Intermediate Programming

Day 12

Outline

- Pointer operations
- Dynamic 2D arrays
- Pointers and `const`
- Review questions

Pointer arithmetic

- A pointer is an unsigned integer value indicating a location in memory.*
  - We can add /subtract integers from a pointer (to get a new pointer):

    ```
    b = a+2;
    b -= 3;
    b++;
    ```

    etc.
  - We can compute the difference between two pointers (to get a signed integer)**:

    ```
    ptrdiff_t d = b-a;
    ```
  - We can print out the value of a pointer:

    ```
    printf( "Address: %p\n" , (void*)a );
    ```

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int *b = a+2;

    return 0;
}
```

* The size of a pointer depends on the architecture. On 64-bit architectures, it is eight bytes long

** ptridiff_t is a predefined type in stddef.h designed to store the difference between pointers

Pointer arithmetic

- A pointer is an unsigned integer value indicating a location in memory.
  - We can add /subtract integers from a pointer

Q: What is the value of b?

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int *b = a+2;

    return 0;
}
```

Pointer arithmetic

- A pointer is an unsigned integer value indicating a location in memory.
  - We can add /subtract integers from a pointer

Q: What is the value of b?

A: b is the address of the integer two in from the start of the array

Note:

- The int two elements in from the start of the array is 8 bytes away in memory
- Because the type of the pointer is known, the compiler automatically deduces that two int lengths correspond to 8 bytes

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int *b = a+2;
    printf( "%d %d\n" , *a , *b );
    return 0;
}
```

```
>> ./a.out
2 6
>>
```

Pointer arithmetic

- A pointer is an unsigned integer value indicating a location in memory.
  - We can add /subtract integers from a pointer

Q: What is the value of b?

A: b is the address of the integer two in from the start of the array

Note:

- The int two elements in from the start of the array is 8 bytes away in memory
- Because the type of the pointer is known, the compiler automatically deduces that two int lengths correspond to 8 bytes

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int *b = a+2;
    printf( "%d %d\n" , (int)(a-b) );
    return 0;
}
```

```
>> ./a.out
-2
>>
```

- Similarly, the difference between pointers is measured in units of elements

Pointer arithmetic

- A pointer is an unsigned integer value indicating a location in memory.
  - We can add /subtract integers from a pointer

Q: What is the value of b?

A: b is the address of the integer two in from the start of the array

Note:

- The int two elements in from the start of the array is 8 bytes away in memory
- Because the type of the pointer is known, the compiler automatically deduces that two int lengths correspond to 8 bytes

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int *b = a+2;
    void *_a =a , *_b = b;
    printf( "%d\n" , b-a );
    printf( "%d\n" , _b-_a );
    return 0;
}
```

```
>> ./a.out
2
8
>>
```

- A pointer of type void* is treated as a raw memory address (v ormation)

Pointer arithmetic

- If `ip` points to `int x`. Then `*ip` can be used anywhere that `x` makes sense:

`printf( "%d\n" , *ip )` ⇔ `printf( "%d\n" , x )`

- Unary ops `&` and `*` bind more tightly than binary arithmetic ops

`*ip += 1` ⇔ `x += 1`

`y = *ip + 1` ⇔ `y = x+1`

- [WARNING] unary operators associate from right to left
  - `++*ip` is the same as `++x`
  - `*ip++` means something else

Pointer arithmetic

- A pointer is an unsigned integer value indicating a location in memory.
  - We can add /subtract integers from a pointer

Q: So what does *b++ mean?

A: It's a combination of four instruction:

1. Increment the pointer b,
2. Return the old pointer's value,*
3. Dereference that
4. Set it to zero

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int* b = a+2;
    *b++ = 0;
    printf( "%d %d %d %d : %d\n" ,
        a[0] , a[1] , a[2] , a[3] , *b );
    return 0;
}
```

```
>> ./a.out
2 4 0 8 : 8
>>
```

*Recall that post-increment/decrement returns the old value

Pointer arithmetic

- We can access a pointer by dereferencing
  `printf( "%d\n" , *b );`
- We can access array elements with []
  `printf( "%d\n" , b[0] );`
- Since pointers and arrays are essentially the same, these are the same operations!

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int* b = a+2;
    printf( "%d\n" , *b );
    printf( "%d\n" , b[0] );
    return 0;
}
```

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

- More generally `*(b+k)` is the same as `b[k]` for any integer `k`

Pointer arithmetic

Though similar, arrays and pointers differ in a couple of ways:

1. The use of **sizeof**

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int* b = a;
    printf( "%d\n" , sizeof(a) );
    printf( "%d\n" , sizeof(b) );
    return 0;
}
```

```
>> ./a.out
16
8
>>
```

Pointer arithmetic

Though similar, arrays and pointers differ in a couple of ways:

1. The use of **sizeof**
2. Arrays are immutable

```
#include <stdio.h>
int main( void )
{
    int a[] = { 2 , 4 , 6 , 8 };
    int b = 10;
    a = &b;
    return 0;
}
```

```
>> gcc ...
foo.c: In function ‘main’:
foo.c:6:4: error: assignment to expression with array type
    6 |  a = &b;
      |    ^
>>
```

Outline

- Pointer operations
- Dynamic 2D arrays
- Pointers and `const`
- Review questions

Dynamic 2D arrays

Q: How do we dynamically declare a 2x3 grid of `int` values?

A1: Declare an array of 6 `int` values

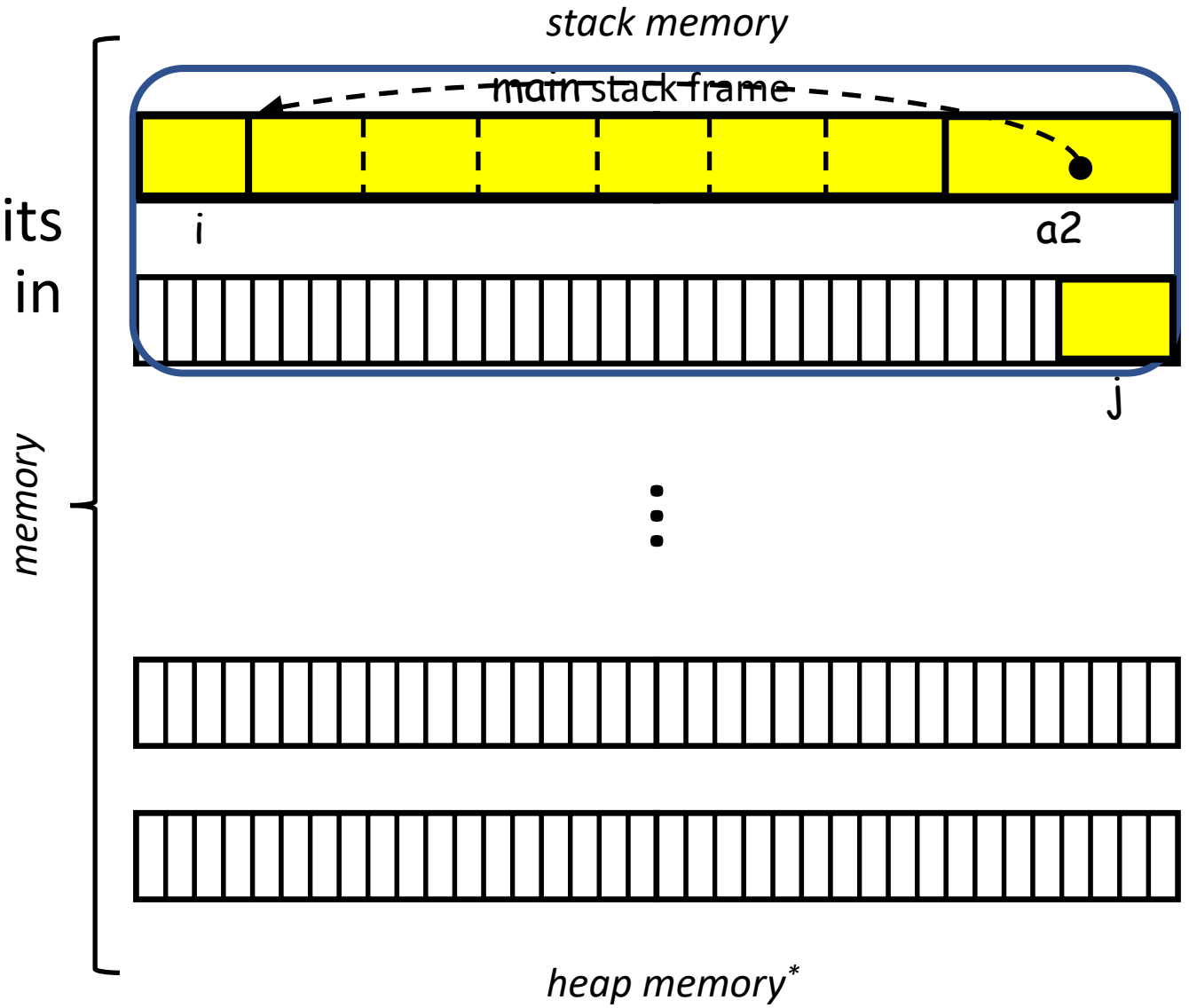
A2: Declare an array (of size 2) containing `int` arrays (of size 3).

Dynamic 2D arrays

Recall:

If we statically declare a 2D array its contents are laid out sequentially in (stack) memory.

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

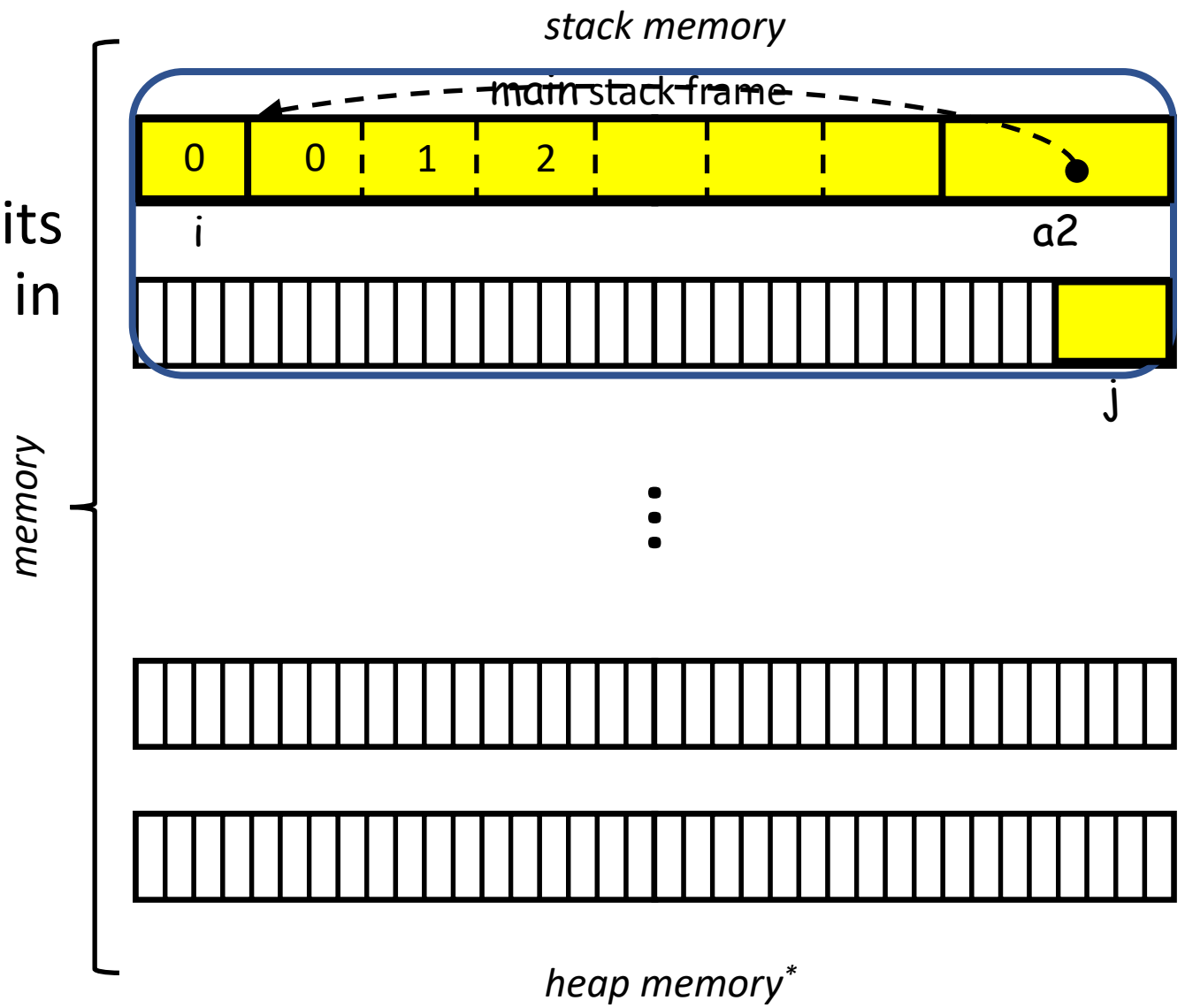


Dynamic 2D arrays

Recall:

If we statically declare a 2D array its contents are laid out sequentially in (stack) memory.

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

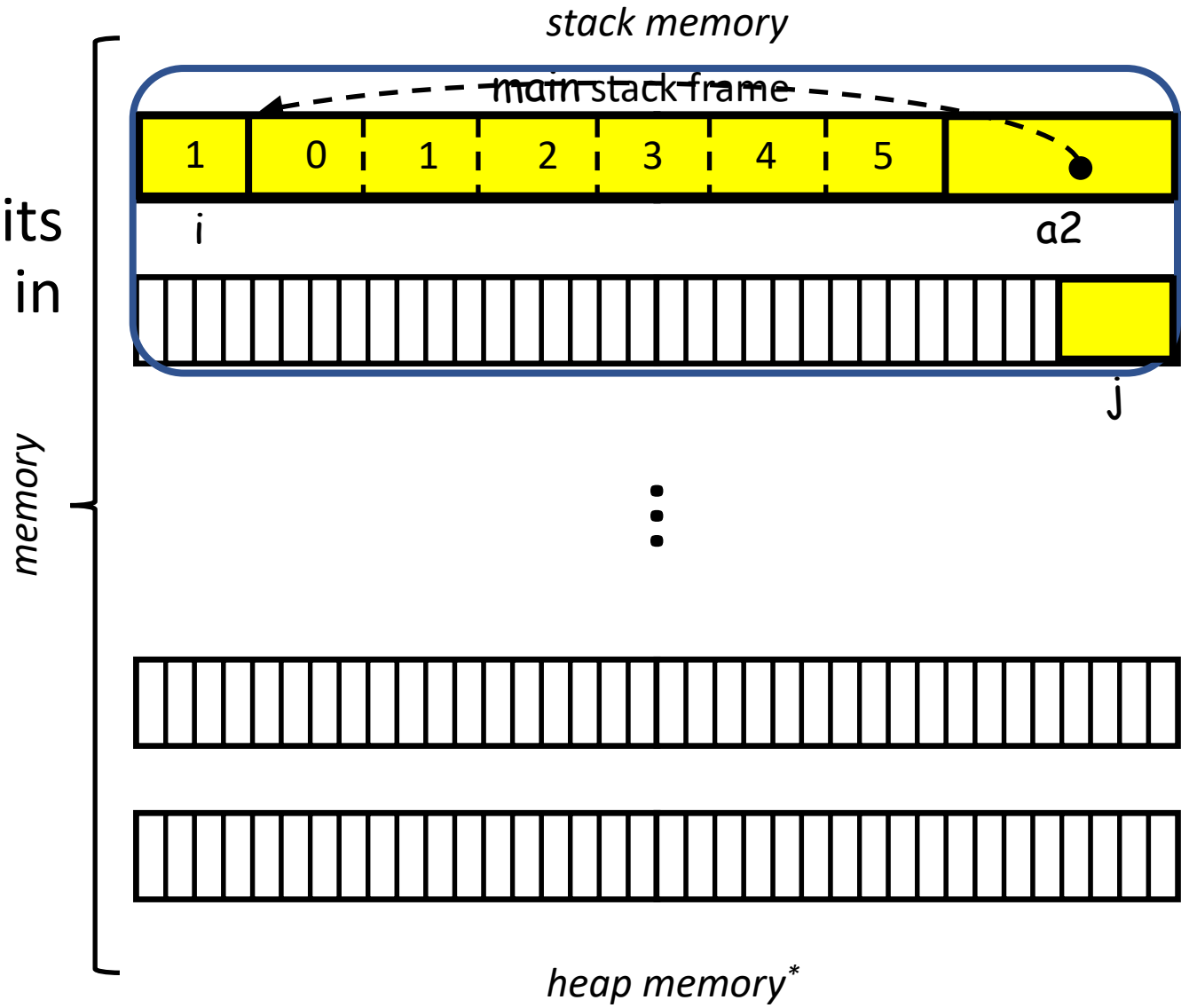


Dynamic 2D arrays

Recall:

If we statically declare a 2D array its contents are laid out sequentially in (stack) memory.

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

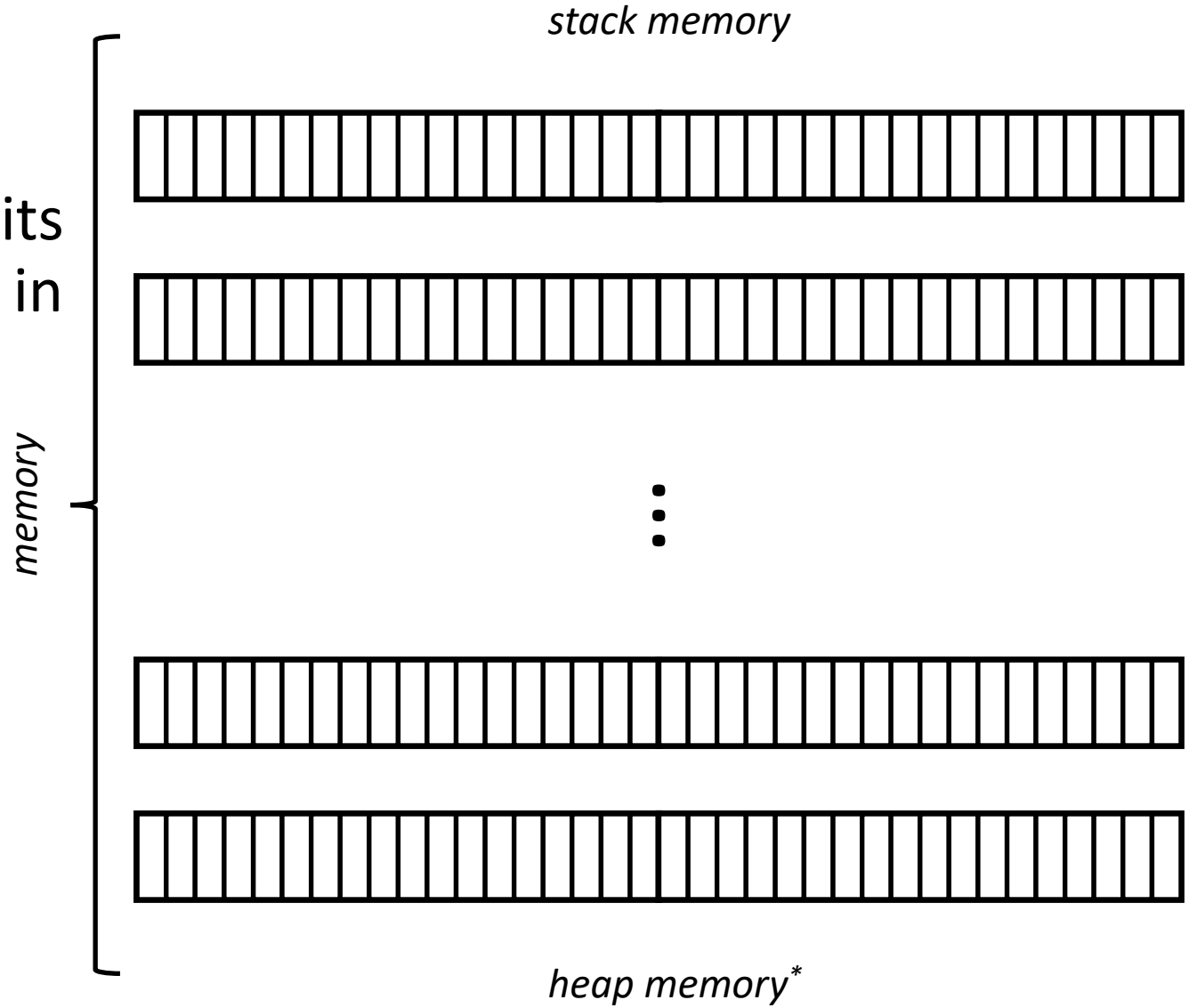


Dynamic 2D arrays

Recall:

If we statically declare a 2D array its contents are laid out sequentially in (stack) memory.

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



Dynamic 2D arrays

Declaring a 2x3 grid of int values:

1. Declare a single array of ints

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays

Declaring a 2x3 grid of int values:

1. Declare a single array of ints
   - Need to allocate/deallocate the int array

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

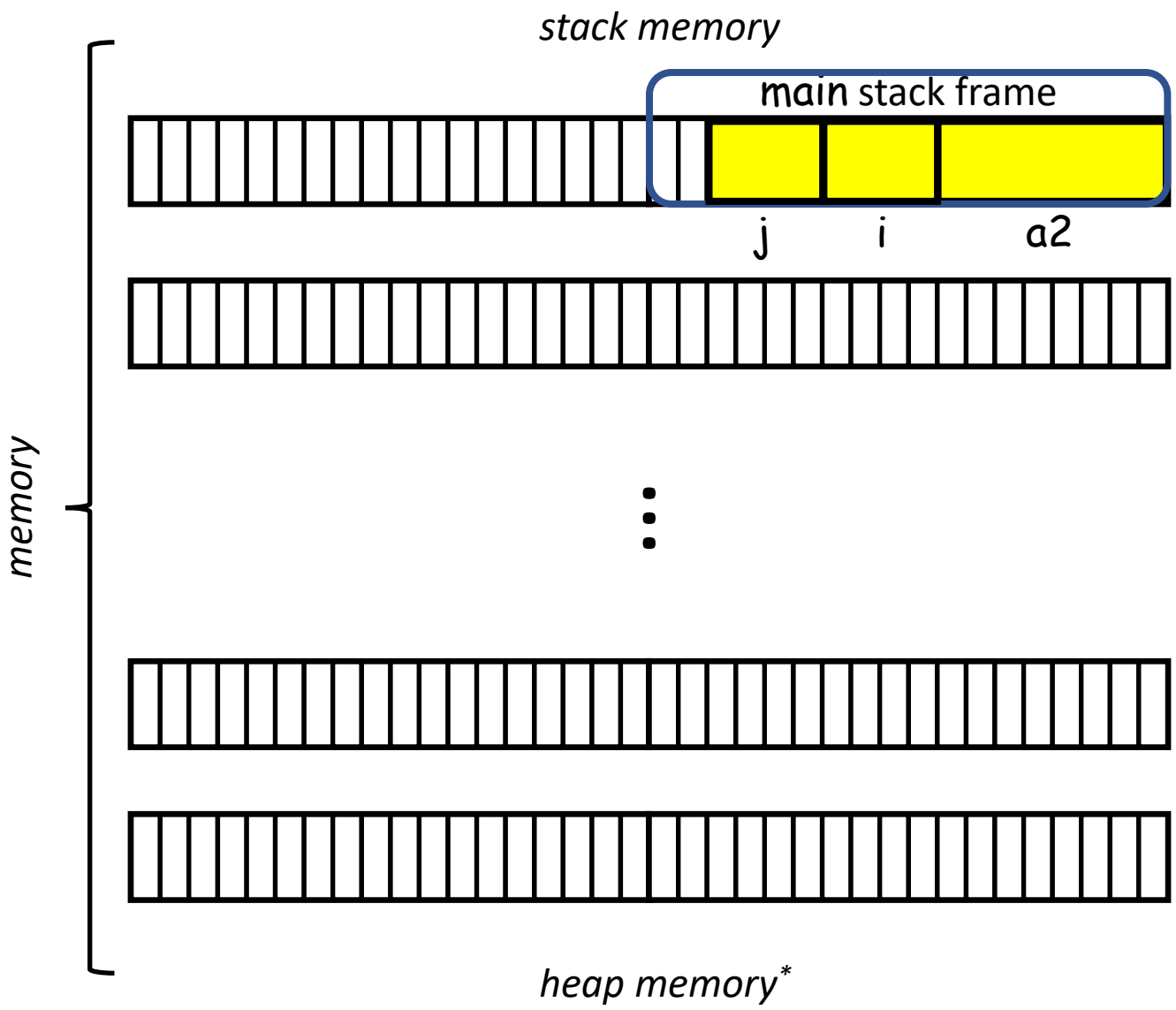
Dynamic 2D arrays

Declaring a 2x3 grid of int values:

1. Declare a single array of ints
   - Need to allocate/deallocate the int array
   - ✗ Indexing is ugly

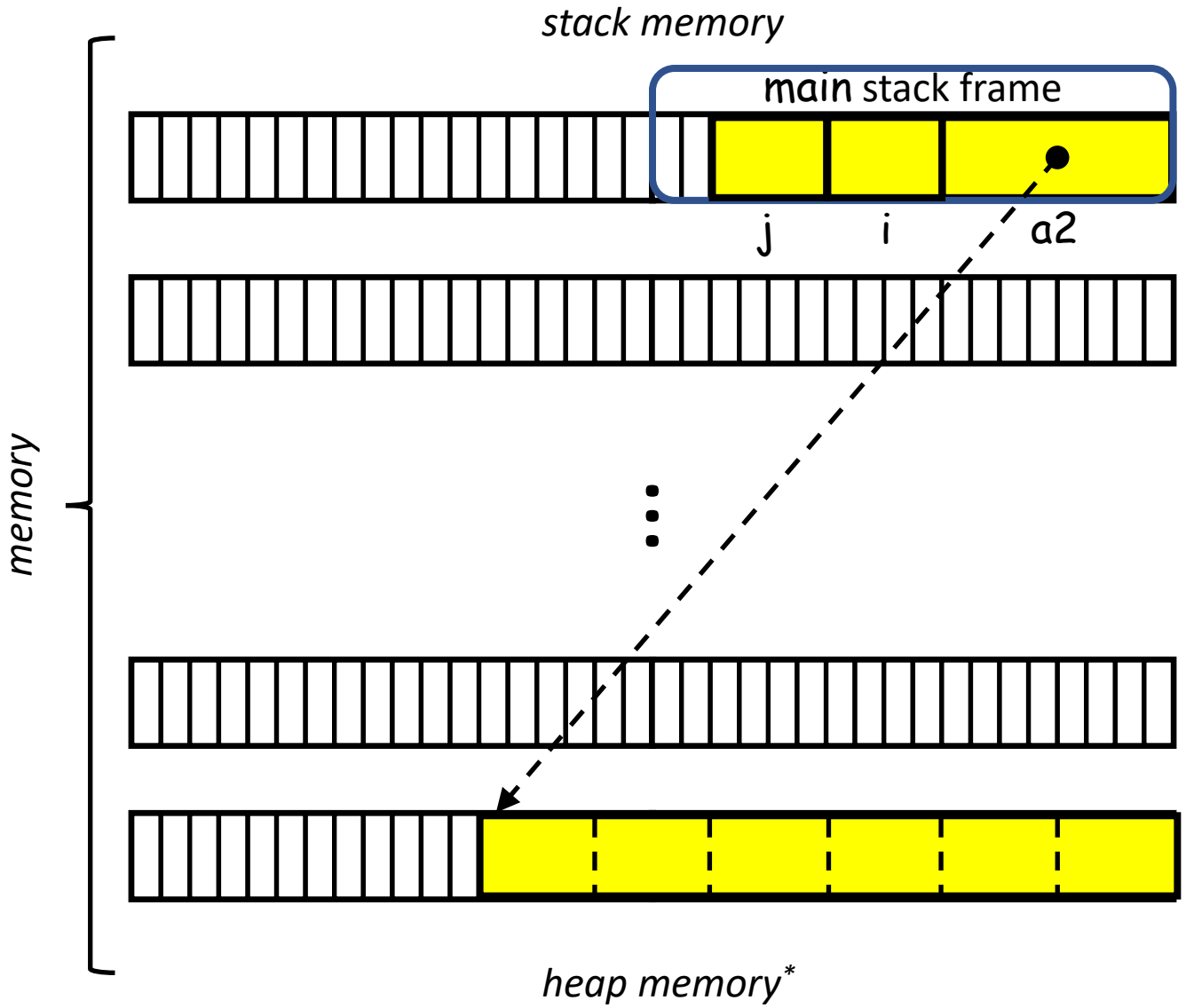
```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



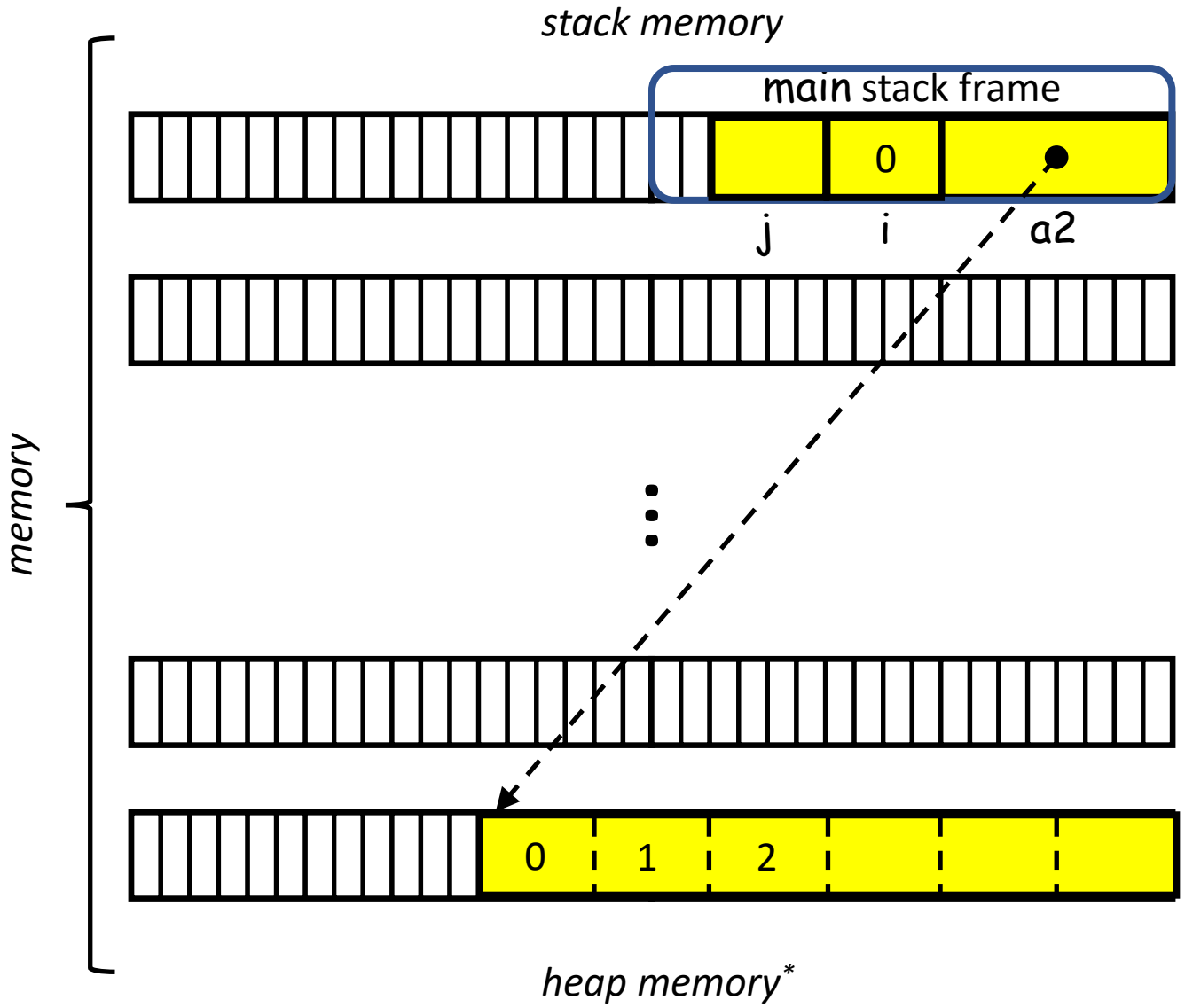

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



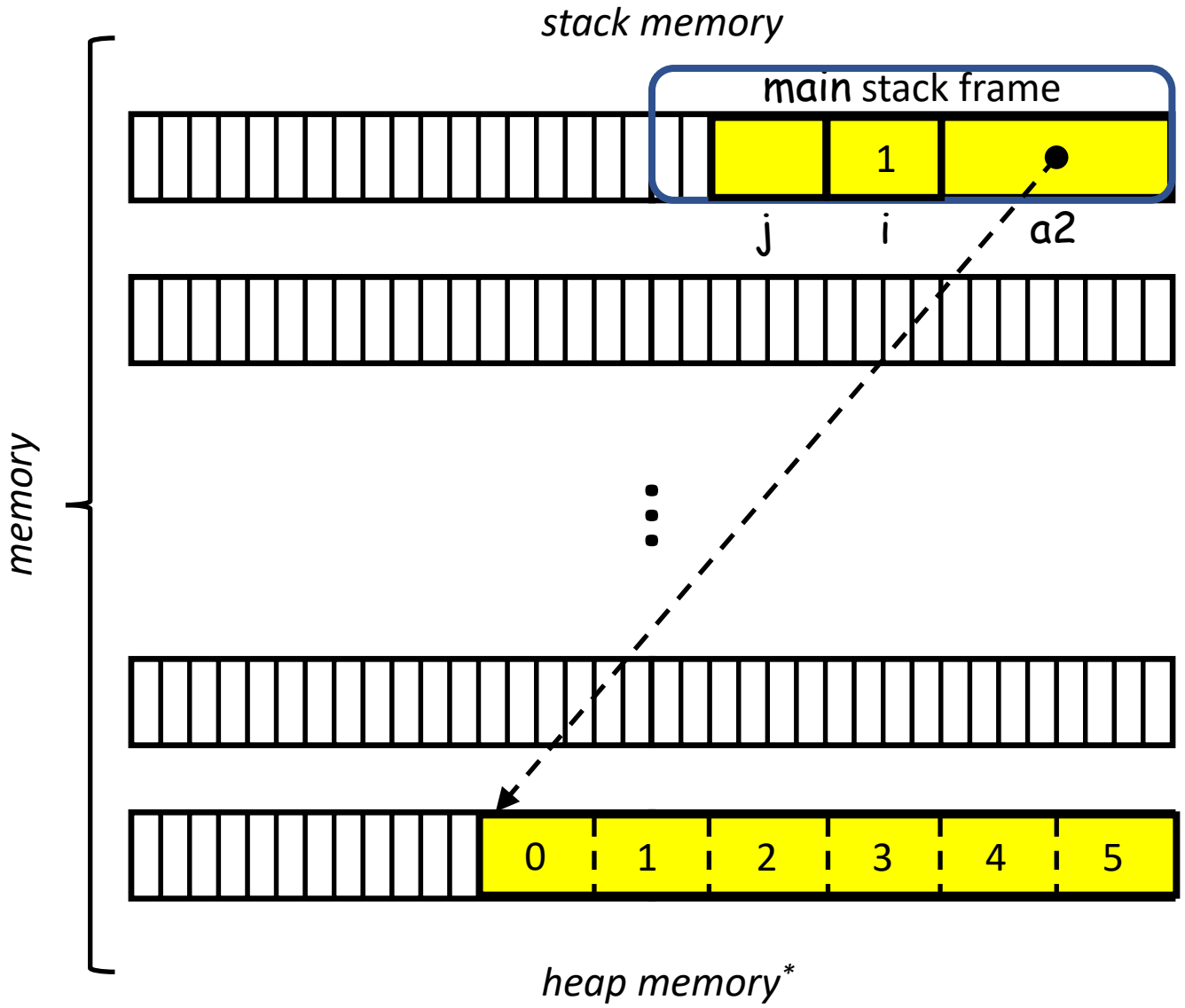

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



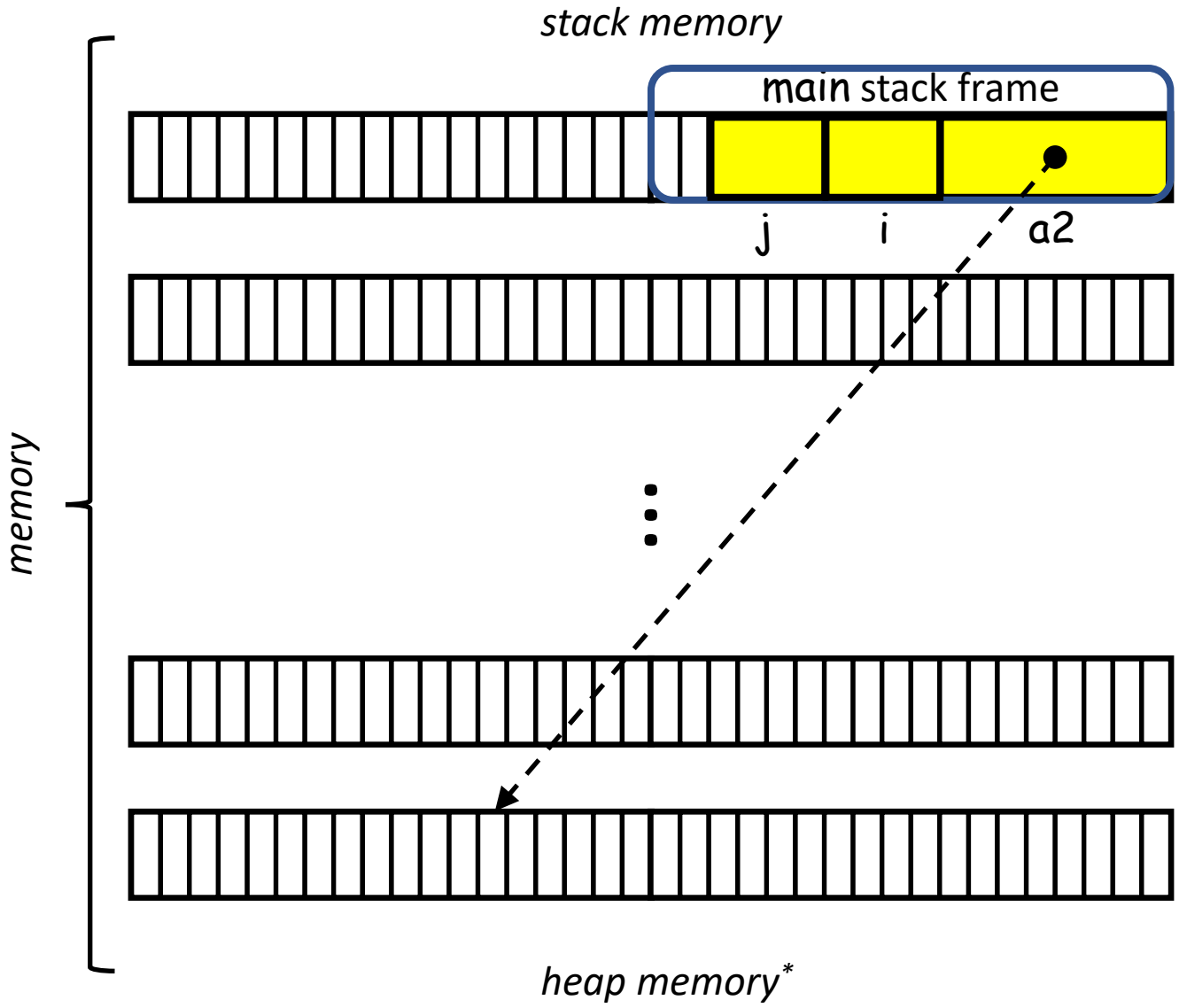

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



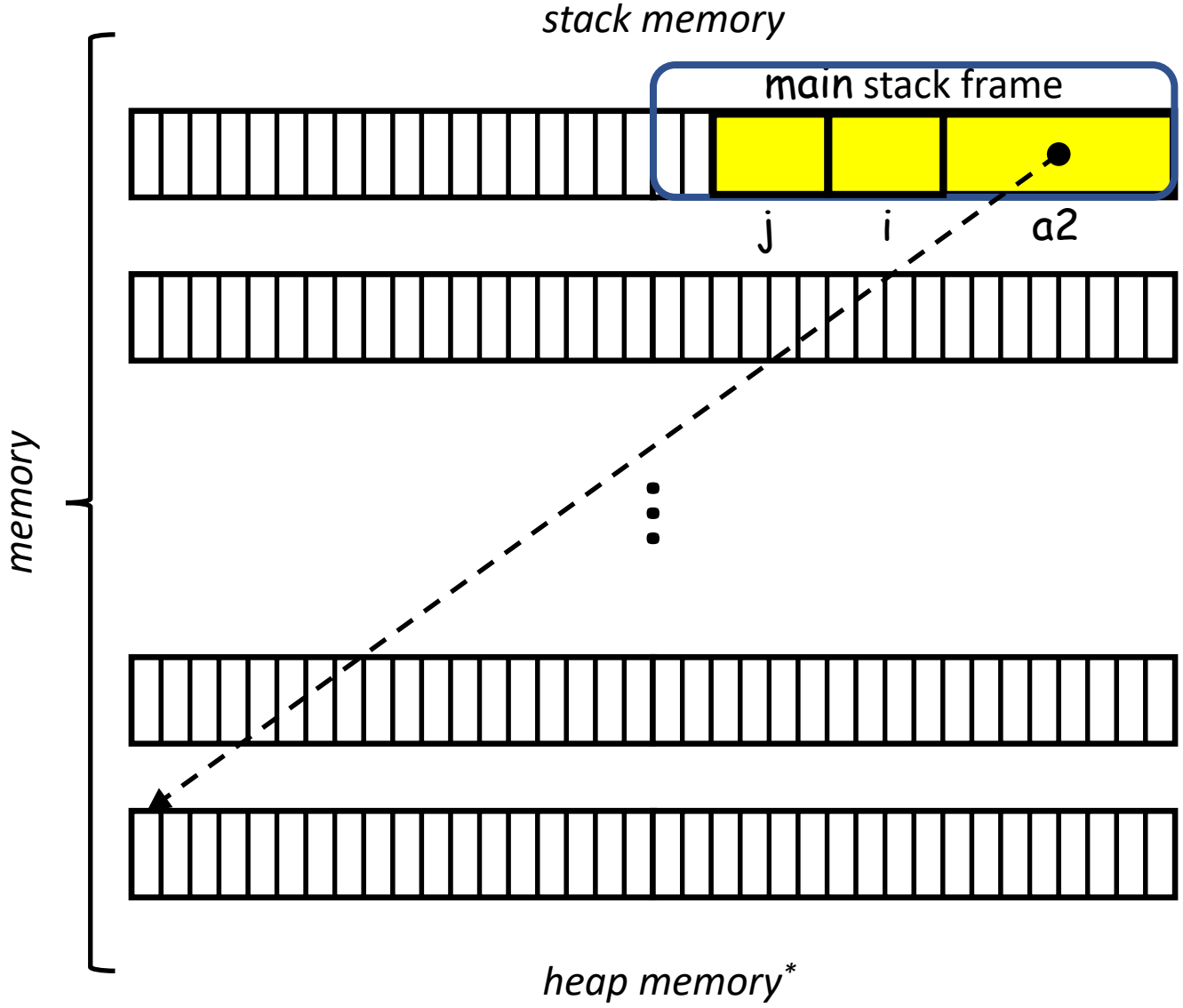

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



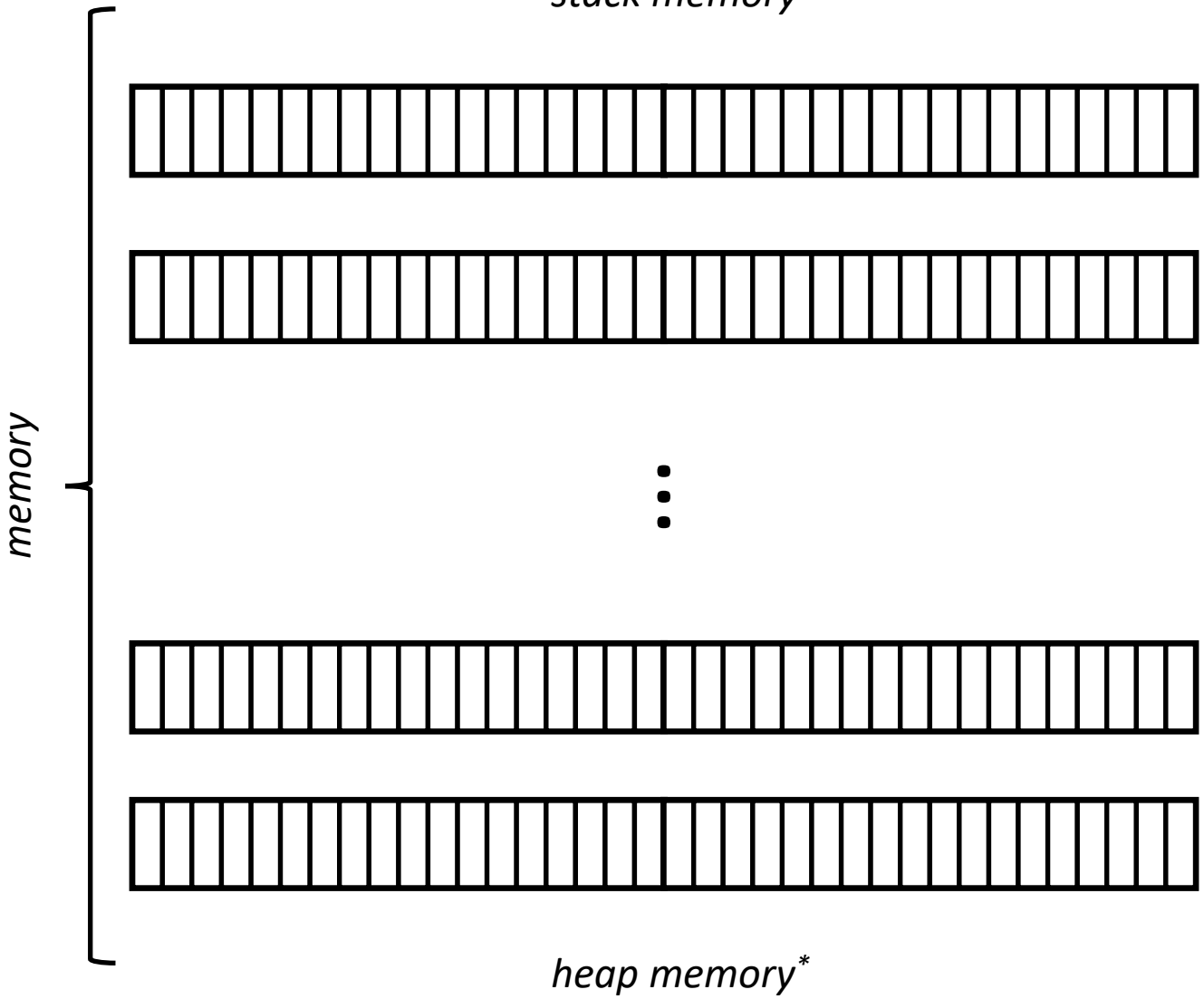

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays




```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays




```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays

Declaring a 2x3 grid of **int** values:

1. Declare a single array of **int**s
2. Declare an array of **int** arrays

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays

Declaring a 2x3 grid of int values:

1. Declare a single array of ints
2. Declare an array of int arrays
   - Need to allocate/deallocate the array of int arrays

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays

Declaring a 2x3 grid of int values:

1. Declare a single array of ints
2. Declare an array of int arrays
   - Need to allocate/deallocate the array of int arrays
   - Need to allocate/deallocate each int array

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays

Declaring a 2x3 grid of int values:

1. Declare a single array of ints
2. Declare an array of int arrays
   - Need to allocate/deallocate the array of int arrays
   - Need to allocate/deallocate each int array

   ✓Indexing is clean

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays

Declaring a 2x3 grid of int values:

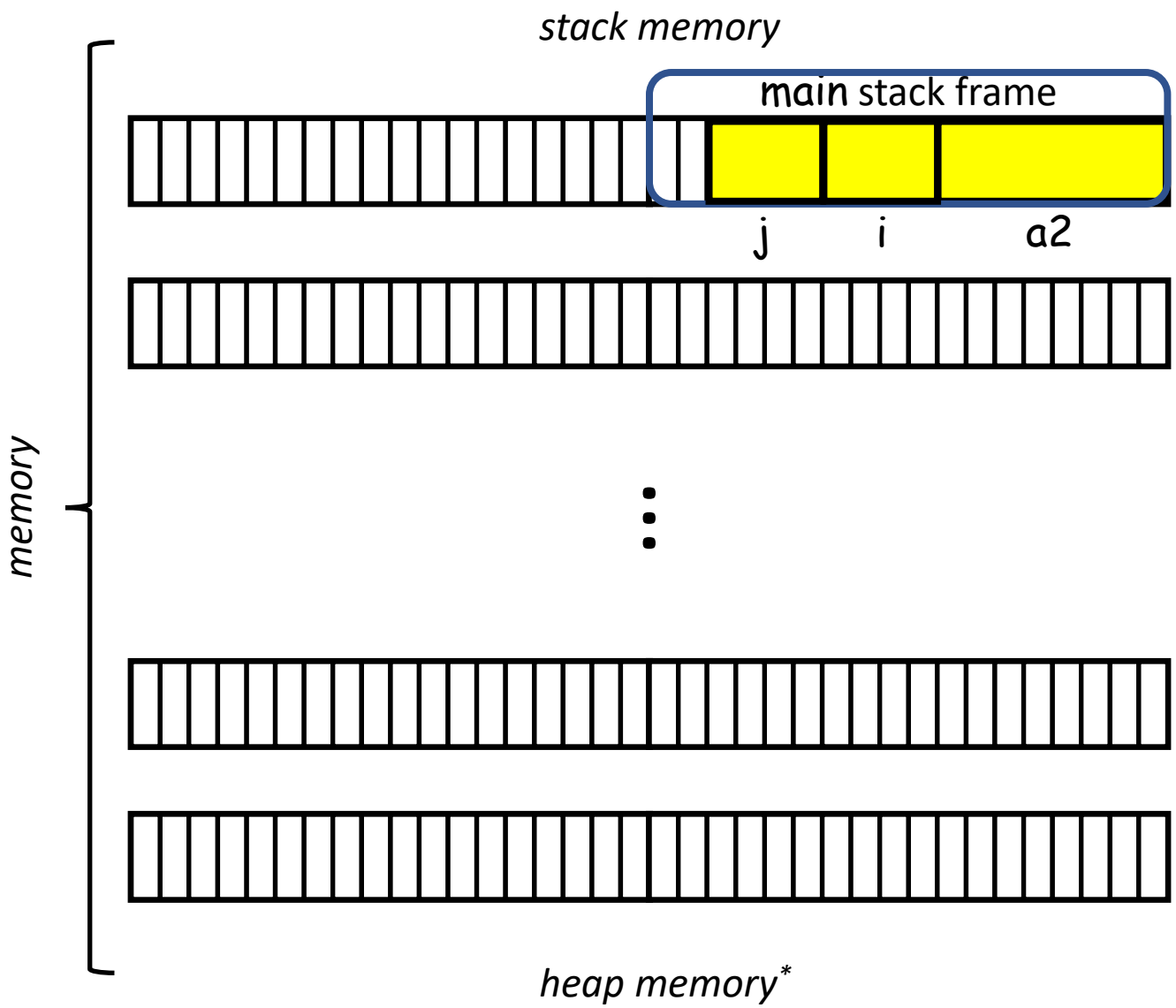
1. Declare a single array of ints
2. Declare an array of int arrays
   - Need to allocate/deallocate the array of int arrays
   - Need to allocate/deallocate each int array

   ✓Indexing is clean

⇒ With dynamic allocation we can have (jagged/non-uniform) 2D arrays with different rows having different sizes.

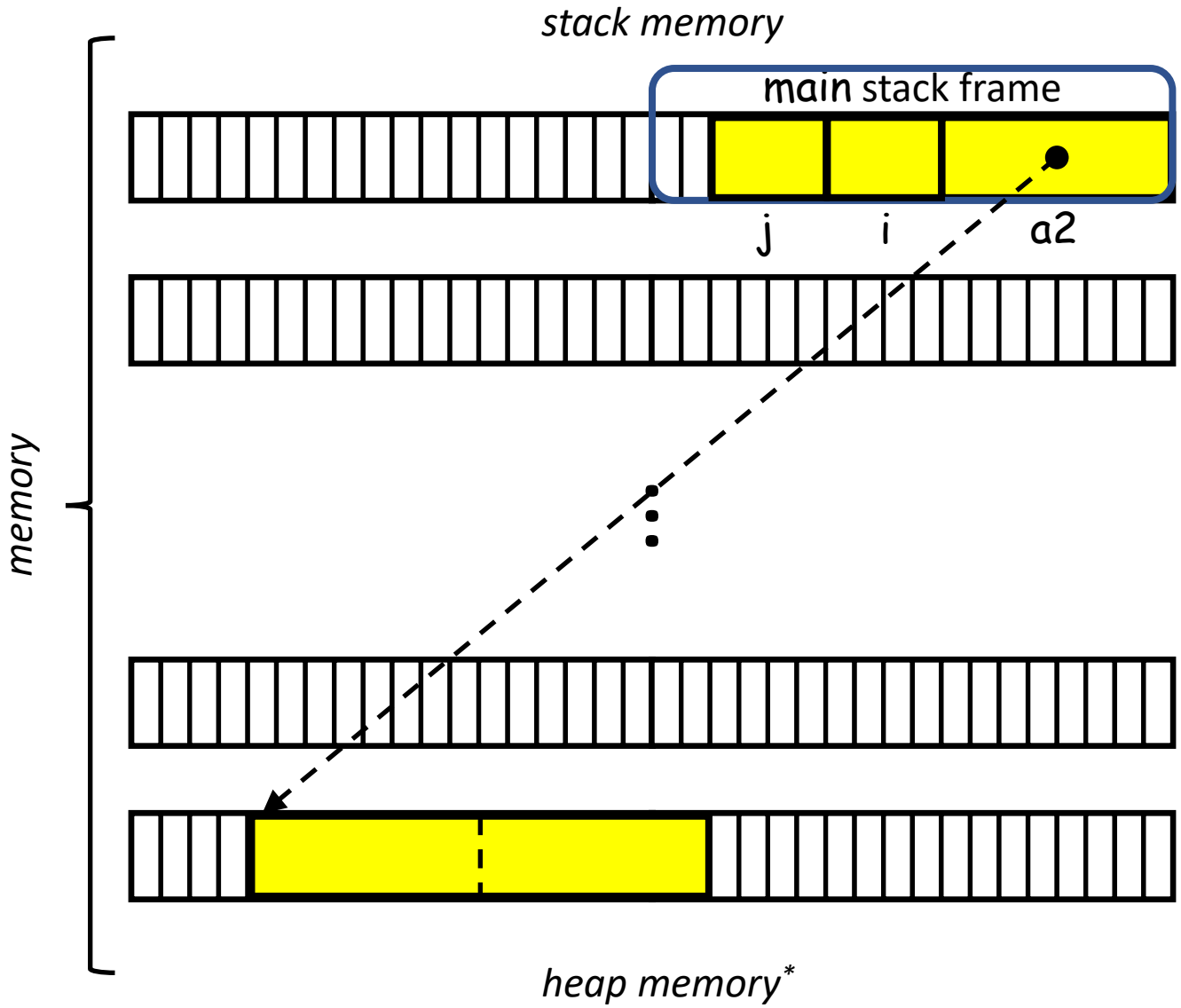
```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



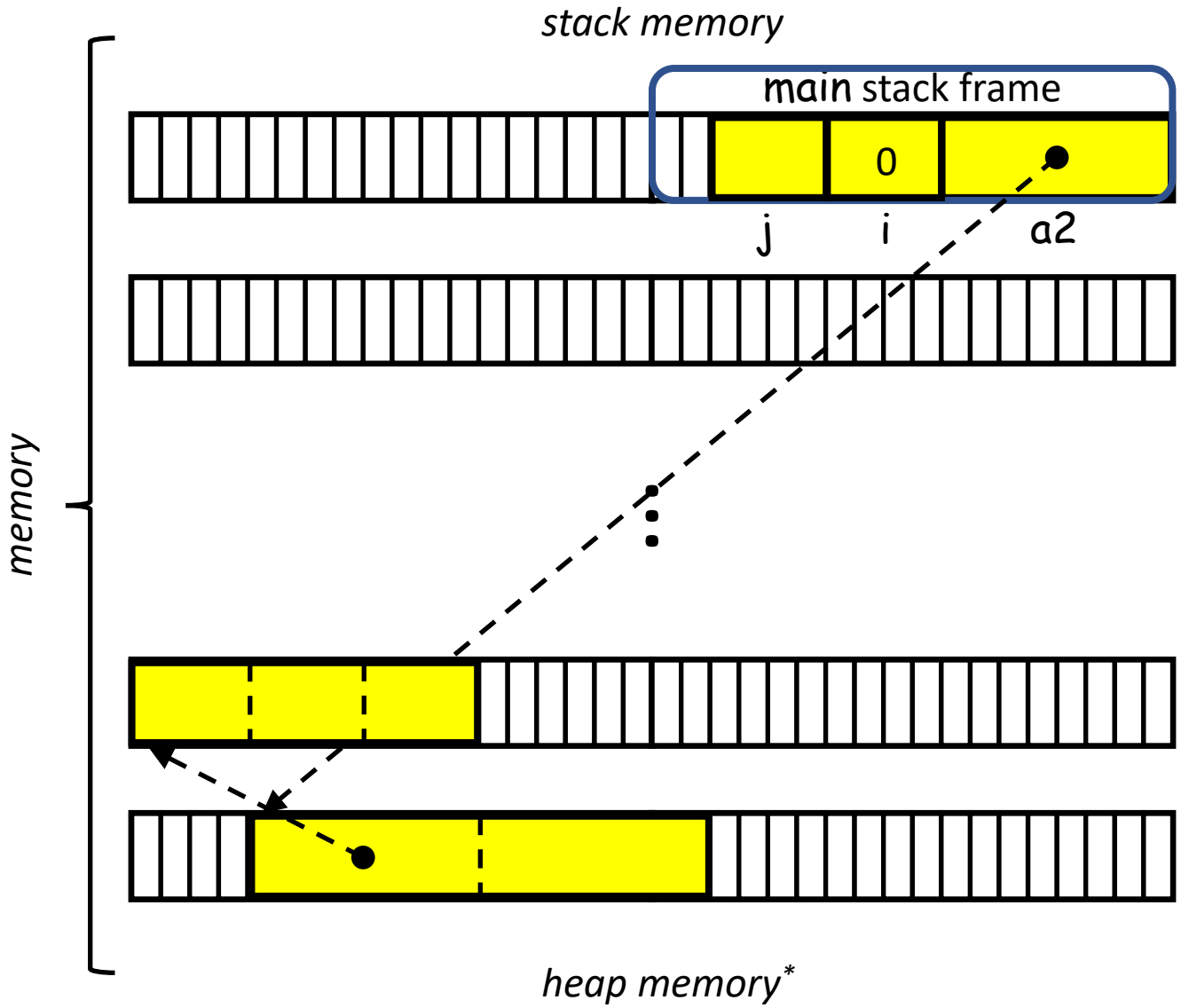

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



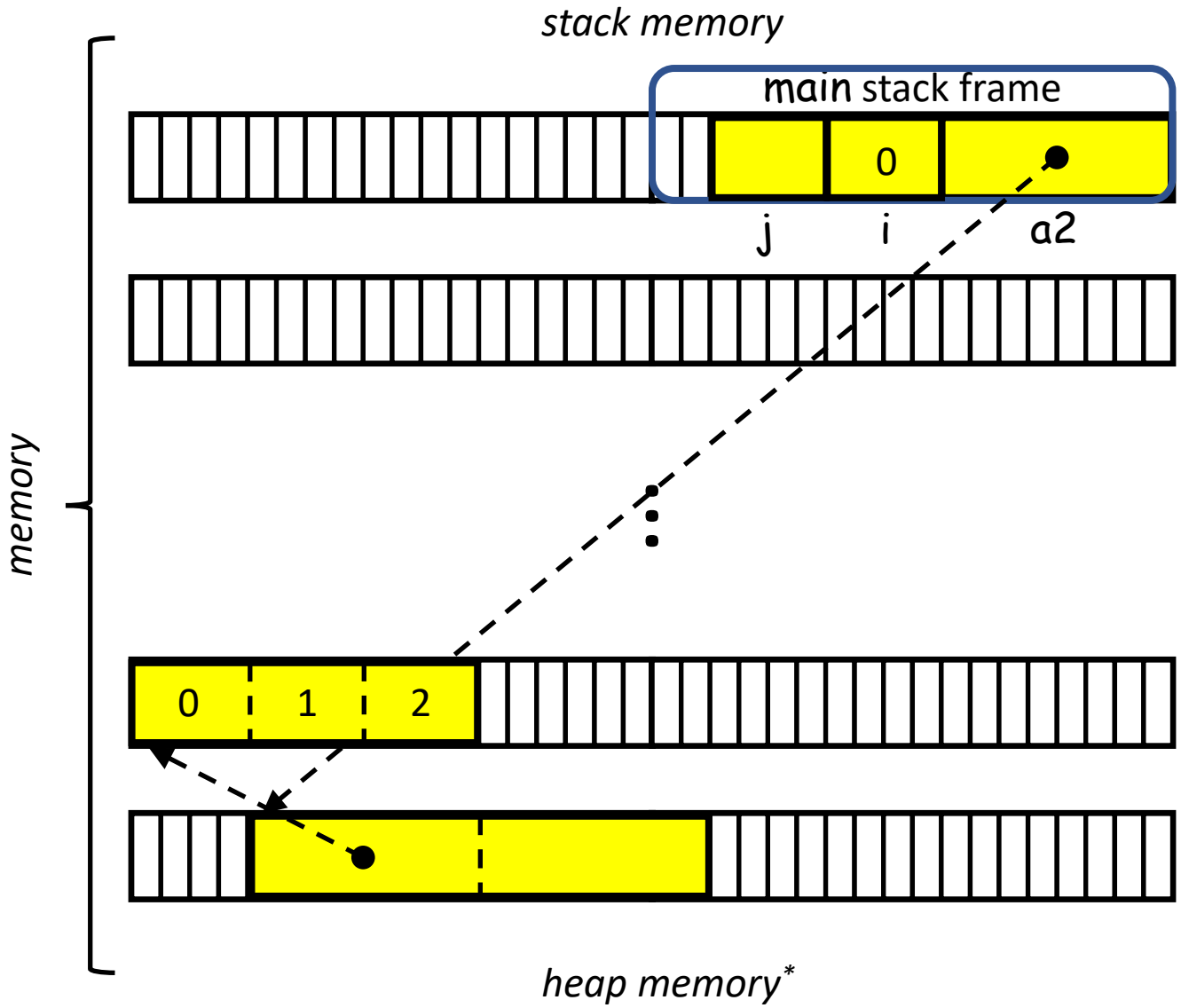

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



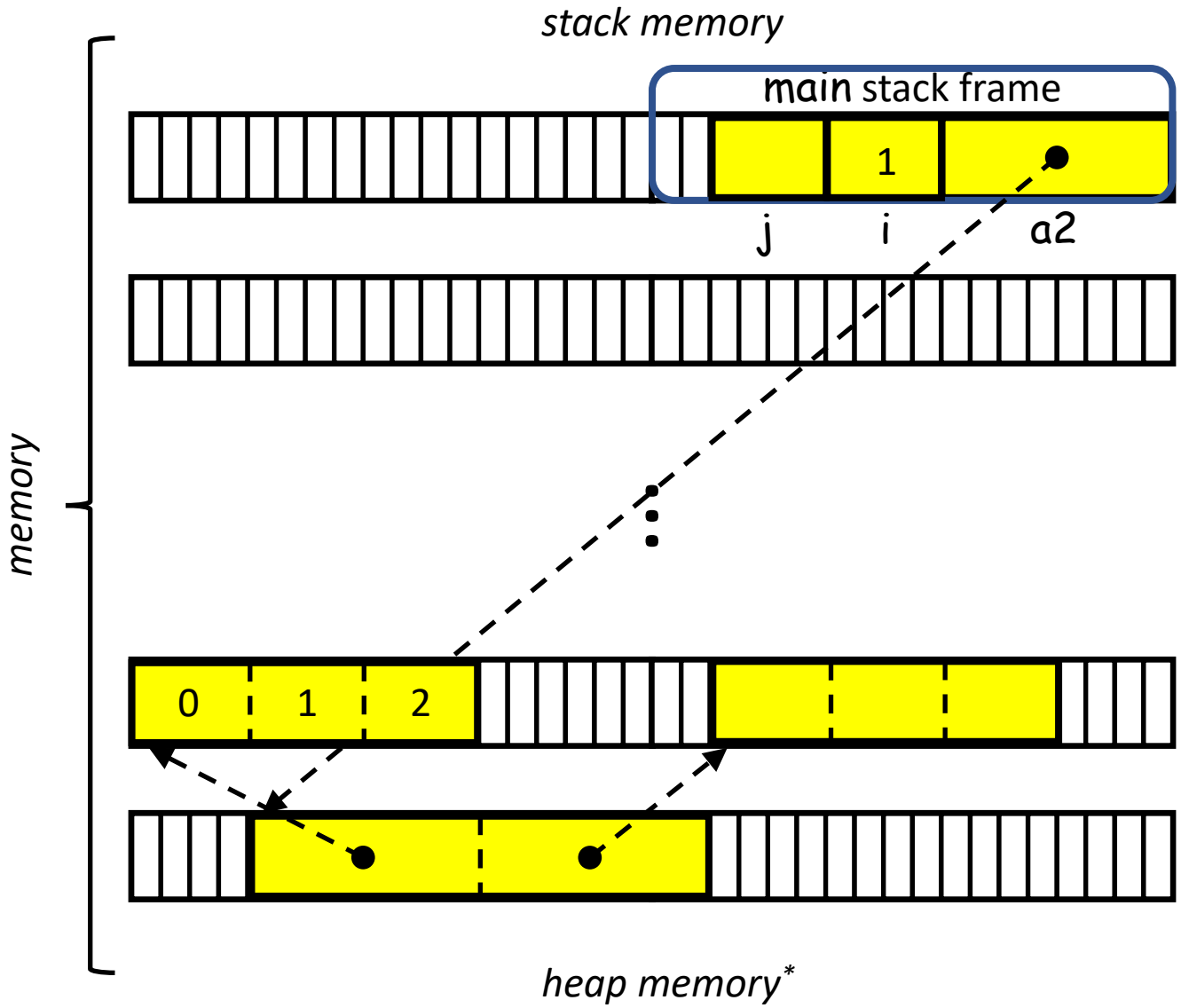

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



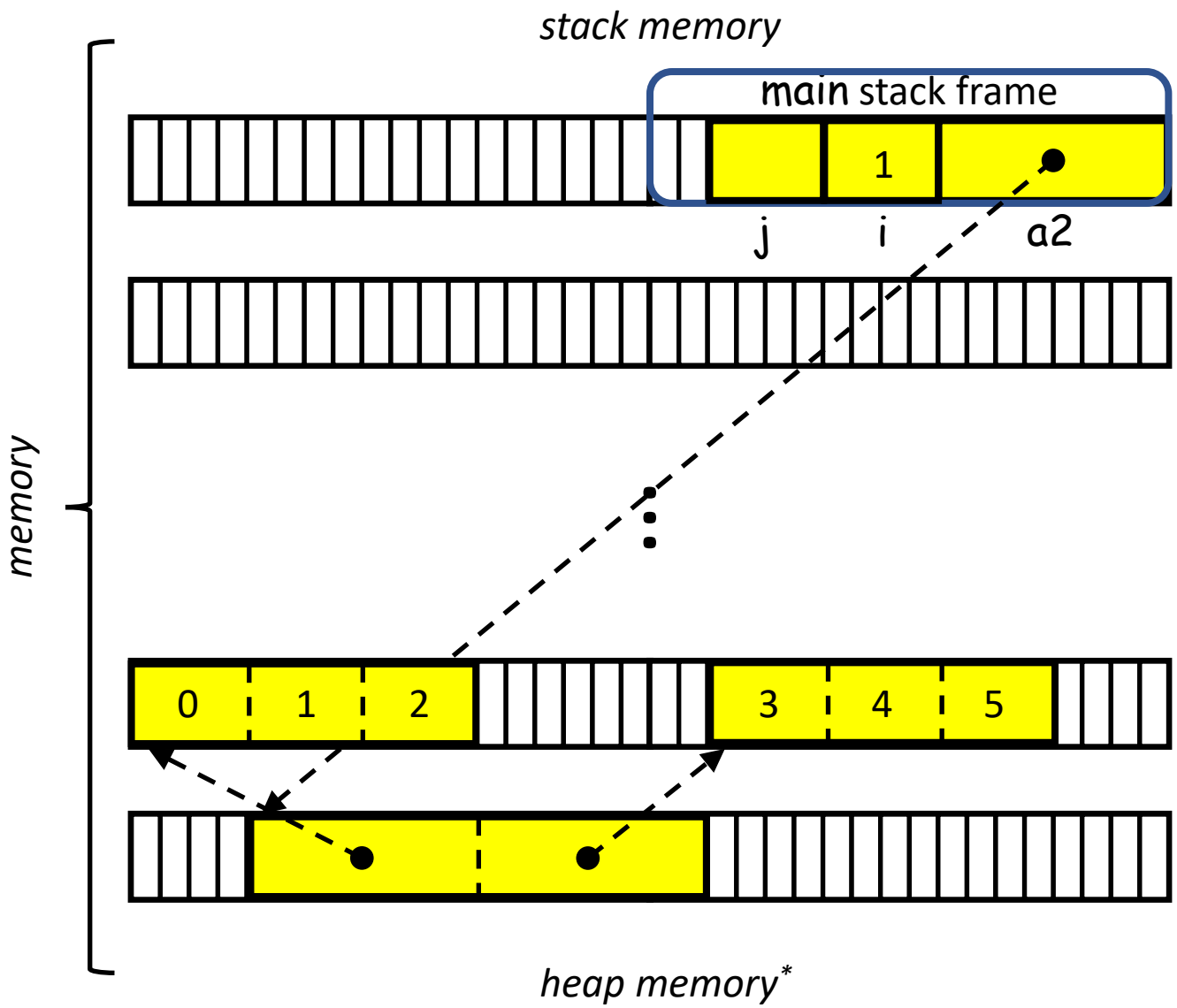

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



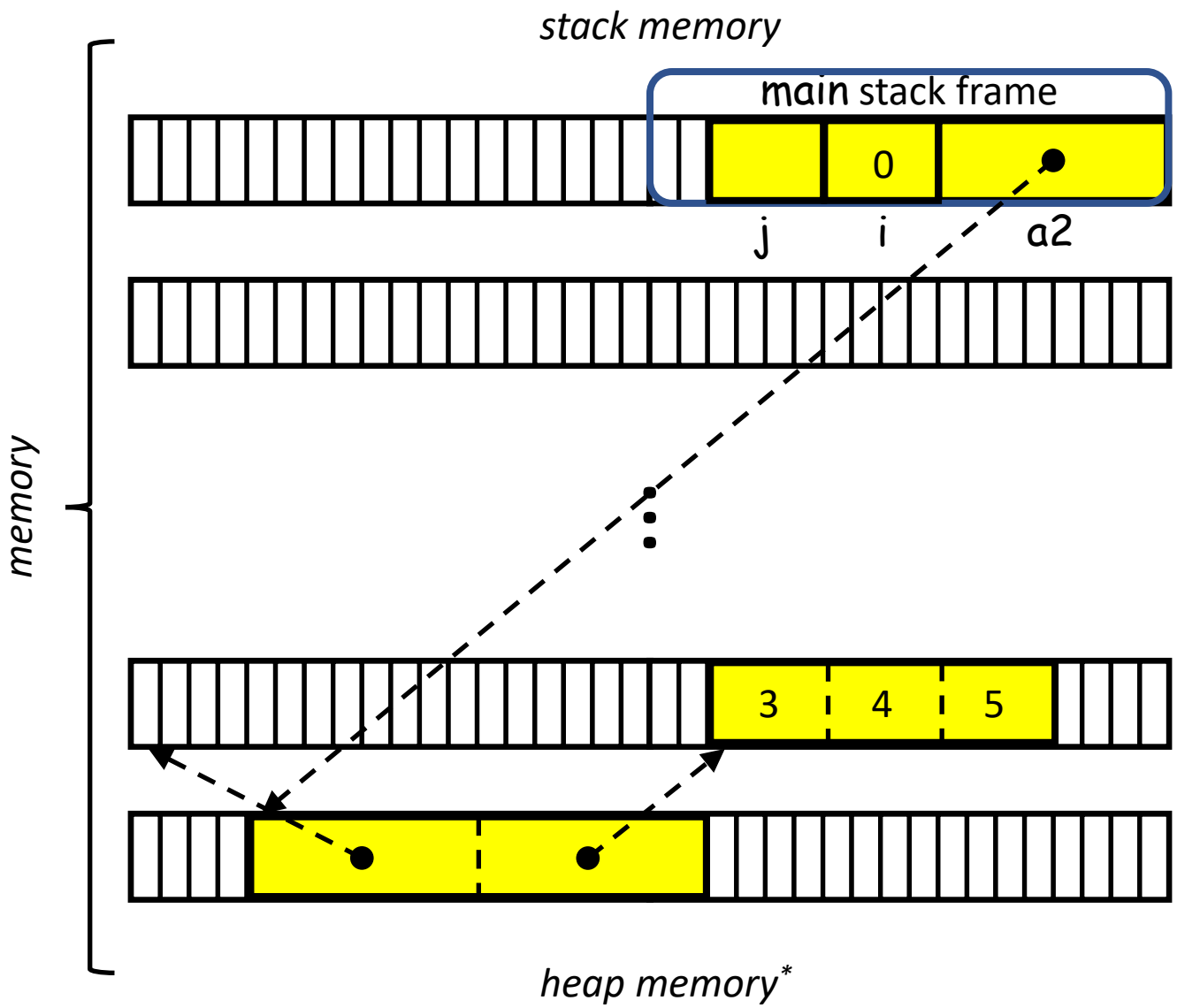

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



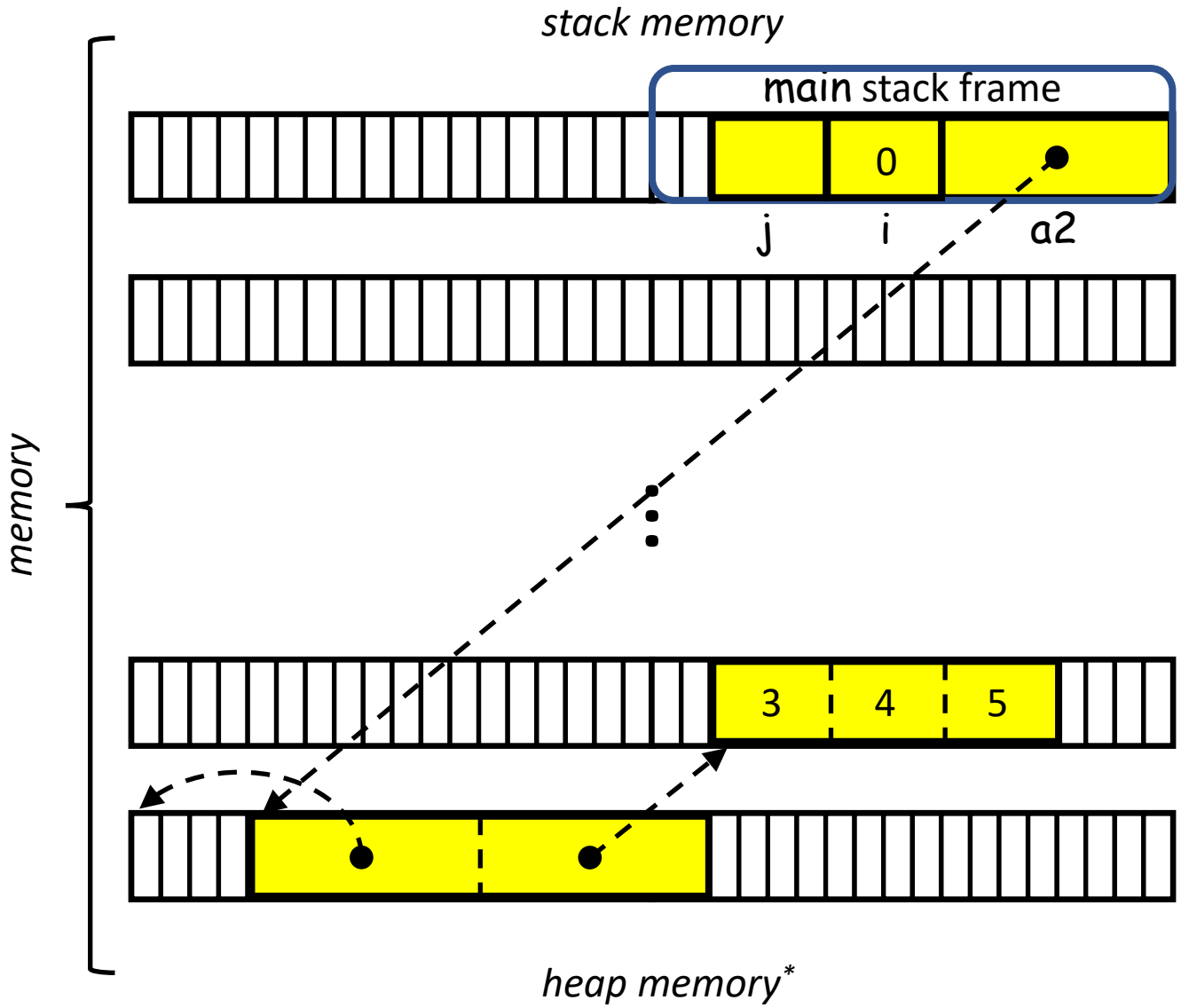

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



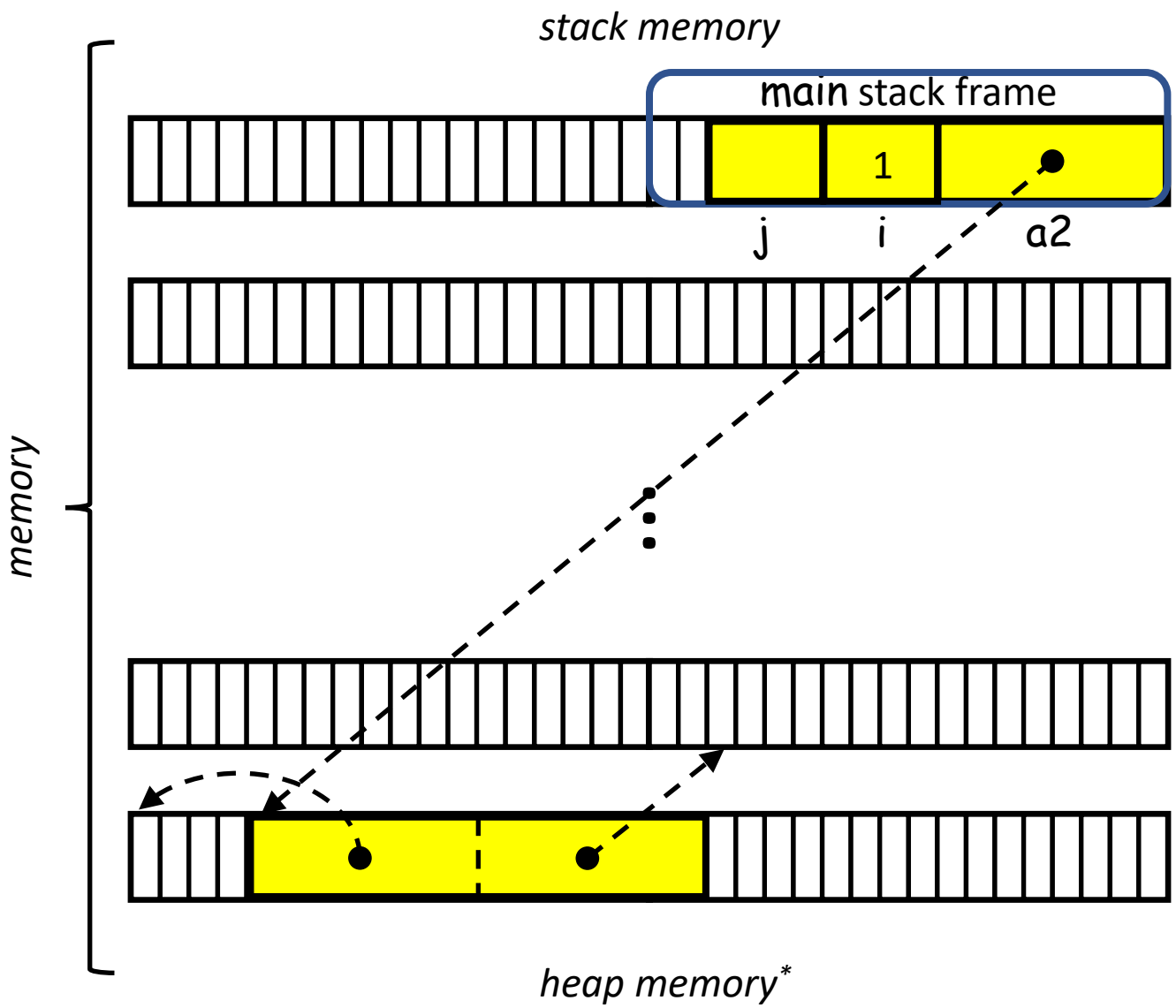

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



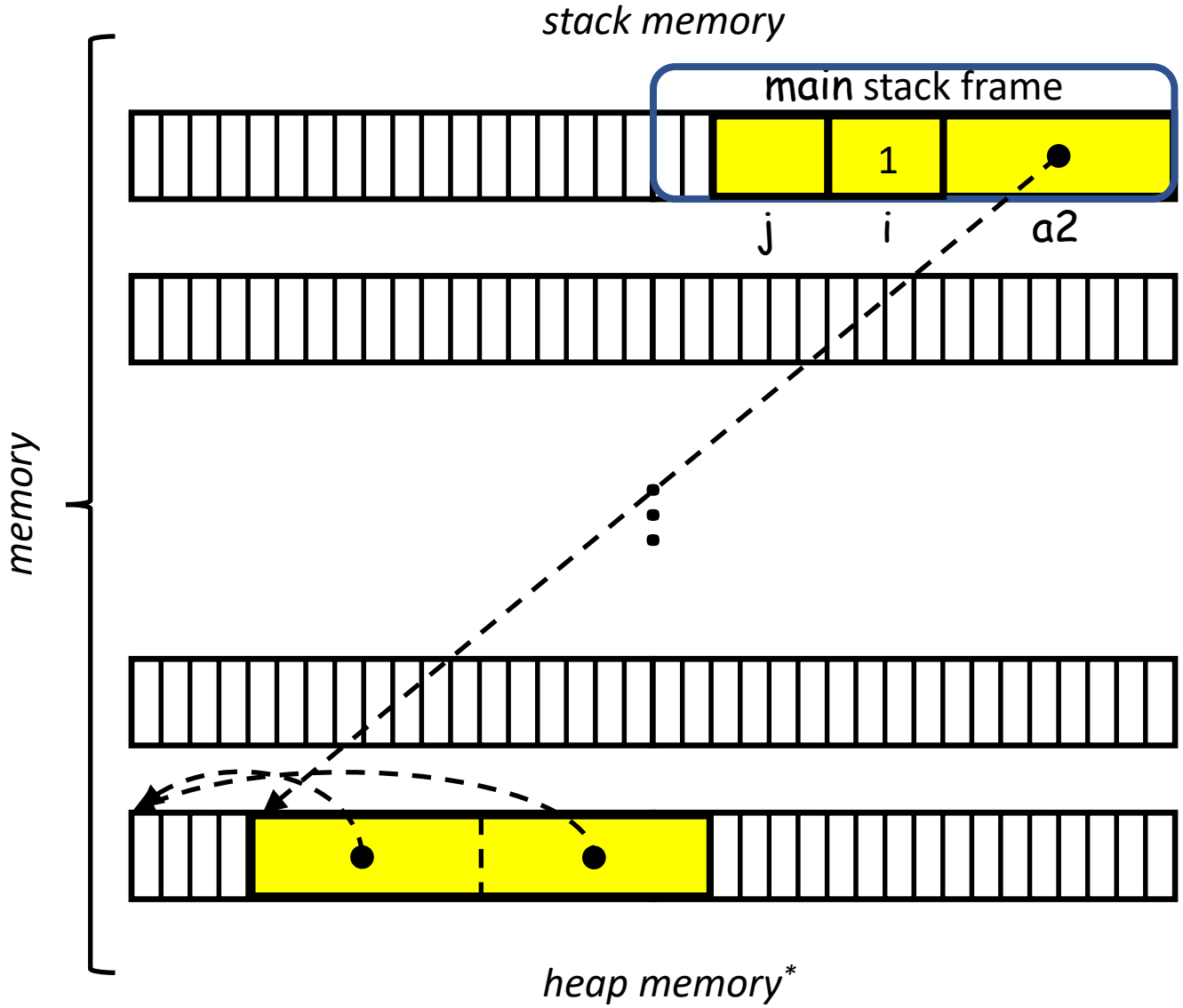

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



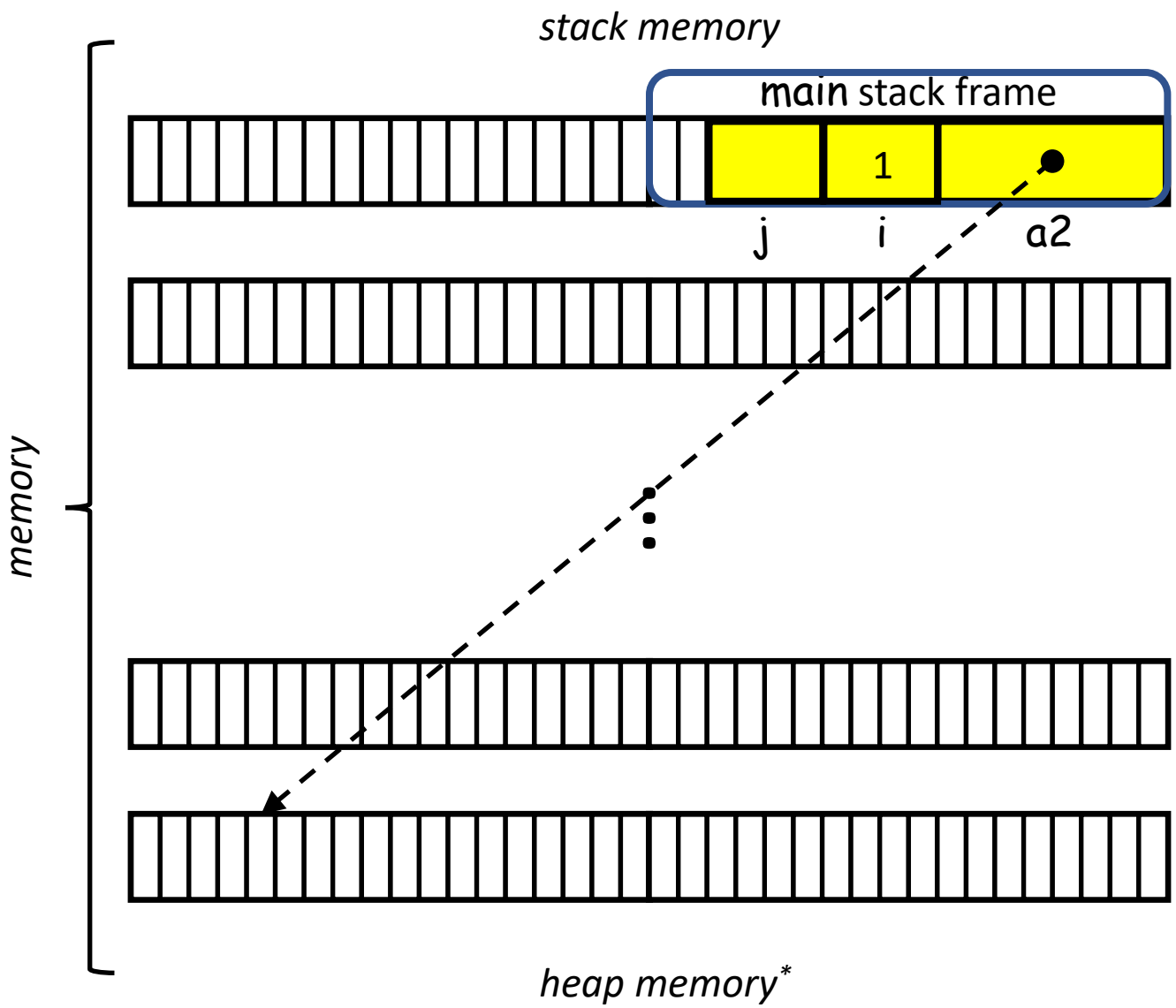

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



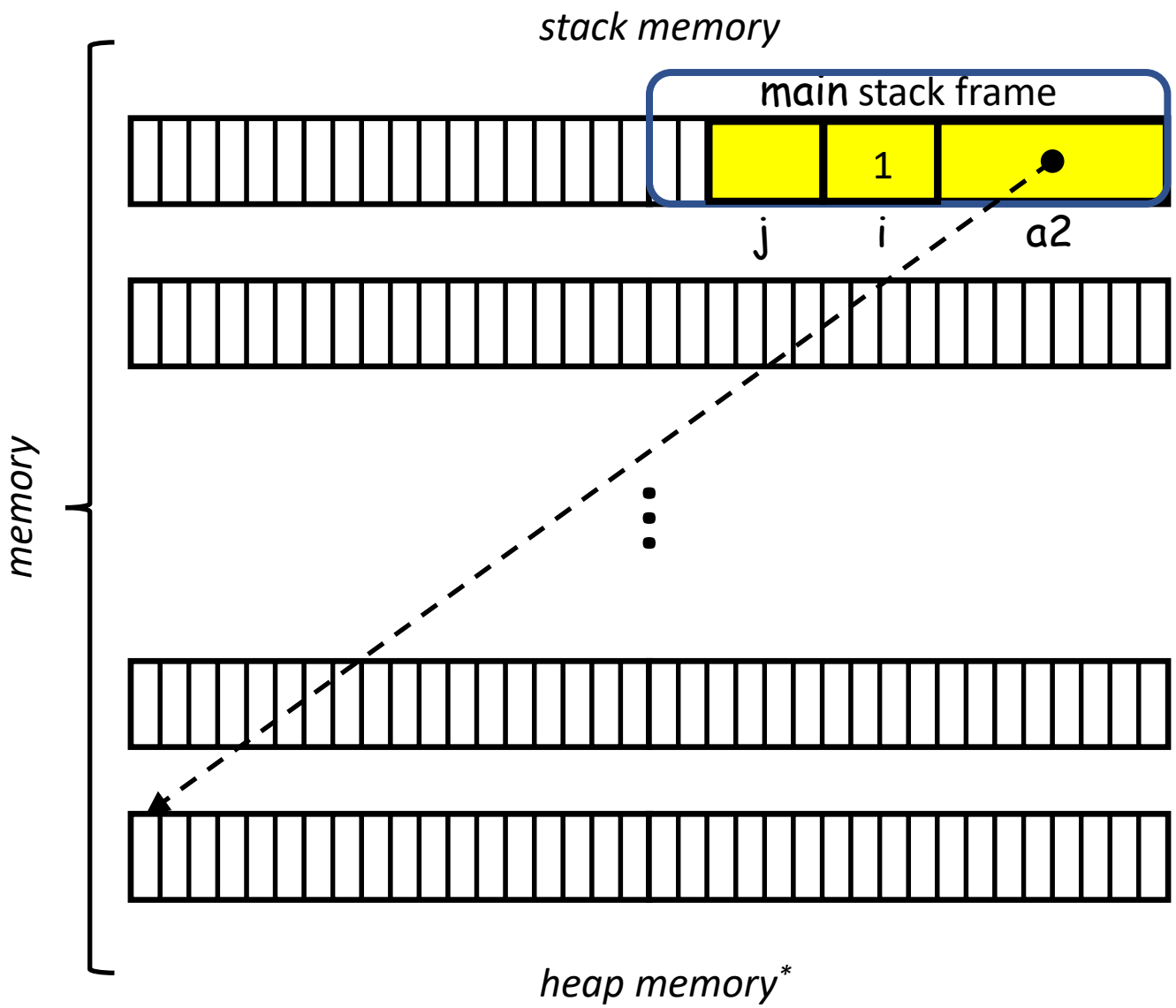

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays



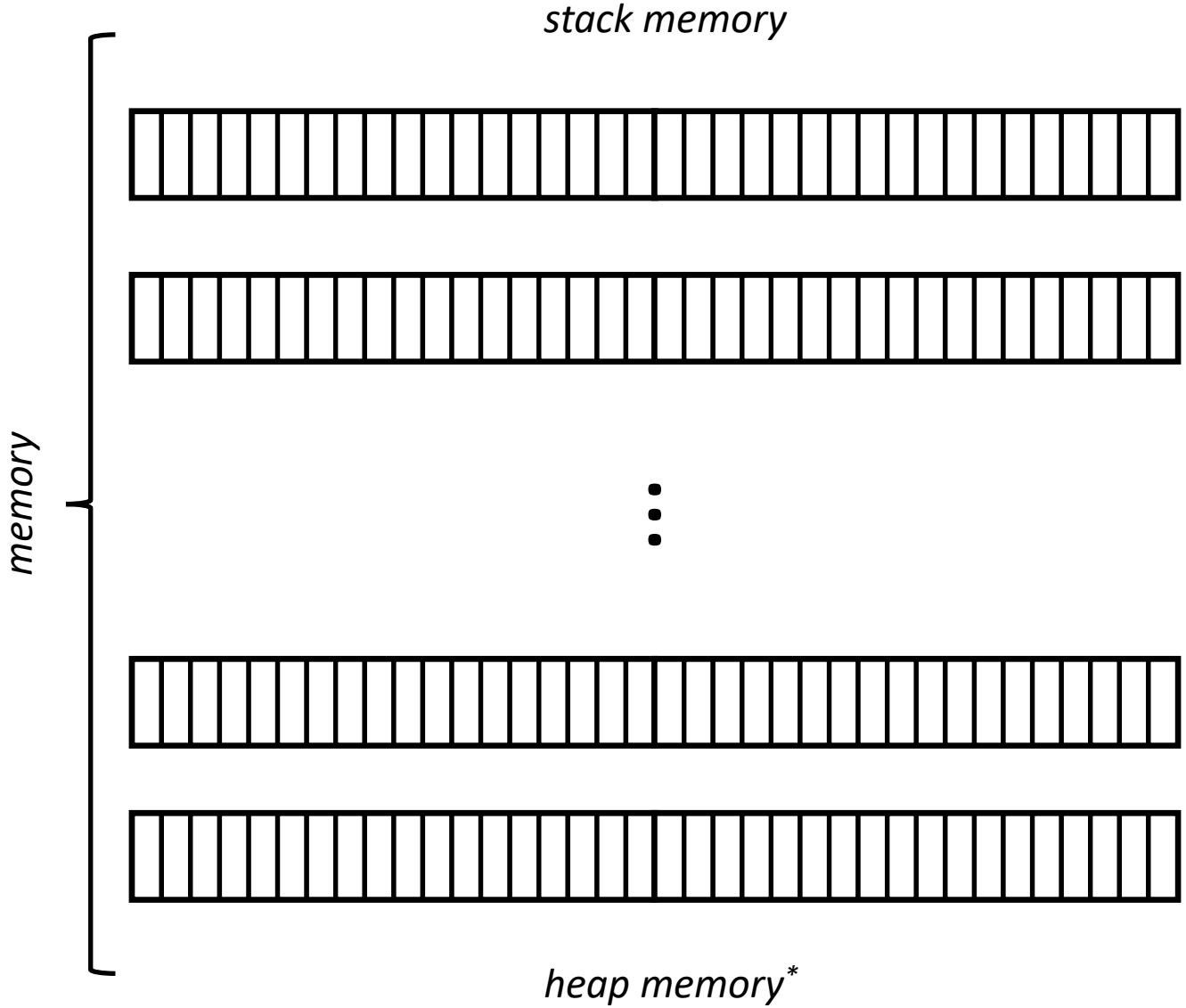

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays




```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Dynamic 2D arrays




```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
    for( int i=0 ; i<2 ; i++ )
    {
        free( a2[i] );
        a2[i] = NULL;
    }
    free( a2 );
    a2 = NULL;
    return 0;
}
```

Outline

- Pointer operations
- Dynamic 2D arrays
- Pointers and `const`
- Review questions

Pointers and **const**

Recall:

When we use the **const** keyword, we are declaring a variable immutable.

```
#include <stdio.h>
int main( void )
{
    const int a = 5;
    a = 0;
    return 0;
}
```

```
>> gcc ...
foo.c:5:4: error: assignment of read-only variable ‘a’
    5 |   a = 0;
      |     ^
>>
```

Pointers and `const`

Q: When we use the `const` keyword with a pointer, who is immutable, the pointer or the pointee?

Pointers and **const**

A: It depends

- If the keyword **const** precedes the type, then the pointee is immutable.

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    const int *c = a;
    c[0] = b;
    return 0;
}
```

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    const int *c = a;
    c = &b;
    return 0;
}
```

Pointers and **const**

A: It depends

- If the keyword **const** precedes the type, then the pointee is immutable.

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    const int *c = a;
    c[0] = b;
    return 0;
}
```

```
>> gcc ...
foo.c: In function ‘main’:
foo.c:8:8: error: assignment of read-only location ‘*c’
    8 |   c[0] = b;
      |        ^
>>
```

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    const int *c = a;
    c = &b;
    return 0;
}
```

```
>> gcc ...
>>
```

Pointers and **const**

A: It depends

- If the keyword **const** follows the type, then the pointer is immutable.

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    int * const c = a;
    c[0] = b;
    return 0;
}
```

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    int * const c = a;
    c = &b;
    return 0;
}
```

Pointers and **const**

A: It depends

- If the keyword **const** follows the type, then the pointer is immutable.

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    int * const c = a;
    c[0] = b;
    return 0;
}
```

```
>> gcc ...
>>
```

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    int * const c = a;
    c = &b;
    return 0;
```

```
>> gcc ...
foo.c: In function ‘main’:
foo.c:8:5: error: assignment of read-only variable ‘c’
    8 |   c = &b;
      |     ^
>>
```

Pointers and **const**

A: It depends

- If the keyword **const** precedes and follows the type, both are immutable.

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    const int * const c = a;
    c[0] = b;
    return 0;
```

```
>> gcc ...
foo.c: In function ‘main’:
foo.c:8:8: error: assignment of read-only location ‘*c’
    8 |   c[0] = b;
      |        ^
>>
```

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 };
    int b = 0;
    const int * const c = a;
    c = &b;
    return 0;
```

```
>> gcc ...
foo.c: In function ‘main’:
foo.c:8:5: error: assignment of read-only variable ‘c’
    8 |   c = &b;
      |     ^
>>
```

Outline

- Pointer operations
- Dynamic 2D arrays
- Pointers and `const`
- Review questions

Review questions

1. What output is printed by the code below?

```
int arr[] = { 94, 69, 35, 72, 9 };
int *p = arr;
int *q = p + 3;
int *r = q - 1;
printf( "%d %d %d\n" , *p , *q , *r );
ptrdiff_t x = q - p;
ptrdiff_t y = r - p;
ptrdiff_t z = q - r;
printf( "%d %d %d\n" , (int)x , (int)y , (int)z );
ptrdiff_t m = p - q;
printf( "%d\n" , (int)m );
int c = ( p<q );
int d = ( q<p );
printf( "%d %d\n" , c , d );
```

Review questions

2. Assume that `arr` is an array of 4 `int` elements. Is the code

```
int *p = arr + 5;
```

legal?

Review questions

3. Assume that arr is an array of 4 int elements. Is the code

```
int *p = arr + 5;
printf( "%d\n" , *p );
```

   legal?

Review questions

4. What output is printed by the code below?

```
#include <stdio.h>
int sum( int a[] , int n )
{
        int x = 0;
        for ( int i=0 ; i<n ; i++ )  x += a[i];
        return x;
}
int main( void )
{
        int data[] = { 23 , 59 , 82 , 42 , 67 , 89 , 76 , 44 , 85 , 81  };
        int result = sum( data + 3 , 4 );
        printf( "result=%d\n" , result );
        return 0;
}
```

Exercise 4-3

- Website -> Course Materials -> Ex4-3