structures
  - Member offset must satisfy the element's alignment requirement
- Overall structure alignment
  - Each structure has alignment requirement **K**
    - ✓ **K** is the largest alignment requirement of the structure elements
    - ✓ Initial address & structure length must be multiples of **K**

Data Alignment

Alignments for structures: Member alignment

- Example)
  - Gap insertion for member alignment
  - Structure alignment K = 8
    (due to **`double`** elements)

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

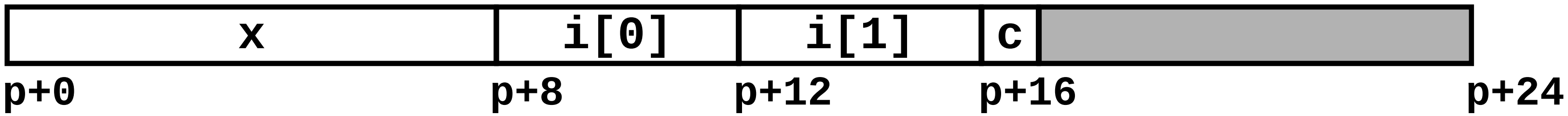


Data Alignment

■ Alignments for structures: Overall alignment

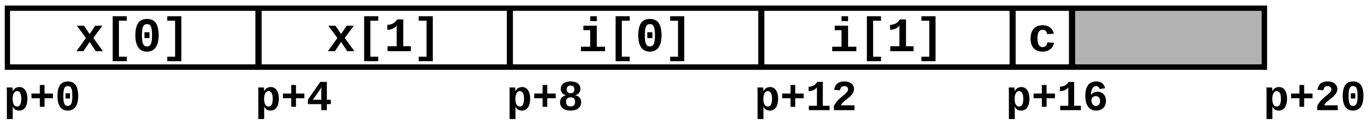
```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

p must be multiple of 8




```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

p must be multiple of 4



Data Alignment

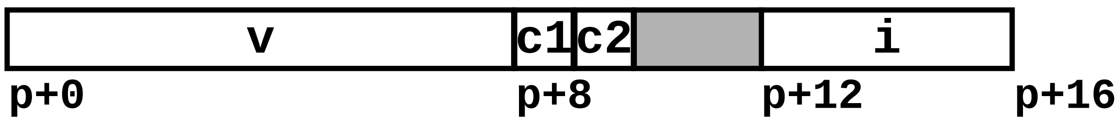
- **Alignments for structures: Ordering elements**

```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space


```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

2 bytes wasted space



Unions

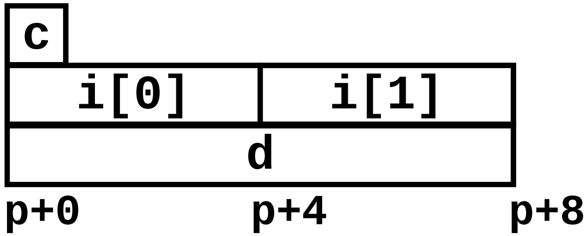
Principles

- Allows an object to be referenced using several different types
  - Overlays union elements
  - Allocates space according to largest elements
  - Usage
    - ✓ Reducing space allocations
    - ✓ Accessing the bit patterns of different data types
- Uses keyword **union** in C

Unions

■ Union example

```
union U1 {
  char c;
  int i[2];
  double d;
} *p;
```



Type	Offset (c)	Offset (i)	Offset (d)	Size
U1	0	0	0	8
S1	0	4	16	24

```
struct S1 {
  char c;
  int i[2];
  double d;
} *q;
```

Unions

Union

- Byte ordering

```
double b2d (unsigned w0, unsigned w1)
{
  union U1 {
    double d;
    unsigned u[2];
  } temp;

  temp.u[0] = w0;
  temp.u[1] = w1;
  return temp.d;
}
```

u[0]	u[1]
d	

- On little-endian machine, (x86-64)
  - **w0** becomes the low-order 4 bytes of **d**
  - **w1** becomes the high-order 4 bytes of **d**
- On big-endian machine,
  - **w0** becomes the high-order 4 bytes of **d**
  - **w1** becomes the low-order 4 bytes of **d**

Structures and Unions

- **Summary**
  - Structure
    - Allocates elements in the order declared
    - Padding in the middle and at end to satisfy alignment
  - Union
    - Overlays elements
    - Way to circumvent type system

Summary

- Array
- Structure
- Union