# Intermediate Programming

## Day 12

# Outline

- Exercise 11
- Pointer operations
- Dynamic 2D arrays
- Pointers and `const`
- Review questions

# Exercise 11

`pairwise_sum.c`:

valgrind to the rescue

```
>> valgrind ./a.out
...
==2131554== Command: ./a.out
==2131554==
==2131554== Invalid write of size 4
==2131554==    at 0x40120E: pairwise_sum (pairwise_sum.c:35)
==2131554==    by 0x4012B5: main (pairwise_sum.c:50)
==2131554==  Address 0x4a8d050 is 0 bytes after a block of size 16 alloc'd
==2131554==    at 0x484186F: malloc (vg_replace_malloc.c:381)
==2131554==    by 0x4011A2: pairwise_sum (pairwise_sum.c:28)
==2131554==    by 0x4012B5: main (pairwise_sum.c:50)
==2131554==
...
```

# Exercise 11

`pairwise_sum.c`:

valgrind to the rescue

*pairwise_sum.c*

```
…
23.  int *pairwise_sum( int *array , int length ){
…
28.        int *fresh = malloc( sizeof(int) * ( length-1 ) );
…
32.        // do the pairwise sum into "fresh"
33.        for( int i=0 ; i<length ; i++ )
34.        {
35.              fresh[i] = array[i] + array[i+1];
36.        }
37.        return fresh;
38.  }
…
```

```
>> valgrind ./a.out
...
==2131554== Command: ./a.out
==2131554==
==2131554== Invalid write of size 4
==2131554==    at 0x40120E: pairwise_sum (pairwise_sum.c:35)
==2131554==    by 0x4012B5: main (pairwise_sum.c:50)
==2131554==  Address 0x4a8d050 is 0 bytes after a block of size 16 alloc'd
==2131554==    at 0x484186F: malloc (vg_replace_malloc.c:381)
==2131554==    by 0x4011A2: pairwise_sum (pairwise_sum.c:28)
==2131554==    by 0x4012B5: main (pairwise_sum.c:50)
==2131554==
...
```

# Exercise 11

`pairwise_sum.c`:

valgrind to the rescue

*pairwise_sum.c*

```
…
23.  int *pairwise_sum( int *array , int length ){
…
28.        int *fresh = malloc( sizeof(int) * ( length-1 ) );
…
32.        // do the pairwise sum into "fresh"
33.        for( int i=0 ; i<length ; i++ )
34.        {
35.              fresh[i] = array[i] + array[i+1];
36.        }
37.        return fresh;
38.  }
…
```

```
>> valgrind ./a.out
...
==2131554== Command: ./a.out
==2131554==
==2131554== Invalid write of size 4
==2131554==    at 0x40120E: pairwise_sum (pairwise_sum.c:35)
==2131554==    by 0x4012B5: main (pairwise_sum.c:50)
==2131554==  Address 0x4a8d050 is 0 bytes after a block of size 16 alloc'd
==2131554==    at 0x484186F: malloc (vg_replace_malloc.c:381)
==2131554==    by 0x4011A2: pairwise_sum (pairwise_sum.c:28)
==2131554==    by 0x4012B5: main (pairwise_sum.c:50)
==2131554==
...
```

# Exercise 11

`pairwise_sum.c`:

valgrind to the rescue

*pairwise_sum.c*

```
…
23.  int *pairwise_sum( int *array , int length ){
…
28.        int *fresh = malloc( sizeof(int) * ( length-1 ) );
…
32.        // do the pairwise sum into "fresh"
33.        for( int i=0 ; i<length-1 ; i++ )
34.        {
35.              fresh[i] = array[i] + array[i+1];
36.        }
37.        return fresh;
38.  }
…
```

```
>> valgrind ./a.out
...
==2186254== HEAP SUMMARY:
==2186254==     in use at exit: 16 bytes in 1 blocks
==2186254==   total heap usage: 4 allocs, 3 frees, 1,068 bytes allocated
==2186254==
==2186254== LEAK SUMMARY:
==2186254==    definitely lost: 16 bytes in 1 blocks
==2186254==    indirectly lost: 0 bytes in 0 blocks
==2186254==      possibly lost: 0 bytes in 0 blocks
==2186254==    still reachable: 0 bytes in 0 blocks
==2186254==         suppressed: 0 bytes in 0 blocks
==2186254== Rerun with --leak-check=full to see details of leaked memory
==2186254==
...
```

# Exercise 11

`pairwise_sum.c`:

valgrind to the rescue

```
>> valgrind ./a.out
...
==2186254== HEAP SUMMARY:
==2186254==     in use at exit: 16 bytes in 1 blo
==2186254==   total heap usage: 4 allocs, 3 frees
==2186254==
==2186254== LEAK SUMMARY:
==2186254==    definitely lost: 16 bytes in 1 blo
==2186254==    indirectly lost: 0 bytes in 0 bloc
==2186254==      possibly lost: 0 bytes in 0 bloc
==2186254==    still reachable: 0 bytes in 0 blocks
==2186254==         suppressed: 0 bytes in 0 blocks
==2186254== Rerun with --leak-check=full to see details of leaked memory
==2186254==
...
```

*pairwise_sum.c*

```
…
23.  int *pairwise_sum( int *array , int length ){
…
28.        int *fresh = malloc( sizeof(int) * ( length-1 ) );
…
38.  }
…
49.  int main(){
…
52.        int *pairsum1 = pairwise_sum(array, 5);
…
57.        free(pairsum1);
58.
59.        int *pairsum2 = pairwise_sum(pairwise_sum(array, 5), 4);
…
63.        free(pairsum2);
64.
65.        return 0;
66.  }
```

# Exercise 11

`pairwise_sum.c`:

valgrind to the rescue

*pairwise_sum.c*

```
…
23.  int *pairwise_sum( int *array , int length ){
…
28.        int *fresh = malloc( sizeof(int) * ( length-1 ) );
…
38.  }
…
49.  int main(){
…
52.        int *pairsum1 = pairwise_sum(array, 5);
…
57.
58.        int *pairsum2 = pairwise_sum(pairsum1, 4);
59.        free(pairsum1);
…
63.        free(pairsum2);
64.
65.        return 0;
66.  }
```

```
>> valgrind ./a.out
...
==2192565== HEAP SUMMARY:
==2192565==     in use at exit: 0 bytes in 0 blocks
==2192565==   total heap usage: 3 allocs, 3 frees, 1,052 bytes allocated
==2192565==
==2192565== All heap blocks were freed -- no leaks are possible
...
```

# Exercise 11

`primes.c`:

valgrind to the rescue

```
>> valgrind ./a.out
...
==2193150== Invalid read of size 4
==2193150==    at 0x4012DD: main (primes.c:64)
==2193150==  Address 0x4b6e070 is 9,808 bytes inside a block of size 10,240 free'd
==2193150==    at 0x48466AF: realloc (vg_replace_malloc.c:1437)
==2193150==    by 0x40122A: set_primes (primes.c:44)
==2193150==    by 0x4012AC: main (primes.c:62)
==2193150==  Block was alloc'd at
==2193150==    at 0x48466AF: realloc (vg_replace_malloc.c:1437)
==2193150==    by 0x40122A: set_primes (primes.c:44)
==2193150==    by 0x4012AC: main (primes.c:62)
...
```

# Exercise 11

`primes.c`:

valgrind to the resc

*primes.c*

```
...
32.  int set_primes( int *list , int capacity ){
33.          int idx;
...
44.                          list = realloc(list, capacity * sizeof(int));
...
49.          return idx;
50.  }
...
53.  int main() {
...
57.          int *list = malloc(capacity * sizeof(int));
...
62.          int prime_count = set_primes( list , capacity );
63.          printf( "Found %d primes in the range [2,%d)\n" , prime_count , MAX_CANDIDATE );
64.          printf( "First and last primes are %d and %d\n" , list[0] , list[ prime_count-1 ] );
...
67.  }
```

```
>> valgrind ./a.out
...
==2193150== Invalid read of size 4
==2193150==    at 0x4012DD: main (primes.c:67)
==2193150==  Address 0x4b6e070 is 9,808 bytes inside a block of size 10,240 free'd
==2193150==    at 0x48466AF: realloc (vg_replace_malloc.c:1437)
==2193150==    by 0x40122A: set_primes (primes.c:44)
==2193150==    by 0x4012AC: main (primes.c:62)
==2193150==  Block was alloc'd at
==2193150==    at 0x48466AF: realloc (vg_replace_malloc.c:1437)
==2193150==    by 0x40122A: set_primes (primes.c:44)
==2193150==    by 0x4012AC: main (primes.c:62)
...
```

# Exercise 11

`primes.c`:

valgrind to the resc

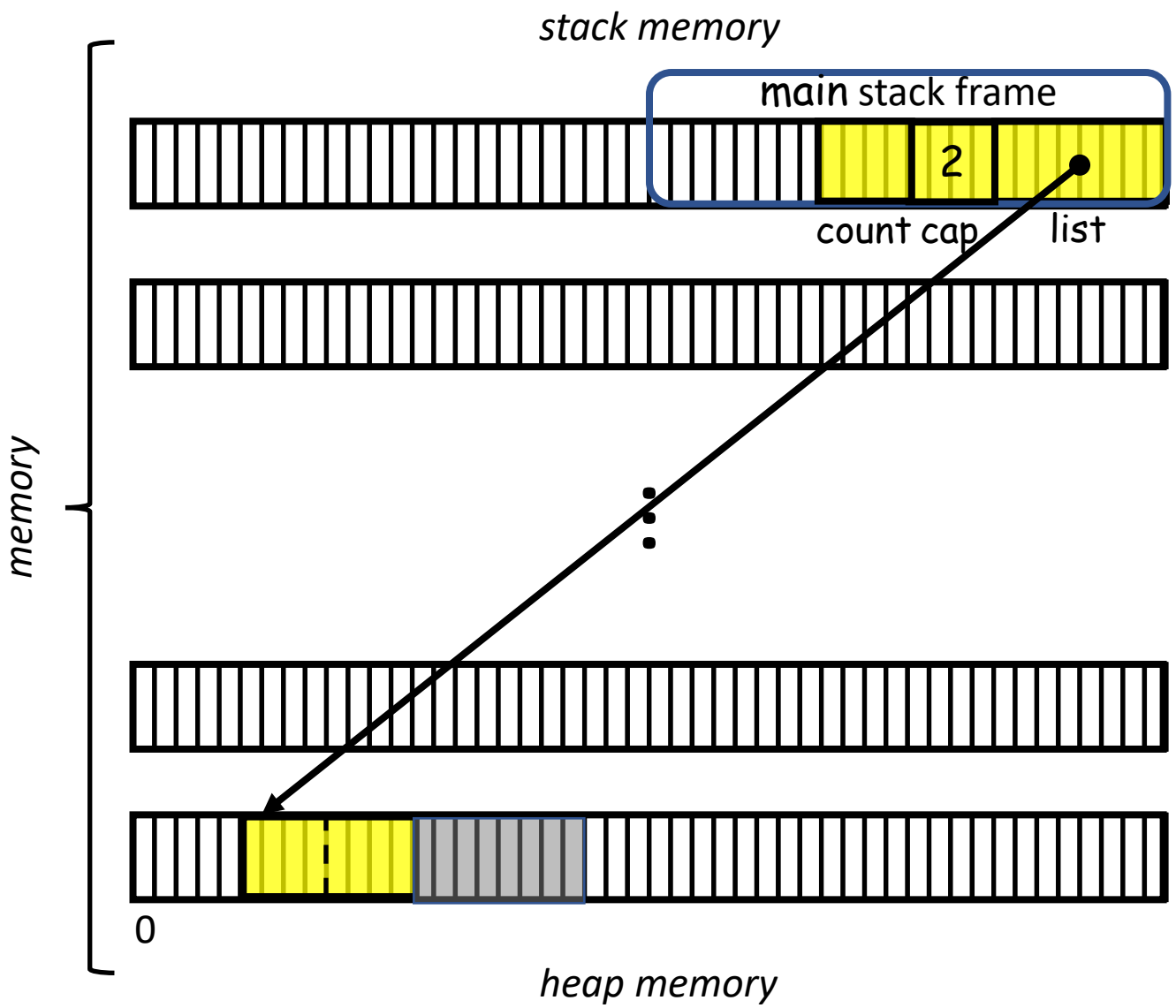
*primes.c*

```
…
32.  int set_primes( int *list , int capacity ){
33.          int idx;
…
44.                          list = realloc(list, capacity * sizeof(int));
…
49.          return idx;
50.  }
…
53.  int main() {
…
57.          int *list = malloc(capacity * sizeof(int));
…
62.          int prime_count = set_primes( list , capacity );
63.          printf( "Found %d primes in the range [2,%d)\n" , prime_count , MAX_CANDIDATE );
64.          printf( "First and last primes are %d and %d\n" , list[0] , list[ prime_count-1 ] );
…
67.  }
```

```
>> valgrind ./a.out
...
==2193150== Invalid read of size 4
==2193150==    at 0x4012DD: main (primes.c:67)
==2193150==  Address 0x4b6e070 is 9,808 bytes inside a block of size 10,240 free'd
==2193150==    at 0x48466AF: realloc (vg_replace_malloc.c:1437)
==2193150==    by 0x40122A: set_primes (primes.c:44)
==2193150==    by 0x4012AC: main (primes.c:62)
==2193150==  Block was alloc'd at
==2193150==    at 0x48466AF: realloc (vg_replace_malloc.c:1437)
==2193150==    by 0x4012
==2193150==    by 0x4012
...
```

The problem is that in line 44, the value of **list** can change (if **realloc** cannot extend the memory). But the **main** function doesn't know this. (It's copy of list does not change).

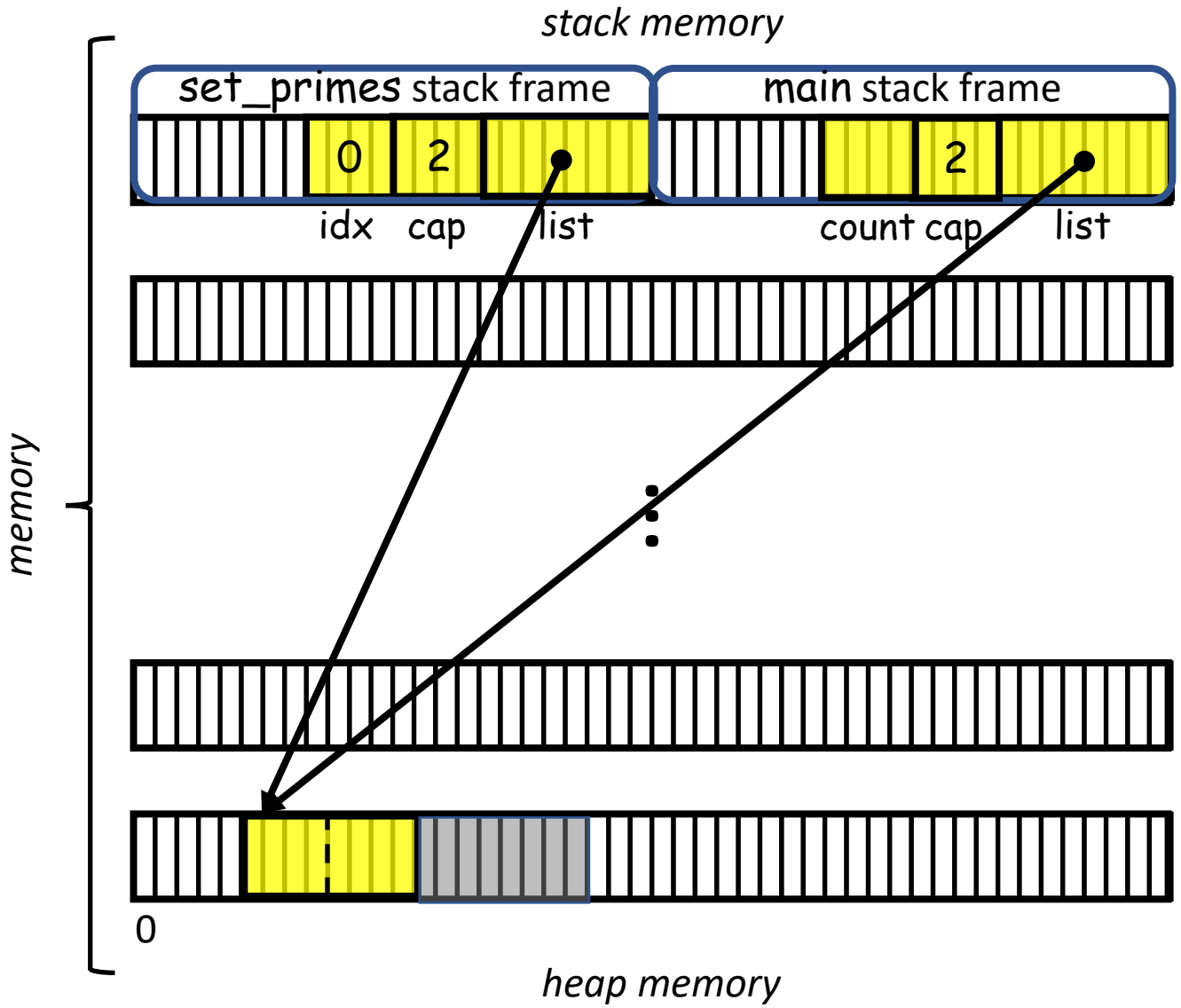
# Exercise 11




*primes.c*

```
...
32.  int set_primes( int *list , int cap ){
33.          int idx=0;
...
44.                          list = realloc( list , cap * sizeof(int));
...
49.          return idx;
50.  }
...
53.  int main() {
...
57.          int *list = malloc( cap * sizeof(int));
...
62.          int count = set_primes( list , cap );
...
67.  }
```

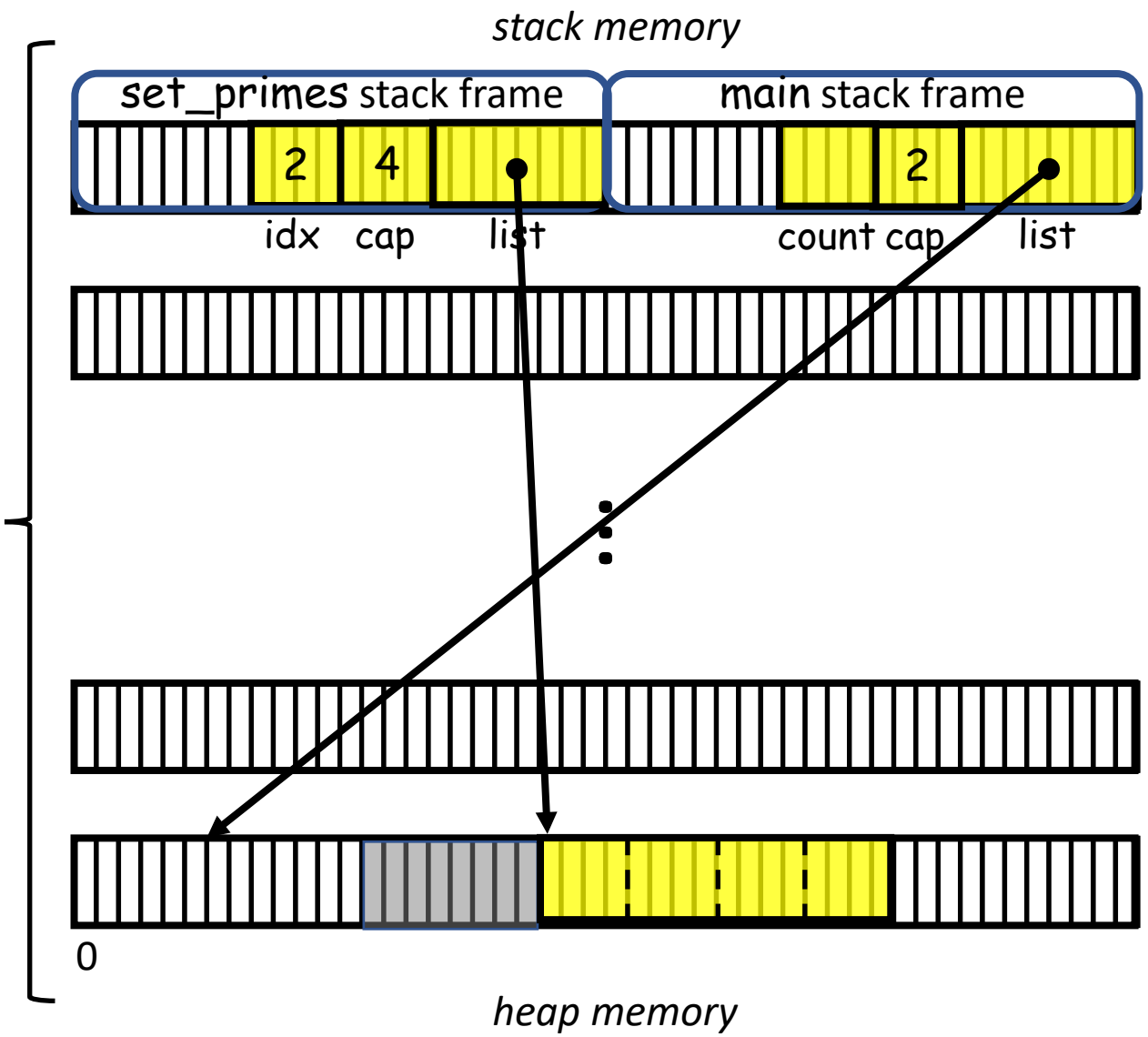
# Exercise 11




*primes.c*

```
...
32.  int set_primes( int *list , int cap ){
33.          int idx=0;
...
44.                          list = realloc( list , cap * sizeof(int));
...
49.          return idx;
50.  }
...
53.  int main() {
...
57.          int *list = malloc( cap * sizeof(int));
...
62.          int count = set_primes( list , cap );
...
67.  }
```

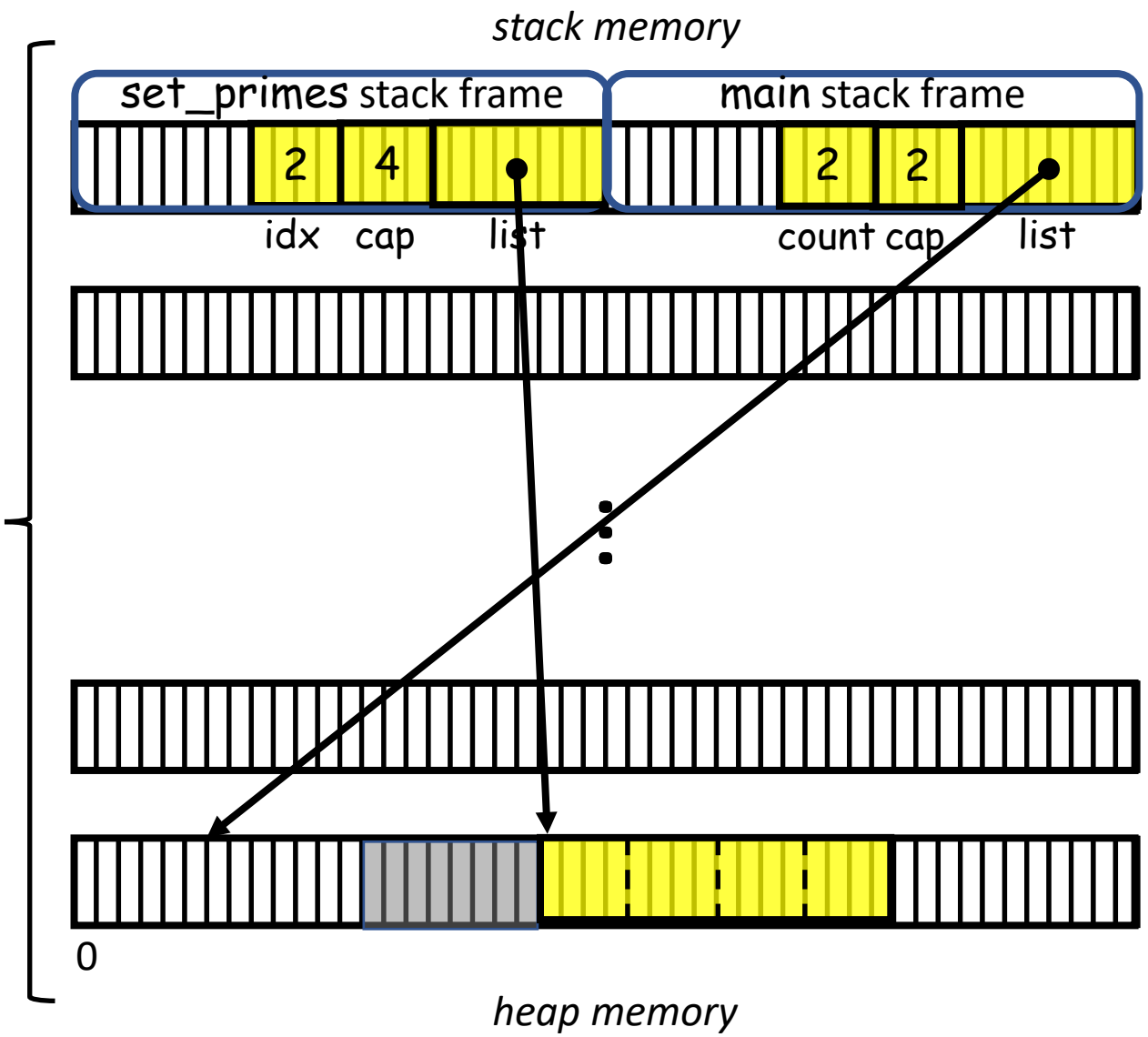
# Exercise 11




*primes.c*

```
...
32.  int set_primes( int *list , int cap ){
33.          int idx=0;
...
44.                          list = realloc( list , cap * sizeof(int));
...
49.          return idx;
50.  }
...
53.  int main() {
...
57.          int *list = malloc( cap * sizeof(int));
...
62.          int count = set_primes( list , cap );
...
67.  }
```

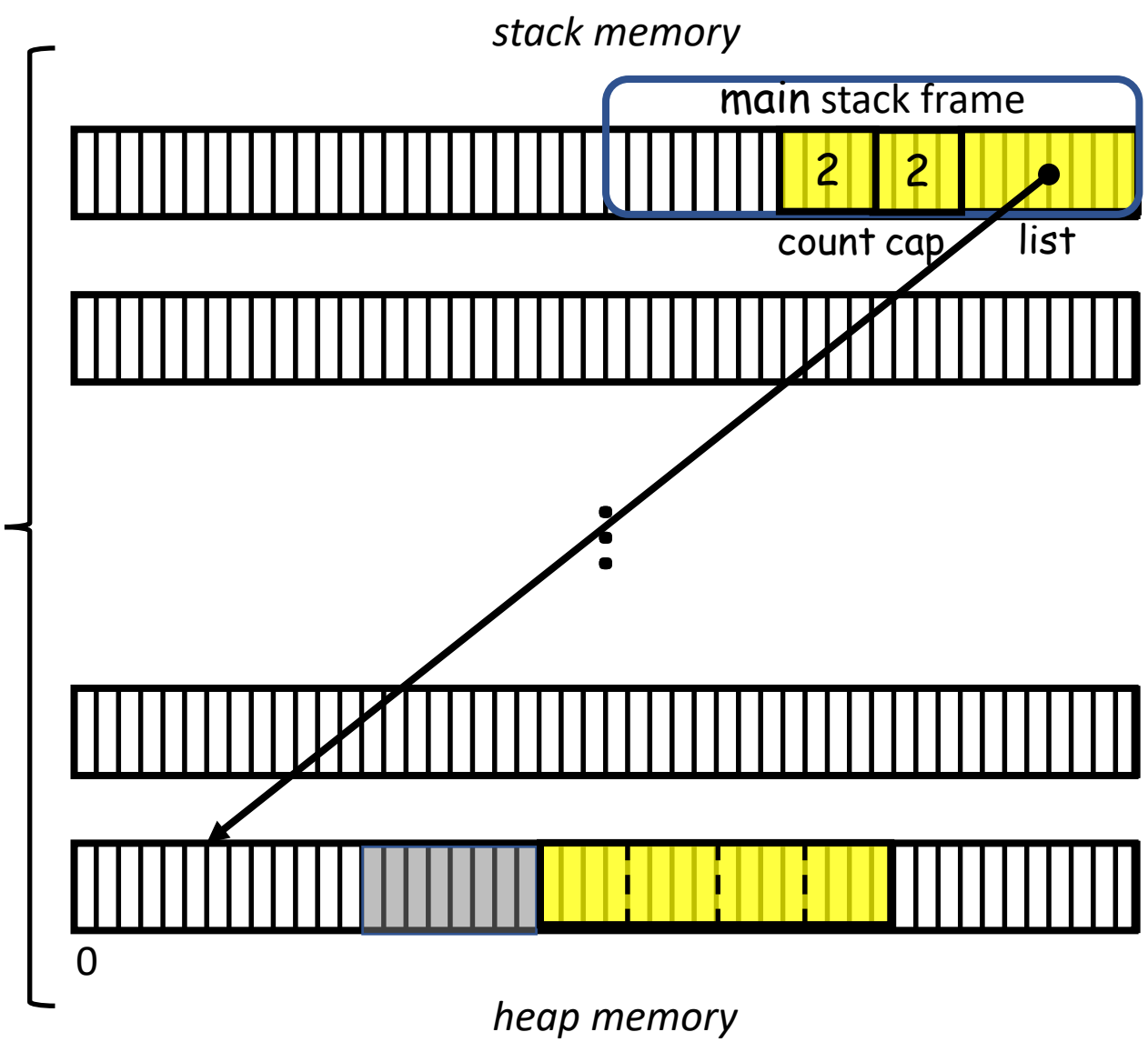
# Exercise 11




*primes.c*

```
...
32.  int set_primes( int *list , int cap ){
33.          int idx=0;
...
44.                          list = realloc( list , cap * sizeof(int));
...
49.          return idx;
50.  }
...
53.  int main() {
...
57.          int *list = malloc( cap * sizeof(int));
...
62.          int count = set_primes( list , cap );
...
67.  }
```

# Exercise 11




*primes.c*

```
...
32.  int set_primes( int *list , int cap ){
33.          int idx=0;
...
44.                          list = realloc( list , cap * sizeof(int));
...
49.          return idx;
50.  }
...
53.  int main() {
...
57.          int *list = malloc( cap * sizeof(int));
...
62.          int count = set_primes( list , cap );
...
67.  }
```

# Exercise 11

`primes.c`:

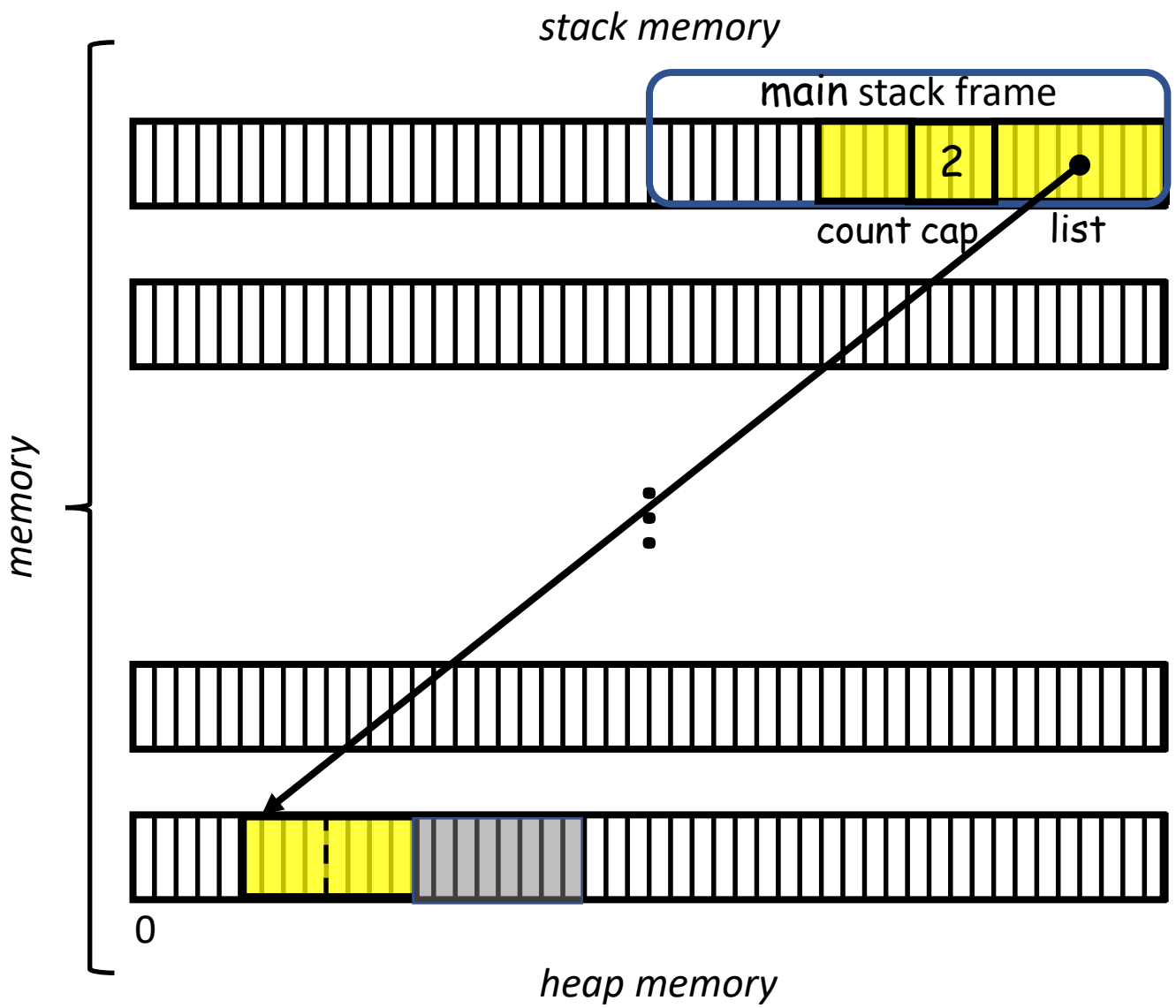
valgrind to the resc

*primes.c*

```
...
32.  int *set_primes( int *list , int capacity , int *p_prime_count ){
33.          int idx;
...
44.                          list = realloc(list, capacity * sizeof(int));
...
49.          *p_prime_count = idx;
50.          return list;
51.  }
...
54.  int main() {
...
58.          int *list = malloc(capacity * sizeof(int));
...
63.          int prime_count;
64.          list = set_primes( list , capacity , &prime_count );
65.          printf( "Found %d primes in the range [2,%d)\n" , prime_count , MAX_CANDIDATE );
66.          printf( "First and last primes are %d and %d\n" , list[0] , list[ prime_count-1 ] );
...
69.  }
```

```
>> valgrind ./a.out
...
==2203638== HEAP SUMMARY:
==2203638==     in use at exit: 0 bytes in 0 blocks
==2203638==   total heap usage: 12 allocs, 12 frees, 82,904 bytes allocated
==2203638==
==2203638== All heap blocks were freed -- no leaks are possible
...
```

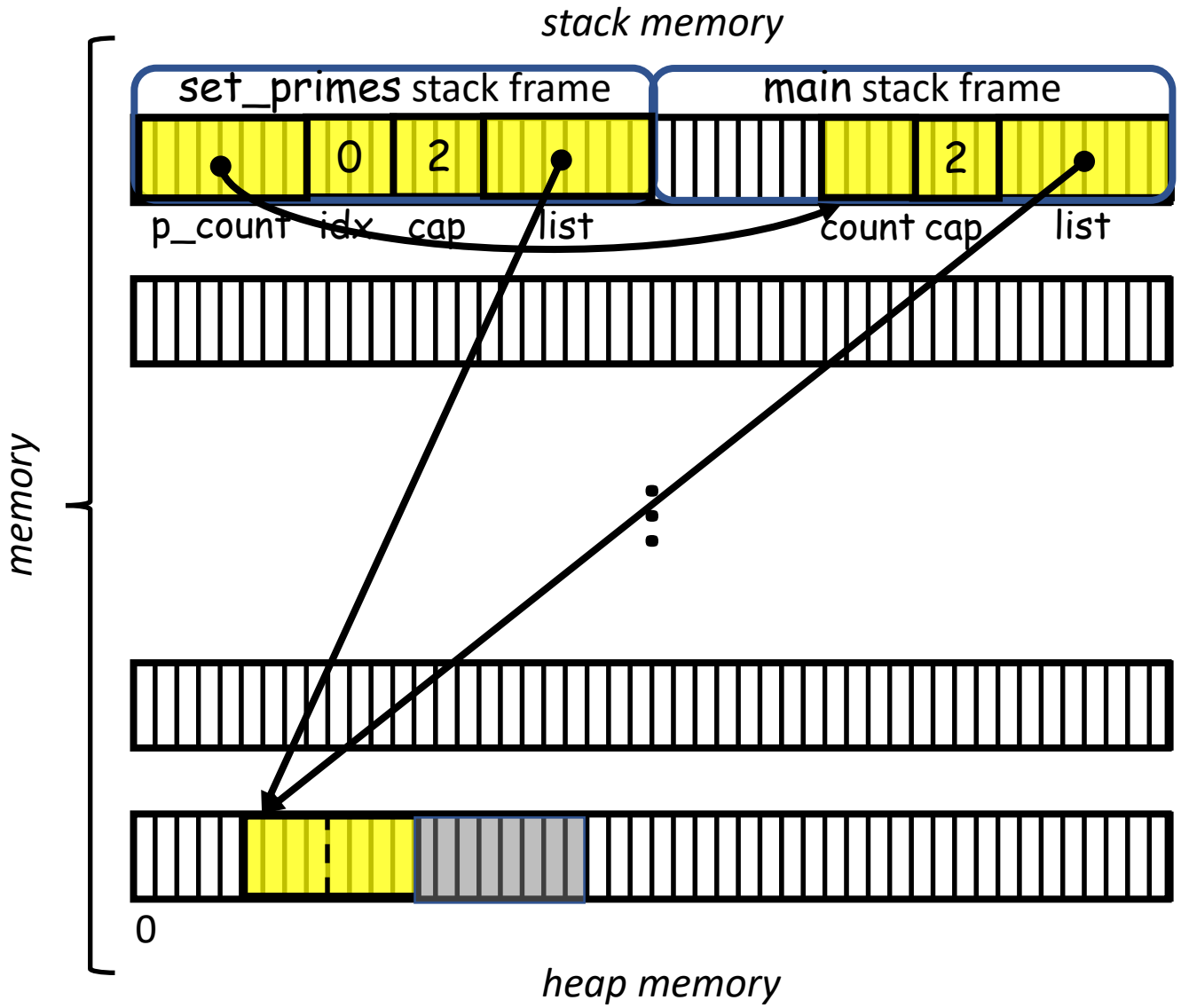
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int *list , int cap , int *p_count ){
33.          int idx=0;
...
44.                          list = realloc( list, cap * sizeof(int));
...
49.          *p_count = idx;
50.          return list;
51.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count;
64.          list = set_primes( list , cap , &count );
...
69.  }
```

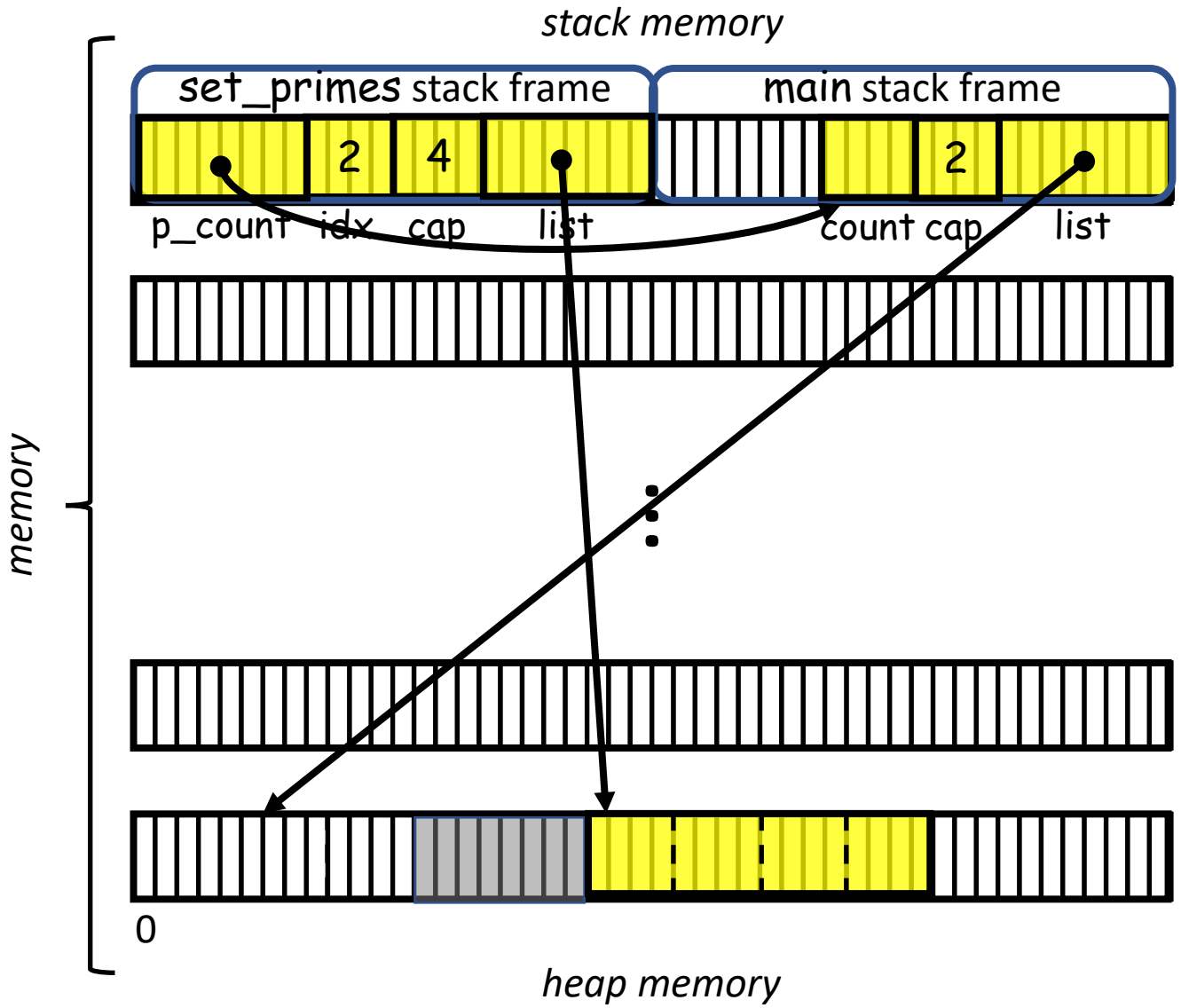
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int *list , int cap , int *p_count ){
33.          int idx=0;
...
44.                          list = realloc( list, cap * sizeof(int));
...
49.          *p_count = idx;
50.          return list;
51.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count;
64.          list = set_primes( list , cap , &count );
...
69.  }
```

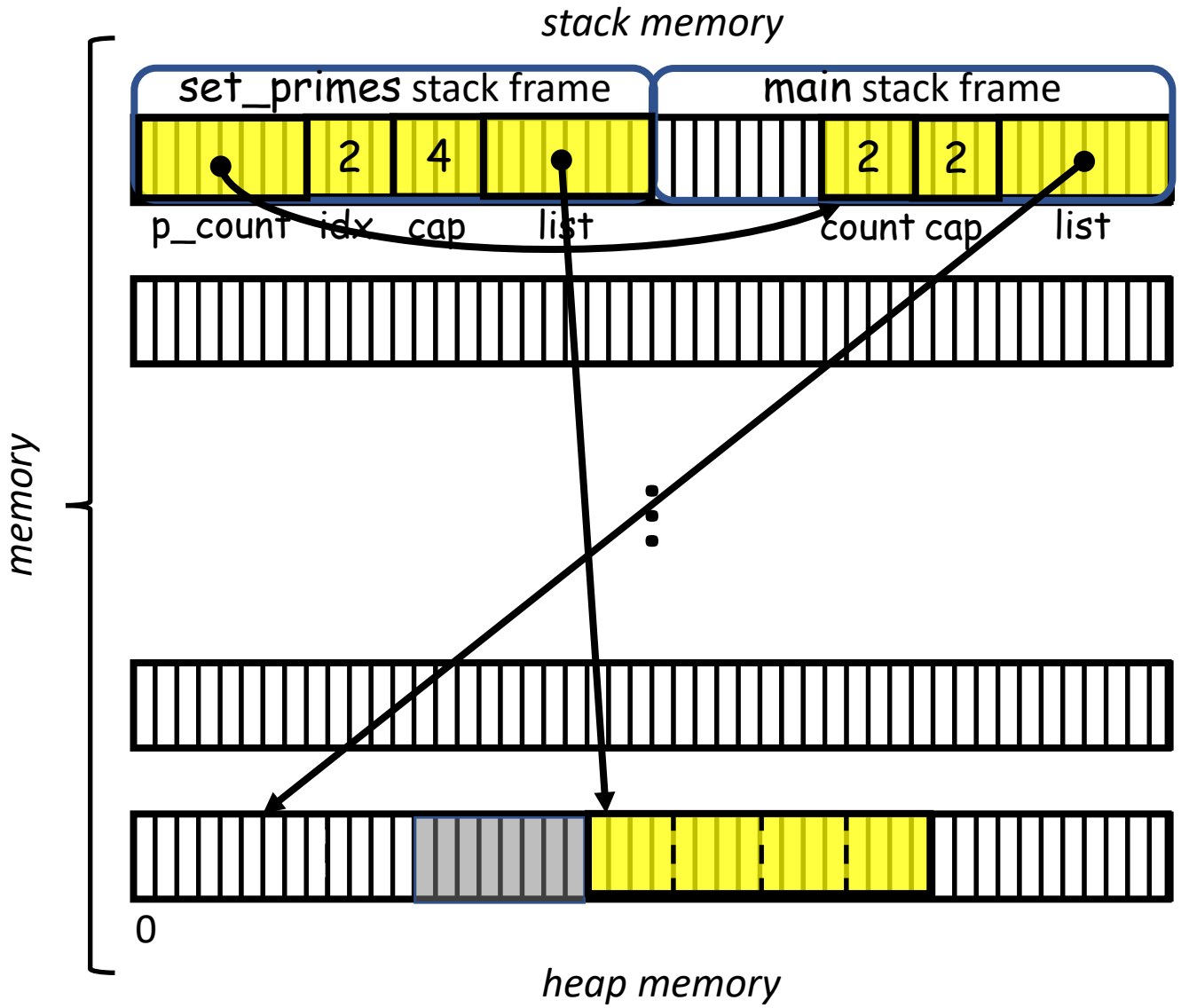
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int *list , int cap , int *p_count ){
33.          int idx=0;
...
44.                          list = realloc( list, cap * sizeof(int));
...
49.          *p_count = idx;
50.          return list;
51.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count;
64.          list = set_primes( list , cap , &count );
...
69.  }
```

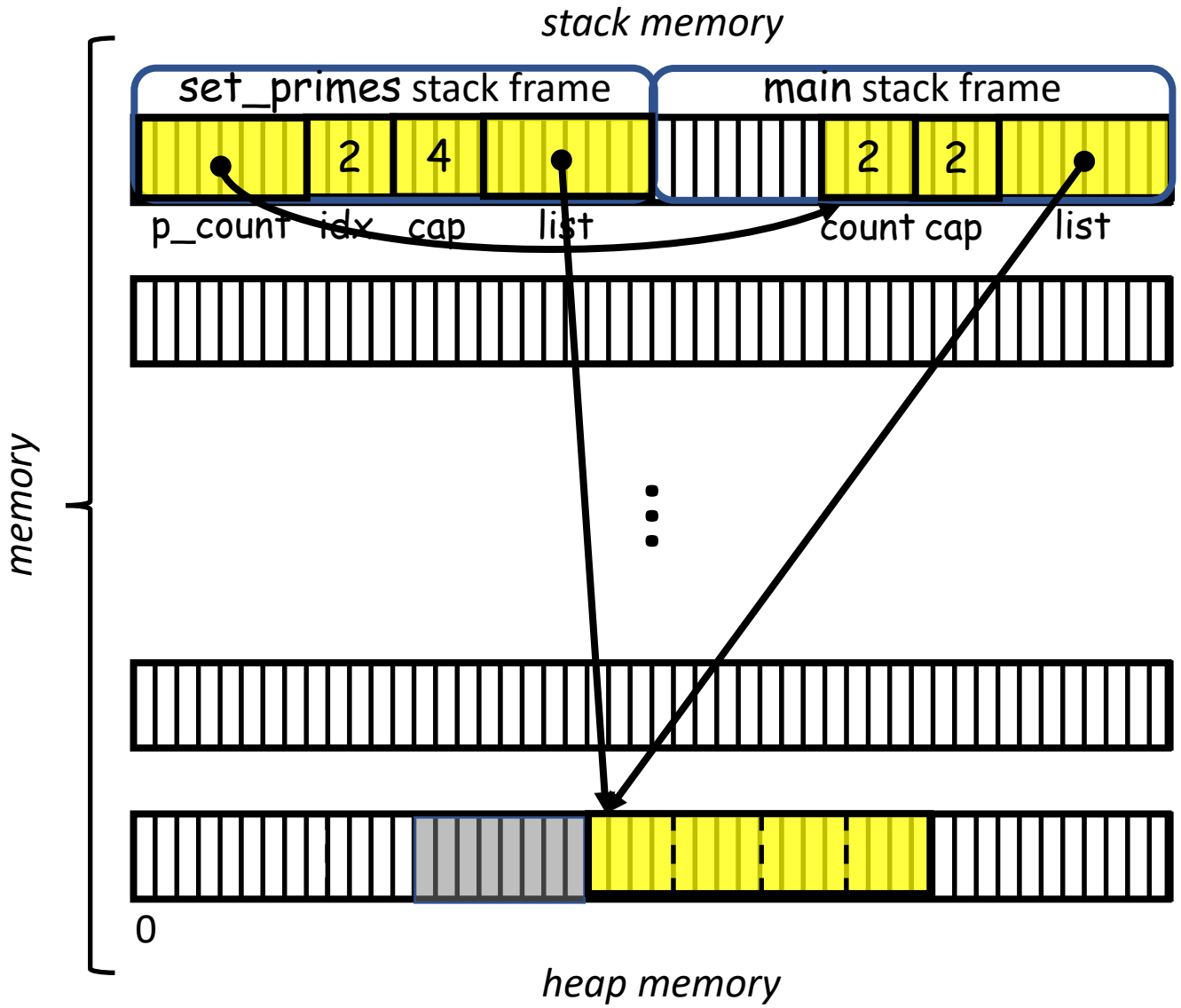
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int *list , int cap , int *p_count ){
33.          int idx=0;
...
44.                          list = realloc( list, cap * sizeof(int));
...
49.          *p_count = idx;
50.          return list;
51.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count;
64.          list = set_primes( list , cap , &count );
...
69.  }
```

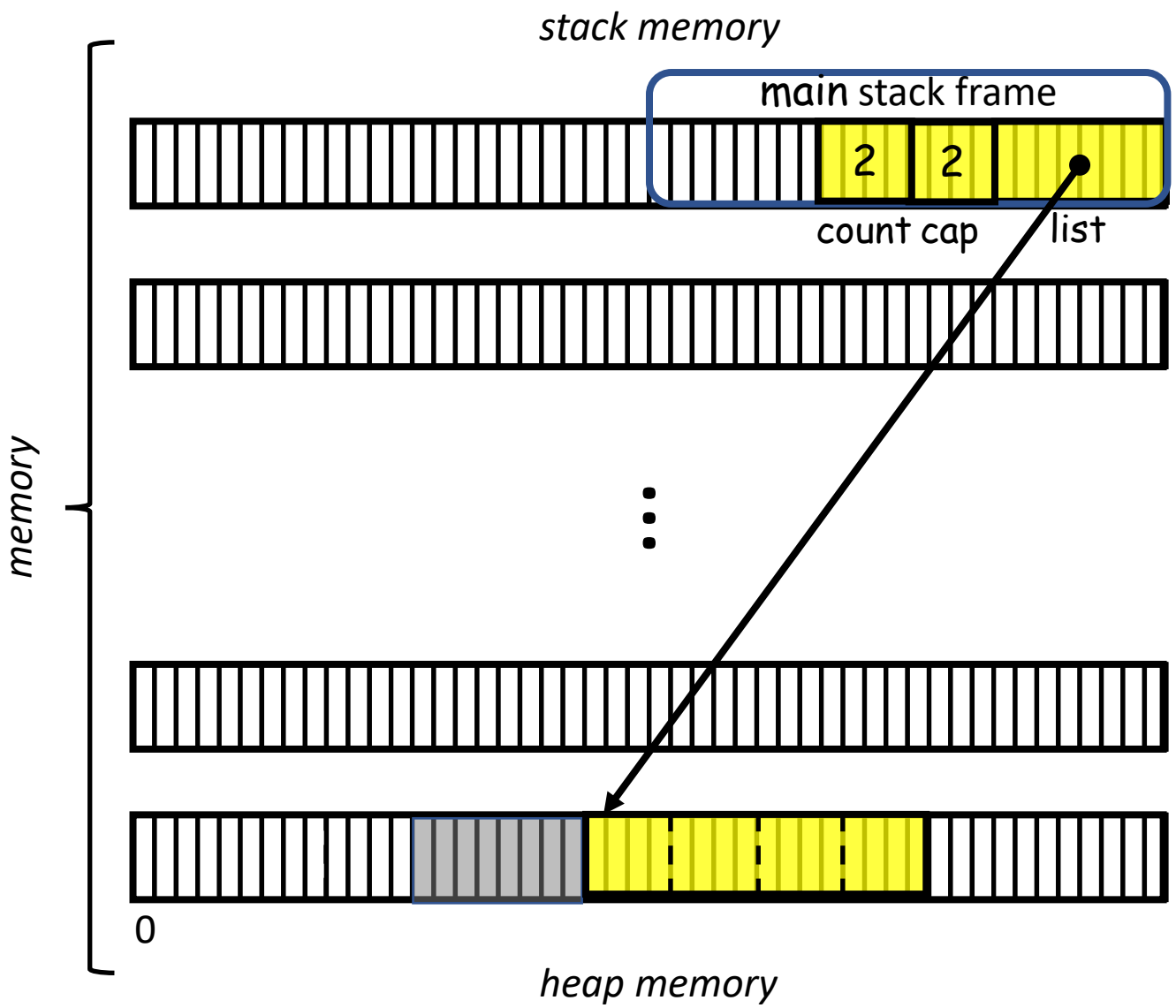
# Exercise 11




*primes.c*

```
...
32.   int *set_primes( int *list , int cap , int *p_count ){
33.           int idx=0;
...
44.                           list = realloc( list, cap * sizeof(int));
...
49.           *p_count = idx;
50.           return list;
51.   }
...
54.   int main() {
...
58.           int *list = malloc( cap * sizeof(int));
...
63.           int count;
64.           list = set_primes( list , cap , &count );
...
69.   }
```

# Exercise 11




*primes.c*

```
...
32.   int *set_primes( int *list , int cap , int *p_count ){
33.           int idx=0;
...
44.                           list = realloc( list, cap * sizeof(int));
...
49.           *p_count = idx;
50.           return list;
51.   }
...
54.   int main() {
...
58.           int *list = malloc( cap * sizeof(int));
...
63.           int count;
64.           list = set_primes( list , cap , &count );
...
69.   }
```

# Exercise 11

`primes.c`:

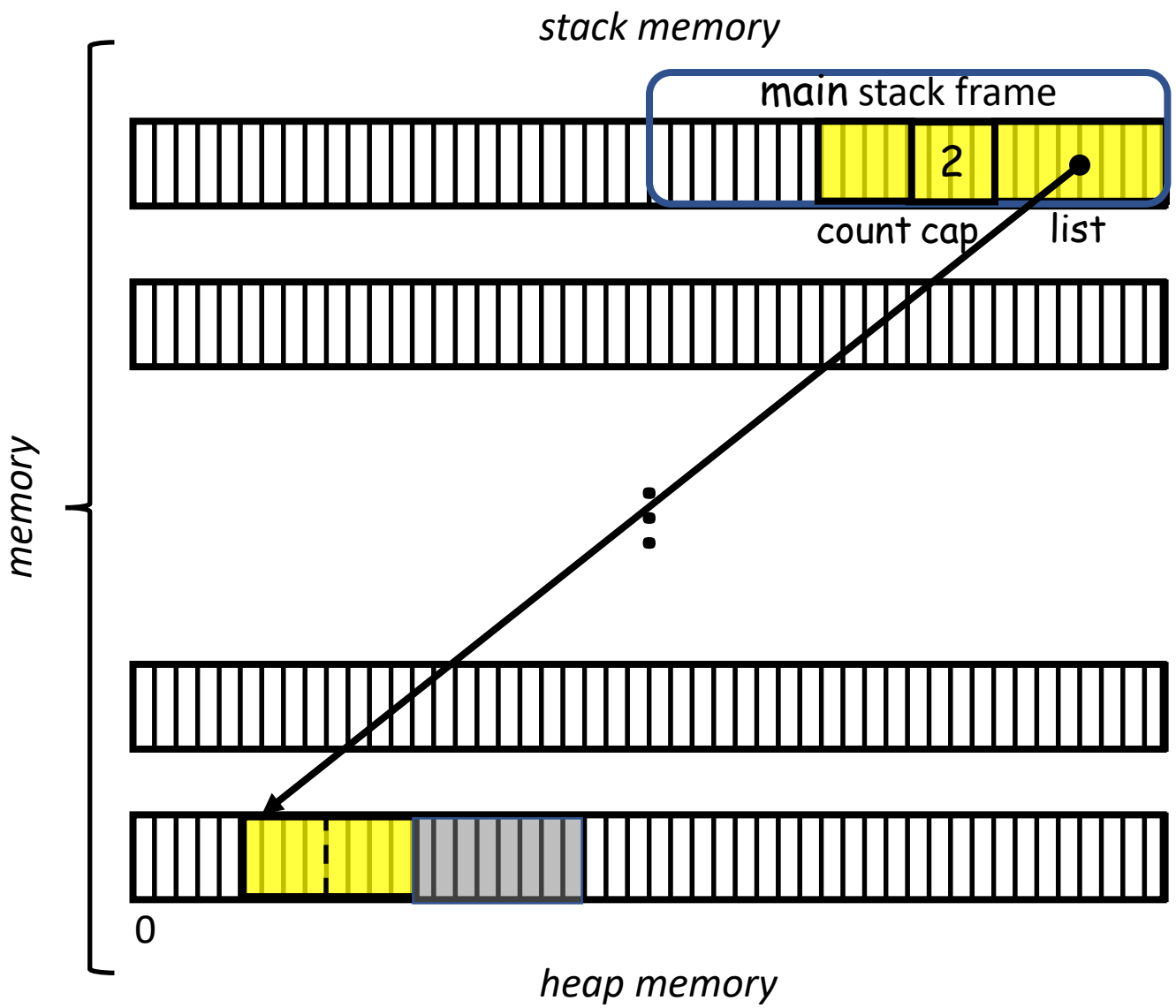
valgrind to the resc

*primes.c*

```
...
32.  int set_primes( int **p_list , int capacity ){
33.          int idx;
...
44.                          *p_list = realloc( *p_list, capacity * sizeof(int));
...
49.          return idx;
50.  }
...
54.  int main() {
...
58.          int *list = malloc(capacity * sizeof(int));
...
63.          int prime_count = set_primes( &list , capacity );
64.          printf( "Found %d primes in the range [2,%d)\n" , prime_count , MAX_CANDIDATE );
65.          printf( "First and last primes are %d and %d\n" , list[0] , list[ prime_count-1 ] );
...
69.  }
```

```
>> valgrind ./a.out
...
==2203638== HEAP SUMMARY:
==2203638==     in use at exit: 0 bytes in 0 blocks
==2203638==   total heap usage: 12 allocs, 12 frees, 82,904 bytes allocated
==2203638==
==2203638== All heap blocks were freed -- no leaks are possible
...
```

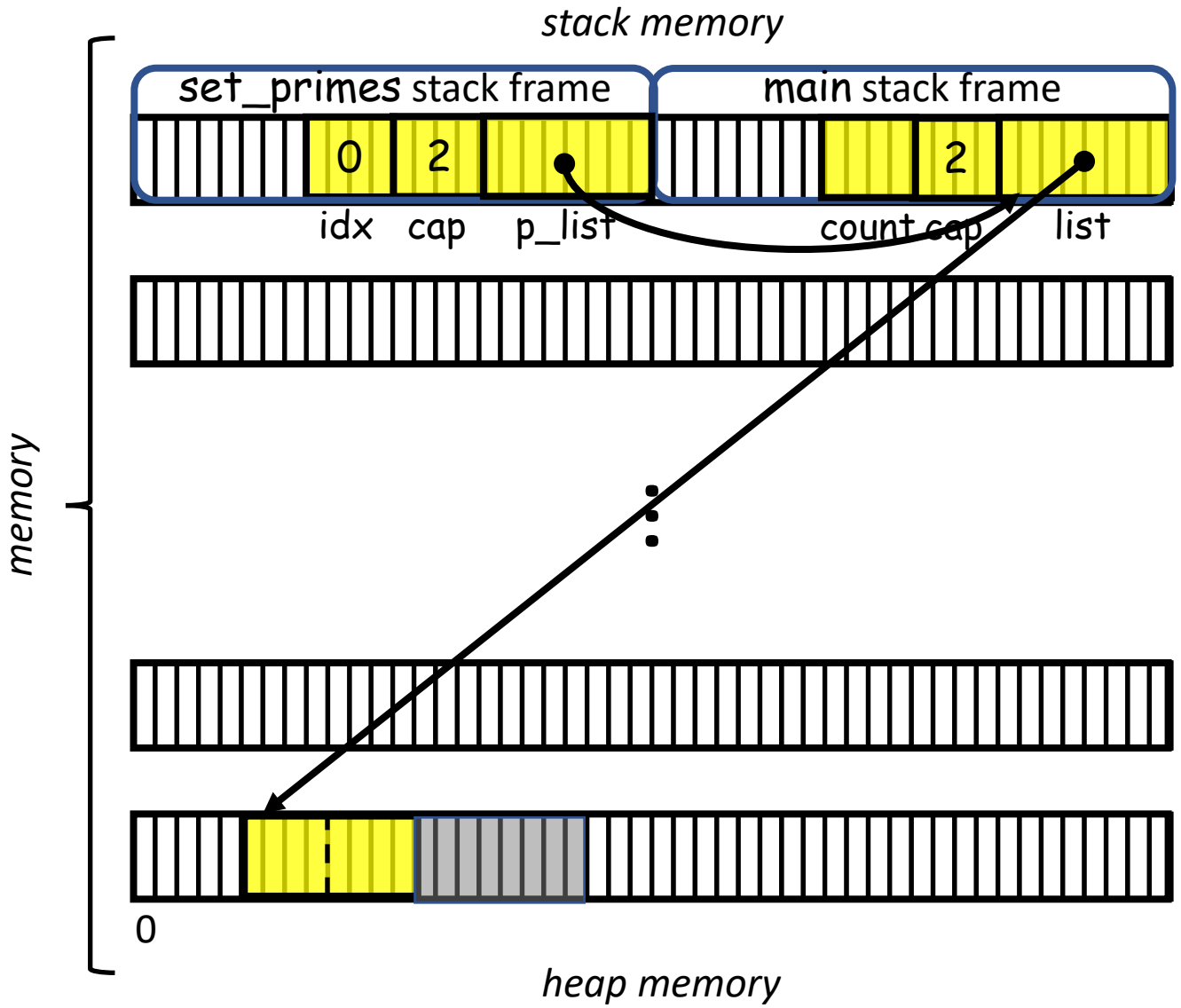
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int **p_list , int cap){
33.          int idx=0;
...
44.                          *p_list = realloc( *p_list, cap * sizeof(int));
...
49.          return idx;
50.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count = set_primes( &list , cap , &count );
...
69.  }
```

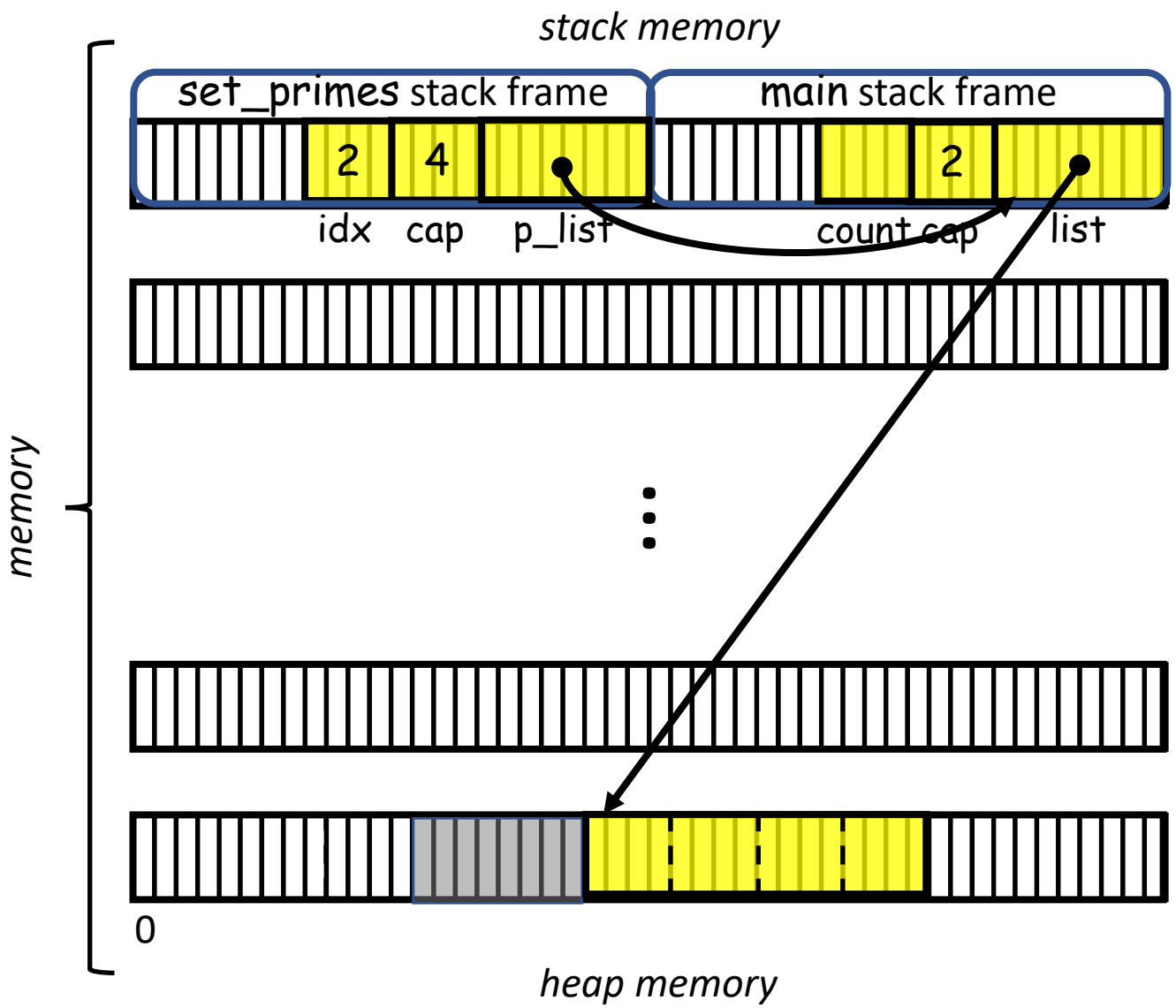
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int **p_list , int cap){
33.          int idx=0;
...
44.                          *p_list = realloc( *p_list, cap * sizeof(int));
...
49.          return idx;
50.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count = set_primes( &list , cap , &count );
...
69.  }
```

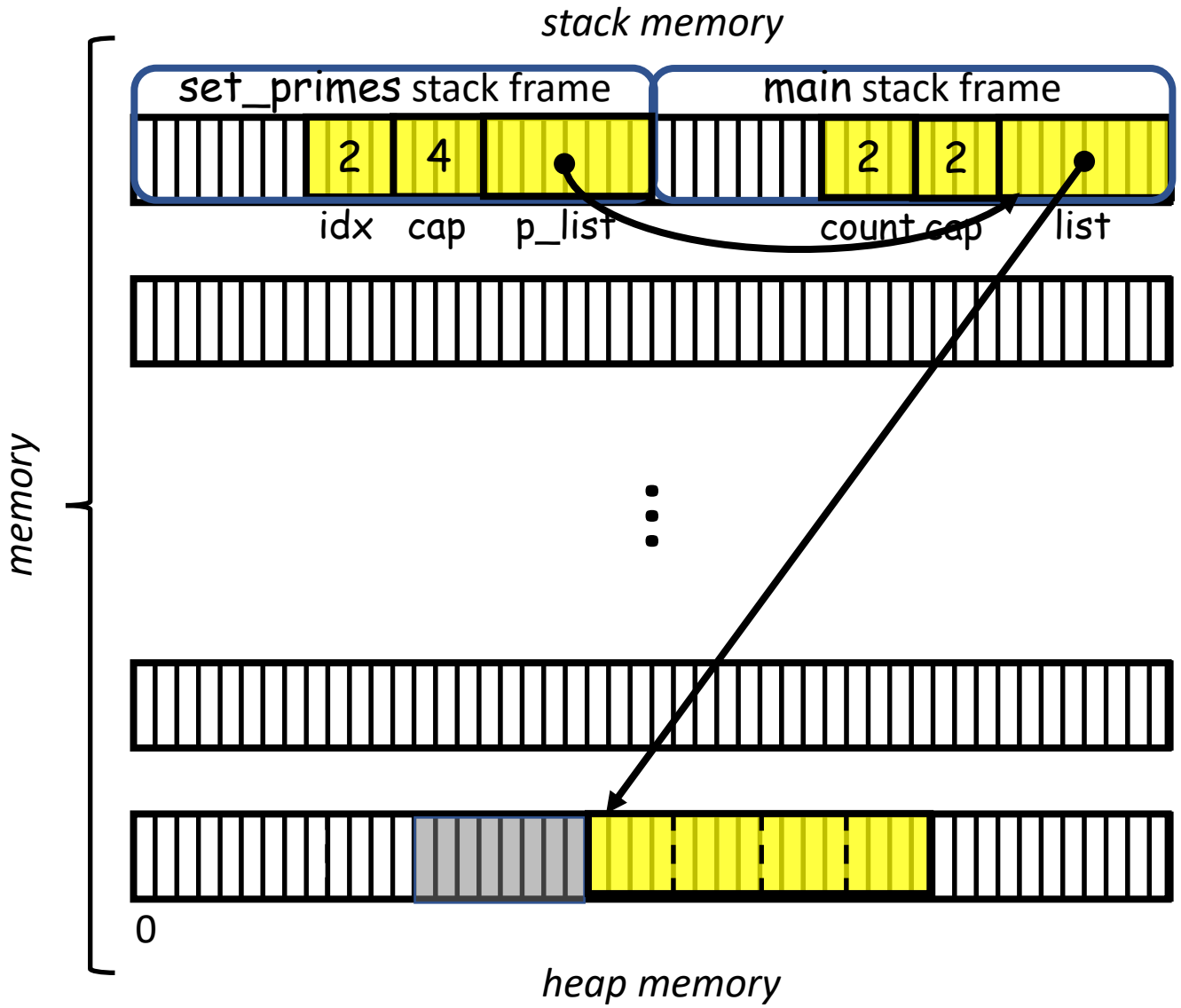
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int **p_list , int cap){
33.          int idx=0;
...
44.                          *p_list = realloc( *p_list, cap * sizeof(int));
...
49.          return idx;
50.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count = set_primes( &list , cap , &count );
...
69.  }
```

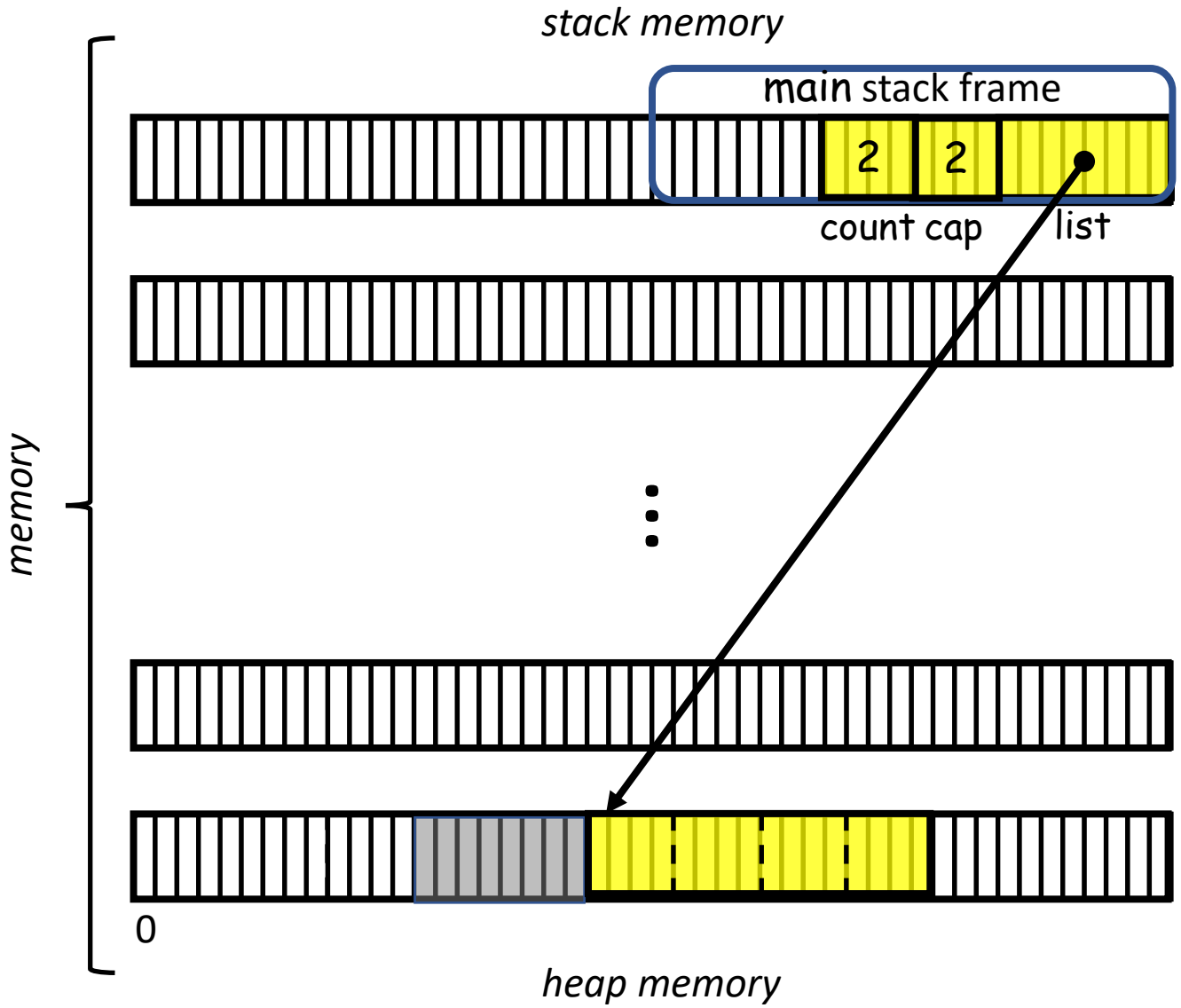
# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int **p_list , int cap){
33.          int idx=0;
...
44.                          *p_list = realloc( *p_list, cap * sizeof(int));
...
49.          return idx;
50.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count = set_primes( &list , cap , &count );
...
69.  }
```

# Exercise 11




*primes.c*

```
...
32.  int *set_primes( int **p_list , int cap){
33.          int idx=0;
...
44.                          *p_list = realloc( *p_list, cap * sizeof(int));
...
49.          return idx;
50.  }
...
54.  int main() {
...
58.          int *list = malloc( cap * sizeof(int));
...
63.          int count = set_primes( &list , cap , &count );
...
69.  }
```

# Outline

- Exercise 11
- 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" , 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 (w/o size information)

# 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++ = 0 do?

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

- Exercise 11
- 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

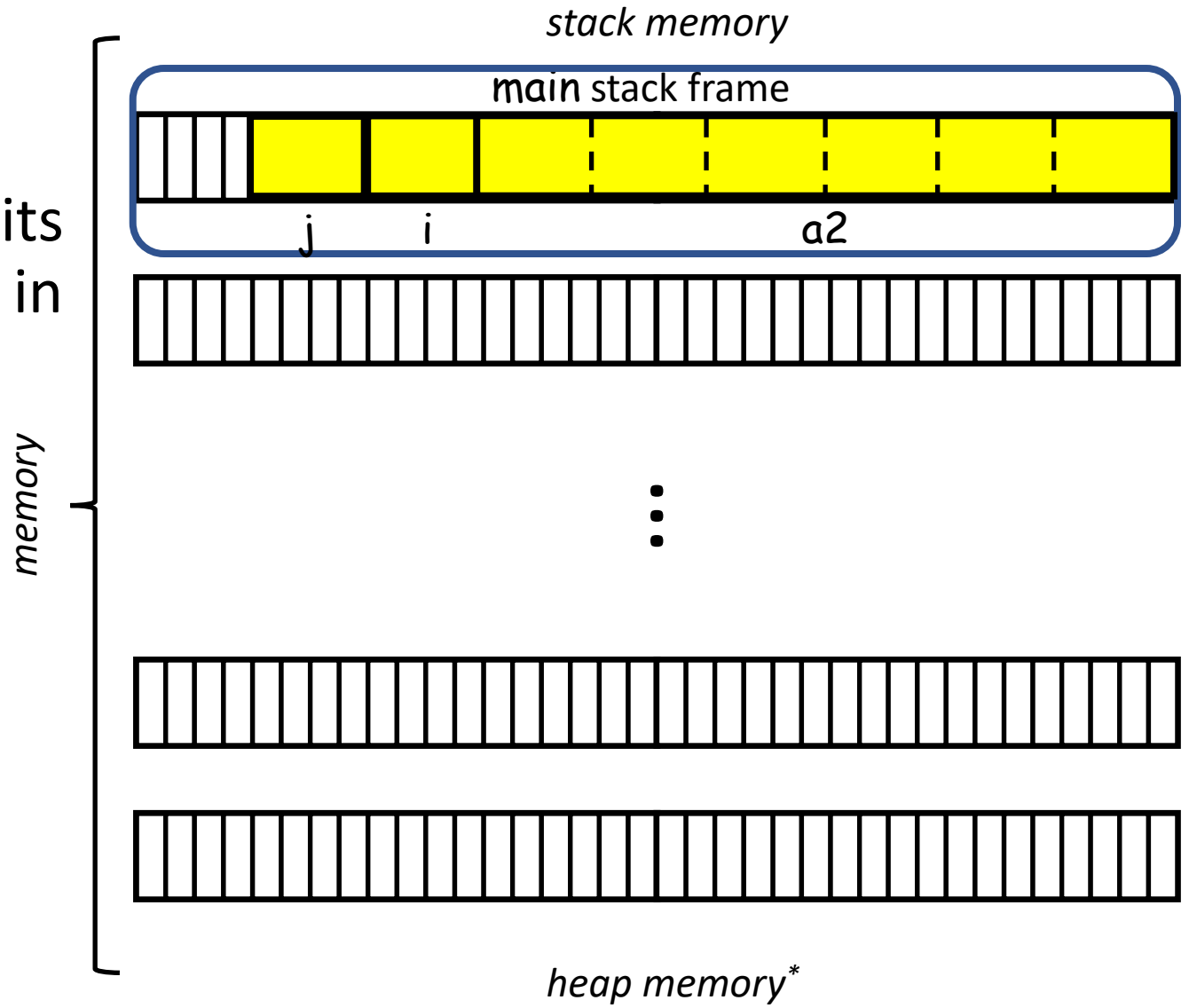
A2: Declare an array (of size 2) containing `int` arrays (of size 3).

# 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;
}
```

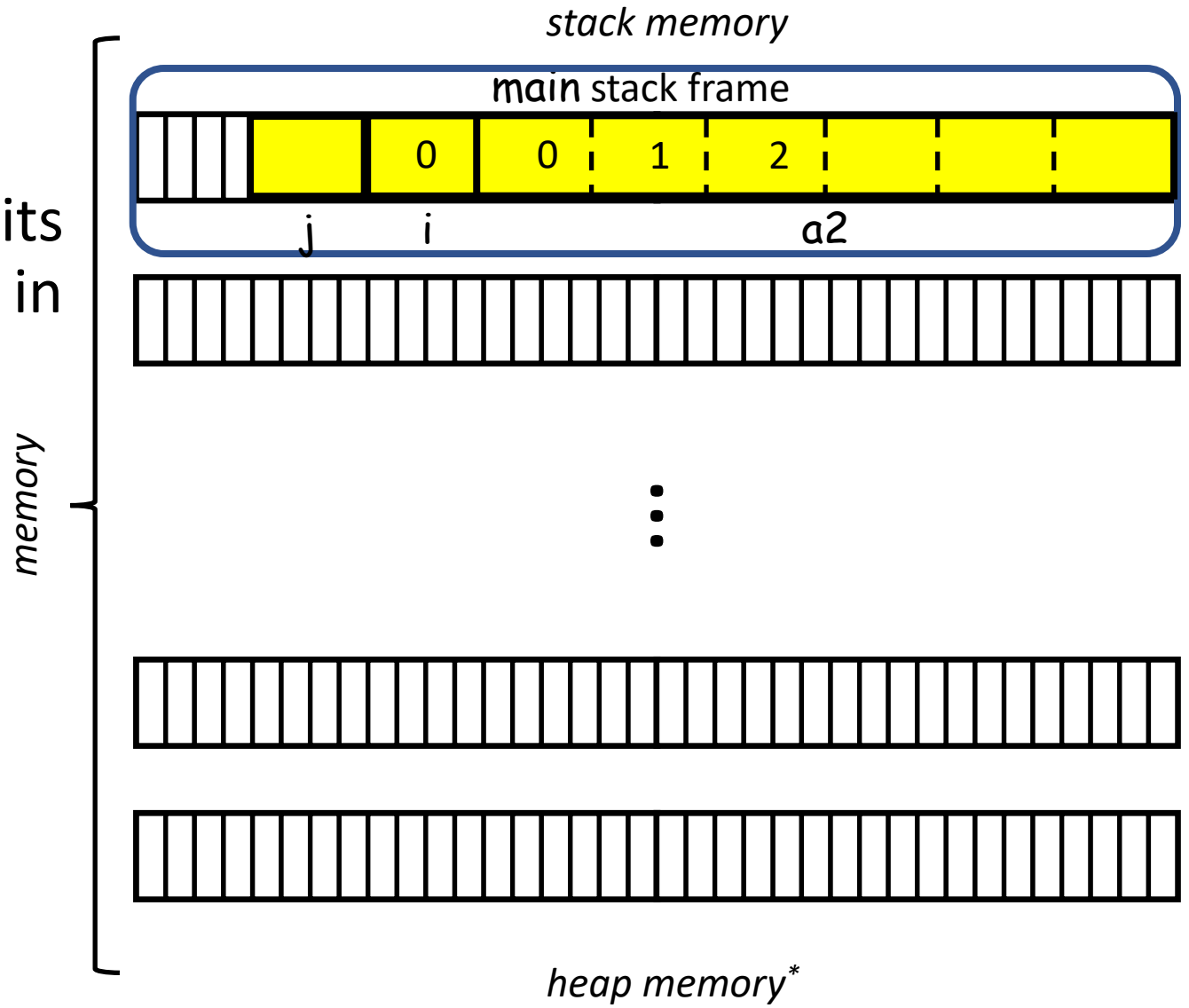


# Dynamic 2D arrays

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;
}
```

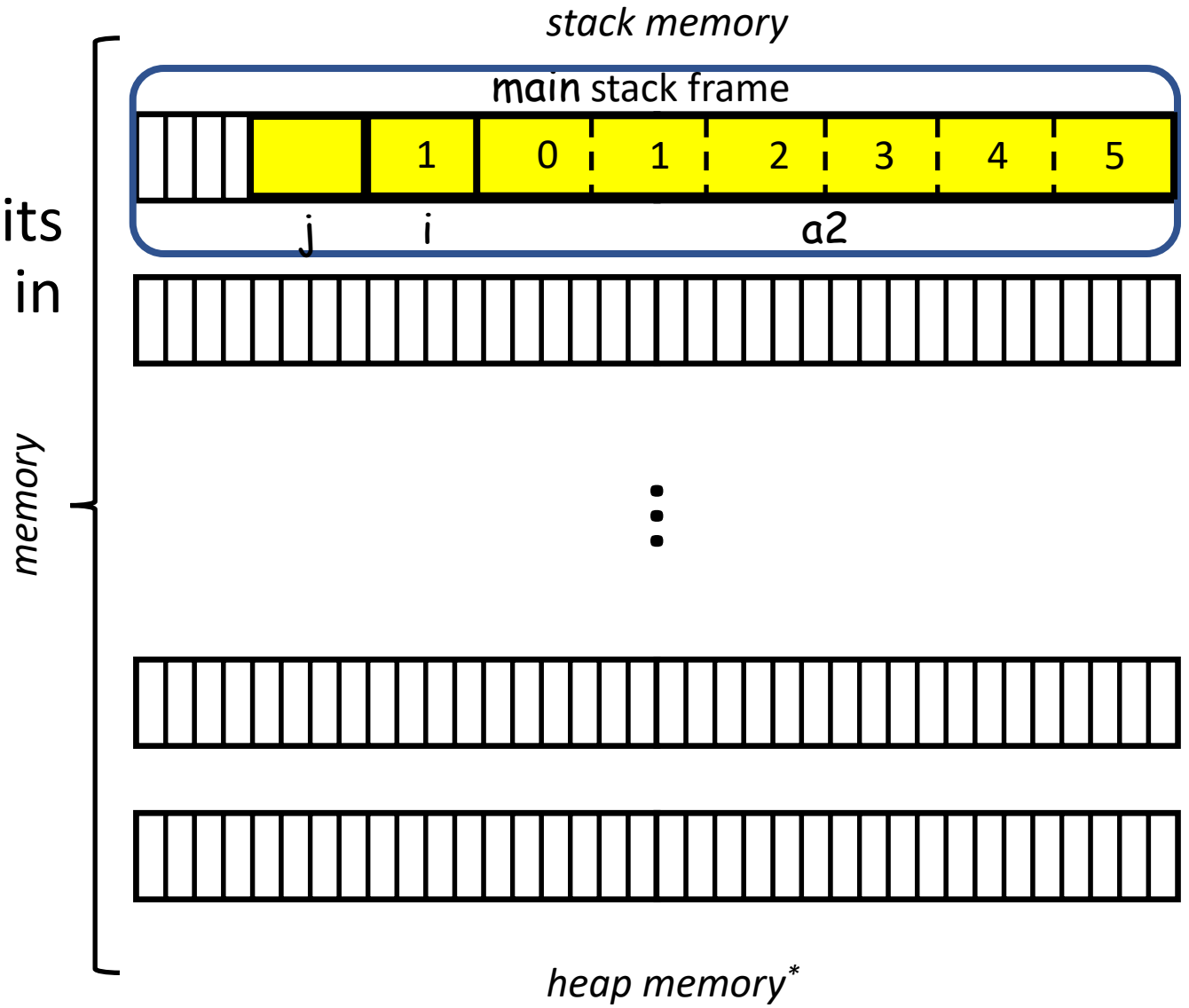


# Dynamic 2D arrays

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;
}
```

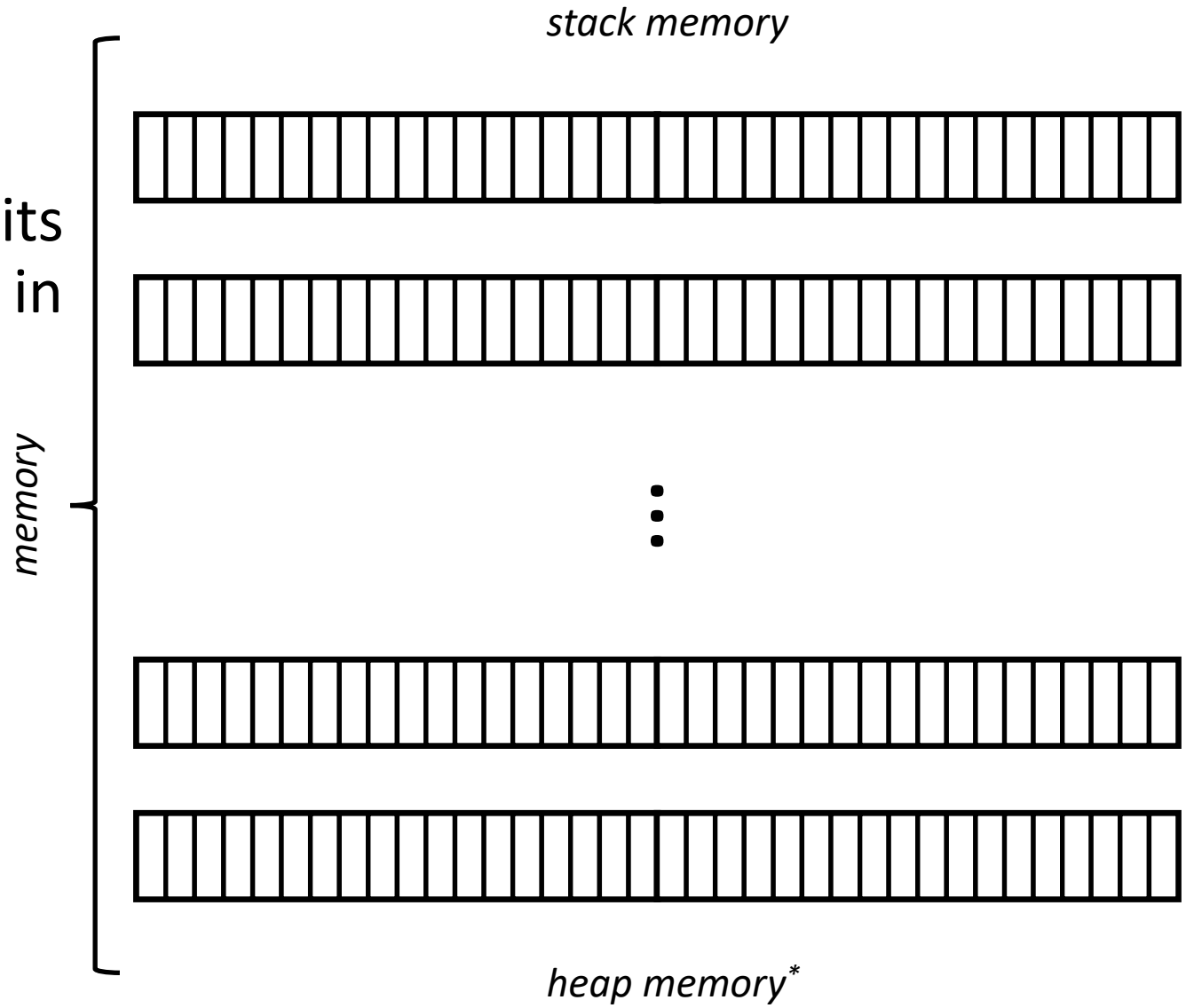


# Dynamic 2D arrays

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;
}
```

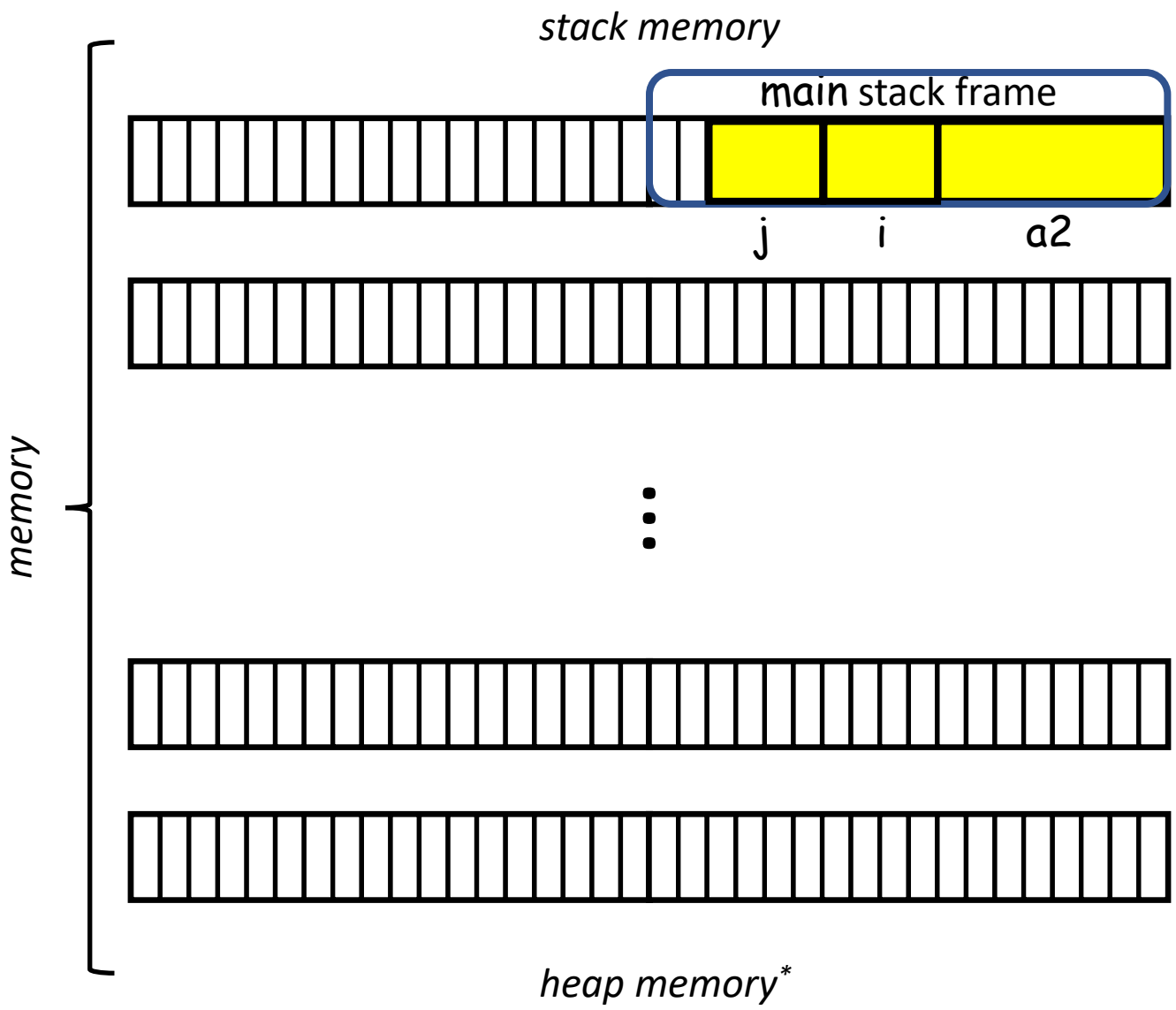
# Dynamic 2D arrays

Declaring a 2x3 grid of int values:

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

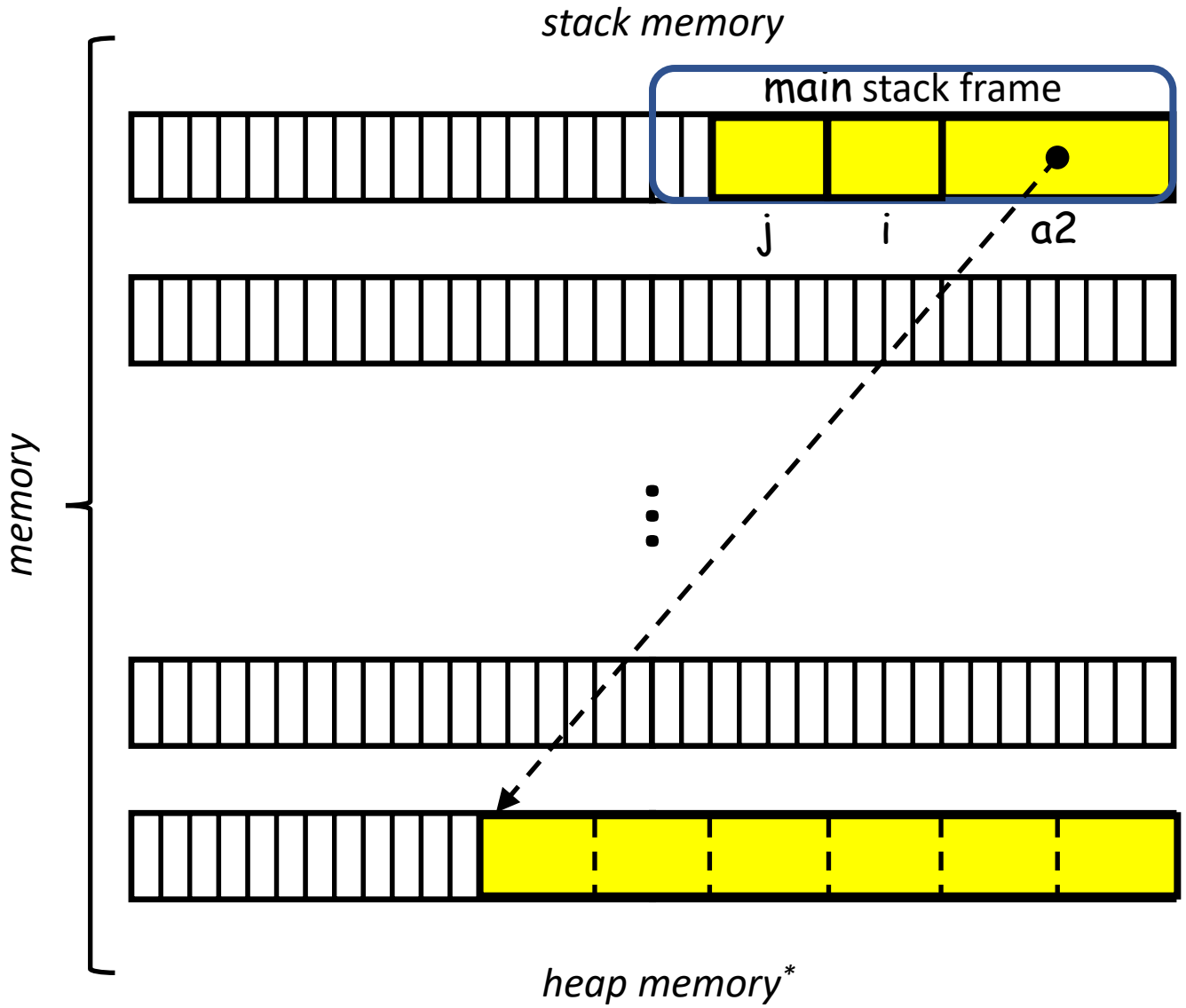
```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

# Dynamic 2D arrays



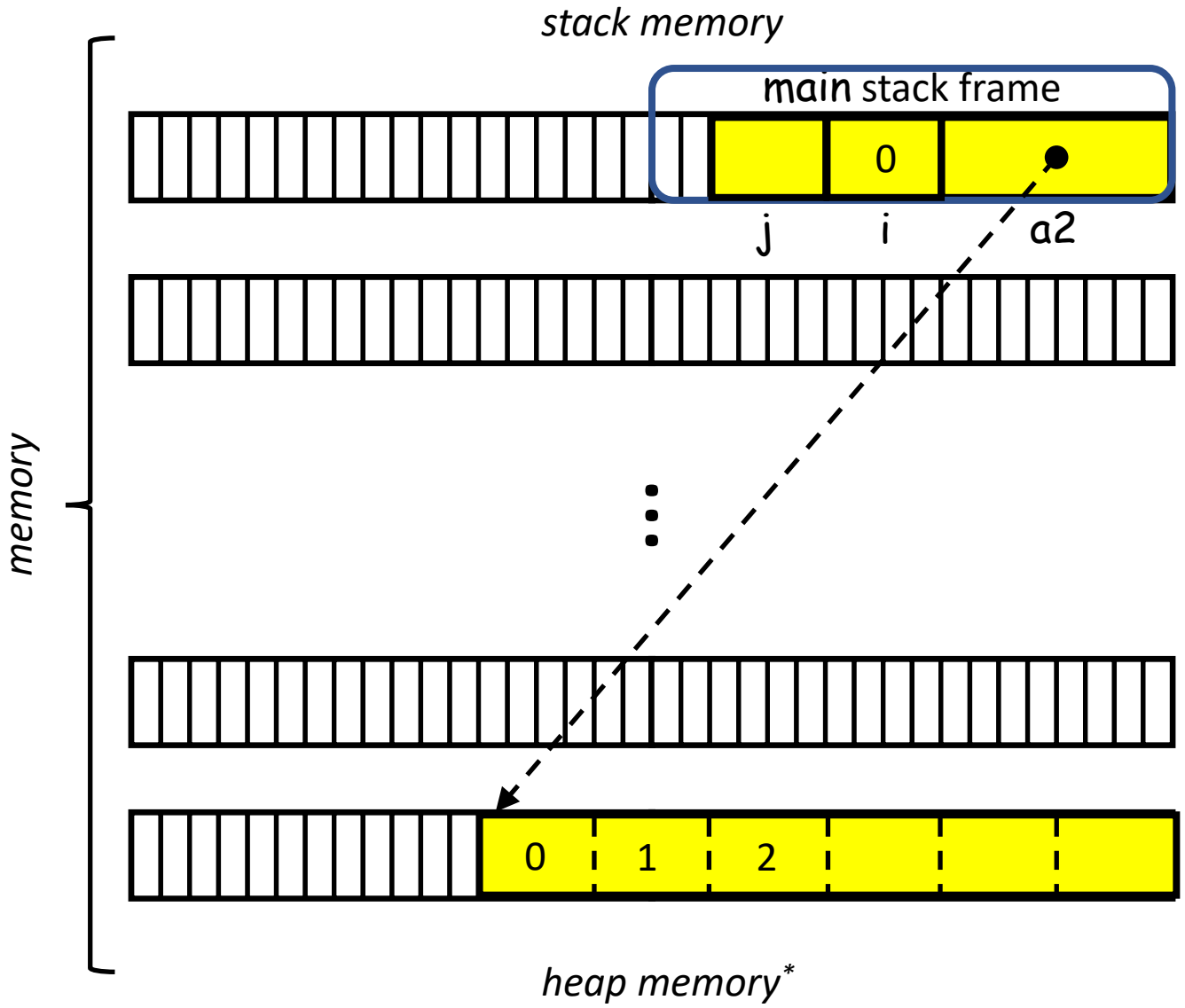

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

# Dynamic 2D arrays



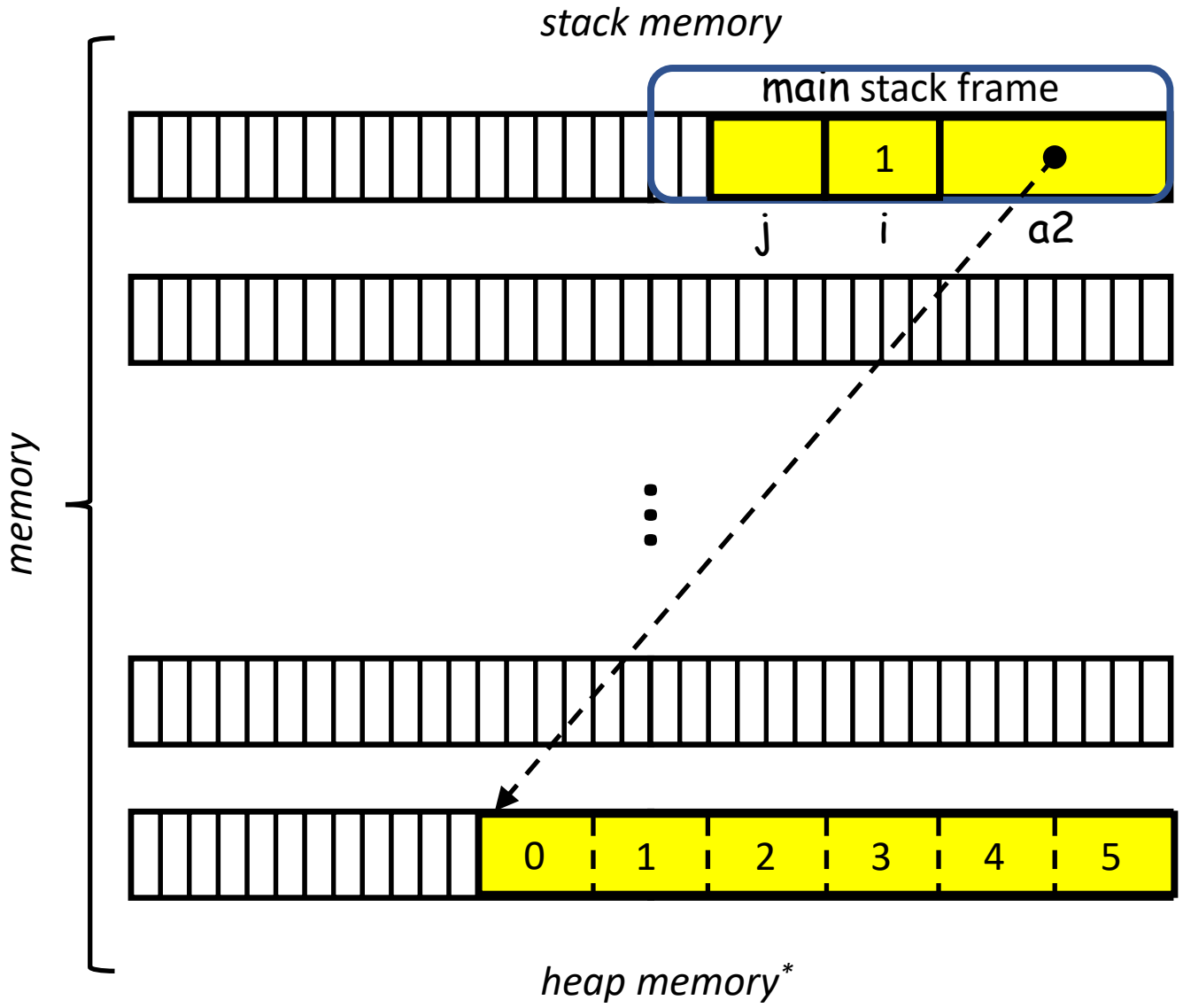

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

# Dynamic 2D arrays



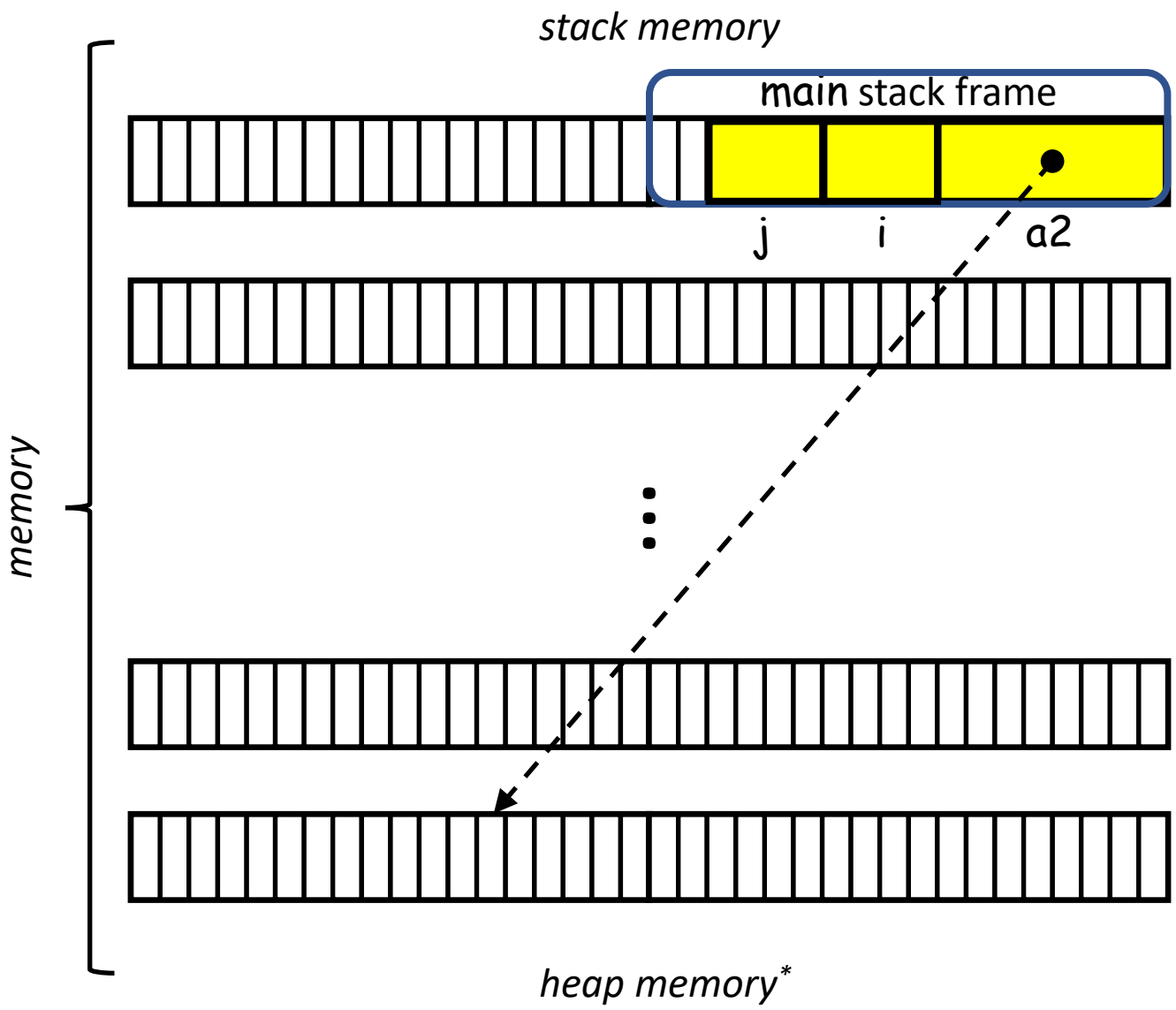

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

# Dynamic 2D arrays



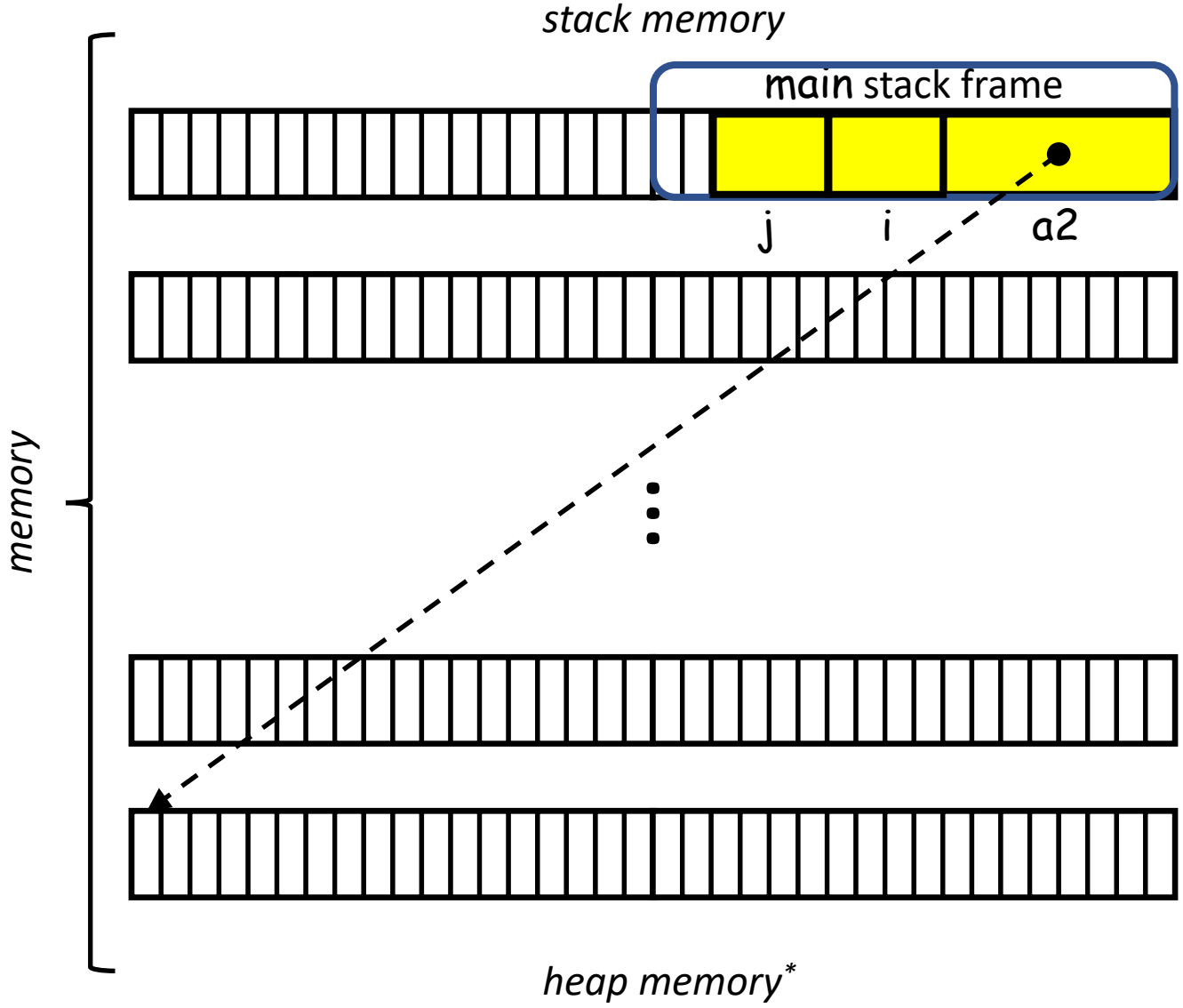

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

# Dynamic 2D arrays



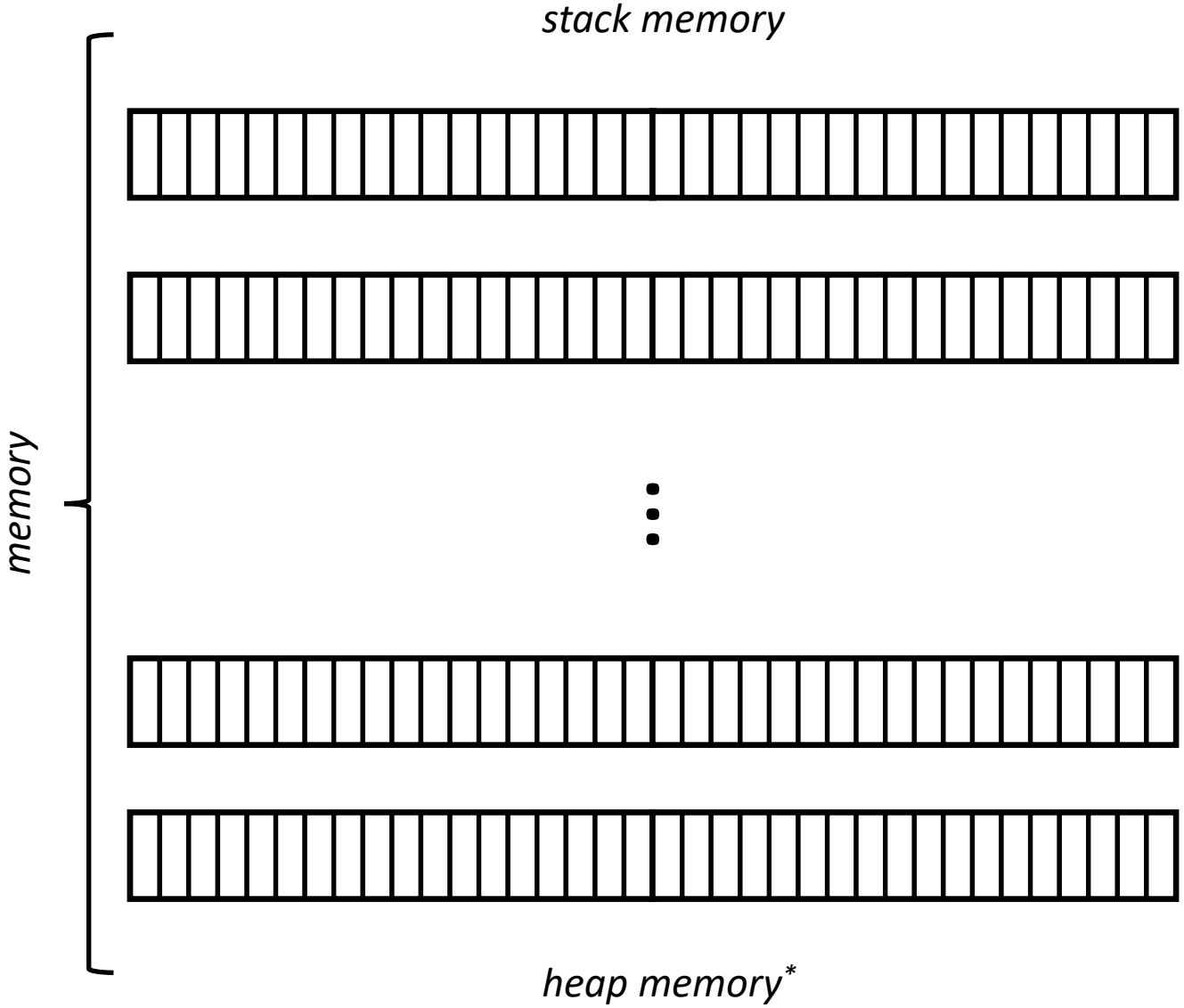

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int *a2 = malloc( sizeof(int) * 2 * 3 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
        for( int j=0 ; j<3 ; j++ )
            a2[3*i+j] = 3*i+j;
    free( a2 );
    a2 = NULL;
    return 0;
}
```

# 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:

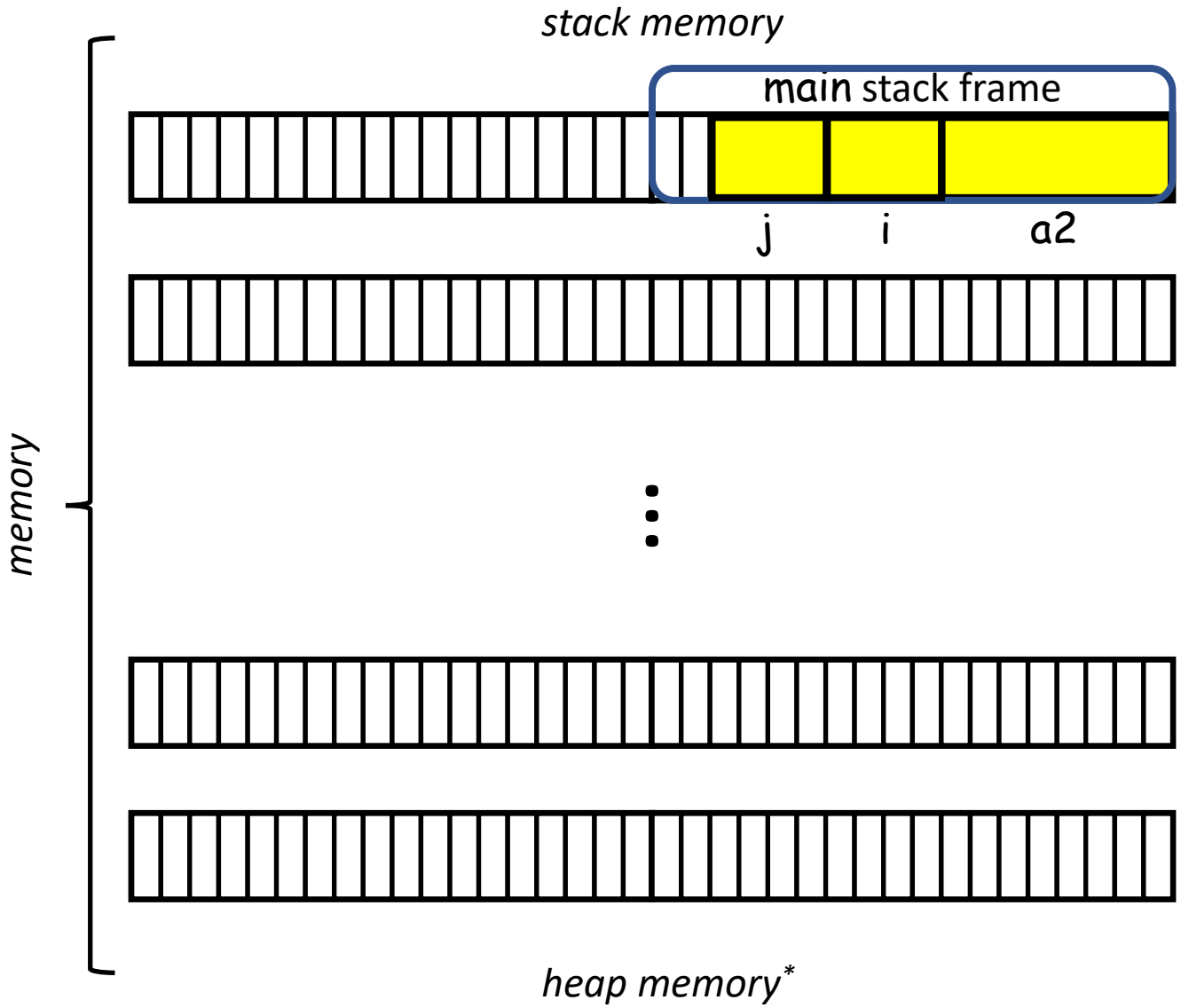
1. Declare a single array of ints
2. Declare an array of int arrays
   - Need to allocate/deallocate the array of int arrays
   - Need to allocate/deallocate each int array

   ✓Indexing is clean

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

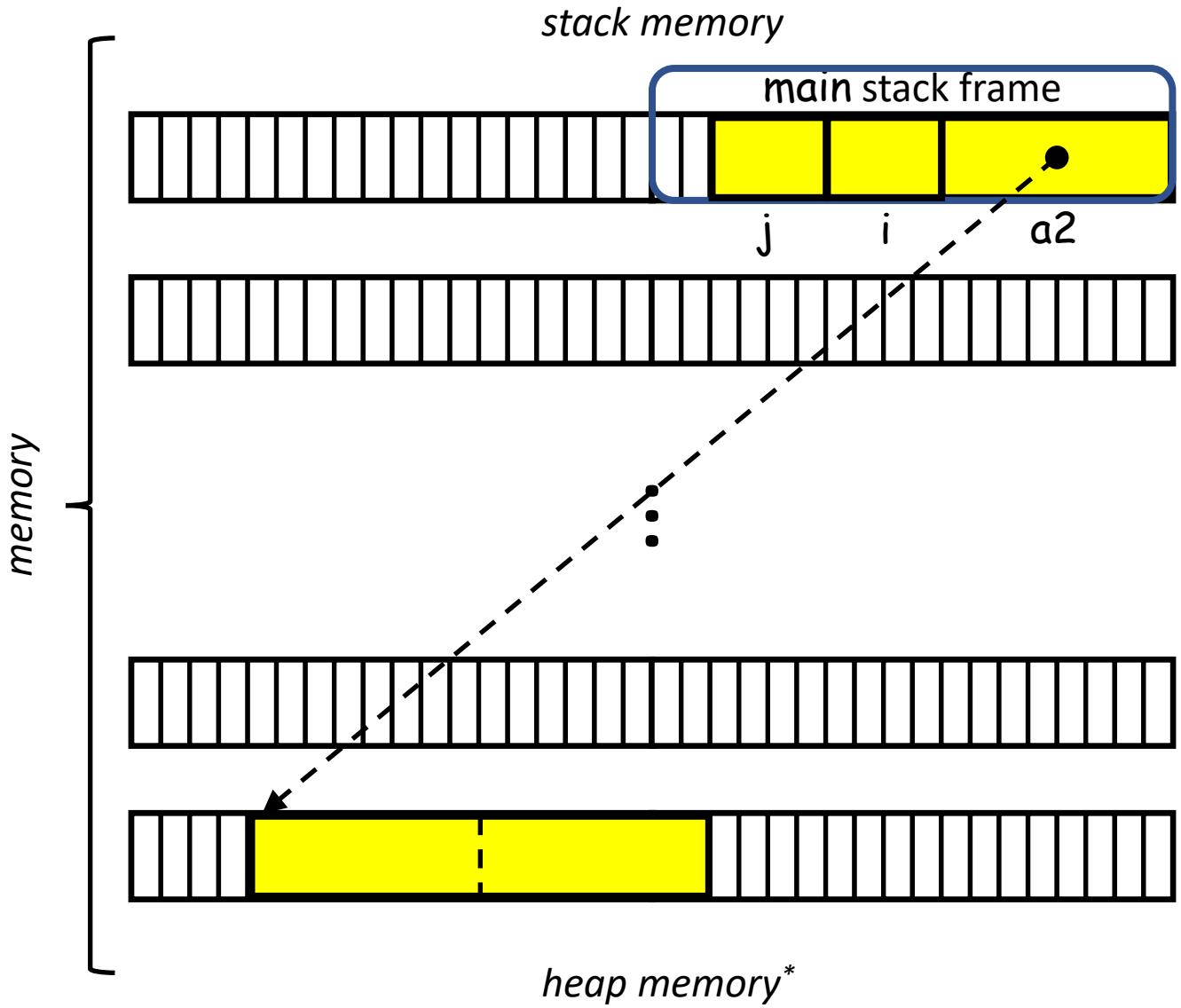
```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



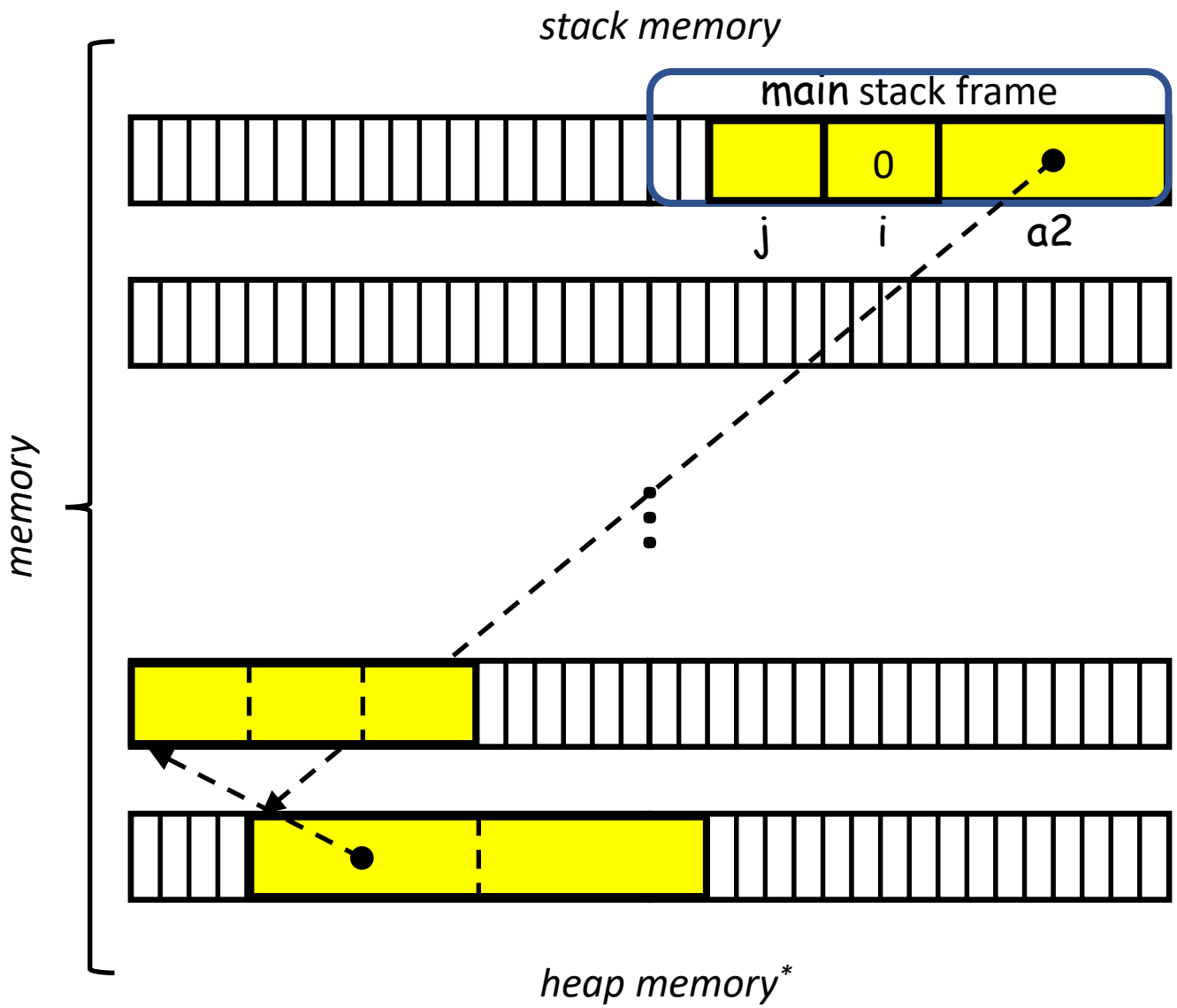

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



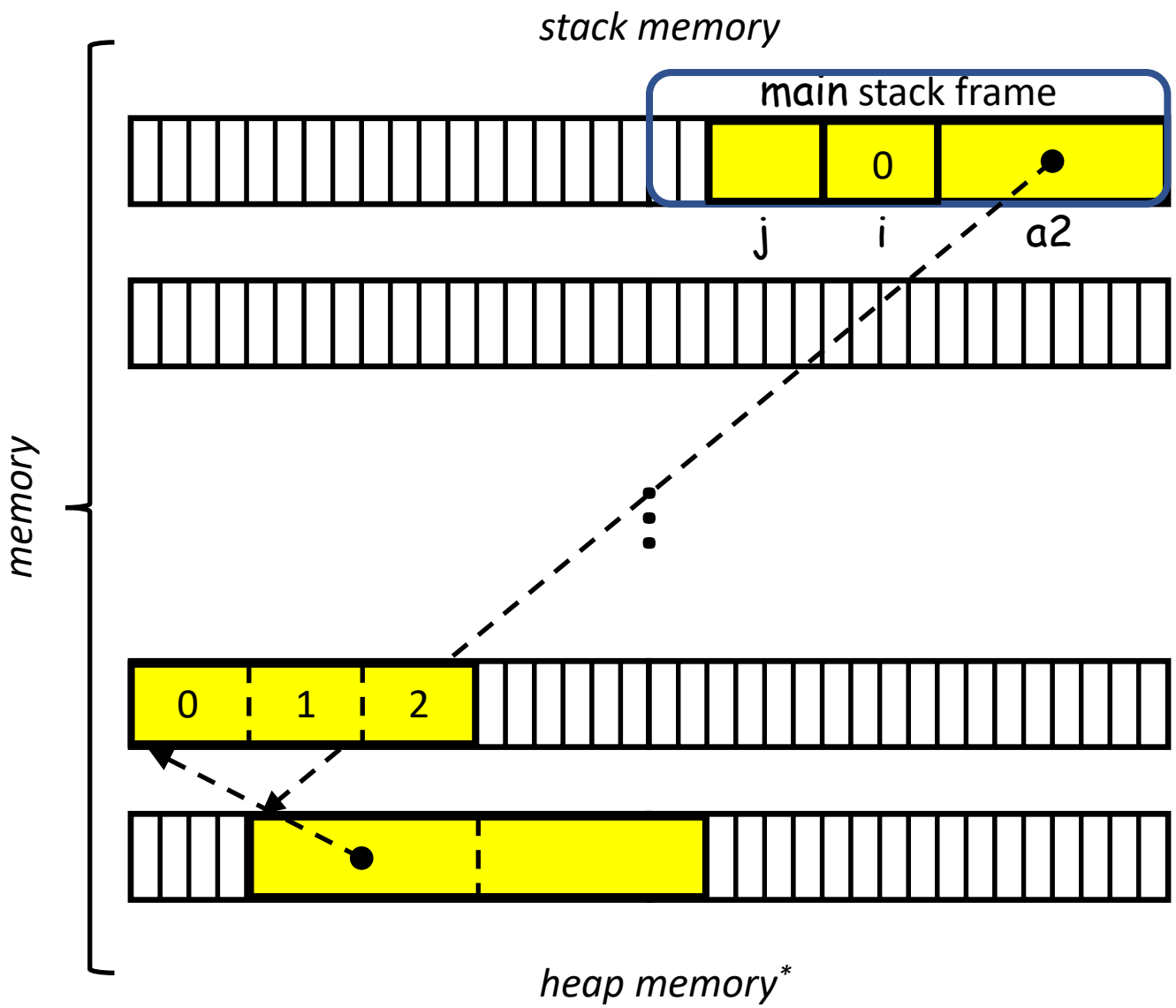

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



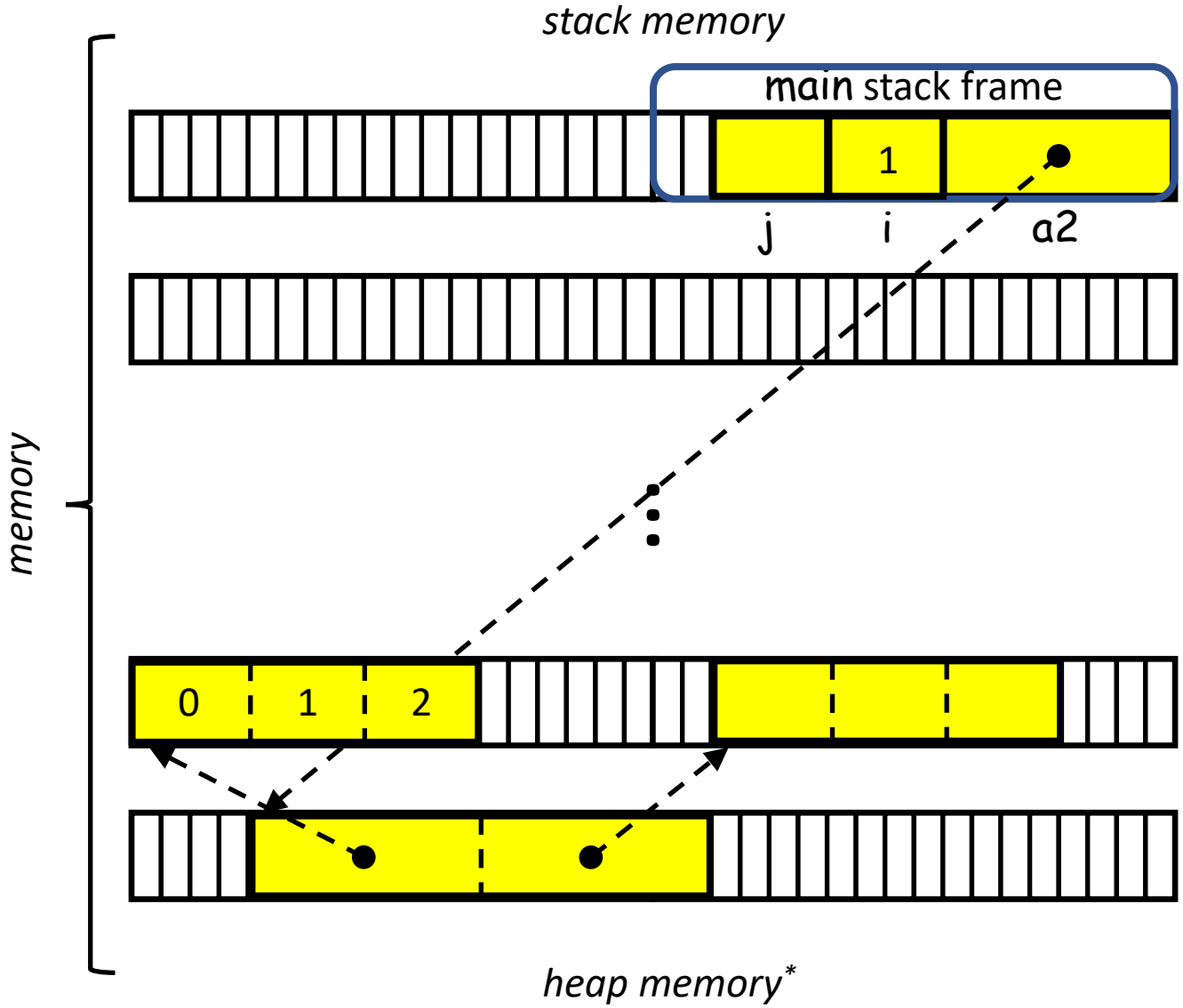

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



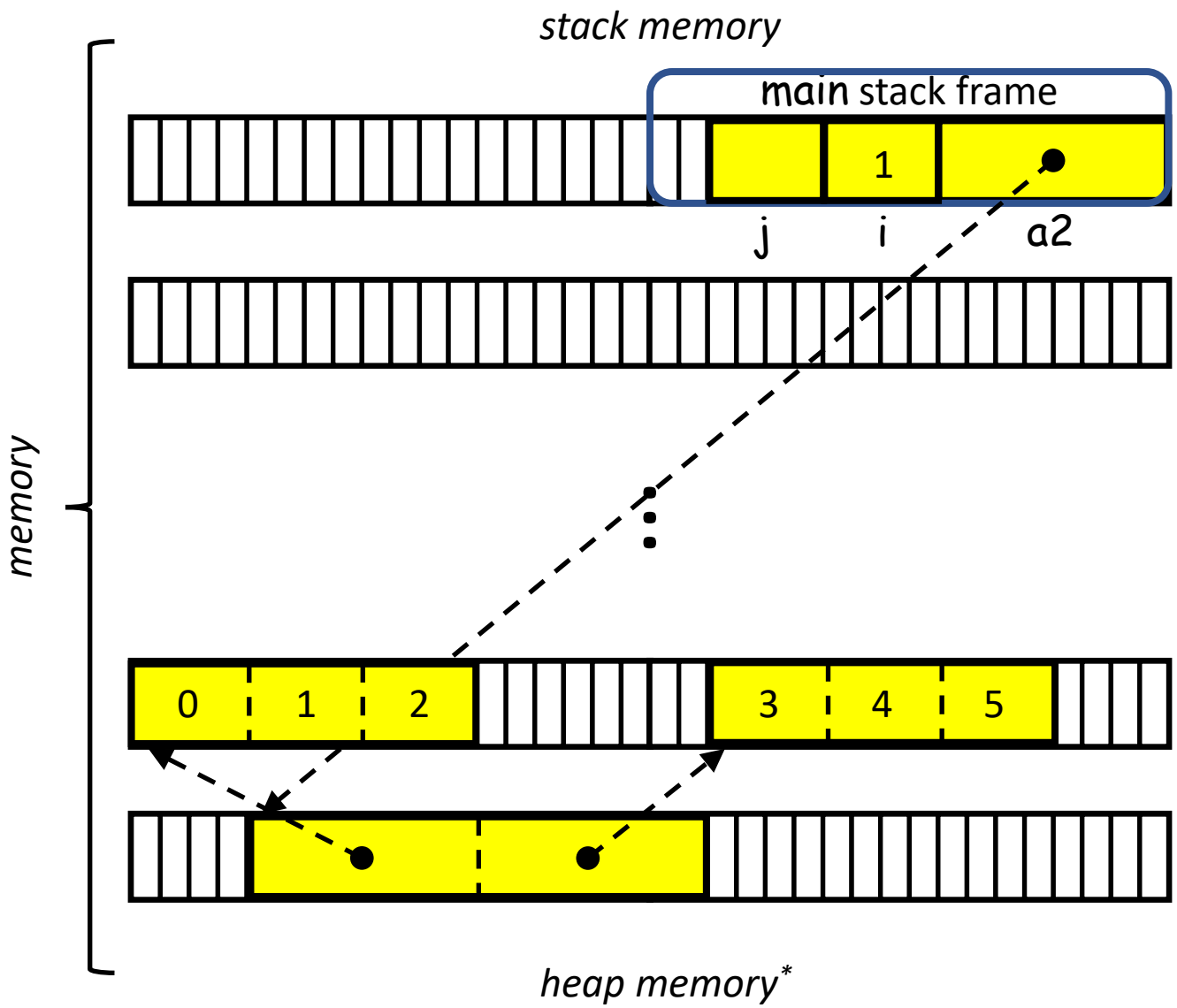

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



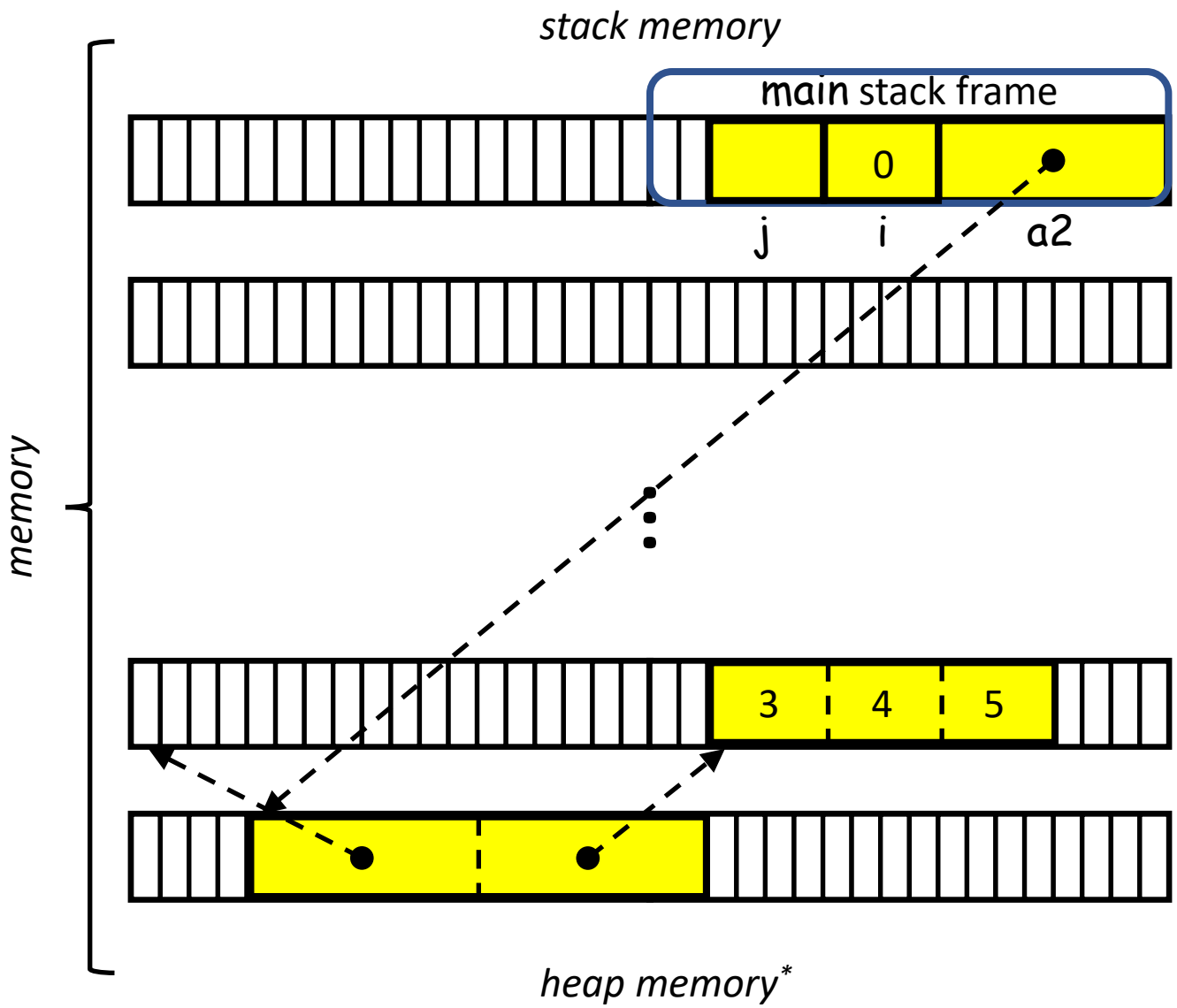

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



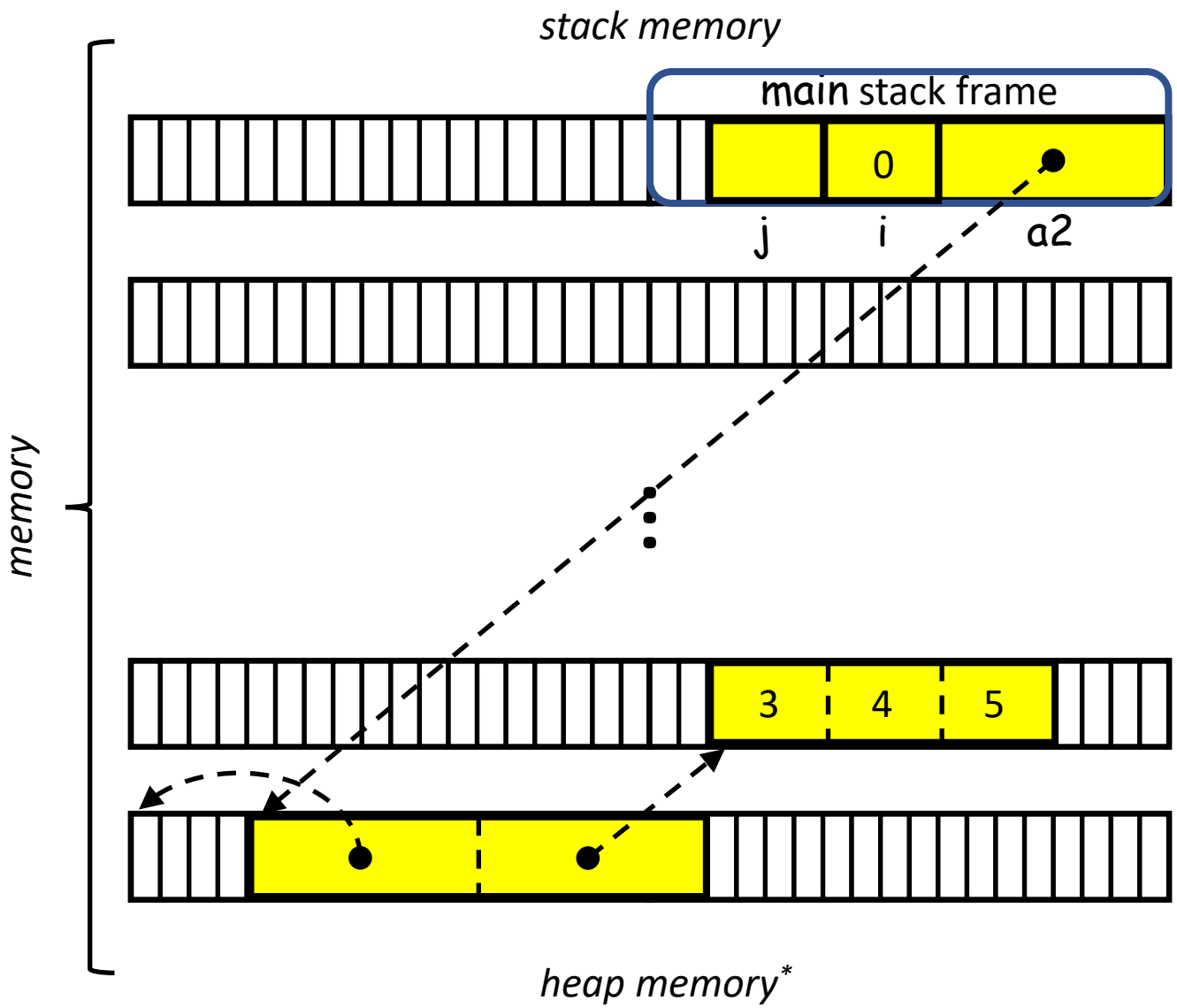

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



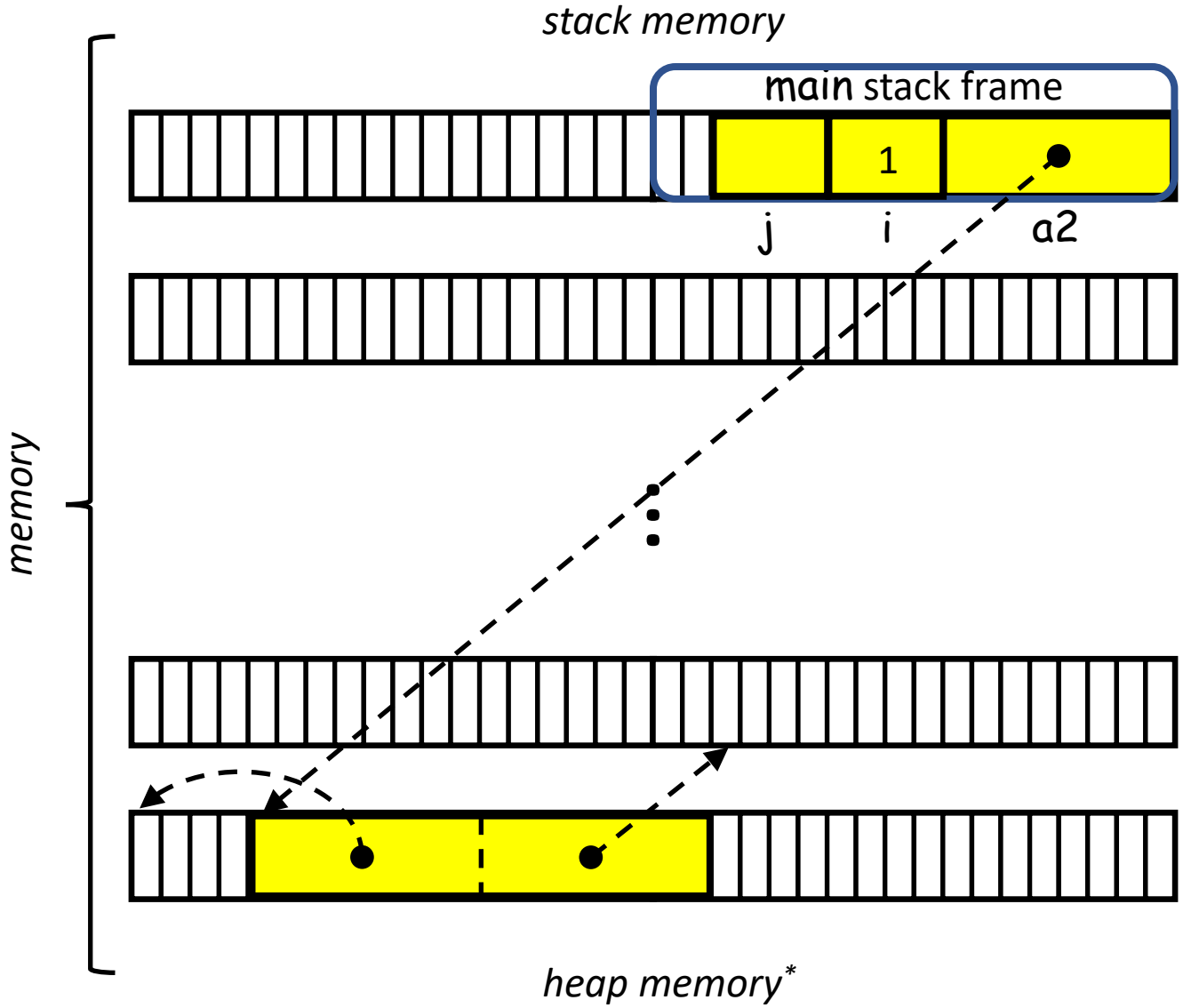

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



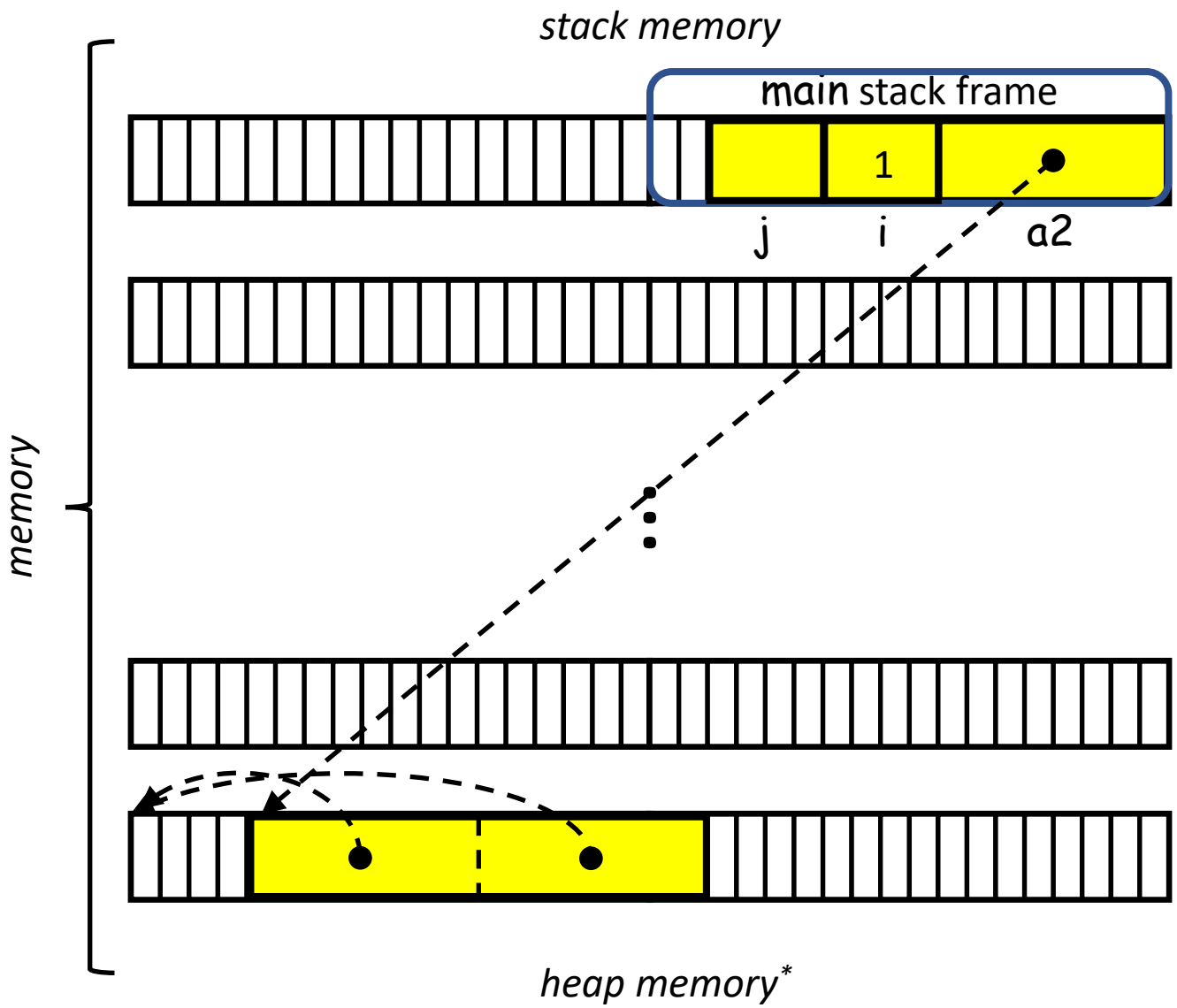

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



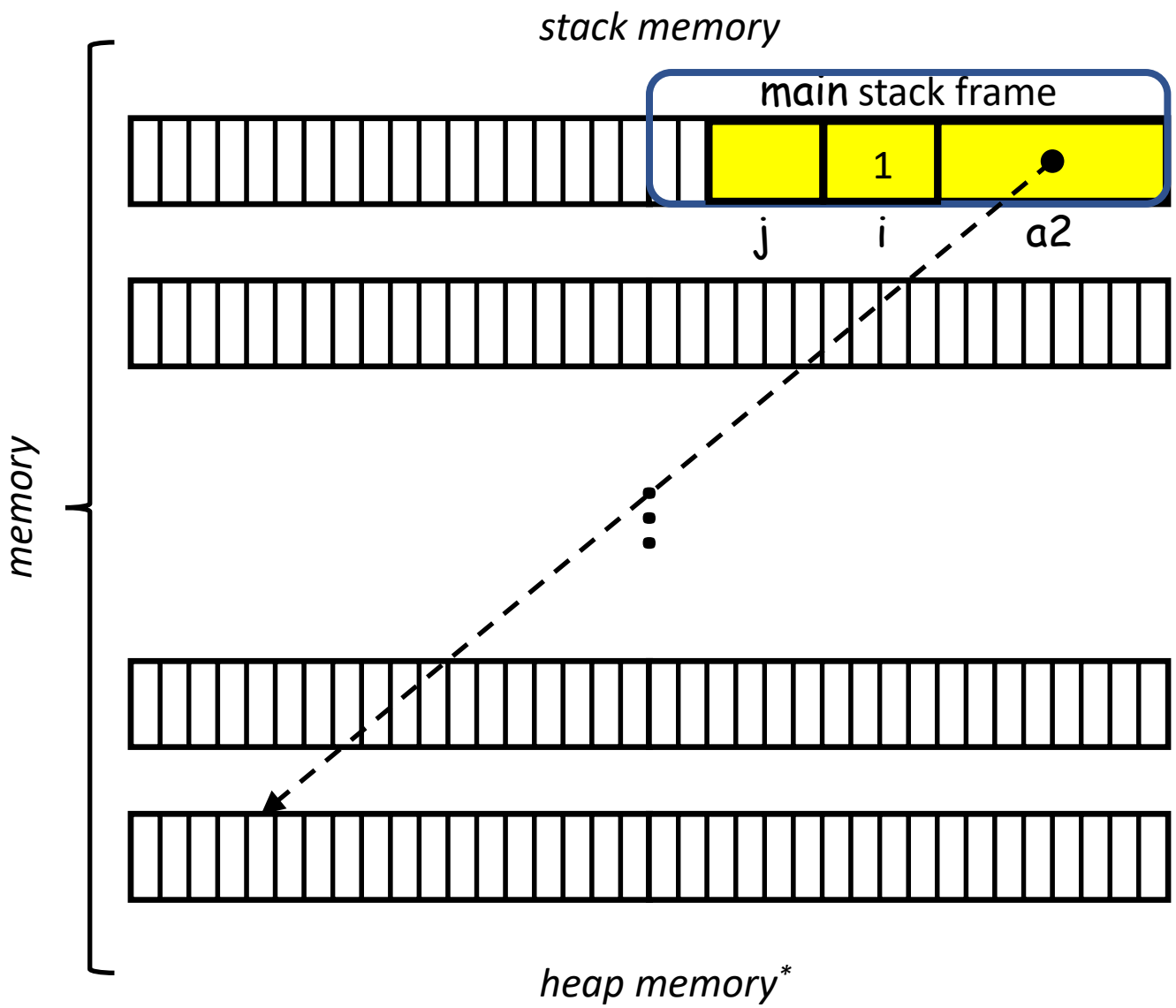

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



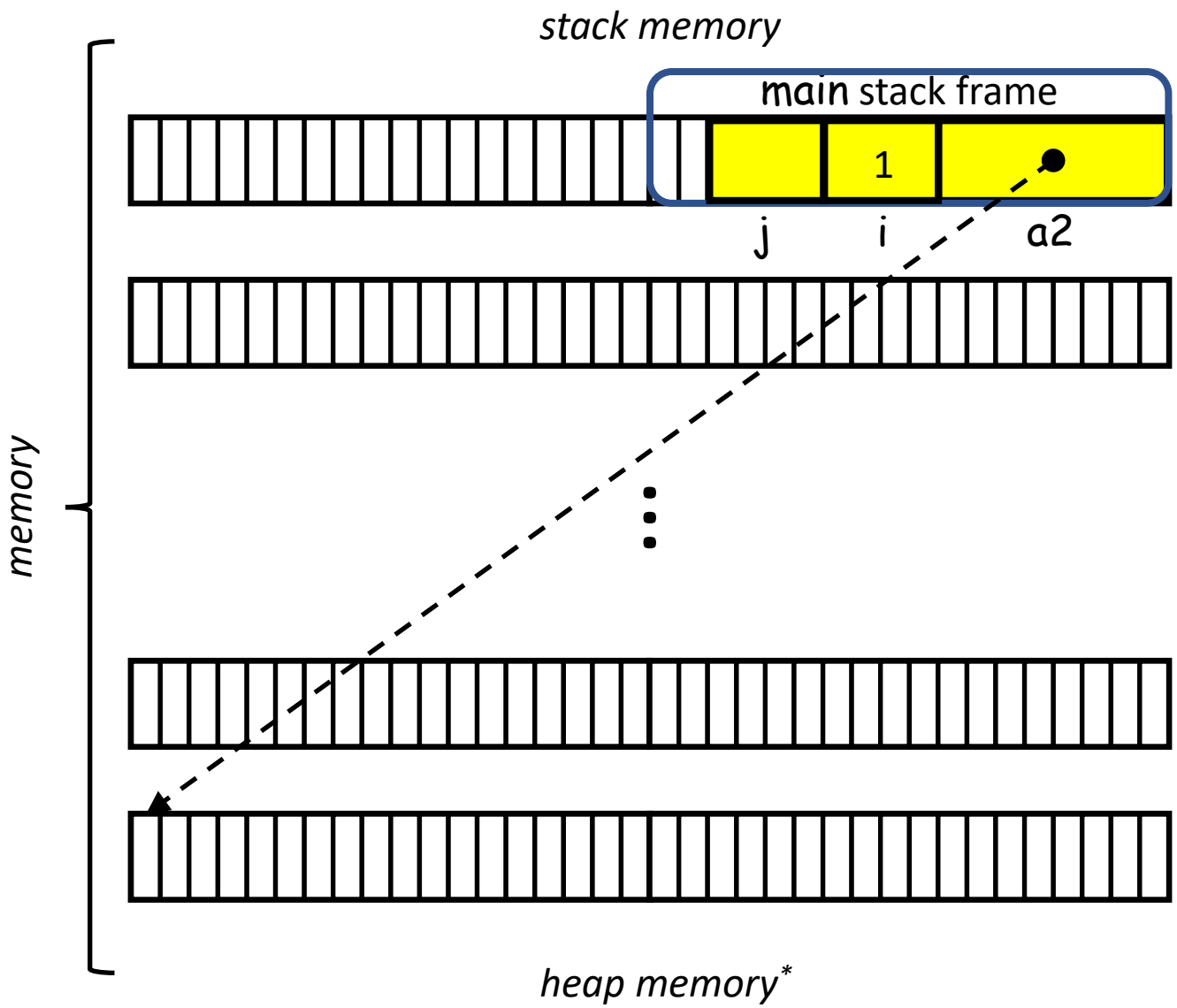

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

# Dynamic 2D arrays



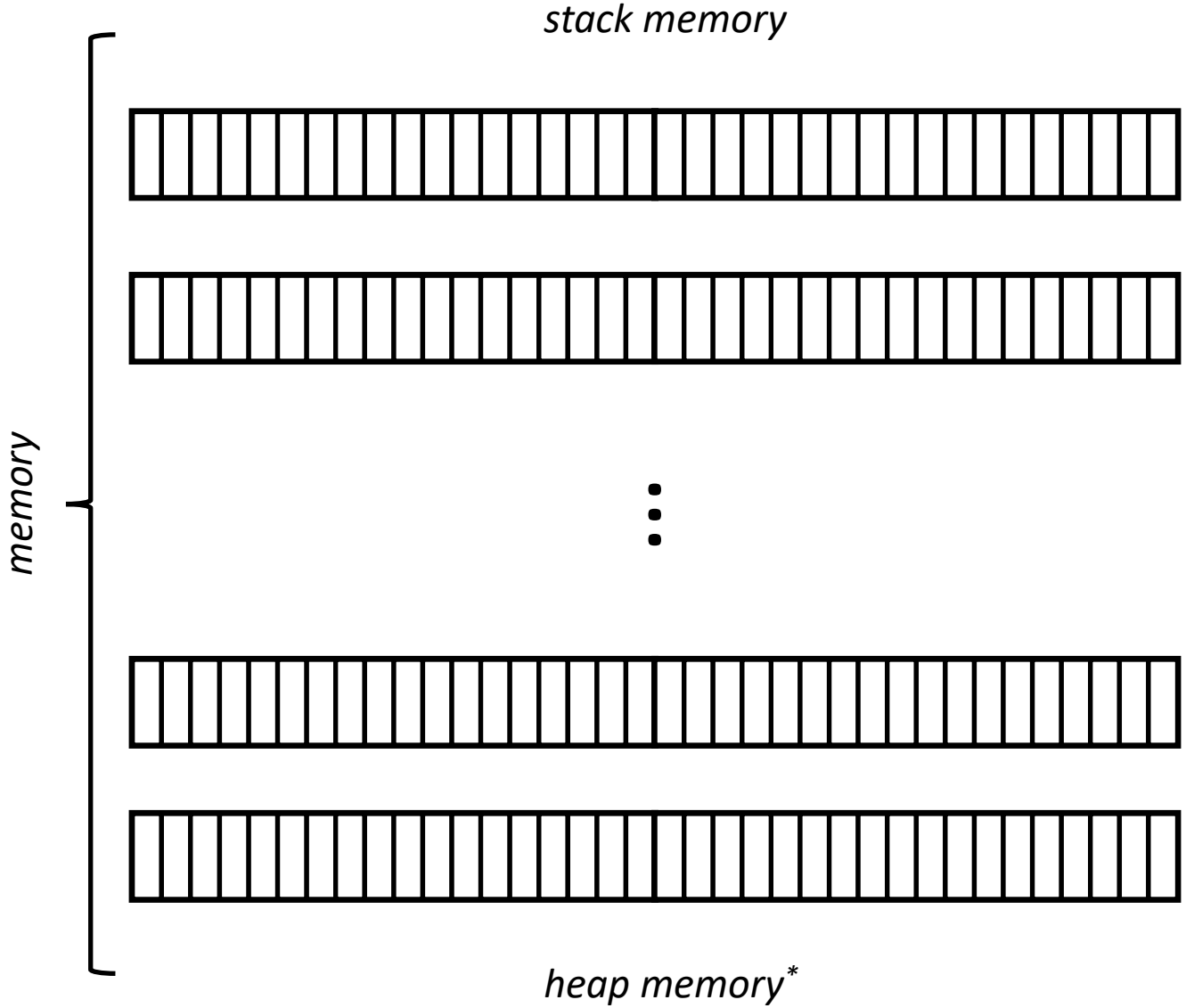

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    int **a2 = malloc( sizeof(int*) * 2 );
    if( !a2 ) return 1;
    for( int i=0 ; i<2 ; i++ )
    {
        a2[i] = malloc( sizeof(int) * 3 );
        if( !a2[i] ) return 1;
        for( int j=0 ; j<3 ; j++ ) a2[i][j] = 3*i+j;
    }
 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

- Exercise 11
- 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

- Exercise 11
- 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 );
```

p points to arr[0]=94
q points to arr[3]=72
r points to arr[2]=35

```
>> ./a.out
94 72 35
3 2 1
-3
1 0
```

# Review questions

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

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

   legal?

Yes. (It's the accessing that's the problem)

# 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?

No. (It's the accessing that's the problem)

# 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;
}
```

Passing in the sub-array a[] = {42,67,89,76}

```
>> ./a.out
result = 274
```

# Review questions

5. Suppose that we have variables:

```
int ra1[10] = { 1 , 2 , 3 };
int *ra2 = ra1;
```

and

```
int fun( int *ra );
```

declaration.
Will

```
fun( ra1 );
fun( ra2 );
```

compile?
What if we change the function declaration to

```
int fun( const int ra[] );
```

Yes and yes.

# Exercise 12

- Website -> Course Materials -> Exercise 12