Intermediate Programming

Day 11

Outline

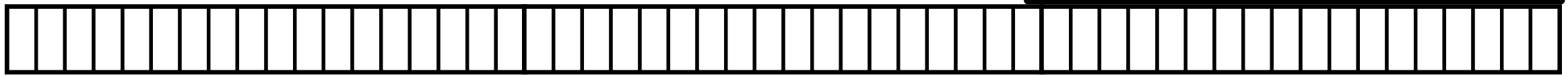
- Dynamic memory allocation
- `Valgrind`
- Review questions

Returning an array in C

Q: Why doesn't this code work?

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

```
>> ./a.out
Segmentation fault (core dumped)
>>
```



memory

Stack frame

- Each function call gets a new, fixed size, *stack frame* “pushed onto” the stack in memory for storing
  - Local variables
  - Copies of function parameters
  - Information about the calling function
  - Other miscellany

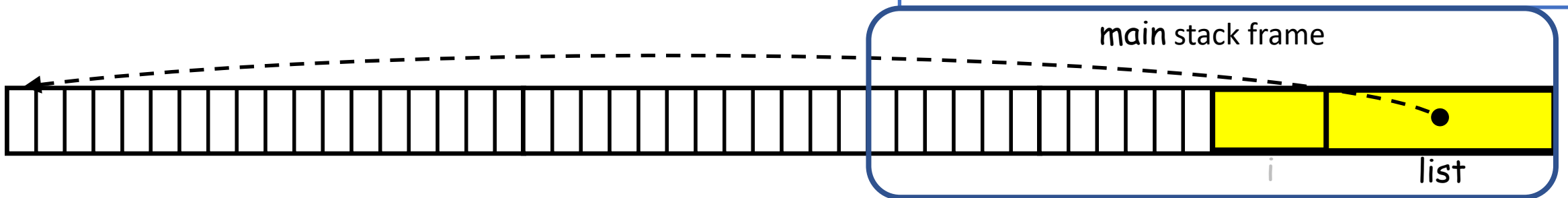
- Once the current function call returns, the stack frame is “popped off” the stack, and execution returns to the calling function
  (so the current function’s frame is always on top)

Returning an array in C

Q: Why doesn't this code work?

- **Frames get pushed on the stack when the function is called**
- And are popped off when it returns

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

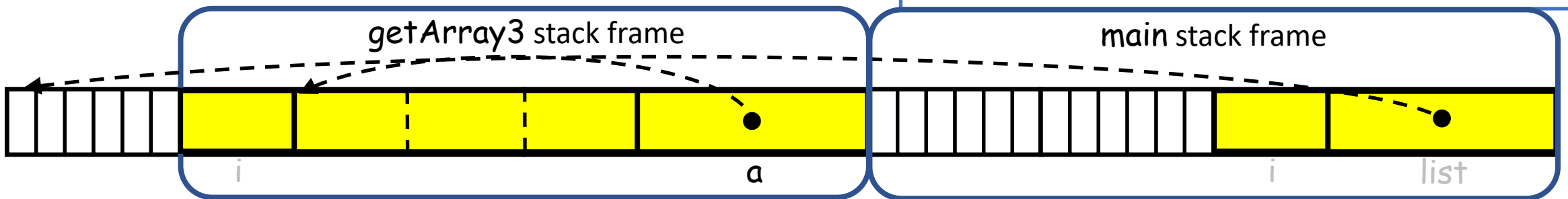


Returning an array in C

Q: Why doesn't this code work?

- **Frames get pushed on the stack when the function is called**
- And are popped off when it returns

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

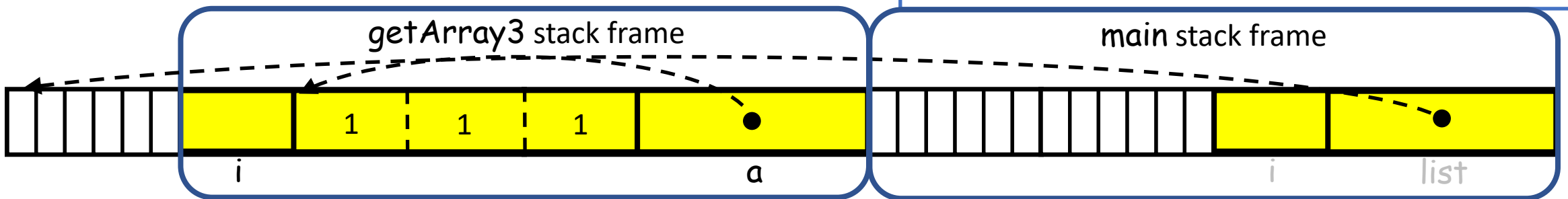


Returning an array in C

Q: Why doesn't this code work?

- Frames get pushed on the stack when the function is called
- And are popped off when it returns

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

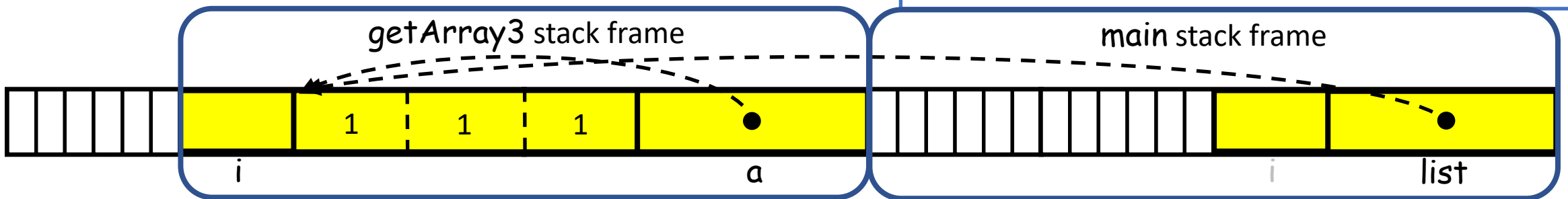


Returning an array in C

Q: Why doesn't this code work?

- Frames get pushed on the stack when the function is called
- And are popped off when it returns

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```



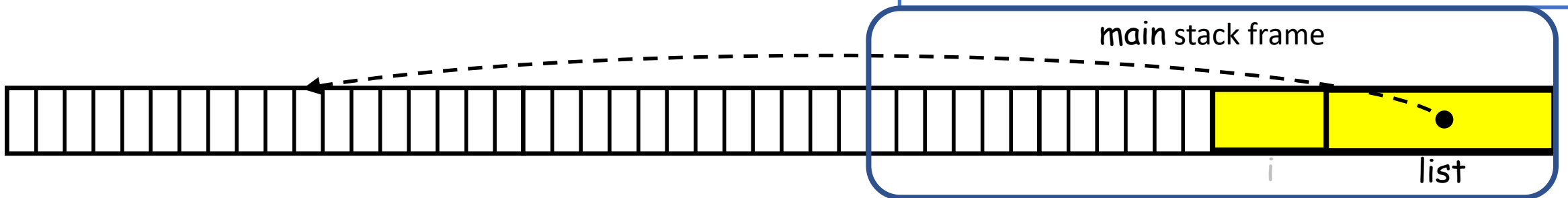
Returning an array in C

Q: Why doesn't this code work?

- Frames get pushed on the stack when the function is called
- **And are popped off when it returns**

A: By the time `list` is assigned the address `a`, the stack frame holding the array values is popped off and the address is no longer valid.

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```



Limitations of arrays allocated in a stack frame

- Arrays (“statically”) allocated on the stack frame have limitations
  - Arrays created on a called function’s frame are not accessible to the calling function once the function returns
    - Returning a is meaningless

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

Limitations of arrays allocated in a stack frame

- Arrays (“statically”) allocated on the stack frame have limitations
  - Arrays created on a called function’s frame are not accessible to the calling function once the function returns
  - Stack frames have fixed (small) size
    - Cannot statically allocate more memory than the size of the stack frame

```
#include <stdio.h>
int main( void )
{
        int values[10000000];
        return 0;
}
```

```
>> ./a.out
Segmentation fault (core dumped)
>>
```

Dynamic Memory Allocation

- Dynamically-allocated memory:
  - Is located on the *heap* – a part of memory separate from the stack
  - Lives as long as we like (until the entire program ends)
    - We don't lose access to it when function call returns
      - ✓ We can return it to a calling function!
      - ✗ We are responsible for managing the memory (and cleaning it up when we're done)
  - Is not subject to frame stack's size limitations because it isn't part of the stack

Dynamic Memory Allocation

- We are responsible for managing the memory
  - Memory should be *deallocated* when it is no longer needed
  - Allocated memory is not available to other programs/users until we deallocate it
  - Failing to deallocate memory is the cause of “memory leaks”

Dynamic Memory Allocation (single)

- We allocate memory using the `malloc` command (in `stdlib.h`)

`void *malloc( size_t );`

  - Input: how much memory (in bytes) to allocate
    - `size_t` is an unsigned integer type:*
      - 4 bytes on 32-bit machines
      - 8 bytes on 64-bit machines
  - Output: the location (on the heap) of the memory
    - `malloc` doesn't need to know what you're going to store just how much memory you need (so it returns a `void*`)

```
…
int *ip = malloc( sizeof( int ) );
```

*On a 32-bit machine you can't ask for more than 2^{32} bytes.

Dynamic Memory Allocation (single)

- We allocate memory using the malloc command (in `stdlib.h`)

  void *malloc(size_t);

- Check that allocation succeeded

```
…
int *ip = malloc( sizeof( int ) );
if( ip==NULL )
{
    fprintf( stderr , "…\n" );
    // do something
}
```

Dynamic Memory Allocation (single)

- We allocate memory using the malloc command (in `stdlib.h`)

  void *malloc(size_t);

- Check that allocation succeeded
- After allocation with malloc, memory cannot be assumed to be initialized

```
…
int *ip = malloc( sizeof( int ) );
if( ip==NULL )
{
     fprintf( stderr , "…\n" );
     // do something
}
*ip = 0;
…
```

Dynamic Memory Allocation (single)

- We allocate memory using the **malloc** command (in `stdlib.h`)

  void *malloc(size_t);

- Check that allocation succeeded
- After allocation with **malloc**, memory cannot be assumed to be initialized
- When done using dynamically-allocated memory, deallocate using **free**

  void free(void *ptr);

  - Input: address of the memory on the heap

```
…
int *ip = malloc( sizeof( int ) );
if( ip==NULL )
{
    fprintf( stderr , "…\n" );
    // do something
}
*ip = 0;
…
free( ip );
```

Dynamic Memory Allocation (single)

- We allocate memory using the malloc command (in `stdlib.h`)

  void *malloc(size_t);

- Check that allocation succeeded
- After allocation with malloc, memory cannot be assumed to be initialized
- When done using dynamically-allocated memory, deallocate using free

  void free(void *ptr);

- It's good practice to set the pointer to NULL to avoid accidental use

```
…
int *ip = malloc( sizeof( int ) );
if( ip==NULL )
{
    fprintf( stderr , "…\n" );
    // do something
}
*ip = 0;
…
if( ip!=NULL ) free( ip );
ip = NULL;
…
```

Dynamic Memory Allocation (multiple)

- We allocate memory using the malloc command (in `stdlib.h`)

  void *malloc(size_t);

- Check that allocation succeeded
- After allocation with malloc, memory cannot be assumed to be initialized
- When done using dynamically-allocate memory, deallocate using free

  void free(void *ptr);

- It's good practice to set the pointer to NULL to avoid accidental use

```
…
int *ip = malloc( sizeof( int ) * sz );
if( ip==NULL )
{
    fprintf( stderr , "…\n" );
    // do something
}
for( int i=0  ;i<sz ; i++ ) ip[i] = 0;
…
if( ip!=NULL ) free( ip );
ip = NULL;
…
```

Dynamic Memory Allocation (multiple)

- We allocate memory using the malloc command (in `stdlib.h`)

  void *malloc(size_t);

- Check that allocation succeeded
- After allocation with malloc, memory cannot be assumed to be initialized
- When done using dynamically-allocate memory, deallocate using free

  void free(void *ptr);

Note:
Call free once for every malloc called.

```
…
int *ip = malloc( sizeof( int ) * sz );
if( ip==NULL )
{
    fprintf( stderr , "…\n" );
    // do something
}
for( int i=0  ;i<sz ; i++ ) ip[i] = 0;
…
if( ip!=NULL ) free( ip );
ip = NULL;
…
```

Deallocation

- Deallocation does not have to happen in the same function where allocation occurred. . .
  - But it does have to happen!
  - Otherwise you can get a *memory leak*

[BADNESS] Only the last allocation was deallocated!

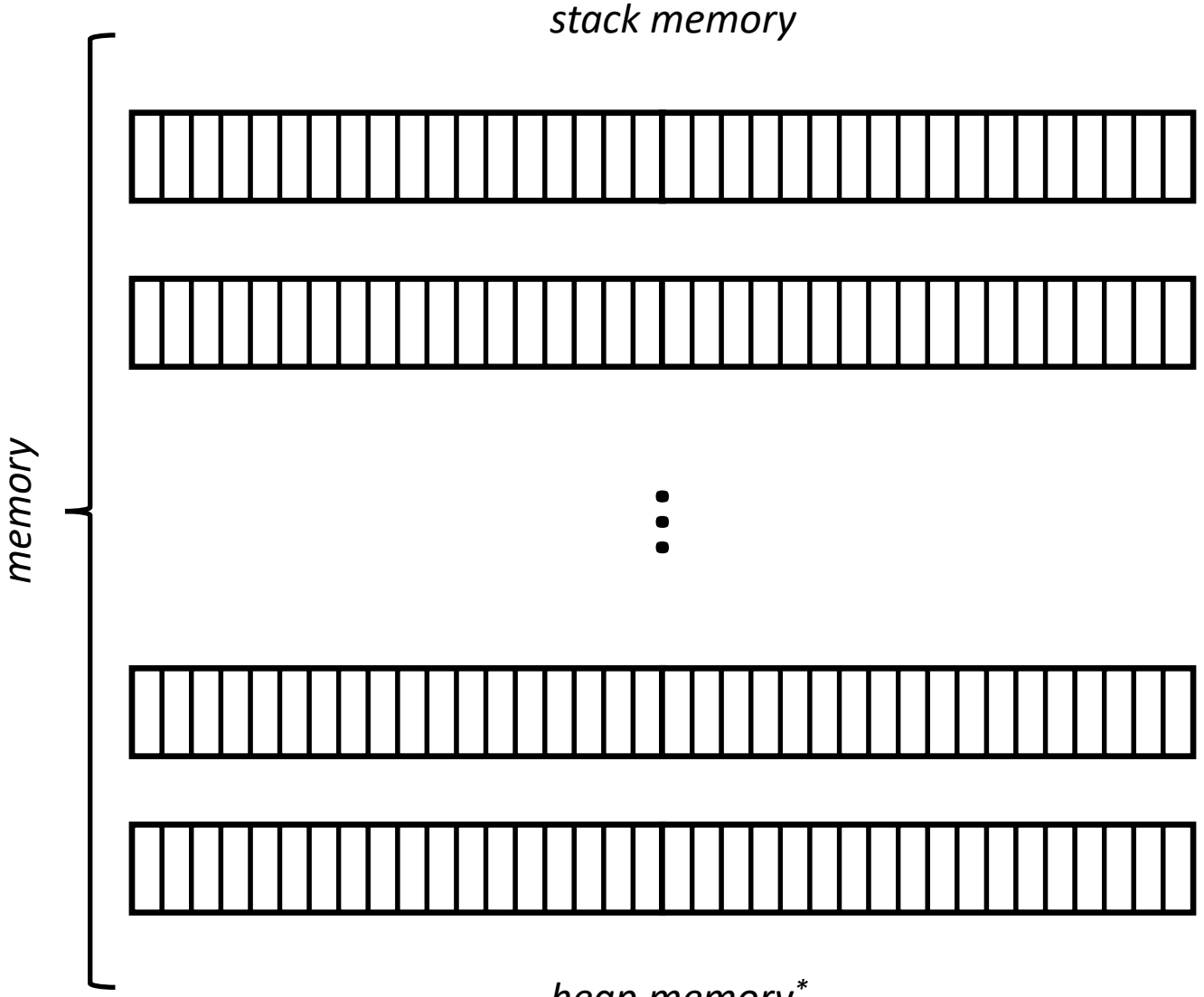
```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 1;
    return a;
}
int main( void )
{
    int *list;
    for( int i=0 ; i<100000 ; i++ )
    {
        list = getArray( 100000 );
        // do something with list
    }
    free( list );
    return 0;
}
```

Deallocation

- Deallocation does not have to happen in the same function where allocation occurred. . .
  - But it does have to happen!
  - Otherwise you can get a *memory leak*

```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 1;
    return a;
}
int main( void )
{
    int *list;
    for( int i=0 ; i<100000 ; i++ )
    {
        list = getArray( 100000 );
        // do something with list
        free( list );
    }
    return 0;
}
```

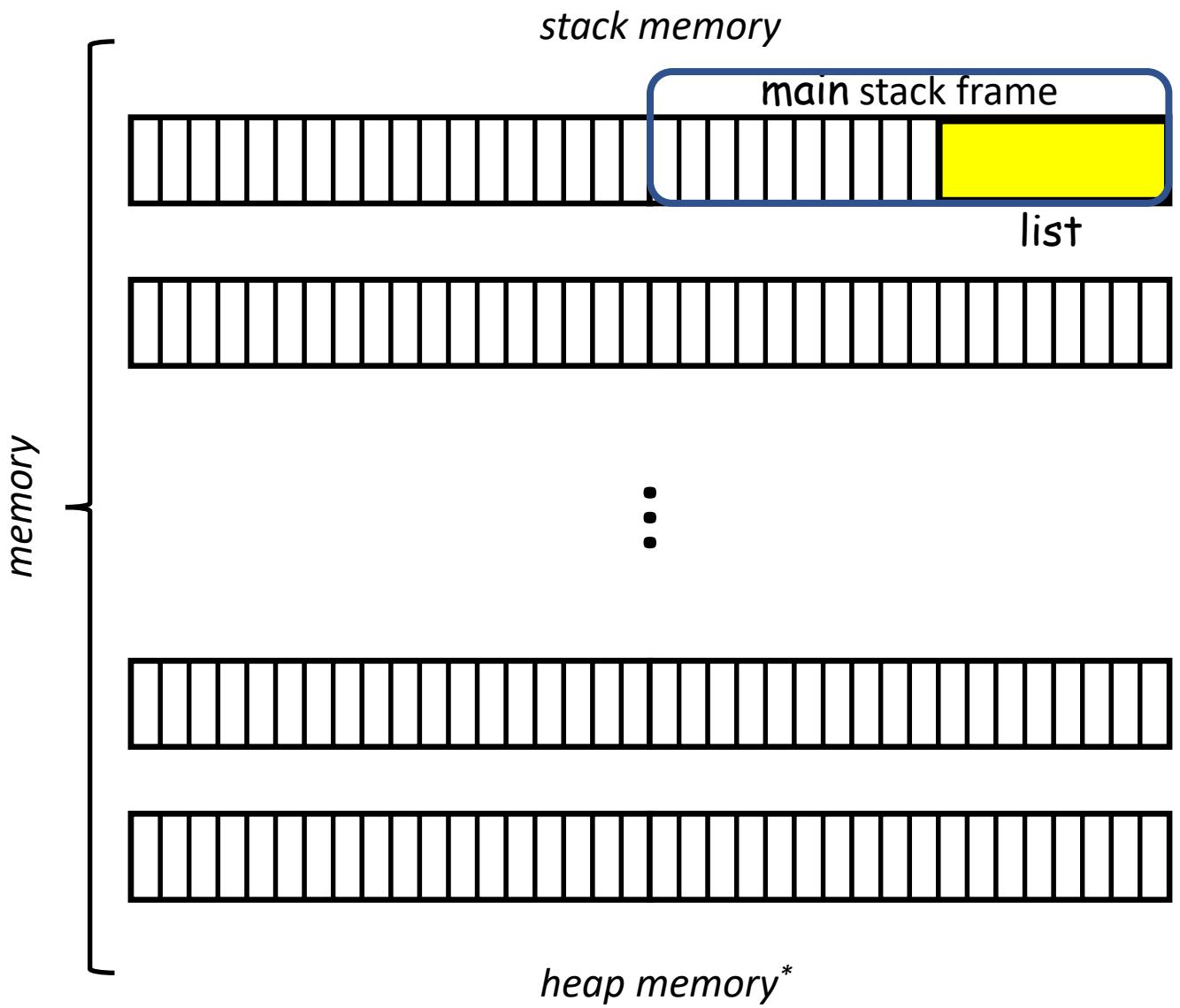
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
      int *a;
      a = malloc( sizeof(int) * sz );
      for( int i=0 ; i<sz ; i++ ) a[i] = 0;
      return a;
}
int main( void )
{
      int *list;
      list = getArray( 8 );
      // do something with list
      free( list );
      list = NULL;
      return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

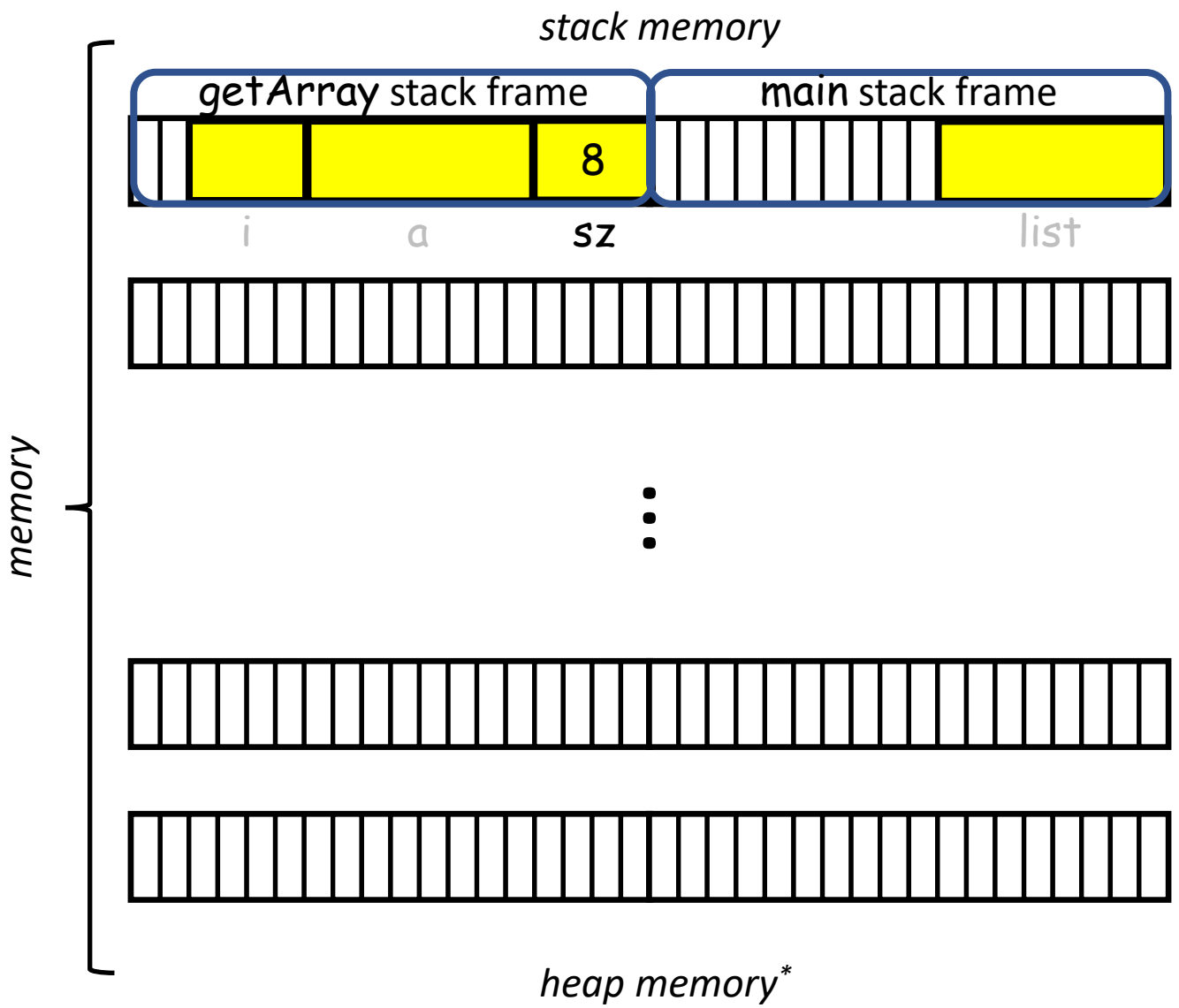
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

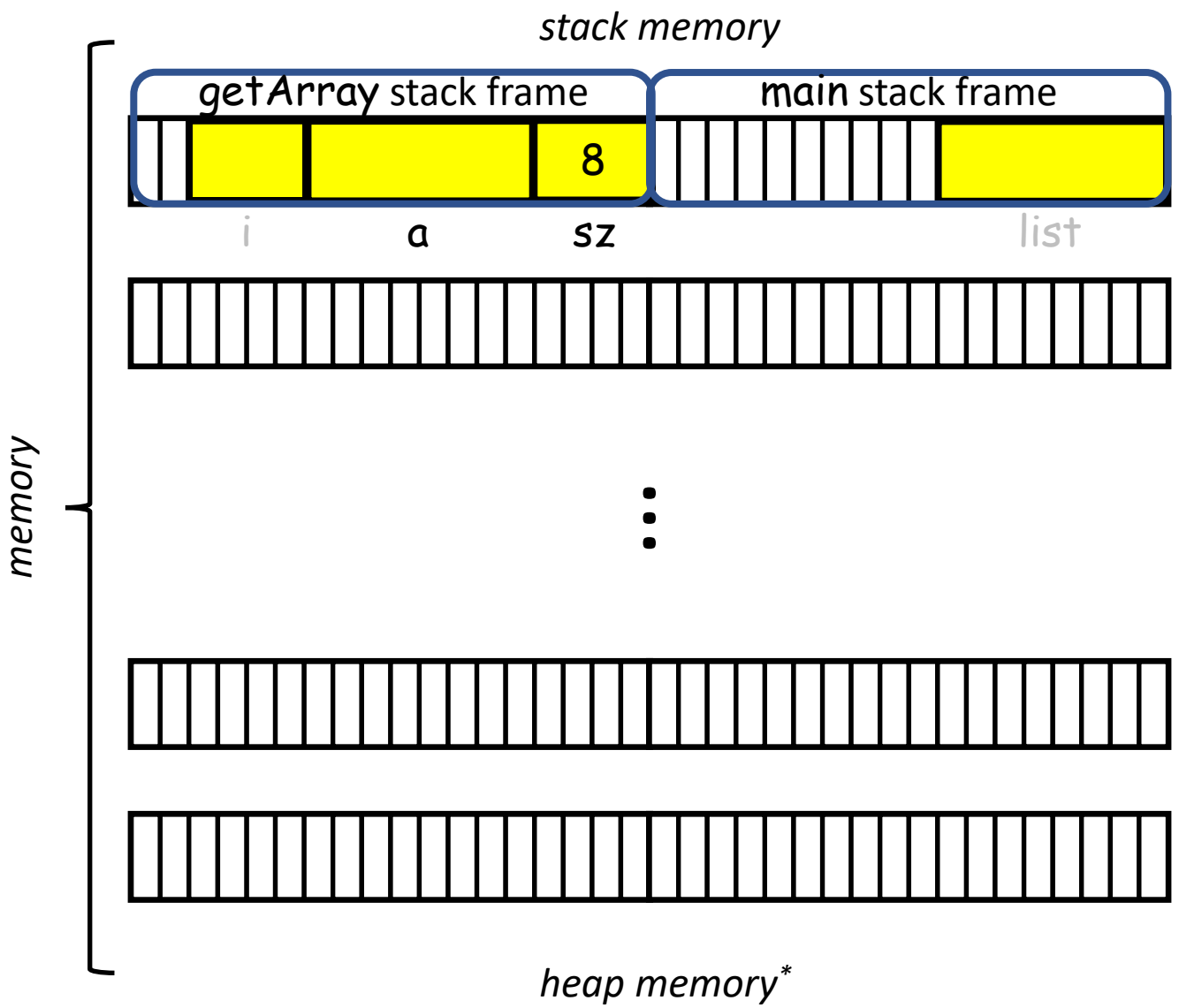
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
      int *a;
      a = malloc( sizeof(int) * sz );
      for( int i=0 ; i<sz ; i++ ) a[i] = 0;
      return a;
}
int main( void )
{
      int *list;
      list = getArray( 8 );
      // do something with list
      free( list );
      list = NULL;
      return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

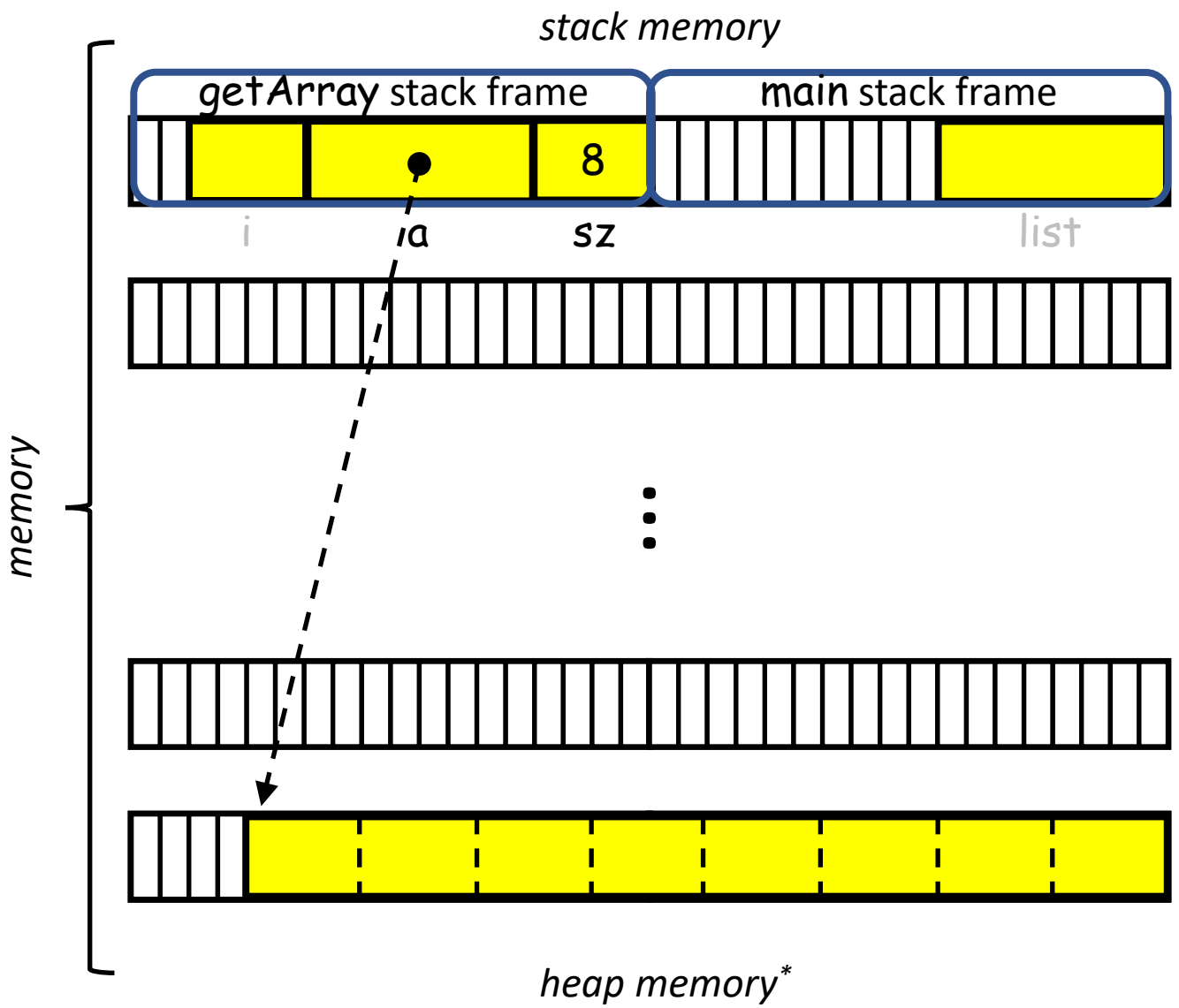
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

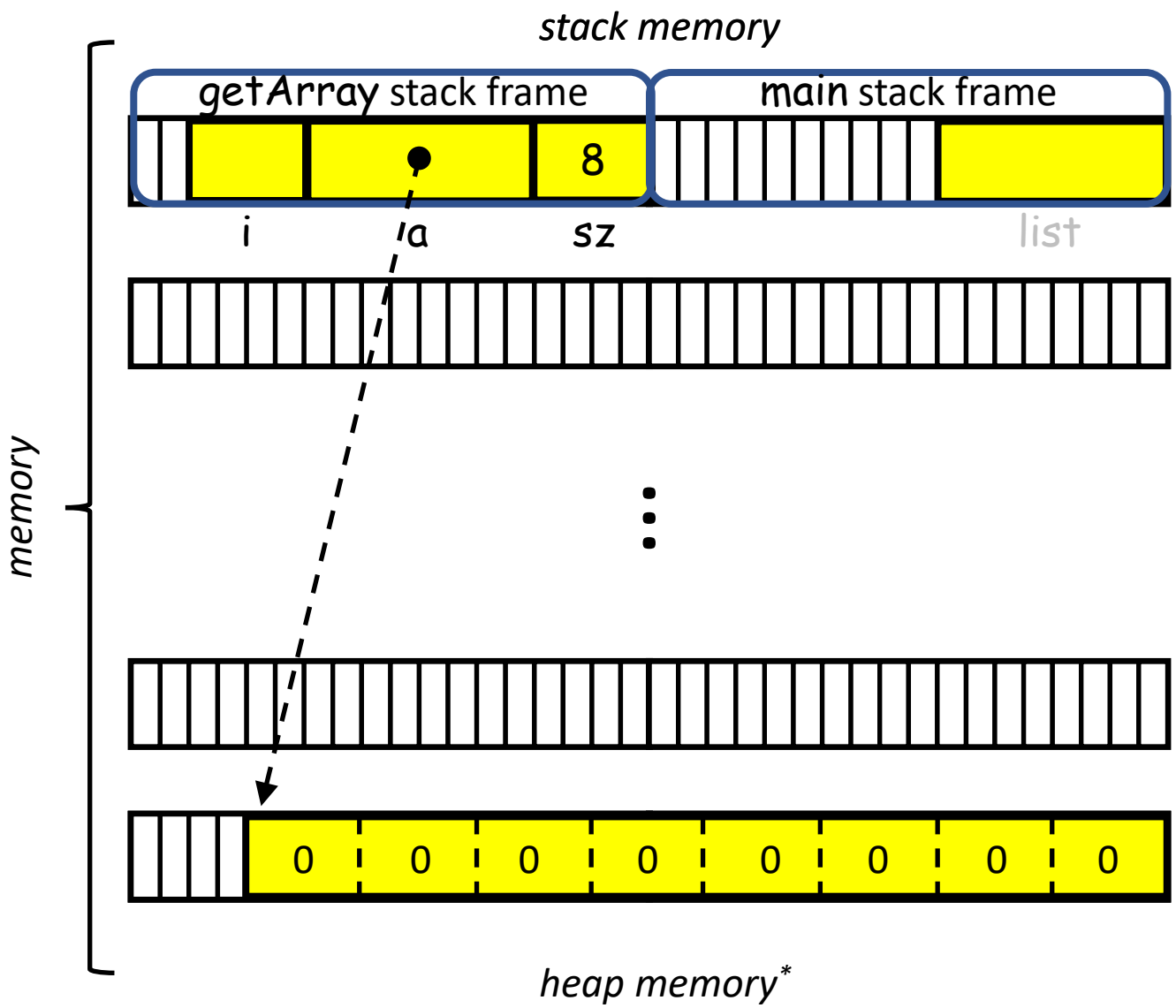
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
      int *a;
      a = malloc( sizeof(int) * sz );
      for( int i=0 ; i<sz ; i++ ) a[i] = 0;
      return a;
}
int main( void )
{
      int *list;
      list = getArray( 8 );
      // do something with list
      free( list );
      list = NULL;
      return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

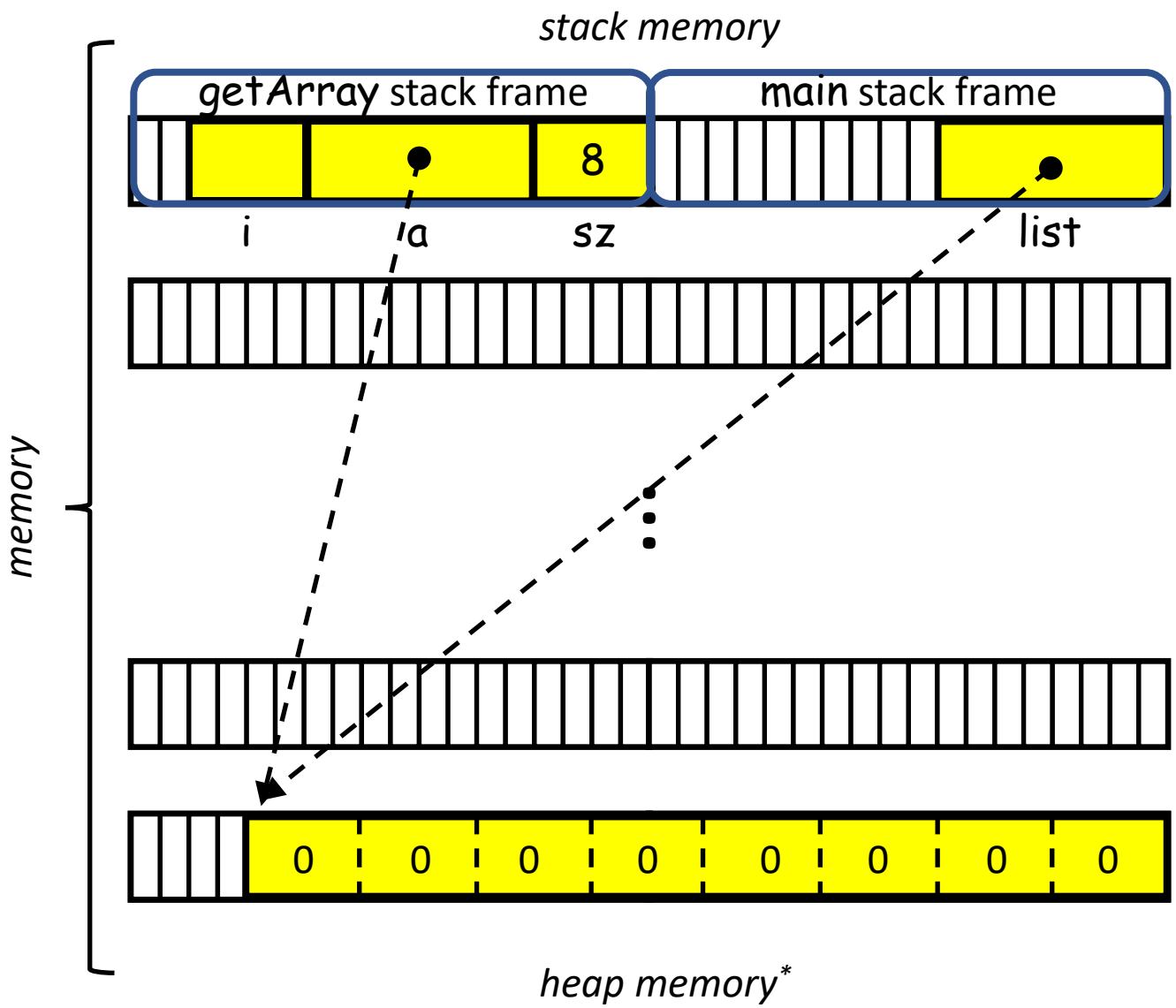
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

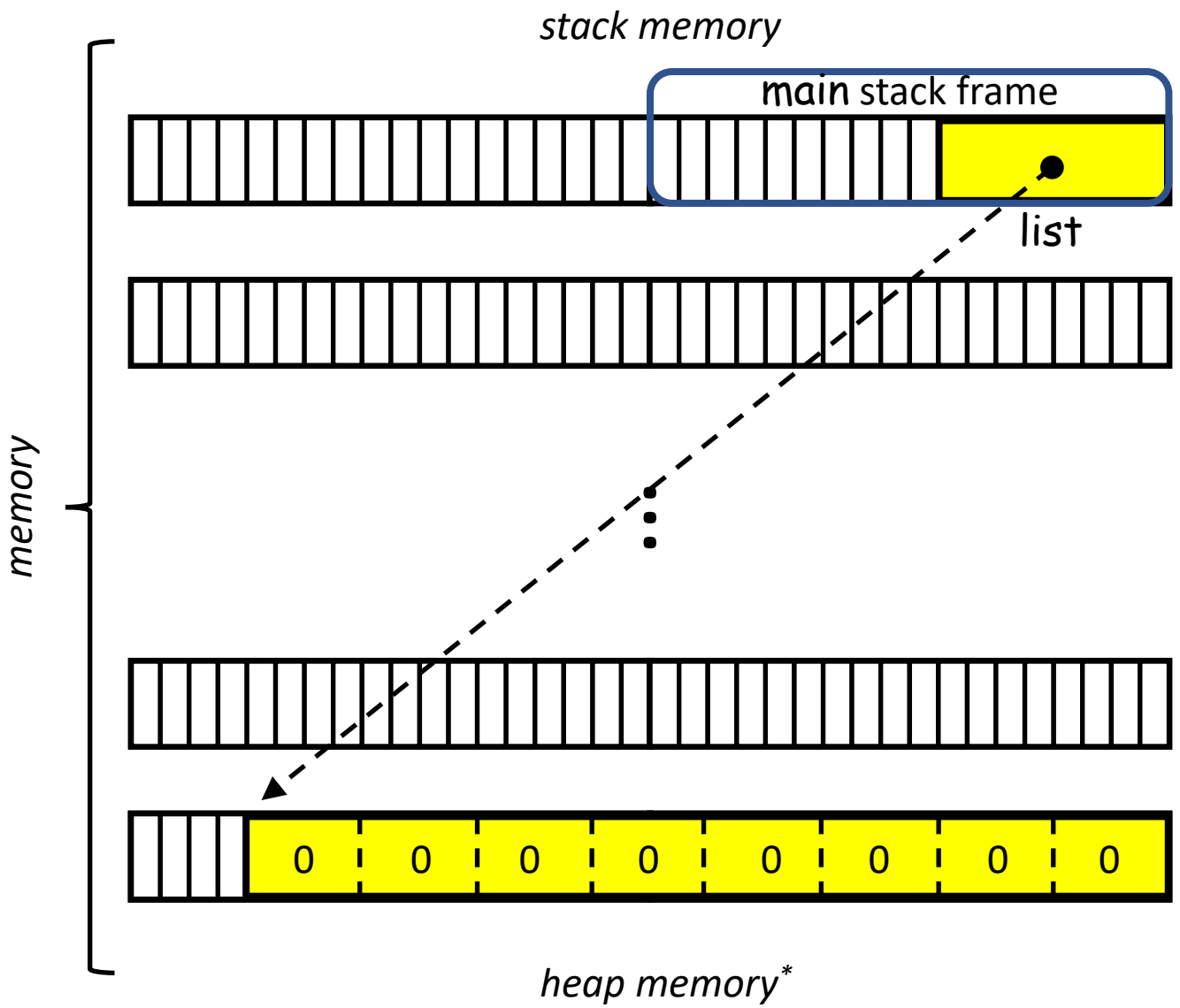
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

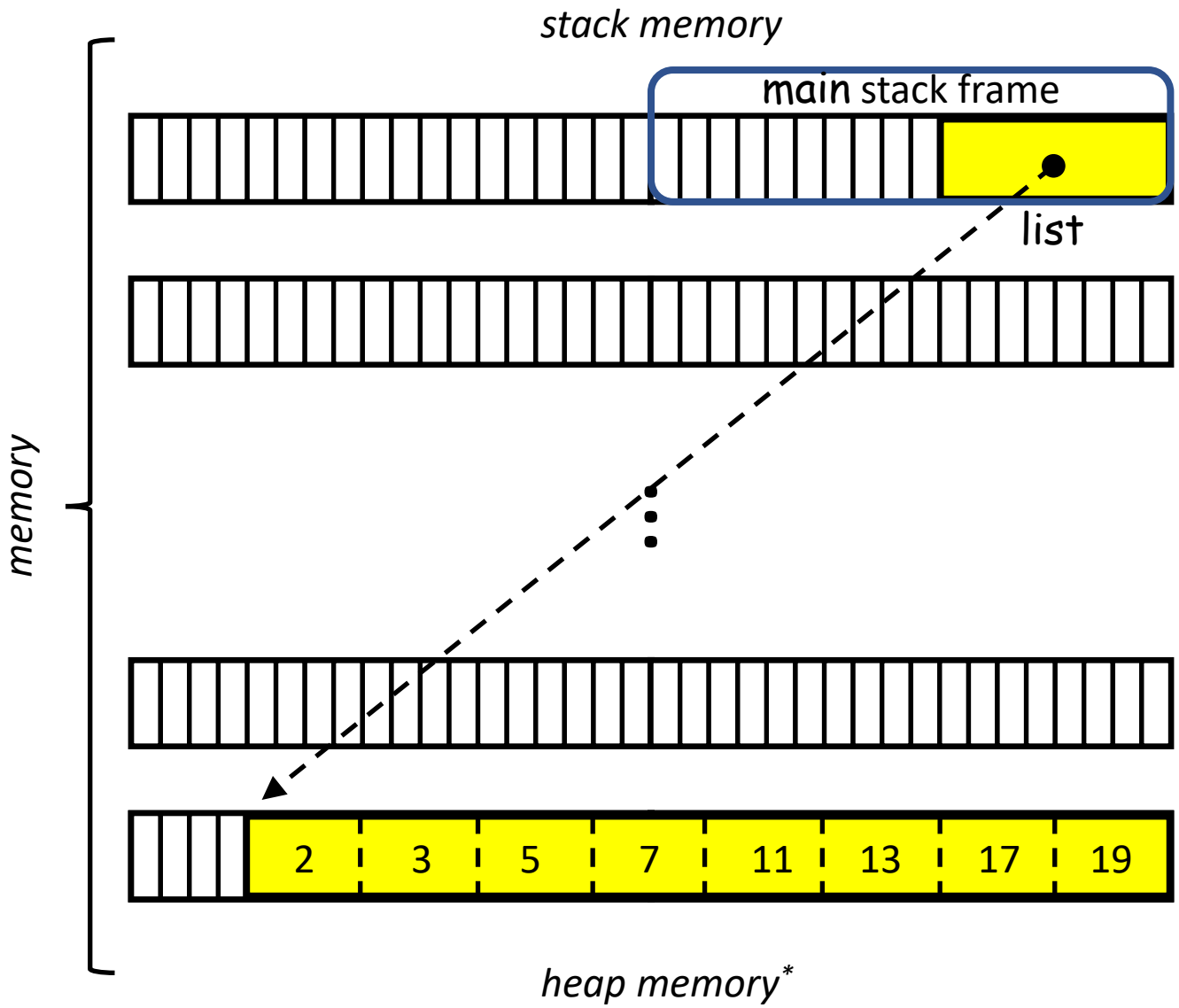
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

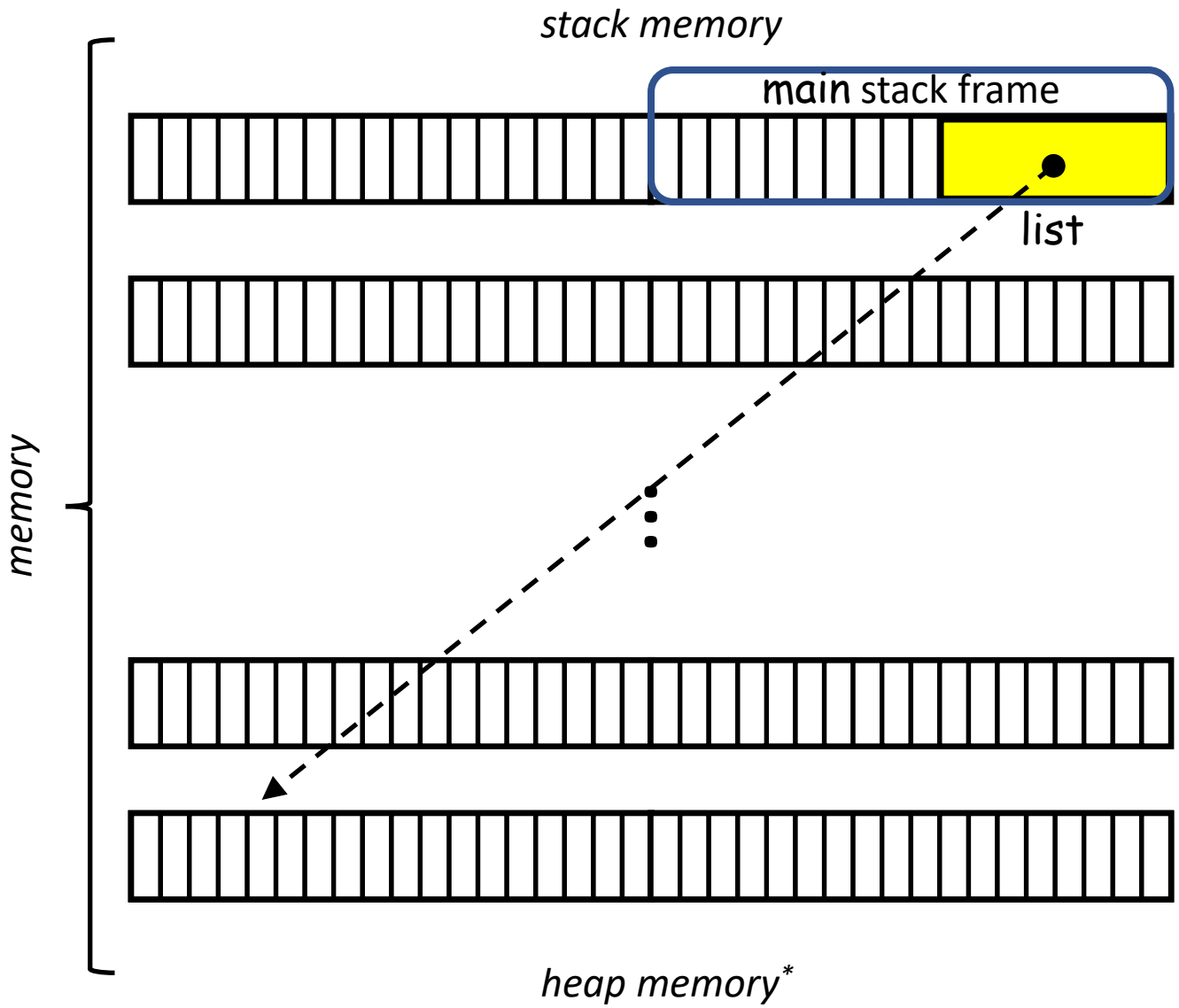
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

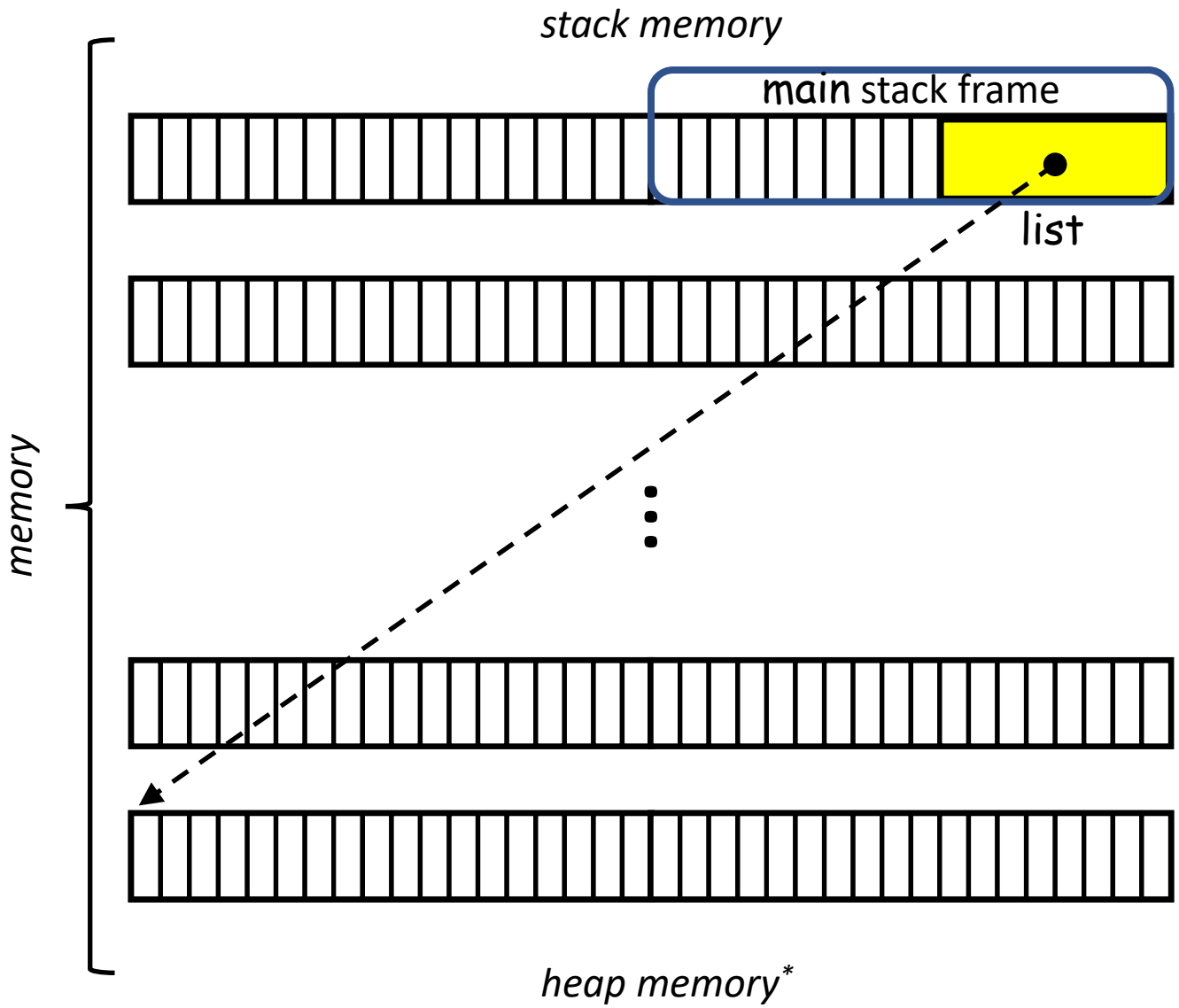
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

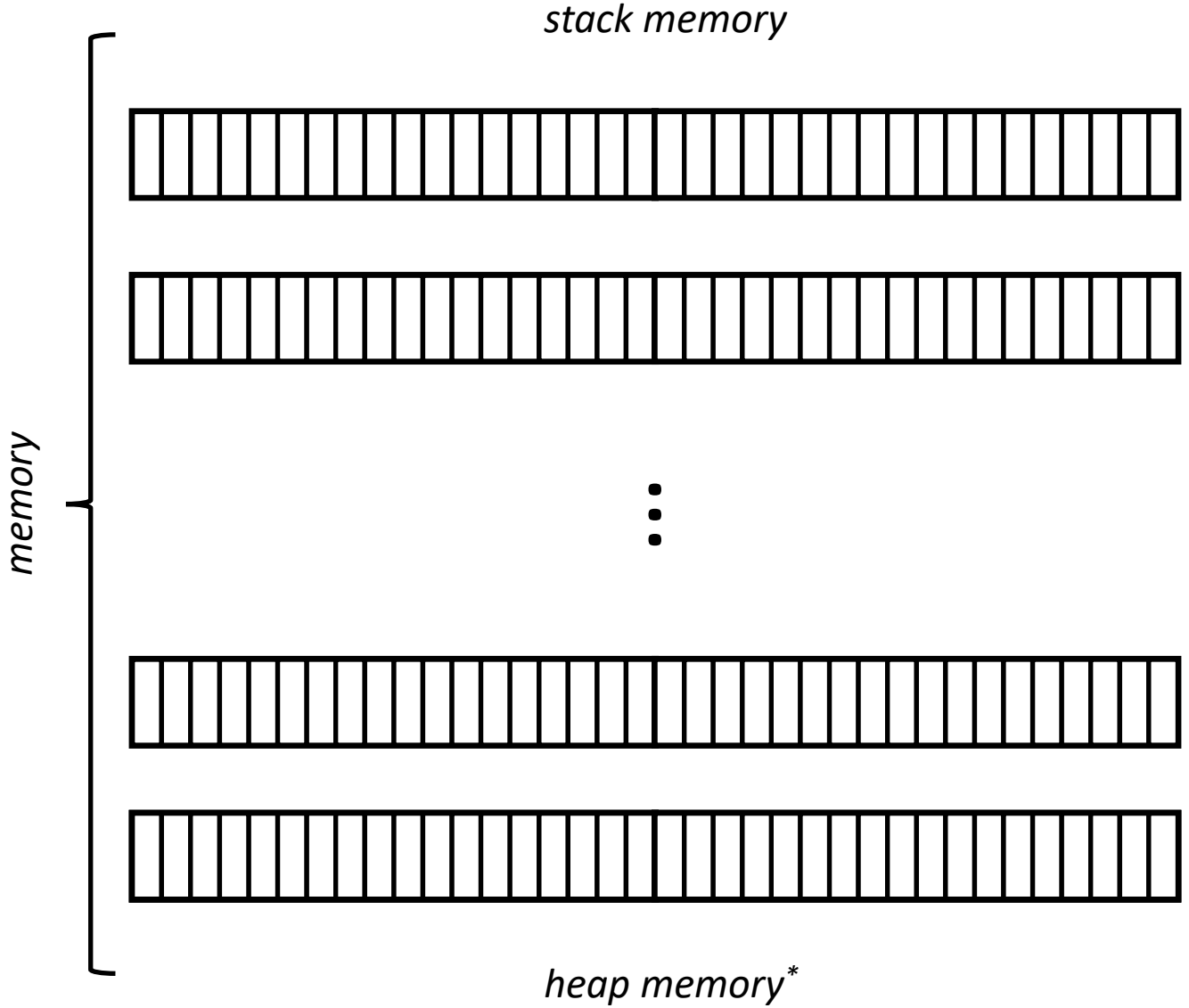
Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

Dynamic Memory Allocation




```
#include <stdio.h>
#include <stdlib.h>
int *getArray( unsigned int sz )
{
    int *a;
    a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 0;
    return a;
}
int main( void )
{
    int *list;
    list = getArray( 8 );
    // do something with list
    free( list );
    list = NULL;
    return 0;
}
```

* Visualization is not to scale:
There is much more room on the heap than on the stack.

Working w/ Memory (`stdlib.h`)

- Allocate and clear memory:

  `void *calloc( size_t num , size_t size );`

  Input:
  - num: the number of elements
  - size: the size of each element (undefined behavior if zero)

  Output:
  - On success: a pointer to the allocated and cleared memory
  - On failure: a null-pointer

```
>> ./a.out
0] 0
1] 0
2] 0
3] 0
>>
```

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
	int *v = calloc( 4 , sizeof(int) );
	if( !v ) …
	for( int i=0 ; i<4 ; i++ ) printf( "%d] %d\n" , i , v[i] );
	free( v );
	return 0;
}
```

Working w/ Memory (`stdlib.h`)

- Change the size of memory pointed to by a pointer while preserving the data (up to the minimum of the old and new sizes):

  `void *realloc( void *ptr , size_t size );`

  Input:
  - `ptr`: pointer to memory on the heap (if `NULL`, `realloc` behaves like `malloc`)
  - `size`: new size for the memory block (if 0, `realloc` behaves like `free`)

  Output:
  - On failure or if `size`=0: a null-pointer
  - On success: a pointer to the new memory with old values preserved up to `num` bytes (may or may not be the address of the old memory)

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
	int *v = malloc( sizeof(int)*3 );
	for( int i=0 ; i<3 ; i++ ) v[i] = i;
	v = realloc( v , sizeof(int)*4 );
	v[3] = 3;
	for( int i=0 ; i<4 ; i++)
		printf( "%d] %d\n" , i , v[i] );
	free( v );
	return 0;
}
```

```
>> ./a.out
0] 0
1] 1
2] 2
3] 3
>>
```

Working w/ Memory (`stdlib.h`)

- Change the size of memory pointed to by a pointer while preserving the data (up to the minimum of the old and new sizes):

```
void *realloc( void *ptr , size_t size );
```

WARNING:
`realloc` will try to extend/shrink the memory in place but is not guaranteed to (e.g. if there isn't sufficient space to grow).
⇒ Do not assume that the input and output pointers are equal

- In that case, `realloc` handles deallocation of the old memory (e.g. there is no call "`free(v);`")

- On failure or if `size`=0: a null-pointer
- On success: a pointer to the new memory with old values preserved up to `num` bytes (may or may not be the address of the old memory)

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
	int *v = malloc( sizeof(int)*3 );
	for( int i=0 ; i<3 ; i++ ) v[i] = i;
	v = realloc( v , sizeof(int)*4 );
	v[3] = 3;
	for( int i=0 ; i<4 ; i++)
		printf( "%d] %d\n" , i , v[i] );
	free( v );
	return 0;
}
```

```
>> ./a.out
0] 0
1] 1
2] 2
3] 3
>>
```

Outline

- Dynamic memory allocation
- `Valgrind`
- Review questions

Valgrind

- Easy-to-use tool for finding memory leaks and other memory issues
- Compile with `-g` to get more helpful output from `valgrind`
- Then run using `valgrind`:

```
valgrind --leak-check=full ./myFile <arg1> <arg2> ...
```

```
#include <stdio.h>
int main( void )
{
    printf( "Hello world!\n" );
    return 0;
}
```

```
>> gcc -std=c99 -Wall -Wextra -g foo.c
>> ./a.out
Hello world!
>>
```

See also http://valgrind.org/docs/manual/QuickStart.html

Valgrind

- Easy-to-use tool for finding memory leaks and other memory issues
- Compile with `-g` to get more helpful output from `valgrind`
- Then run using `valgrind`:

va

```
>> valgrind --leak-check=full ./a.out
```

header

```
==12133== Command: ./a.out
==12133==
Hello World!
==12133==
==12133== HEAP SUMMARY:
==12133==     in use at exit: 0 bytes in 0 blocks
==12133==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==12133==
==12133== All heap blocks were freed -- no leaks are possible
==12133==
==12133== For counts of detected and suppressed errors, rerun with: -v
==12133== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
>>
```

>

world!\n");

Wextra -g foo.c

See also http://va

Valgrind

- Your program output is interspersed with messages from `valgrind`
- Kinds of issues flagged by `valgrind`
  - Invalid reads or writes:
    Attempts to dereference pointers to memory that's not yours
  - Memory leaks:
    Failing to deallocate a block of memory you allocate previously
    - Info about leaks appears in HEAP SUMMARY section

Valgrind

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <assert.h>
5.  char *string_copy( const char *in )
6.  {
7.      char *out = malloc( strlen( in ) );
8.      assert( out!= NULL );
9.      return strcpy( out , in );
10. }
11. int main( void )
12. {
13.     char *str = string_copy( "hello" );
14.     assert( str!=NULL );
15.     printf( "%s\n" , str );
16.     return 0;
17. }
```

```
>> ./a.out
hello
>>
```

```
>> valgrind --leak-check=full ./a.out
```

header

```
==17647== Command: ./a.out
==17647==
==17647== Invalid write of size 1
==17647==    at 0x4C30CB7: strcpy (vg_replace_strmem.c:506)
==17647==    by 0x40067C: string_copy (foo.c:9)
==17647==    by 0x400690: main (foo.c:13)
==17647==  Address 0x5200045 is 0 bytes after a block of size 5 alloc'd
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
==17647== Invalid read of size 1
==17647==    at 0x4C30BC4: strlen (vg_replace_strmem.c:454)
==17647==    by 0x4EAAAA1: puts (in /usr/lib64/libc-2.24.so)
==17647==    by 0x4006C0: main (foo.c:15)
==17647==  Address 0x5200045 is 0 bytes after a block of size 5 alloc'd
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
hello
==17647==
```

first half of output

Valgrind

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <assert.h>
5.  char *string_copy( const char *in )
6.  {
7.      char *out = malloc( strlen( in ) );
8.      assert( out!= NULL );
9.      return strcpy( out , in );
10. }
11. int main( void )
12. {
13.     char *str = string_copy( "hello" );
14.     assert( str!=NULL );
15.     printf( "%s\n" , str );
16.     return 0;
17. }
```

```
>> ./a.out
hello
>>
```

second half of output

```
==17647== HEAP SUMMARY:
==17647==     in use at exit: 5 bytes in 1 blocks
==17647==   total heap usage: 2 allocs, 1 frees, 1,029 bytes allocated
==17647==
==17647== 5 bytes in 1 blocks are definitely lost in loss record 1 of 1
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
==17647== LEAK SUMMARY:
==17647==    definitely lost: 5 bytes in 1 blocks
==17647==    indirectly lost: 0 bytes in 0 blocks
==17647==      possibly lost: 0 bytes in 0 blocks
==17647==    still reachable: 0 bytes in 0 blocks
==17647==         suppressed: 0 bytes in 0 blocks
==17647==
==17647== For counts of detected and suppressed errors, rerun with: -v
==17647== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
>>
```

Valgrind

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <assert.h>
5.  char *string_copy( const char *in )
6.  {
7.      char *out = malloc( strlen( in ) );
8.      assert( out!= NULL );
9.      return strcpy( out , in );
10. }
11. int main( void )
12. {
13.     char *str = string_copy( "hello" );
14.     assert( str!=NULL );
15.     printf( "%s\n" , str );
16.     return 0;
17. }
```

```
>> ./a.out
hello
>>
```

```
>> valgrind --leak-check=full ./a.out
```

header

```
==17647== Command: ./a.out
==17647==
==17647== Invalid write of size 1
==17647==    at 0x4C30CB7: strcpy (vg_replace_strmem.c:506)
==17647==    by 0x40067C: string_copy (foo.c:9)
==17647==    by 0x400690: main (foo.c:13)
==17647==  Address 0x5200045 is 0 bytes after a block of size 5 alloc'd
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
==17647== Invalid read of size 1
==17647==    at 0x4C30BC4: strlen (vg_replace_strmem.c:454)
==17647==    by 0x4EAAAA1: puts (in /usr/lib64/libc-2.24.so)
==17647==    by 0x4006C0: main (foo.c:15)
==17647==  Address 0x5200045 is 0 bytes after a block of size 5 alloc'd
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
hello
==17647==
```

first half of output

Valgrind

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <assert.h>
5.  char *string_copy( const char *in )
6.  {
7.      char *out = malloc( strlen( in ) );
8.      assert( out!= NULL );
9.      return strcpy( out , in );
10. }
11. int main( void )
12. {
13.     char *str = string_copy( "hello" );
14.     assert( str!=NULL );
15.     printf( "%s\n" , str );
16.     return 0;
17. }
```

```
>> ./a.out
hello
>>
```

```
>> valgrind --leak-check=full ./a.out
```

header

```
==17647== Command: ./a.out
==17647==
==17647== Invalid write of size 1
==17647==    at 0x4C30CB7: strcpy (vg_replace_strmem.c:506)
==17647==    by 0x40067C: string_copy (foo.c:9)
==17647==    by 0x400690: main (foo.c:13)
==17647==  Address 0x5200045 is 0 bytes after a block of size 5 alloc'd
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
==17647== Invalid read of size 1
==17647==    at 0x4C30BC4: strlen (vg_replace_strmem.c:454)
==17647==    by 0x4EAAAA1: puts (in /usr/lib64/libc-2.24.so)
==17647==    by 0x4006C0: main (foo.c:15)
==17647==  Address 0x5200045 is 0 bytes after a block of size 5 alloc'd
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
hello
==17647==
```

first half of output

Valgrind

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <assert.h>
5.  char *string_copy( const char *in )
6.  {
7.      char *out = malloc( strlen( in ) );
8.      assert( out!= NULL );
9.      return strcpy( out , in );
10. }
11. int main( void )
12. {
13.     char *str = string_copy( "hello" );
14.     assert( str!=NULL );
15.     printf( "%s\n" , str );
16.     return 0;
17. }
```

```
>> ./a.out
hello
>>
```

second half of output

```
==17647== HEAP SUMMARY:
==17647==     in use at exit: 5 bytes in 1 blocks
==17647==   total heap usage: 2 allocs, 1 frees, 1,029 bytes allocated
==17647==
==17647== 5 bytes in 1 blocks are definitely lost in loss record 1 of 1
==17647==    at 0x4C2DB9D: malloc (vg_replace_malloc.c:299)
==17647==    by 0x400645: string_copy (foo.c:7)
==17647==    by 0x400690: main (foo.c:13)
==17647==
==17647== LEAK SUMMARY:
==17647==    definitely lost: 5 bytes in 1 blocks
==17647==    indirectly lost: 0 bytes in 0 blocks
==17647==      possibly lost: 0 bytes in 0 blocks
==17647==    still reachable: 0 bytes in 0 blocks
==17647==         suppressed: 0 bytes in 0 blocks
==17647==
==17647== For counts of detected and suppressed errors, rerun with: -v
==17647== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
>>
```

Valgrind

- So what was wrong?
  - An invalid write
  - An invalid read
  - A block of memory that wasn't `free`d

- On `ugradx`, this program didn't crash and seemed to work properly

⇒ Not every bad memory access leads to error (or bad output)

⇒ `valgrind` is a useful tool to help us find problematic code

Valgrind

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <assert.h>
5.  char *string_copy( const char *in )
6.  {
7.      char *out = malloc( strlen( in )+1 );
8.      assert( out!= NULL );
9.      return strcpy( out , in );
10. }
11. int main( void )
12. {
13.     char *str = string_copy( "hello" );
14.     assert( str!=NULL );
15.     printf( "%s\n" , str );
16.     free( str );
17.     return 0;
18. }
```

```
>> ./a.out
hello
>>
```

```
>> valgrind --leak-check=full ./a.out
                                                    header
==30398== Command: ./a.out
==30398==
hello
==30398==
==30398== HEAP SUMMARY:
==30398==     in use at exit: 0 bytes in 0 blocks
==30398==   total heap usage: 2 allocs, 2 frees, 1,030 bytes allocated
==30398==
==30398== All heap blocks were freed -- no leaks are possible
==30398==
==30398== For counts of detected and suppressed errors, rerun with: -v
==30398== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
>>
```

Outline

- Dynamic memory allocation
- `Valgrind`
- Review questions

Review questions

1. What is the difference between stack and heap memory?

Review questions

2. What is dynamic memory allocation in C?

Review questions

3. What is the memory leak problem?

Review questions

4. What is the difference between malloc, realloc, and calloc?

Review questions

5. What do we use `valgrind` to check for?

Review questions

6. Consider the exclaim function below. Do you see any problems with this function?

```
// Return a C character string containing n exclamation points.
// n must be less than 20.
char *exclaim( int n )
{
        char s[20];
        assert( n<20 );
        for( int i=0 ; i<n ; i++ ) s[i] = '!';
        s[n] = '\0';
        return s;
}
```

Exercise 4-2

- Website -> Course Materials -> Ex4-2