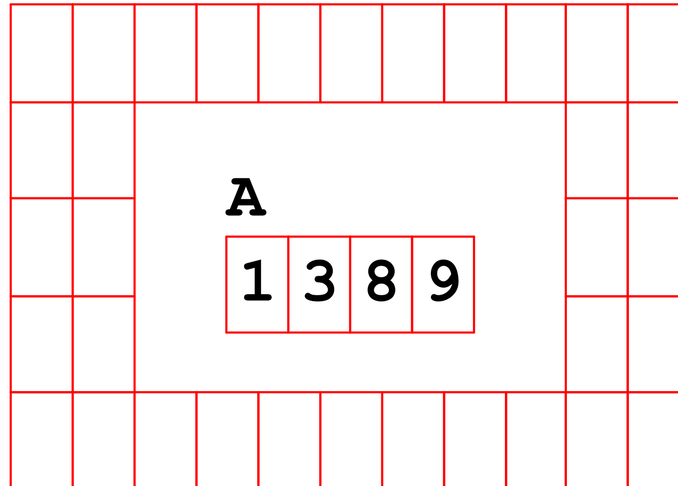
C Arrays

```
int A[4] = {1, 3, 8, 9};
```



C Arrays

```
int A[4] = {1, 3, 8, 9};
```

If `A` is at address `0x400`:

0x400	0x404	0x408	0x40C
1	3	8	9

```
A == (int *)0x400
```

C Arrays

```
int A[4] = {1, 3, 8, 9};
```

If **A** is at address **0x400**:

0x400	0x404	0x408	0x40C
1	3	8	9

```
A[1] == 3
```

C Arrays

```
int A[4] = {1, 3, 8, 9};
```

If **`A`** is at address **`0x400`**:

0x400	0x404	0x408	0x40C
1	3	8	9

```
*(A+1) == 3
```

C Arrays

```
int A[4] = {1, 3, 8, 9};
```

If `A` is at address `0x400`:

0x400	0x404	0x408	0x40C
1	3	8	9

```
A+1 == (int *)0x404
```

C Arrays

```
int A[4] = {1, 3, 8, 9};
```

If **`A`** is at address **`0x400`**:

0x400	0x404	0x408	0x40C
1	3	8	9

```
&A[1] == (int *)0x404
```

C Arrays

```
int A[4] = {1, 3, 8, 9};
```

If **A** is at address **0x400**:

0x400	0x404	0x408	0x40C
1	3	8	9

```
*(int *)0x404 == 3
```

C Arrays

```
int A[4] = {1, 3, 8, 9};
```

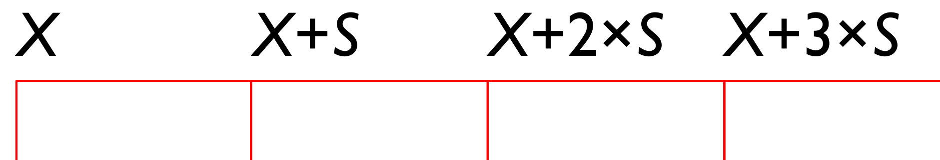
If **A** is at address *X*:

X	*X*+4	*X*+8	*X*+12
1	3	8	9

C Arrays

```
T A[4] = {...};
```

If **`A`** is at address X and S = **`sizeof(T)`**:

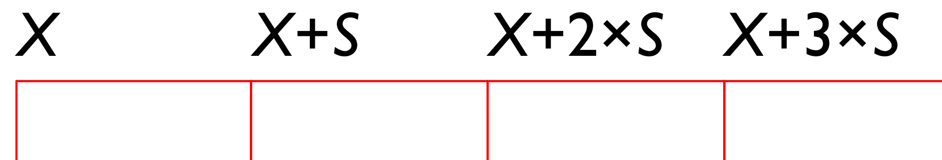


`A == (T *)`X

C Arrays

```
T A[4] = {...};
```

If **`A`** is at address X and S = **`sizeof(T)`**:

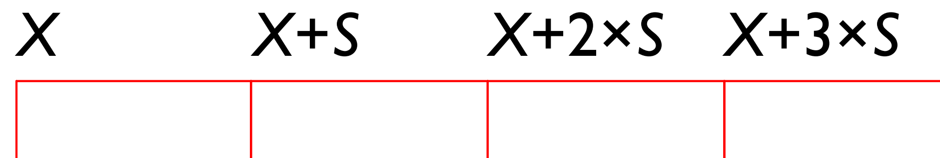


`A+1 == (T *)``(`$X+S$`)`

C Arrays

```
T A[4] = {...};
```

If **`A`** is at address *X* and *S* = **`sizeof(T)`**:

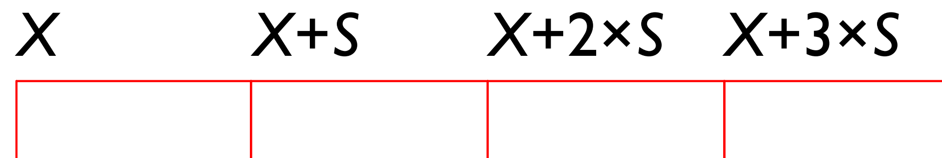


`&A[1] == (T *)`(*X*+*S*)

C Arrays

```
T A[4] = {...};
```

If **`A`** is at address X and S = **`sizeof(T)`**:



`A+`*i* **`== (T *)`**$(X+i*S)$

Pointer Arithmetic

- **OK** to add an integer to a pointer

```
T A[4] = { ... };
T *P1 = A + 1;
T *P2 = A + 3;
```

Pointer Arithmetic

- **OK** to add an integer to a pointer
- **OK** to subtract an integer from a pointer

```
T A[4] = { ... };
T *P1 = A + 3;
T *P2 = P1 - 2;
```

Pointer Arithmetic

- **OK** to add an integer to a pointer
- **OK** to subtract an integer from a pointer
- **OK** to subtract a pointer from a pointer

```
T A[4] = { ... };
T *P1 = A + 1;
T *P2 = A + 3;
int pdelta = P2 - P1; // == 2
int padelta = P2 - A; // == 3
```

Pointer Arithmetic

- **OK** to add an integer to a pointer
- **OK** to subtract an integer from a pointer
- **OK** to subtract a pointer from a pointer
- **NOT OK** to add a pointer to a pointer

```
T A[4] = { ... };
T *P1 = A + 1;
T *P2 = A + 3;
... P1 + P2 ...
```

Arrays versus Pointers

Given

```
int A[4];
```

or

```
int *A = malloc(sizeof(int) * 4);
```

then **A** mostly behaves the same either way

```
A[0]
```

Arrays versus Pointers

Given

```
int A[4];
```

or

```
int *A = malloc(sizeof(int) * 4);
```

then **`A`** mostly behaves the same either way

```
A[1] = 3
```

Arrays versus Pointers

Given

```
int A[4];
```

or

```
int *A = malloc(sizeof(int) * 4);
```

then **A** mostly behaves the same either way

```
int *P = A+2
```

Arrays versus Pointers

Given

```
int A[4];
```

or

```
int *A = malloc(sizeof(int) * 4);
```

then **A** mostly behaves the same either way

```
int *P = &A[2]
```

Arrays versus Pointers

Given

```
int A[4];
```

or

```
int *A = malloc(sizeof(int) * 4);
```

then **`A`** mostly behaves the same either way

```
f(A)
```

Arrays versus Pointers

Given

```
int A[4];
```

or

```
int *A = malloc(sizeof(int) * 4);
```

then **`A`** mostly behaves the same either way

Differences:

`&A`

`int A[4]`: **`&A`** is the same **`int*`** as **`A`**
`int *A`: **`&A`** is a **`int**`**

Arrays versus Pointers

Given

```
int A[4];
```

or

```
int *A = malloc(sizeof(int) * 4);
```

then **`A`** mostly behaves the same either way

Differences:

`&A`

`sizeof(A)`

`int A[4]`: **`sizeof(A)`** is **`4*sizeof(int)`**
`int *A`: **`sizeof(A)`** is **`sizeof(int*)`**

Arrays as Arguments

```
void f(int v) {
   v = 2;
}

void call_f() {
   int n = 1;
   f(n);
   printf("%d\n", n);
}
```

`call_f()` prints the original value **1**, not **2**

Arrays as Arguments

```
void f(int *v) {
   v[0] = 2;
}

void call_f() {
   int *p = malloc(sizeof(int));
   *p = 1;
   f(p);
   printf("%d\n", *p);
}
```

`call_f()` prints 2, not the original value **1**

Arrays as Arguments

```
void f(int v[1]) {
   v[0] = 2;
}

void call_f() {
   int a[1] = { 1 };
   f(a);
   printf("%d\n", a[0]);
}
```

`call_f()` prints 2, not the original value **`1`**

... because the value of **`v`** is an address

Arrays as Arguments

```
void f(int v[1]) { .... }
```

is the same as

```
void f(int v[100]) { .... }
```

is the same as

```
void f(int v[]) { .... }
```

is the same as

```
void f(int *v) { .... }
```

Arrays as Arguments

```
#include <stdio.h>

void f(int v[100]) {
  printf("%ld\n", sizeof(v));
}

int main() {
  int *p = NULL;
  f(p);
  return 0;
}
```

Copy

Prints same result as `sizeof(int*)`

Type Definitions in C

```
#include <stdio.h>

typedef int number;


number twice(number v) {
  return v + v;
}

void set_twice(number *p) {
  *p = twice(*p);
}

int main() {
  number v = 1;
  set_twice(&v);
  printf("%d\n", v);
  return 0;
}
```

Copy

Type Definitions in C

```
#include <stdio.h>

typedef int number;
typedef int *number_ptr;

number twice(number v) {
  return v + v;
}

void set_twice(number_ptr p) {
  *p = twice(*p);
}

int main() {
  number v = 1;
  set_twice(&v);
  printf("%d\n", v);
  return 0;
}
```

Copy

Type Definitions in C

A type definition looks like a variable definition, but prefixed with **`typedef`**

```
int v;
```

defines **`v`** to hold an **`int`** value

```
typedef int T;
```

defines **`T`** as an alias for **`int`**

```
int *p;
```

defines **`p`** to hold an **`int`** pointer

```
typedef int *T;
```

defines **`T`** as an alias for **`int*`**

```
char *s;
```

defines **`s`** to hold a string

```
typedef char *String;
```

defines **`String`** as an alias for **`char*`**

Type Definitions in C

A type definition looks like a variable definition, but prefixed with **`typedef`**

```
int **p;
```

defines **`p`** to hold an **`int*`** pointer

```
typedef int **T;
```

defines **`T`** as an alias for **`int**`**

```
int a[5];
```

defines **`a`** to hold 5 **`int`**s

```
typedef int zip_t[5];
```

defines **`zip_t`** to mean 5 **`int`**s

```
zip_t uofu = { 8, 4, 1, 1, 2 };

void mail_to(zip_t z);
```

Array Access in Machine Code

```
typedef int zip_t[5];

int get_digit(zip_t z, int digit) {
  return z[digit];
}
```

```
movl (%rdi,%rsi,4), %eax
```

```
%rdi = z
%rsi = digit
```

```
zip_t uofu = { 8, 4, 1, 1, 2 };
get_digit(uofu, 2);
```

z	z+4	z+8	z+12	z+16
8	4	1	1	2

Array Access in Machine Code

```
void zincr(zip_t z) {
  size_t i;
  for (i = 0; i < 5; i++)
    z[i]++;
}
```

```
                                  #   %rdi = z
   movl    $0, %eax               #   i = 0
   jmp     .L3                    #   goto middle
.L4:                              # loop:
   addl    $1, (%rdi,%rax,4)      #   z[i]++
   addq    $1, %rax               #   i++
.L3:                              # middle:
   cmpq    $4, %rax               #   compare i to 4
   jbe     .L4                    #   if <=, goto loop
   rep ret                        #   return
```

`gcc -Og`

Array Access in Machine Code

```
void zincr(zip_t z) {
  size_t i;
  for (i = 0; i < 5; i++)
    z[i]++;
}
```

```
                                  #   %rdi = z
   xorl     %eax, %eax            #   i = 0
.L3:                              # loop:
   addl     $1, (%rdi,%rax,4)     #   z[i]++
   addq     $1, %rax              #   i++
   cmpq     $5, %rax              #   compare i to 5
   jne      .L3                   #   if !=, goto loop
   rep ret                        #   return
```

gcc -O2

Arrays and Typedefs

```
typedef int number;
number a[3];
```

An array of 3 **`ints`**:

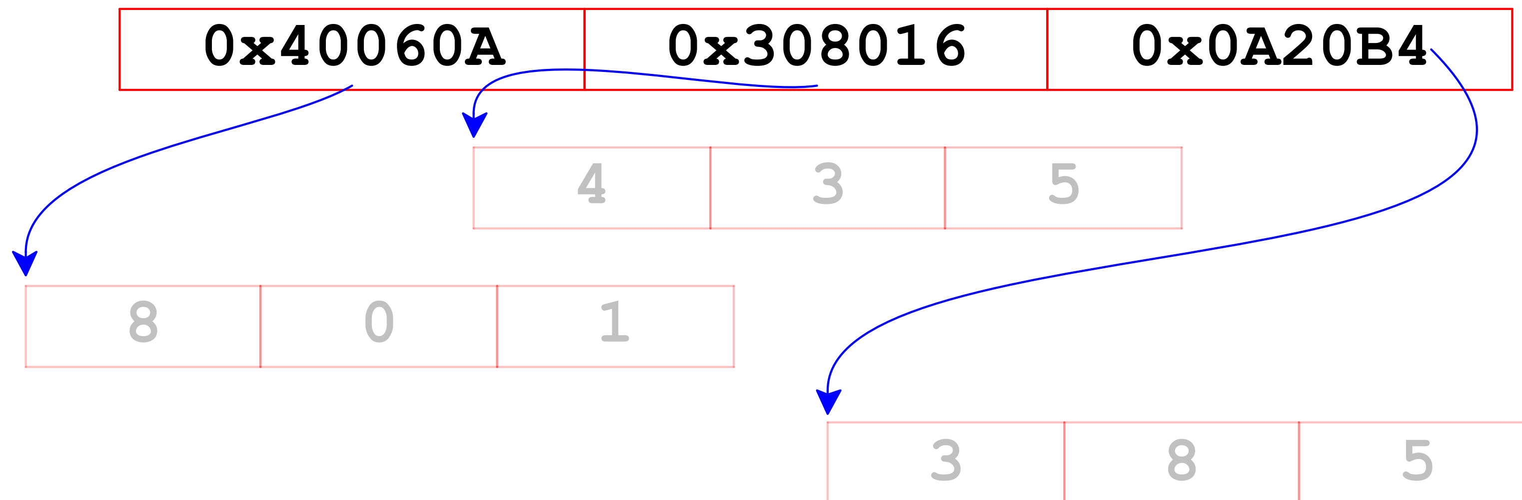
8	0	1

Total size of **`a`**: **`3*sizeof(int) = 12`**

Arrays and Typedefs

```
typedef int *number_ptr;
number_ptr a[3];
```

An array of 3 pointers:



Total size of **a**: **3*sizeof(int*) = 24**

Arrays and Typedefs

```
typedef int area_t[3];
area_t a[3];
```

An array of 3 arrays:

Arrays and Typedefs

```
typedef int area_t[3];
area_t a[3];
```

An array of 3 arrays:

8	0	1	4	3	5	3	8	5

Total size: `3*sizeof(area_t)` =
`3*(3*sizeof(int))` = 36

Arrays and Typedefs

```
typedef int area_t[3];
area_t a[2];
```

An array of 2 arrays:

8	0	1

4	3	5

Arrays and Typedefs

```
typedef int area_t[3];
area_t a[2];
```

An array of 2 arrays:

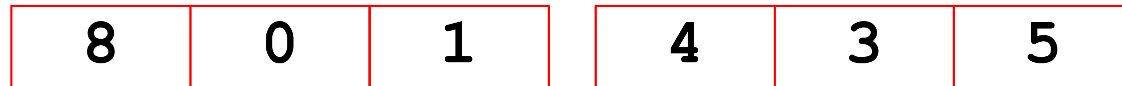
8	0	1	4	3	5

Total size: `2*sizeof(area_t)` = `2*(3*sizeof(int))` = **24**

Arrays and Typedefs

```
int a[2][3];
```

An array of 2 arrays:



Arrays and Typedefs

```
int a[2][3];
```

An array of 2 arrays:

8	0	1	4	3	5

Total size: `2*3*sizeof(int_t) = 24`

Arrays and Typedefs

```
int a[2][3] = { {8, 0, 1},
                {4, 3, 5} };
```

An array of 2 arrays:

8	0	1

4	3	5

Arrays and Typedefs

```
int a[2][3] = { {8, 0, 1},
                {4, 3, 5} };
```

An array of 2 arrays:

8	0	1	4	3	5

```
a[1][0] == 4
```

because

```
a[1] == {4, 3, 5}
```

Arrays and Typedefs

```
int a[2][3] = { {8, 0, 1},
                {4, 3, 5} };
```

An array of 2 arrays:

8	0	1	4	3	5

```
int *p = a[1]
```

OK

Arrays and Typedefs

```
int a[2][3] = { {8, 0, 1},
                {4, 3, 5} };
```

An array of 2 arrays:

8	0	1	4	3	5

```
int **p = a;
```

NOT OK

Arrays and Typedefs

```
int a[2][3] = { {8, 0, 1},
                {4, 3, 5} };
```

An array of 2 arrays:

8	0	1	4	3	5

```
int **p = (int **)a;
```

NOT OK

Arrays and Typedefs

```
int a[2][3] = { {8, 0, 1},
                {4, 3, 5} };
```

An array of 2 arrays:

8	0	1	4	3	5

```
int *p = (int *)a;
```

sortof OK...

Two-Dimensional Array Layout

```
int A[M][N];
```

Memory:

A[0][0]	A[0][1]	A[0][2]	...	A[0][N-1]
A[1][0]	A[1][1]	A[1][2]	...	A[1][N-1]
A[2][0]	A[2][1]	A[2][2]	...	A[2][N-1]
⋮	⋮	⋮	⋮	⋮
A[M-1][0]	A[M-1][1]	A[M-1][2]	...	A[M-1][N-1]

(N columns, M rows)

Memory, linear view:

$M \times N$

A[0][0]	A[0][1]	A[0][2]	...	A[0][N-1]	A[1][0]	.

This is ***row-major order***

Two-Dimensional Array Layout

```
int A[M][N];
```

Memory:

A[0][0]	A[0][1]	A[0][2]	...	A[0][N-1]
A[1][0]	A[1][1]	A[1][2]	...	A[1][N-1]
A[2][0]	A[2][1]	A[2][2]	...	A[2][N-1]
⋮	⋮	⋮	⋮	⋮
A[M-1][0]	A[M-1][1]	A[M-1][2]	...	A[M-1][N-1]

(width: N; height: M)

Memory, linear view:

A[0][0]	A[0][1]	A[0][2]	...	A[0][N-1]	A[1][0]	...

(length: $M \times N$)

```
A[i][j] = 17;
```

```
int *pA = (int *)A;
pA[i*N + j] = 17;
```

Two-Dimensional Array Access

```
#define N 16
typedef int matrix16_t[N][N];

int product_at(matrix16_t A, matrix16_t B, int i, int k) {
  int j, result;

  for (j = 0, result = 0; j < N; j++)
    result += A[i][j] * B[j][k];

  return result;
}
```

Copy

Two-Dimensional Array Access

```
#define N 16
typedef int matrix16_t[N][N];

int product_at(matrix16_t A, matrix16_t B, int i, int k) {
  int j, result;

  for (j = 0, result = 0; j < N; j++)
    result += A[i][j] * B[j][k];

  return result;
}
```

Copy

```
.L3:
   movl     (%rdi), %esi
   addq     $64, %rcx
   addq     $4, %rdi
   imull    -64(%rcx), %esi
   addl     %esi, %eax
   cmpq     %rdx, %rcx
   jne      .L3
   rep ret
```

gcc -O2

Two-Dimensional Array Access

```
int product_at(matrix16_t A, matrix16_t B, int i, int k) {
  int *Aptr, *Bptr, cnt, result;
  Aptr = &A[i][0];
  Bptr = &B[0][k];
  cnt = N-1;
  result = 0;

  do {
    result += (*Aptr) * (*Bptr);
    Aptr++;
    Bptr += N;
    cnt--;
  } while(cnt >= 0);

  return result;
}
```

```
.L3:
   movl    (%rdi), %esi
   addq    $64, %rcx
   addq    $4, %rdi
   imull   -64(%rcx), %esi
   addl    %esi, %eax
   cmpq    %rdx, %rcx
   jne     .L3
   rep ret
```

gcc -O2

Unspecified Row Count: Ok

```
int f(int A[][N])
```

Memory:

A[0][0]	A[0][1]	A[0][2]	...	A[0][N−1]
A[1][0]	A[1][1]	A[1][2]	...	A[1][N−1]
A[2][0]	A[2][1]	A[2][2]	...	A[2][N−1]
⋮	⋮	⋮	⋮	⋮

N (row width); ? (row count)

```
A[i][j] = 17;
```

```
int *pA = (int *)A;
pA[i*N + j] = 17;
```

Unspecified Column Count: Not Ok

```
int f(int A[M][])
```

Memory:

A[0][0]	A[0][1]	A[0][2]	...
A[1][0]	A[1][1]	A[1][2]	...
A[2][0]	A[2][1]	A[2][2]	...
⋮	⋮	⋮	⋮
A[M-1][0]	A[M-1][1]	A[M-1][2]	...

(rows: M; row width: ?)

```
A[i][j] = 17;
```

```
int *pA = (int *)A;
pA[i*? + j] = 17;
```

Exercise: Determining Array Dimensions

```
#define M ??
#define N ??

int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j) {
  return mat1[i][j] + mat2[j][i];
}
```

```
leaq 0(,%rsi,4),%rax
leaq 0(,%rdi,8),%rdx
subq %rdi,%rdx
addq %rax,%rsi
salq $2,%rsi
movl mat2(%rsi,%rdi,4),%eax
addl mat1(%rax,%rdx,4),%eax
```

Exercise: Determining Array Dimensions

```
#define M ??
#define N ??

int mat1[M][N], pmat1 = (int*)mat1;
int mat2[N][M], pmat2 = (int*)mat2;

int sum_element(int i, int j) {
  return mat1[i][j] + mat2[j][i];
}
```

```
mat1[i][j]
pmat1[i*N+j]
pmat1 @ (i*N+j)*4
N = 7
pmat1 @ (i*7+j)*4
pmat1 @ (i7+j)*4
pmat1 @ i7*4+j4
```

```
                                %rdi = i   %rsi = j
leaq 0(,%rsi,4),%rax            j4 = 4*j
leaq 0(,%rdi,8),%rdx            i8 = i*8
subq %rdi,%rdx                  i7 = i8-i
addq %rax,%rsi                  j5 = j4+j
salq $2,%rsi                    j20 = 4*j5
movl mat2(%rsi,%rdi,4),%eax     pmat2 @ j20+4*i
addl mat1(%rax,%rdx,4),%eax     pmat1 @ i7*4+j4
```

Exercise: Determining Array Dimensions

```
#define M ??
#define N ??

int mat1[M][N], pmat1 = (int*)mat1;
int mat2[N][M], pmat2 = (int*)mat2;

int sum_element(int i, int j) {
  return mat1[i][j] + mat2[j][i];
}
```

```
mat2[j][i]
pmat2[j*M+i]
pmat2 @ (j*M+i)*4
M = 5
pmat2 @ (j*5+i)*4
pmat2 @ j20+4*i
```

```
                                 %rdi = i   %rsi = j
leaq 0(,%rsi,4),%rax             j4 = 4*j
leaq 0(,%rdi,8),%rdx             i8 = i*8
subq %rdi,%rdx                   i7 = i8-i
addq %rax,%rsi                   j5 = j4+j
salq $2,%rsi                     j20 = 4*j5
movl mat2(%rsi,%rdi,4),%eax      pmat2 @ j20+4*i
addl mat1(%rax,%rdx,4),%eax      pmat1 @ i7*4+j4
```