600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

**Much of the code in these examples is not commented because it would otherwise not fit on the slides. This is bad coding practice in general and you should not follow my lead on this.*

Outline

- The `const` keyword
- Memory layout
- Arrays
- Miscellanea

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

```
#include <stdio.h>
int main( void )
{
    const int i= 1;
    printf( "%d\n" , i );
    return 0;
}
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

```
#include <stdio.h>
int main( void )
{
    const int i= 1;
    printf( "%d\n" , i++ );
```

```
>> gcc -std=c99 -pedantic -Wall -Wextra foo.c
foo.c: In function main:
foo.c:5:21: error: increment of read-only variable i
   printf( "%d\n" , i++ );
                    ^~
>>
```

The `const` keyword

- Using the `const` keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a `const` variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
  - **const int * ap;**
    "**ap** is a pointer to a constant **int**"

```
#include <stdio.h>
int main( void )
{
    int a =1 , b = 2;
    const int * ap = &a;
    ap = &b;
    return 0;
}
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
  - **const int * ap;**
    "**ap** is a pointer to a constant **int**"

```
#include <stdio.h>
int main( void )
{
    int a =1 , b = 2;
    const int * ap = &a;
    *ap = b;
    return 0;
}
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
  - **const int * ap;**
    "**ap** is a pointer to a constant **int**"

```
#include <stdio.h>
int main( void )
{
    int a =1 , b = 2;
    const int * ap = &a;
```

```
>> gcc -std=c99 -pedantic -Wall -Wextra foo.c
foo.c: In function main:
foo.c:6:7: error: assignment of read-only variable ap
 *ap = b;
       ^
>>
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
  - **const int * ap;**
    "**ap** is a pointer to a constant **int**"
  - **int * const ap;**
    "**ap** is a constant pointer to an **int**"

```
#include <stdio.h>
int main( void )
{
    int a =1 , b = 2;
    int * const ap = &a;
    *ap = b;
    return 0;
}
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
  - **const int * ap;**
    "**ap** is a pointer to a constant **int**"
  - **int * const ap;**
    "**ap** is a constant pointer to an **int**"

```
#include <stdio.h>
int main( void )
{
    int a =1 , b = 2;
    int * const ap = &a;
    ap = &b;
    return 0;
}
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
  - const int * ap;
    "ap is a pointer to a constant int"
  - int * const ap;
    "ap is a constant pointer to an int"

```
#include <stdio.h>
int main( void )
{
    int a =1 , b = 2;
    int * const ap = &a;
```

```
>> gcc -std=c99 -pedantic -Wall -Wextra foo.c
foo.c: In function main:
foo.c:6:6: error: assignment of read-only variable ap
   ap = &b;
      ^
>>
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
- A pointer to a non-constant variable can be either constant or not

```
#include <stdio.h>
int main( void )
{
    int a =1;
    const int * ap1 = &a;
    int * ap2 = &a;
    return 0;
}
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration we read from right to left
- A pointer to a non-constant variable can be either constant or not
- A pointer to a constant variable should point to a constant value

```
#include <stdio.h>
int main( void )
{
    const int a =1;
    const int * ap1 = &a;
    int * ap2 = &a;
    return 0;
}
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- When parsing a variable declaration
- A
  can b
- A poi
  shoul

```
#include <stdio.h>
int main( void )
{
    const int a =1;
    const int * ap1 = &a;
```

Note that this a warning, not an error.
The code will still compile and run.

```
>> gcc -std=c99 -pedantic -Wall -Wextra foo.c
foo.c: In function main:
foo.c:7:15: warning: initialization discards const qualifier from pointer target type ...
   int * ap2 = &a;
               ^
>>
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- We can almost always use an explicit cast to remove the **const** qualifier

```
#include <stdio.h>
int main( void )
{
    const int a =1;
    printf( "%d\n" , a );
    int * ap = (int*)&a;
    (*ap)++;
    printf( "%d\n" , a );
    return 0;
}
```

```
>> ./a.out
1
2
>>
```

The **const** keyword

- Using the **const** keyword in a declaration indicates that the item being declared, once initialized, should be "read-only"
  - Attempts to change the value of a **const** variable are caught by the compiler
  - Knowing that a value won't change allows the compiler to optimize the code

Note:

- We can almost always use an explicit cast to remove the **const** qualifier
  - If the variable is both **const** and **static**, it is allocated in a special part of memory where values are not allowed to change

```
#include <stdio.h>
int main( void )
{
    static const char c_hello[] = "hello";
    char* hello = (char*)c_hello;
    printf( "%s\n" , c_hello );
    hello[4] = 0;
    printf( "hi\
    return 0;
}
```

```
>> ./a.out
hello
Segmentation fault (core dump)
>>
```

Outline

- The `const` keyword
- Memory layout
- Arrays
- Miscellanea

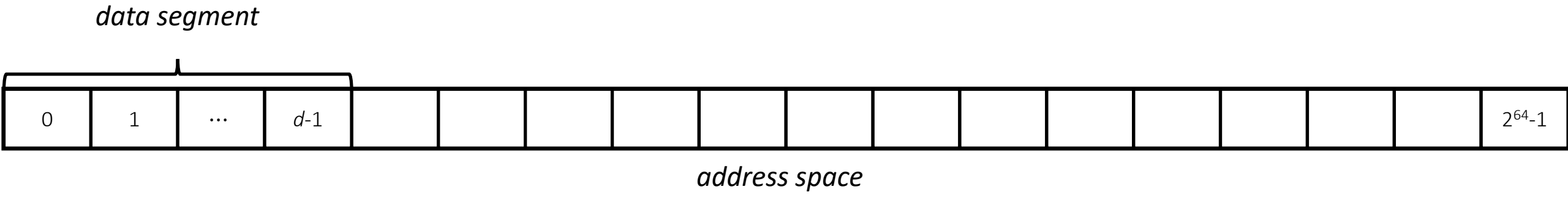
Memory layout

- On 64-bit machines, the operating system (OS) pretends like each program has access to 2^{64} bytes of a memory
  - A pointer points to an address in the range $[0,2^{64}-1]$

address space

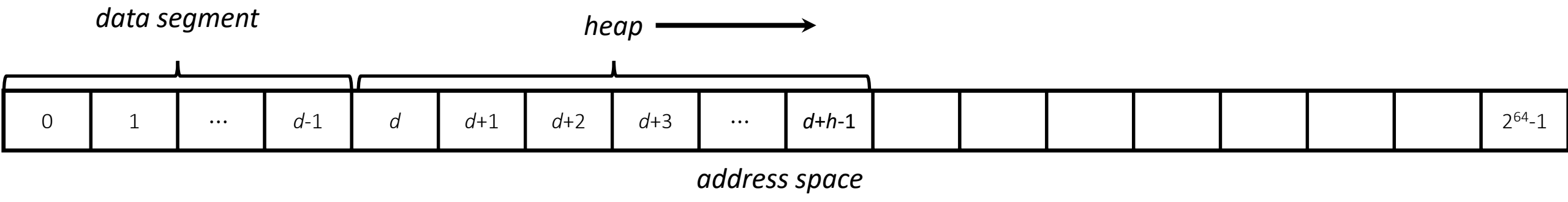
Memory layout

- On 64-bit machines, the operating system (OS) pretends like each program has access to 2^{64} bytes of a memory
  - A pointer points to an address in the range $[0,2^{64}-1]$
    - The (fixed size) data segment resides at the low end of the address space



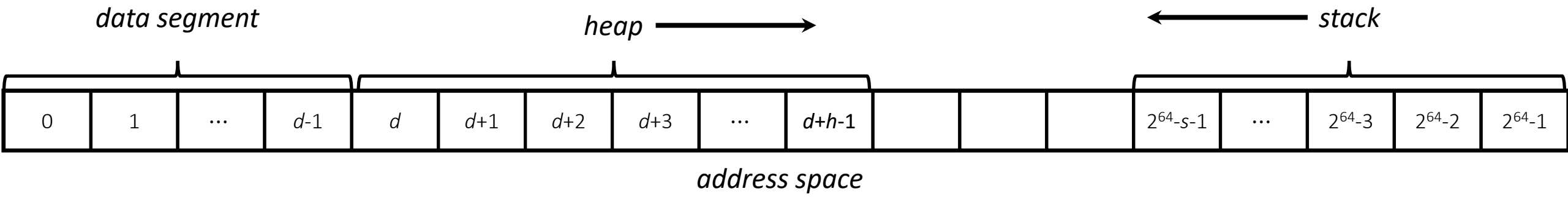
Memory layout

- On 64-bit machines, the operating system (OS) pretends like each program has access to 2^{64} bytes of a memory
  - A pointer points to an address in the range $[0,2^{64}-1]$
    - The (fixed size) data segment resides at the low end of the address space
    - The heap starts after the data segment and grows up



Memory layout

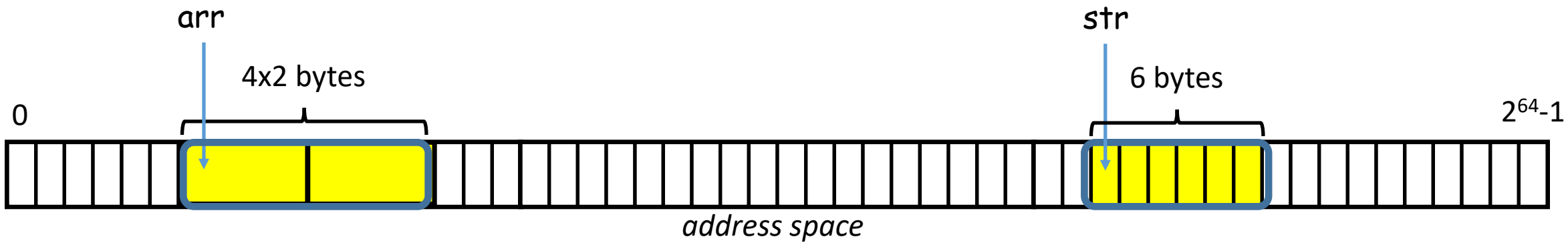
- On 64-bit machines, the operating system (OS) pretends like each program has access to 2^{64} bytes of a memory
  - A pointer points to an address in the range $[0,2^{64}-1]$
    - The (fixed size) data segment resides at the low end of the address space
    - The heap starts after the data segment and grows up
    - The stack starts at the high end of the address space and grows down



Memory layout

- When we allocate an array of size **n**, the OS finds a contiguous block of **n** bytes and returns a pointer to the location of the first byte

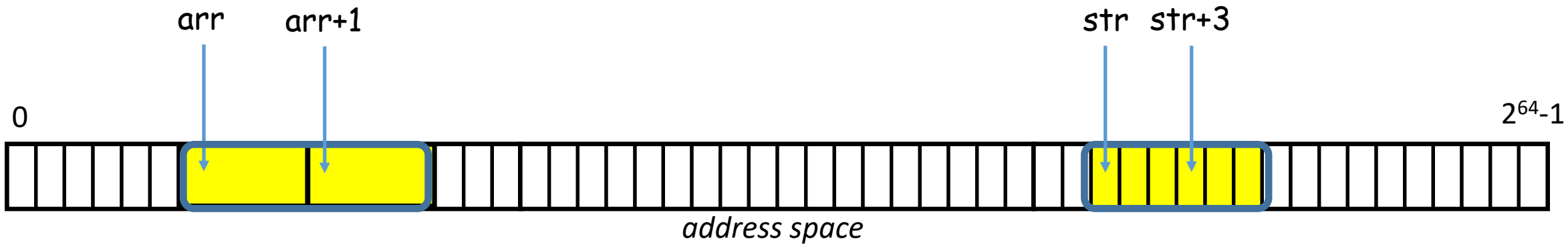
```
#include <stdio.h>
int main( void )
{
    char* str = malloc( 6 );
    int* arr = malloc( sizeof(int)*2 );
    return 0;
}
```



Memory layout

- When we allocate an array of size n, the OS finds a contiguous block of n bytes and returns a pointer to the location of the first byte
  - Adding an integer k to a pointer is the same as stepping k * sizeof(type) bytes forward in memory

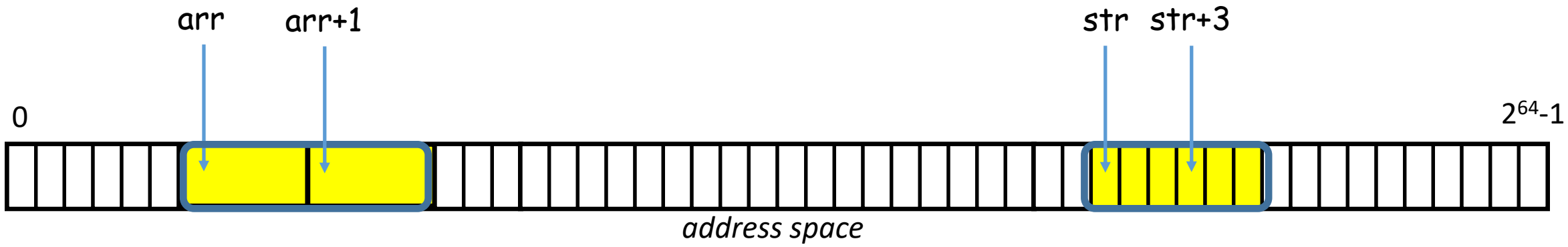
```
#include <stdio.h>
int main( void )
{
    char* str = malloc( 6 );
    int* arr = malloc( sizeof(int)*2 );
    char * str3 = str+3;
    int * arr1 = arr+1;
    return 0;
}
```



Memory layout

```
#include <stdio.h>
int main( void )
{
    char* str = malloc( 6 );
    int* arr = malloc( sizeof(int)*2 );
    str[3] = '.';      // or *(str+3) = '.';
    arr[1] = 0;        // or *(arr+1) = 0;
    return 0;
}
```

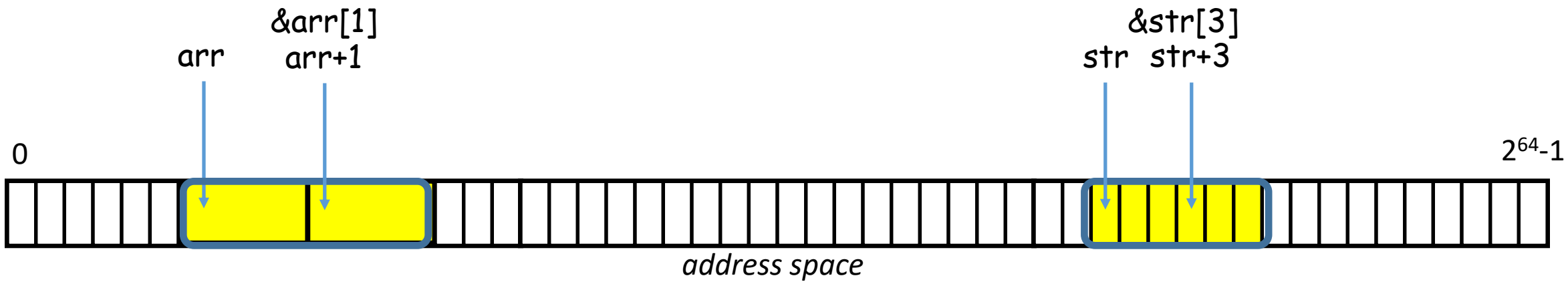
- When we allocate an array of size **n**, the OS finds a contiguous block of **n** bytes and returns a pointer to the location of the first byte
  - Adding an integer **k** to a pointer is the same as stepping **k * sizeof(type)** bytes forward in memory
  - Accessing at index **k** is the same as adding **k** and dereferencing the pointer



Memory layout

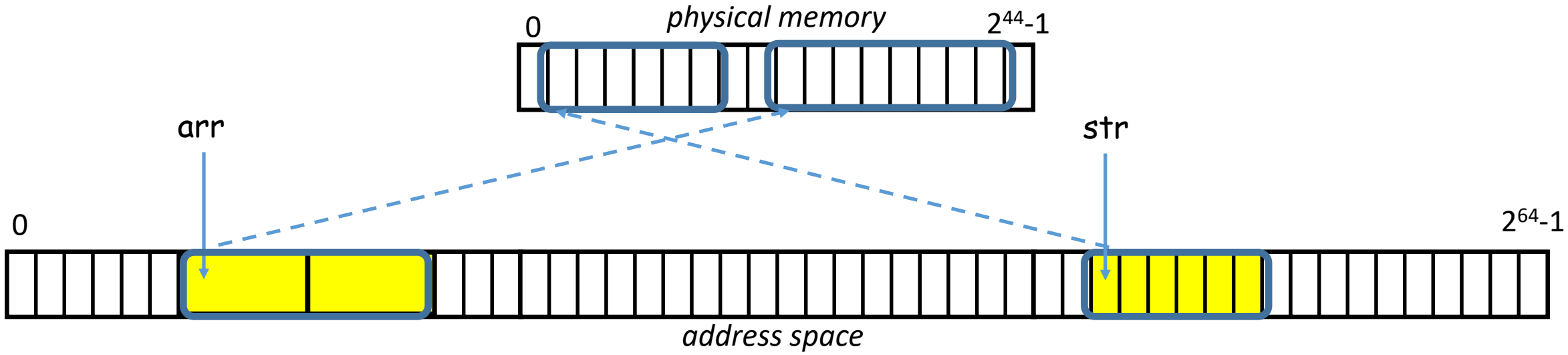
```
#include <stdio.h>
int main( void )
{
    char* str = malloc( 6 );
    int* arr = malloc( sizeof(int)*2 );
    char * str3 = &str[3];
    int * arr1 = &arr[1];
    return 0;
}
```

- When we allocate an array of size n, the OS finds a contiguous block of n bytes and returns a pointer to the location of the first byte
  - Adding an integer k to a pointer is the same as stepping k * sizeof(type) bytes forward in memory
  - Accessing at index k is the same as adding k and dereferencing the pointer
  - Accessing at index k and taking the address is the same as adding k to the pointer



Memory layout

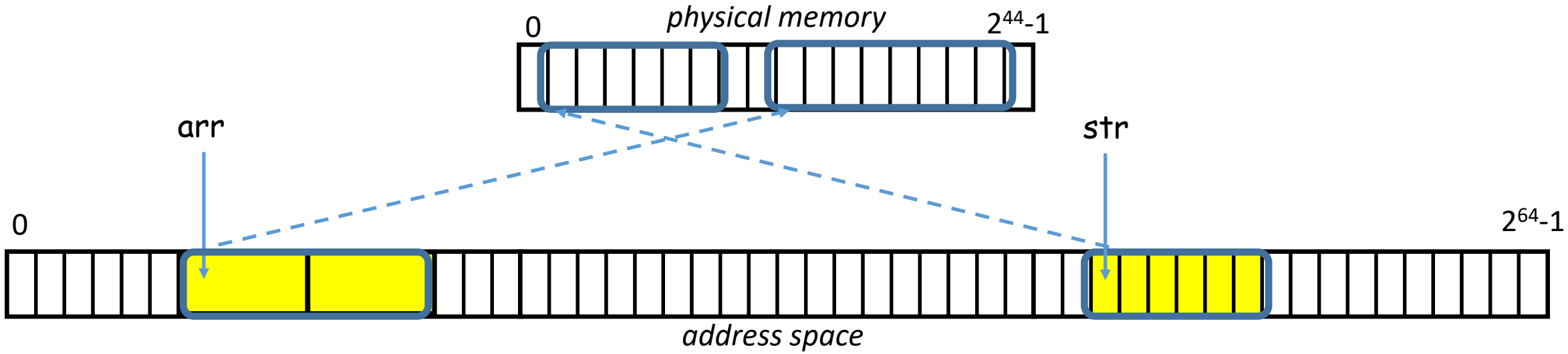
- On 64-bit machines, the operating system (OS) pretends like each program has access to 2^{64} bytes of a memory
  - The value of the pointers is the *virtual address*
  - The OS maps the virtual address to a physical location (e.g in RAM)



Memory layout

- On 64-bit machines, the operating system (OS) pretends like each program has access to 2^{64} bytes of a memory
  - The value of the pointers is the *virtual address*
  - The OS maps the virtual address to a physical location (e.g in RAM)

  ⇒ Two programs running at the same time on the same machine can have pointers with the same (virtual) addresses



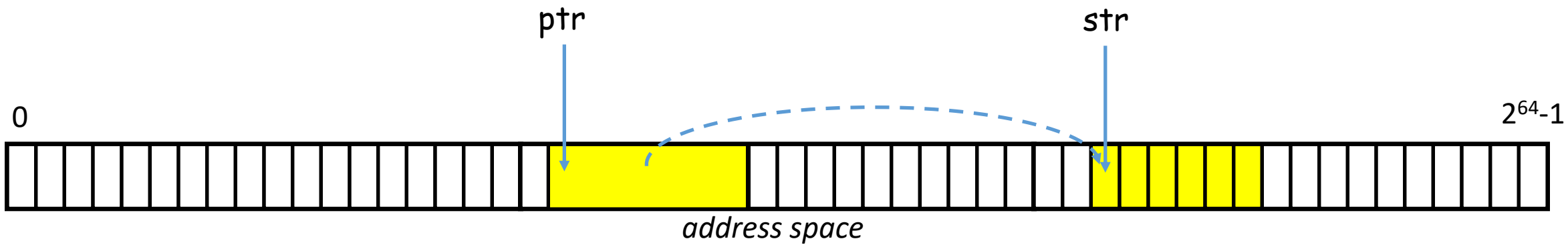
Outline

- The `const` keyword
- Memory layout
- Arrays
- Miscellanea

Arrays

- When we declare an array on the stack, we get a block of memory
- When we declare a pointer, we get something that stores an address

```
#include <stdio.h>
int main( void )
{
    char str[] = "hello";
    char * ptr = str;
    return 0;
}
```

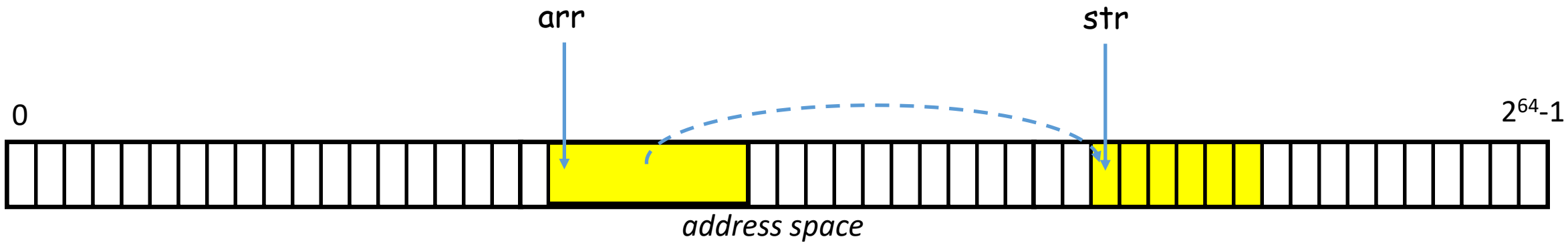


Arrays

- When we declare an array on the stack, we get a block of memory
- When we declare a pointer, we get something that stores an address
  - These are not the same
    - They refer to data of different sizes
    - What a pointer points to can change
    - A pointer doesn't have to point to anything

```
#include <stdio.h>
int main( void )
{
    char str[] = "hello";
    char * ptr = str;
    printf( "%d\n" , sizeof(str) );
    printf( "%d\n" , sizeof( ptr ) );
    return 0;
}
```

```
>>
./a.out
6
8
>>
```

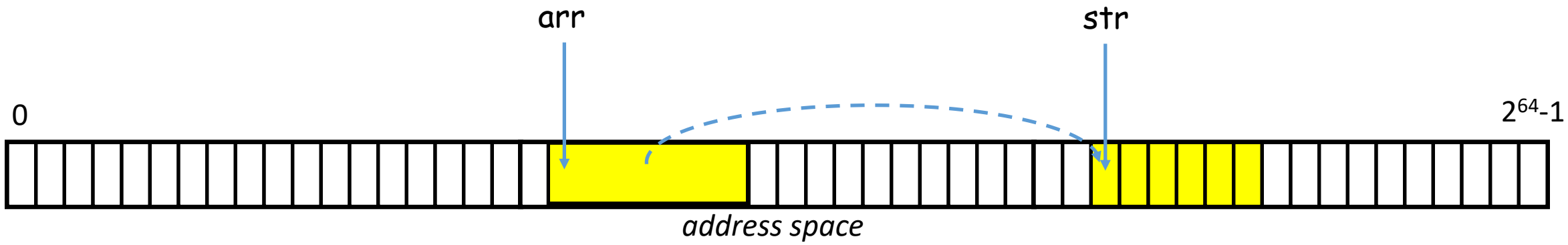


Arrays

- When we declare an array on the stack, we get a block of memory
- When we declare a pointer, we get something that stores an address
  - These are not the same
  - However, they are equivalent when it comes to pointer arithmetic and indexing

```
#include <stdio.h>
int main( void )
{
    char str[] = "hello";
    char * ptr = str;
    printf( "%c %c\n" , str[0] , *(str+1) );
    printf( "%c %c\n" , ptr[0] , *(ptr+1) );
    return 0;
}
```

```
>>
./a.out
h e
h e
>>
```

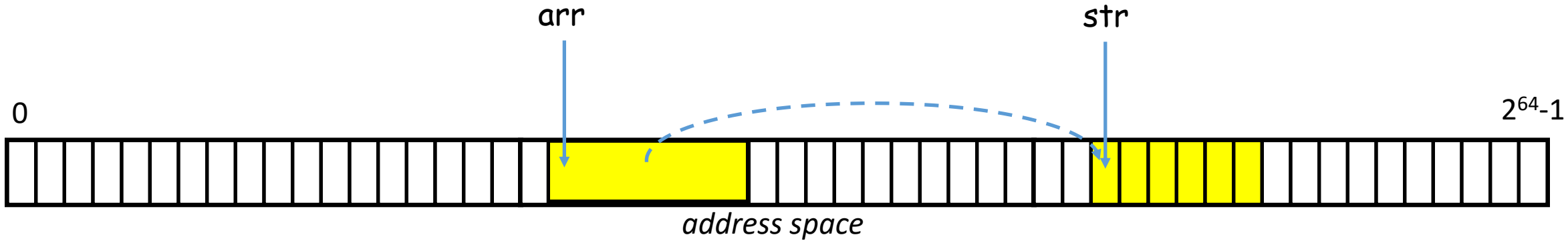


Arrays

- When we declare an array on the stack, we get a block of memory
- When we declare a pointer, we get something that stores an address
  - These are not the same
  - However, they are equivalent when it comes to pointer arithmetic and indexing
  - And, when we pass an array as an argument, it is turned into a pointer that points to the first element of the array

```
#include <stdio.h>
void foo( char* ptr )
{
    printf( "%d\n" , sizeof( ptr ) );
}
int main( void )
{
    char str[] = "hello";
    char * ptr = str;
    foo( str );
    foo( ptr );
    return 0;
}
```

```
>>
./a.out
8
8
>>
```



2-dimensional arrays

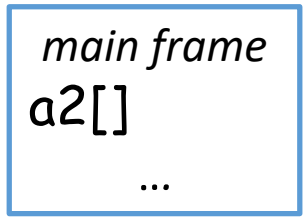
Q: How do we declare a 4x5 grid of values?

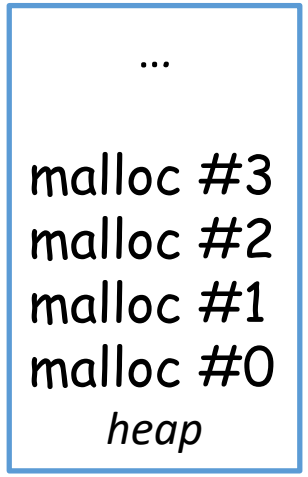
2-dimensional arrays

Q: How do we declare a 4x5 grid of values?

A: We can declare an array of pointers and separately allocate each row

- ✓ Accessing grid entries is clean
- ✗ The rows are on the heap and will need to be deallocated
- ✗ Accessing an entry requires following two pointers
- ✗ Memory is not contiguous





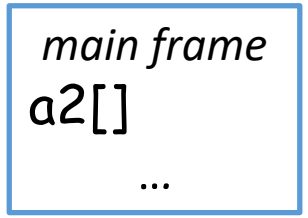

```
#include <stdio.h>
int main( void )
{
    char* a2[4];
    for( int i=0 ; i<4 ; i++ )
        a2[i] = malloc( 5 );
    for( int i=0 ; i<4 ; i++ )
        for( int j=0 ; j<5 ; j++ )
            a2[i][j] = i+j;
    return 0;
}
```

2-dimensional arrays

Q: How do we declare a 4x5 grid of values?

A: We can declare a single array of integers and use row-major indexing

- ✓The entire grid is on the stack
- ✓Memory is contiguous
- ✓Accessing is more efficient
- ✗But indexing is ugly



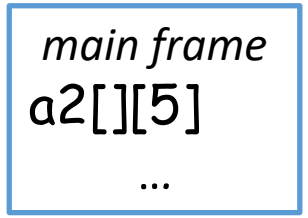

```
#include <stdio.h>
int main( void )
{
    char a2[4*5];
    for( int i=0 ; i<4 ; i++ )
        for( int j=0 ; j<5 ; j++ )
            a2[5*i+j] = i+j;
    return 0;
}
```

2-dimensional arrays

Q: How do we declare a 4x5 grid of values?

A: We can declare as a 2D array and let the compiler handle indexing

- ✓The entire grid is on the stack
- ✓Memory is contiguous
- ✓Accessing is more efficient
- ✓And it's clean

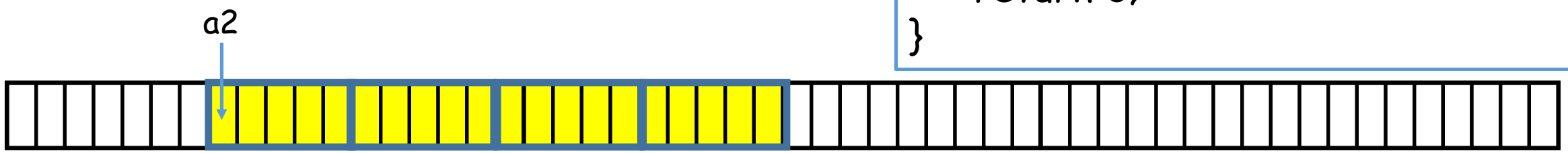



```
#include <stdio.h>
int main( void )
{
    char a2[4][5];
    for( int i=0 ; i<4 ; i++ )
        for( int j=0 ; j<5 ; j++ )
            a2[i][j] = i+j;
    return 0;
}
```

2-dimensional arrays

- The compiler allocates a single block of memory with 4x5 elements
  - As with 1D arrays, a2 can be used as the address of the first element

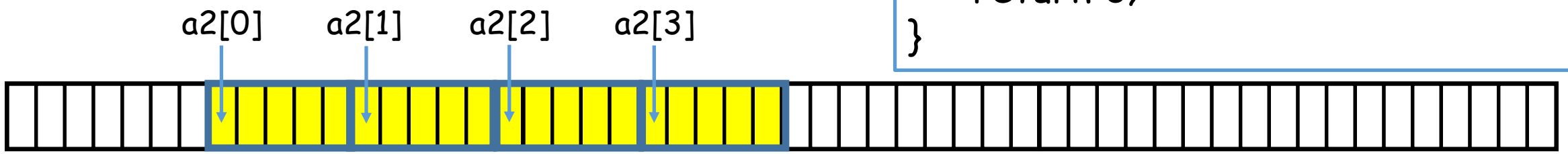
```
#include <stdio.h>
int main( void )
{
    char a2[4][5];
    for( int i=0 ; i<4 ; i++ )
        for( int j=0 ; j<5 ; j++ )
            a2[i][j] = i+j;
    return 0;
}
```



2-dimensional arrays

- The compiler allocates a single block of memory with 4x5 elements
  - As with 1D arrays, a2 can be used as the address of the first element
  - Furthermore a2[i] can be used as the address of the i-th sub-block

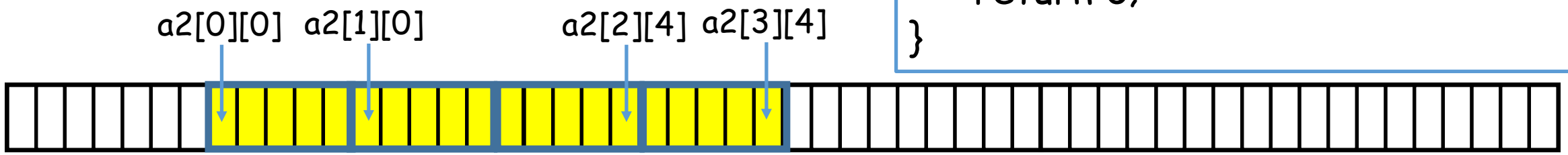
```
#include <stdio.h>
int main( void )
{
    char a2[4][5];
    for( int i=0 ; i<4 ; i++ )
        for( int j=0 ; j<5 ; j++ )
            a2[i][j] = i+j;
    return 0;
}
```



2-dimensional arrays

- The compiler allocates a single block of memory with 4x5 elements
  - As with 1D arrays, a2 can be used as the address of the first element
  - Furthermore a2[i] can be used as the address of the i-th sub-block
  - When we access at index [i][j], the compiler automatically turns this into the 1D index i*5+j
    - Needs the size of the second dimension (5)
    - Don't need the size of the first (4)

```
#include <stdio.h>
int main( void )
{
    char a2[4][5];
    for( int i=0 ; i<4 ; i++ )
        for( int j=0 ; j<5 ; j++ )
            a2[i][j] = i+j;
    return 0;
}
```

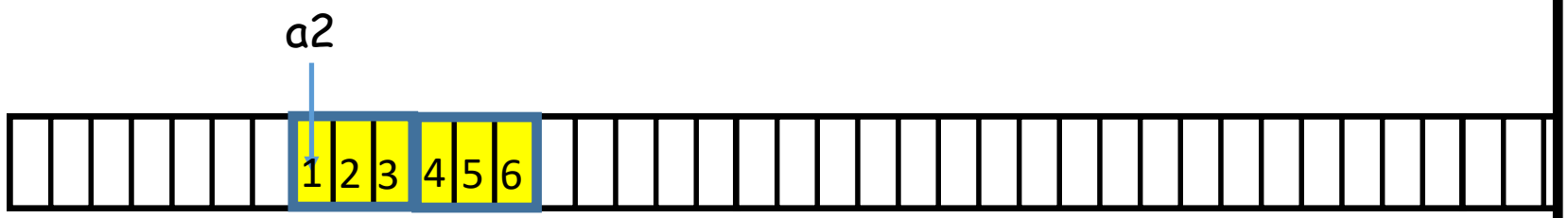


2-dimensional arrays

- We can declare and assign using nested braces (i.e. arrays of arrays)
  - The values are in the order that they appear in memory
  - We need to specify the second dimensions

```
#include <stdio.h>
int main( void )
{
    char a2[][3] = { { 1 , 2 , 3 } , { 4 , 5 , 6 } };
    char * a1 = a2;
    for( int i=0 ; i<2 ; i++ ) for( int j=0 ; j<3 ; j++ )
        printf( "%d %d\n" , a2[i][j] , a1[i*3+j] );
    return 0;
}
```

```
>> ./a.out
1 1
2 2
3 3
4 4
5 5
6 6
>>
```

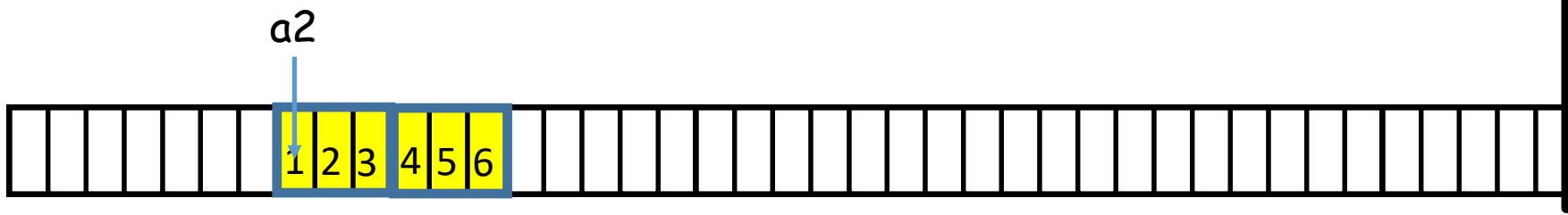


2-dimensional arrays

- When passing a 2D array as a function argument:
  - You can pass it as a pointer
    - This is the address of a2[0][0]

```
#include <stdio.h>
void foo( char * a1 , int sz )
{
    for( int i=0 ; i<sz ; i++ ) printf( "%d\n" , a1[i] );
}
int main( void )
{
    char a2[][3] = { { 1 , 2 , 3 } , { 4 , 5 , 6 } };
    foo( (char*)a2 , 6 );
    return 0;
}
```

```
>> ./a.out
1
2
3
4
5
6
>>
```

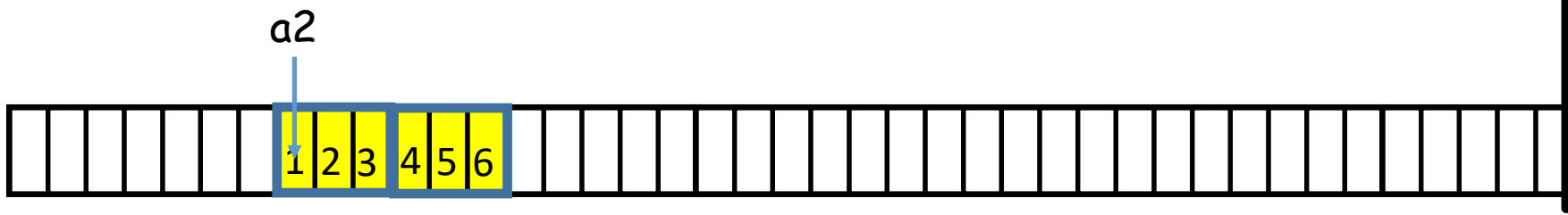


2-dimensional arrays

- When passing a 2D array as a function argument:
  - You can pass it as a pointer
  - Or as a 2D array
    - We only pass the dimension of the first index, **sz**, since the second is already provided

```
#include <stdio.h>
void foo( char a2[][3] , int sz )
{
    for( int i=0 ; i<sz ; i++ ) for( int j=0 ; j<3 ; j++ )
        printf( "%d\n" , a2[i][j] );
}
int main( void )
{
    char a2[][3] = { { 1 , 2 , 3 } , { 4 , 5 , 6 } };
    foo( a2 , 2 );
    return 0;
}
```

```
>> ./a.out
1
2
3
4
5
6
>>
```

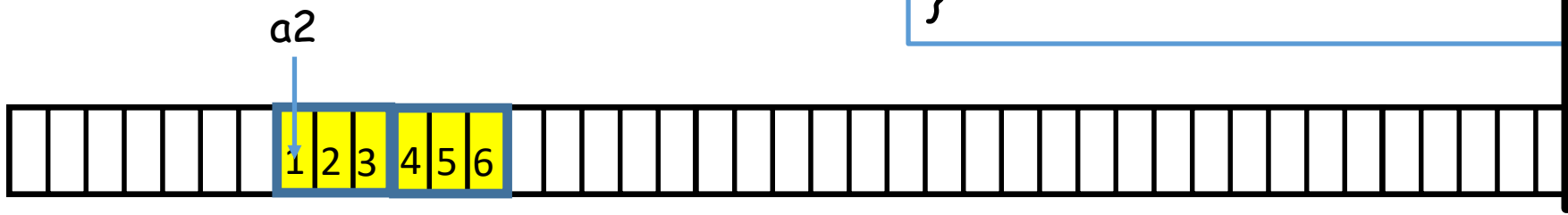


2-dimensional arrays

- When passing a 2D array as a function argument:
  - You can pass it as a pointer
  - Or as a 2D array
  - Or slice by slice as 1D arrays

```
#include <stdio.h>
void foo( char a1[] , int sz )
{
    for( int i=0 ; i<sz ; i++ )
        printf( "%d\n" , a1[i] );
}
int main( void )
{
    char a2[][3] = { { 1 , 2 , 3 } , { 4 , 5 , 6 } };
    for( int i=0 ; i<2 ; i++ )
        foo( a2[i] , 3 );
    return 0;
}
```

```
>> ./a.out
1
2
3
4
5
6
>>
```

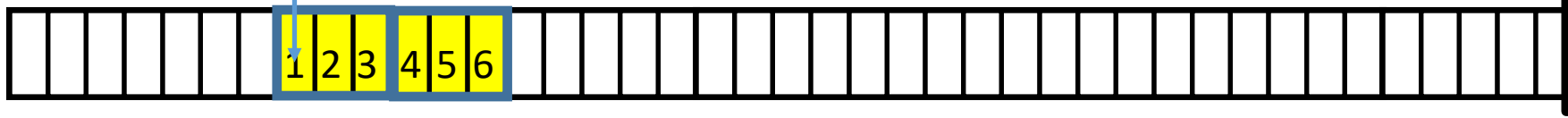


2-dimensional arrays

- When passing a 2D array as a function argument:
  - You can pass it as a pointer
  - Or as a 2D array
  - Or slice by slice as 1D arrays

```
#include <stdio.h>
void foo( char a1[] , int sz )
{
    for( int i=0 ; i<sz ; i++ )
        printf( "%d\n" , a1[i] );
}
int main( void )
{
    char a2[][3] = { { 1 , 2 , 3 } , { 4 , 5 , 6 } };
    for( int i=0 ; i<2 ; i++ )
        foo( a2[i] , 3 );
```

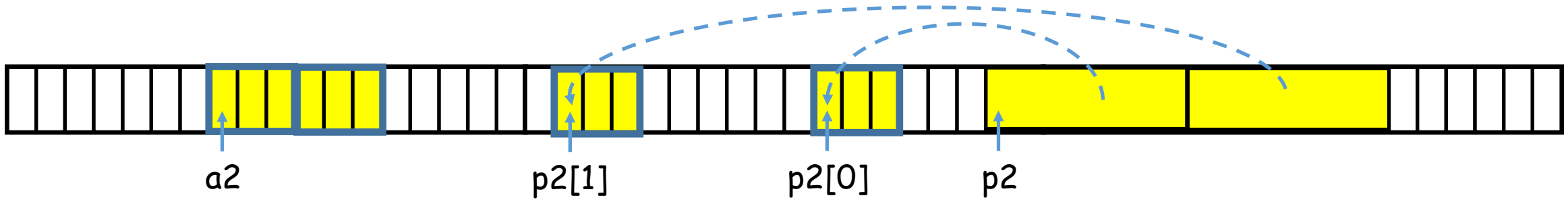
Whichever way you do it, `foo` is passed a pointer to memory in `main`'s stack frame and can change the values.




```
>> ./a.out
1
2
3
4
5
6
>>
```

2D arrays vs. pointers to pointers

```
#include <stdio.h>
int main( void )
{
    char a2[2][3];
    char ** p2 = malloc( sizeof( char* )*2 );
    for( int i=0 ; i<2 ; i++ ) p2[i] = malloc( 3 );
    ...
    for( int i=0 ; i<2 ; i++ ) free( p2[i] );
    free( p2 );
    return 0;
}
```

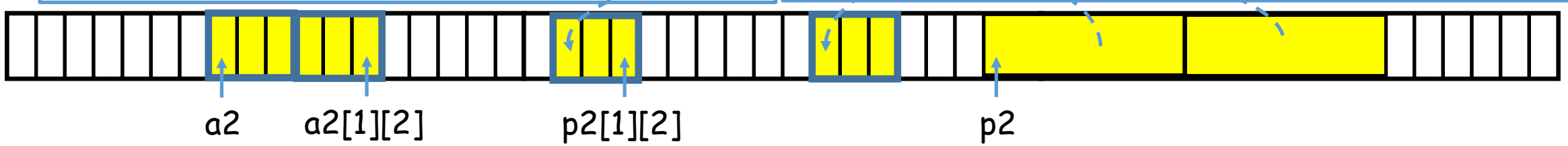


2D arrays vs. pointers to pointers

- Similar:
  - Entries accessed via double indices

```
{
    …
    a2[1][2] = 0;
    p2[1][2] = 0;
    …
}
```

```
#include <stdio.h>
int main( void )
{
    char a2[2][3];
    char ** p2 = malloc( sizeof( char* )*2 );
    for( int i=0 ; i<2 ; i++ )
        p2[i] = malloc( 3 );
    …
    return 0;
}
```

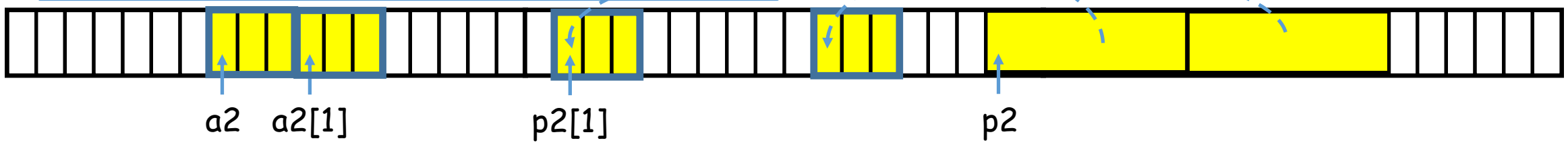


2D arrays vs. pointers to pointers

- Similar:
  - Entries accessed via double indices
  - 1D slices accessed via single indices

```
void foo( char * ip );
…
{
    …
    foo( a2[1] );
    foo( p2[1] );
    …
}
```

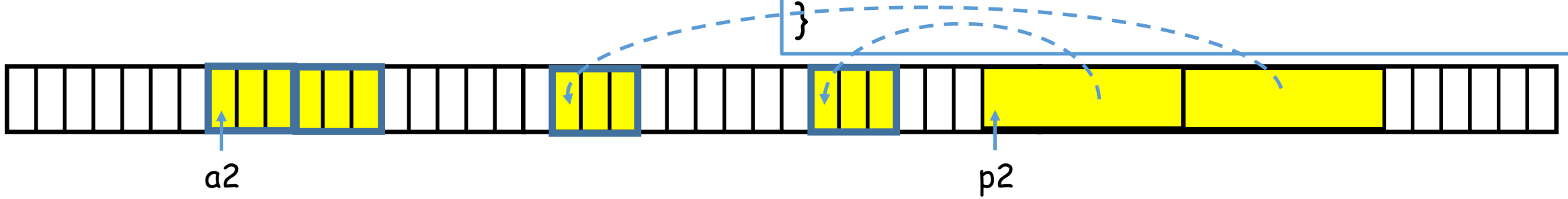
```
#include <stdio.h>
int main( void )
{
    char a2[2][3];
    char ** p2 = malloc( sizeof( char* )*2 );
    for( int i=0 ; i<2 ; i++ )
        p2[i] = malloc( 3 );
    …
    return 0;
}
```



2D arrays vs. pointers of pointers

- Similar:
  - Entries accessed via double indices
  - 1D slices accessed via single indices
- Different:
  - 2D Arrays are contiguous

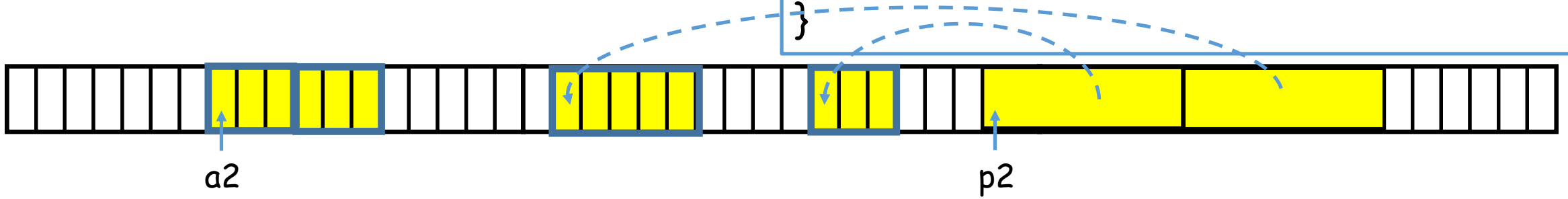
```
#include <stdio.h>
int main( void )
{
    char a2[2][3];
    char ** p2 = malloc( sizeof( char* )*2 );
    for( int i=0 ; i<2 ; i++ )
        p2[i] = malloc( 3 );
    …
    return 0;
}
```



2D arrays vs. pointers of pointers

- Similar:
  - Entries accessed via double indices
  - 1D slices accessed via single indices
- Different:
  - 2D Arrays are contiguous
  - Pointers of pointers don't have to have the same number of elements in each slice

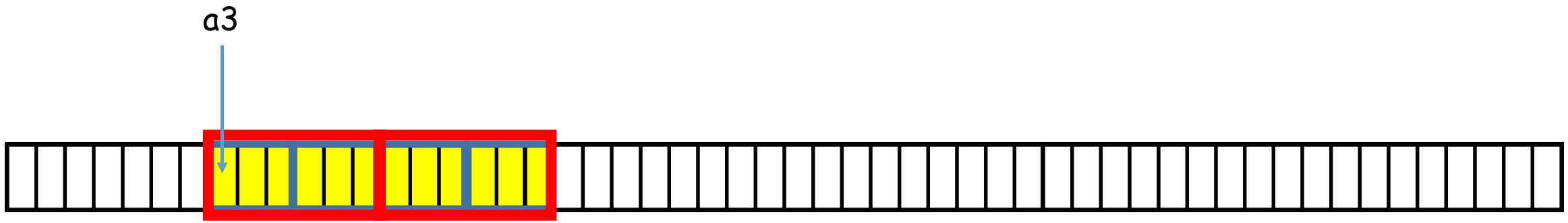
```
#include <stdio.h>
int main( void )
{
    char a2[2][3];
    char ** p2 = malloc( sizeof( char* )*2 );
    for( int i=0 ; i<2 ; i++ )
        p2[i] = malloc( 3 );
    …
    return 0;
}
```



Multi-dimensional arrays

- The compiler allocates a single block of memory with 3x4x5 elements
  - The variable points to the first element in the block

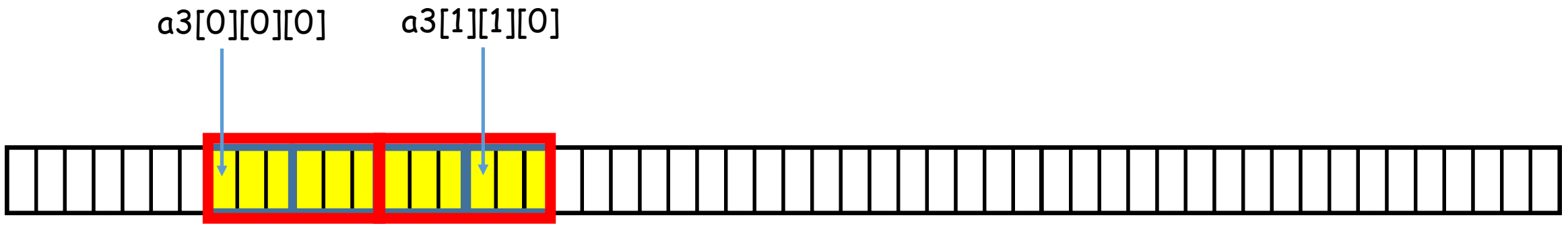
```
#include <stdio.h>
int main( void )
{
    char a3[2][2][3];



    return 0;
}
```



Multi-dimensional arrays

- The compiler allocates a single block of memory with 3x4x5 elements
  - The variable points to the first element in the block
  - The 3D index [i][j][k] corresponds to the 1D index i*2*3 + j*3 +k

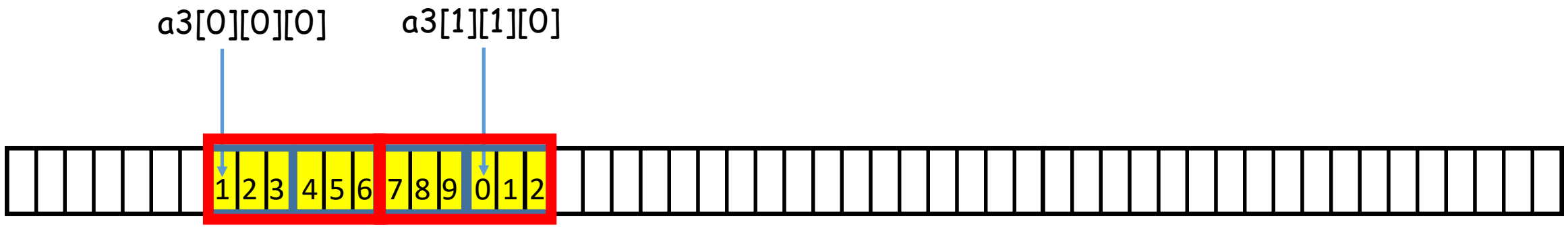
```
#include <stdio.h>
int main( void )
{
    char a3[3][4][5];
    for( int i=0 ; i<3 ; i++ )
        for( int j=0 ; j<4 ; j++ )
            for( int k=0 ; k<5 ; k++ )
                a3[i][j][k] = i + j + k;
    return 0;
}
```



Multi-dimensional arrays

- The compiler allocates a single block of memory with 2x2x3 elements
  - We can declare and assign using nested braces

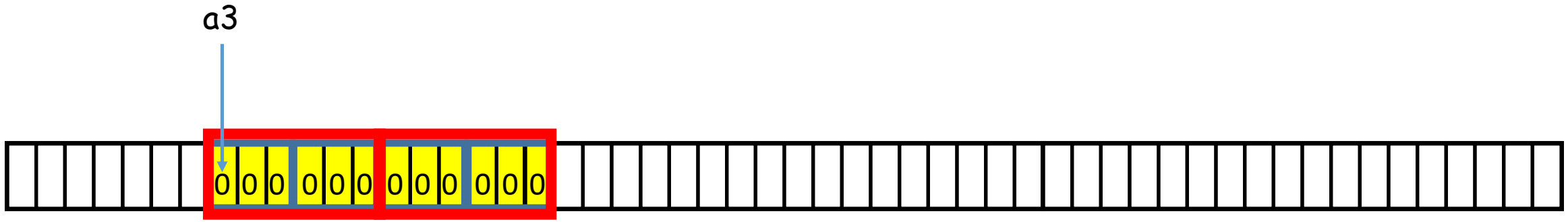
```
#include <stdio.h>
int main( void )
{
    char a3[][2][3] =
    {
        { { 1 , 2 , 3 } , { 4 , 5 , 6 } } ,
        { { 7 , 8 , 9 } , { 0 , 1 , 1 } }
    };
    return 0;
}
```



Multi-dimensional arrays

- We can pass the array to a function:
  - As a pointer

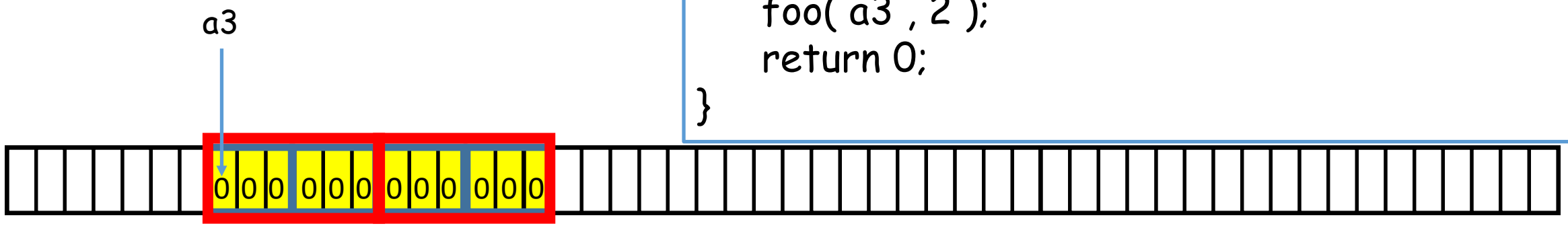
```
#include <stdio.h>
void foo( char* a1 , int sz )
{
    for( int i=0 ; i<sz ; i++ ) a1[i] = 0;
}
int main( void )
{
    char a3[2][2][3];
    foo( (char*)a3 , 2*2*3 );
    return 0;
}
```



Multi-dimensional arrays

- We can pass the array to a function:
  - As a pointer
  - As a 3D array

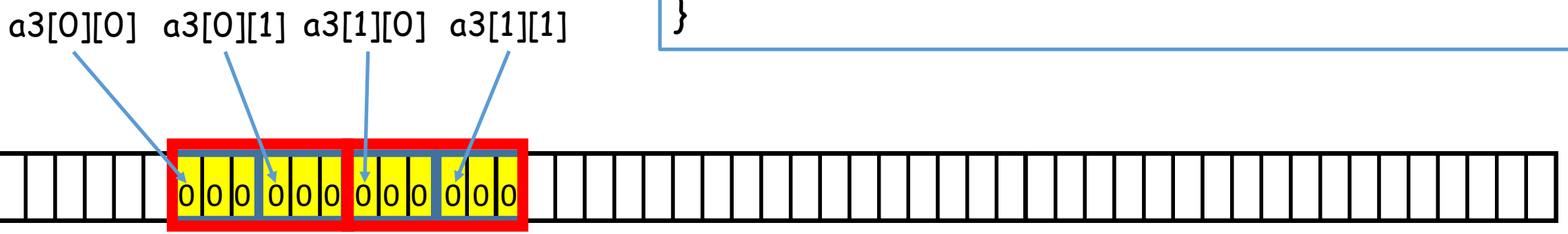
```
#include <stdio.h>
void foo( char a3[][2][3] , int sz )
{
    for( int i=0 ; i<sz ; i++ )
        for( int j=0 ; j<2 ; j++ )
            for( int k=0 ; k<3 ; k++ )
                a3[i][j][k] = 0;
}
int main( void )
{
    char a3[2][2][2];
    foo( a3 , 2 );
    return 0;
}
```



Multi-dimensional arrays

- We can pass the array to a function:
  - As a pointer
  - As a 3D array
  - As 1D slices

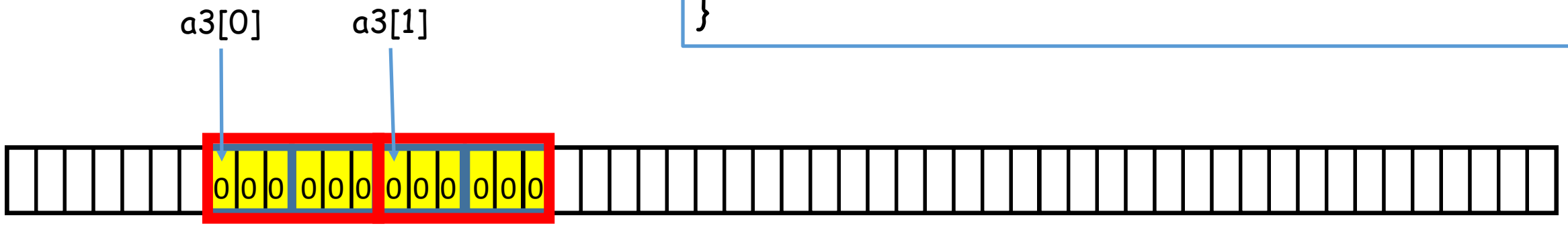
```
#include <stdio.h>
void foo( char * a1 , int sz )
{
    for( int i=0 ; i<sz ; i++ ) a1[i] = 0;
}
int main( void )
{
    char a3[2][2][3];
    for( int i=0 ; i<2 ; i++ ) for( int j=0 ; j<2 ; j++ )
        foo( a3[i][j] , 3 );
    return 0;
}
```



Multi-dimensional arrays

- We can pass the array to a function:
  - As a pointer
  - As a 3D array
  - As 1D slices
  - As 2D slices

```
#include <stdio.h>
void foo( char a2[][3] , int sz )
{
    for( int i=0 ; i<sz ; i++ ) for( int j=0 ; j<3 ; j++ )
        a2[i][j] = 0;
}
int main( void )
{
    char a3[2][2][3];
    for( int i=0 ; i<2 ; i++ ) foo( a3[i] , 2 );
    return 0;
}
```



Outline

- The `const` keyword
- Memory layout
- Arrays
- Miscellanea

Miscellanea

- When declaring a pointer, the "*" can go:
  - Next to the type: `int* foo;`
  - Next to the name: `int *foo;`
  - Between the two: `int * foo;`

- Technically, it should go next to the name

Miscellanea

- C allows us to declare two variables at once

```
#include <stdio.h>
int main( void )
{
    int a=1 , b=2;
    …
    return 0;
}
```

```
#include <stdio.h>
int main( void )
{
    int a=1;
    int b=2;
    …
    return 0;
}
```

Miscellanea

- C allows us to declare two variables at once
  - If the types are pointers, we need a "*" before each variable

```
#include <stdio.h>
int main( void )
{
    int * a=NULL , * b=NULL;
    …
    return 0;
}
```

```
#include <stdio.h>
int main( void )
{
    int* a=NULL;
    int* b=NULL;
    …
    return 0;
}
```

Miscellanea

- C allows us to declare two variables at once
  - If the types are pointers, we need a "*" before each variable
  - Otherwise, only the first will be a pointer

```
#include <stdio.h>
int main( void )
{
    int * a=NULL , b=NULL;
    …
    return 0;
}
```

```
#include <stdio.h>
int main( void )
{
    int* a=NULL;
    int b=NULL;
    …
    return 0;
}
```

Miscellanea

- When parsing a declaration, we start with the variable name and work our way out:

`const int * foo;` ⇔ `foo` is a pointer to a constant `int`

`int foo[3][4];` ⇔ `foo` is a size-three array of quadruples of `int`s

- If we have qualifiers on both sides, we alternate, starting on the right

`int * foo[3];` ⇔ `foo` is size-three array whose elements are pointers to `int`s

```
int main( int argc , const char * argv[] );
```

*For more details, see the Clockwise/Spiral rule (http://c-faq.com/decl/spiral.anderson.html)

Miscellanea

- We can declare/define a function taking an array with brackets:

```
void foo( int a[] );
```

- However, the compiler automatically turns this into a pointer:

```
void foo( int * a );
```