600.120 Intermediate Programming, Spring 2017*

Misha Kazhdan

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

Anouncements

- Homework 2 is due Wednesday

Outline

- Function calls and stack frames
- Pointers
- Using statically-allocated arrays
- Dynamic memory allocation basics

Writing a **swap** function in C

Q: Why doesn't this code work?

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b =2;
        swap( a ,b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

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

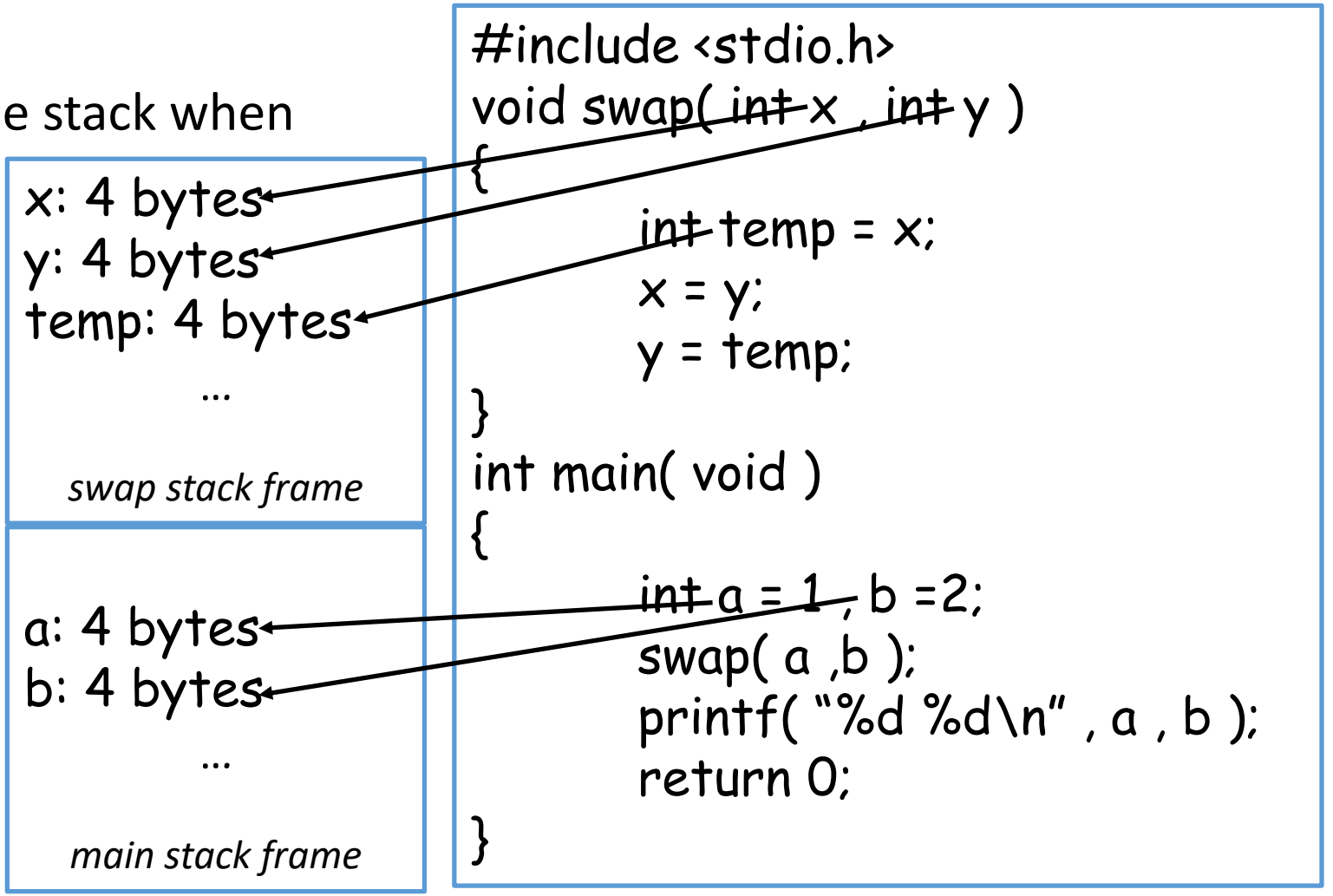
Local variables

- Each function call gets a new *stack frame* “pushed onto” the stack in memory for storing
  - Local variables
  - Copies of function parameters
  - 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)

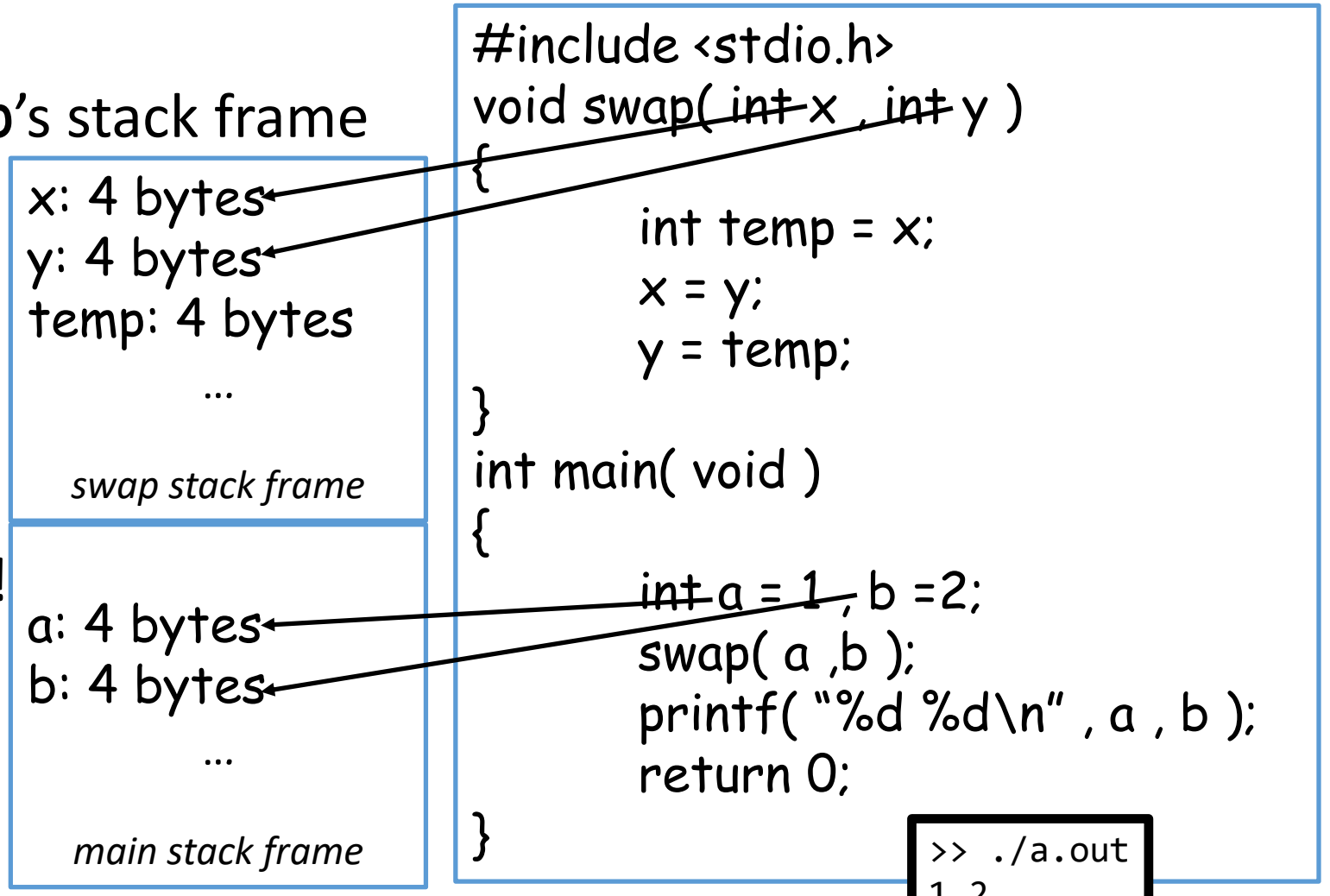
Local variables

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



Writing a **swap** function in C

- Adjusting values in **swap**'s stack frame has no impact on values in **main**'s stack frame
- What we need is to give **swap** function access to **main**'s stack frame
- This is a job for *pointers*!



Pointers

- A *pointer* is a variable that contains the address of a variable.
  - Every pointer points to a specific data type (except a *pointer to void*, more on that later)
  - Declare a pointer using type of variable it will point to, and a "*":
    - "int *iP" is a pointer to an int
    - "double *dP" is a pointer to a double
    - etc.
- Operations related to pointers
  - address-of operator &: returns address of whatever follows it
  - dereferencing operator *: returns value being pointed to

Pointers

- A *pointer* is a variable that contains the address of a variable.
  - Every pointer points to a specific data type (except a *pointer to void*, more on that later)
  - Declare a pointer using type of variable it will point to, and a "*":
    - "int *iP" is a pointer to an int
    - "double *dP" is a pointer to a double
    - etc.
- Operations related to pointers
  - address-of operator &: returns address of whatever follows it

Note:
When declaring a pointer, the "*" needs to be associated with the variable name, not the type

- int * a , b; ⇔ declares a pointer to an int called a and an int called b
- int * a , * b; ⇔ declares a pointer to an int called a and a pointer to an int called b

Pointers

- A *pointer* is a variable that contain
  - Every pointer points to a specific da
    (except a *pointer to void*, but more
  - Declare a pointer using type of varia
    - "int *iP" is a pointer to an int
    - "double *dP" is a pointer to a double
    - etc.
- Operations related to pointers
  - address-of operator &: returns addr
  - dereferencing operator *: returns v

```
#include <stdio.h>
int main( void )
{
        int x = 1 , y =2; // ints
        int *iP;          // a pointer to an int
        iP = &x;          // that points to x
        y = *iP;          // y has the value of
                          // what iP points to (x)
        *iP = 0;          // what iP points to (x)
                          // has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```

```
>> ./a.out
0 1
>>
```

A working **swap** function

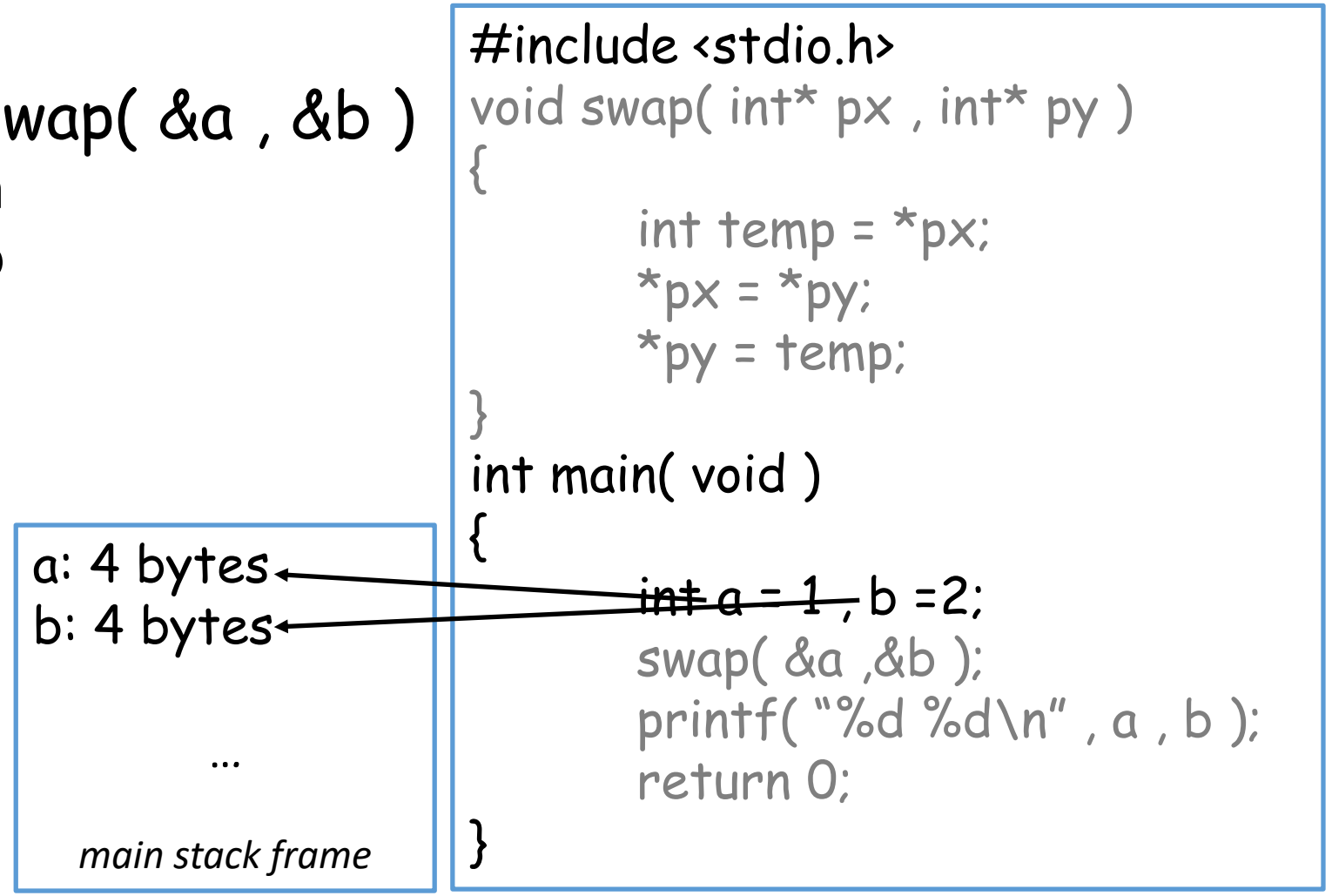
- The call in **main** is now **swap(&a , &b)** since we need to send in the addresses of **a** and **b**
- Pointer arguments allow a called function to access and modify values in the calling function

```
#include <stdio.h>
void swap( int* px , int* py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b =2;
        swap( &a ,&b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

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

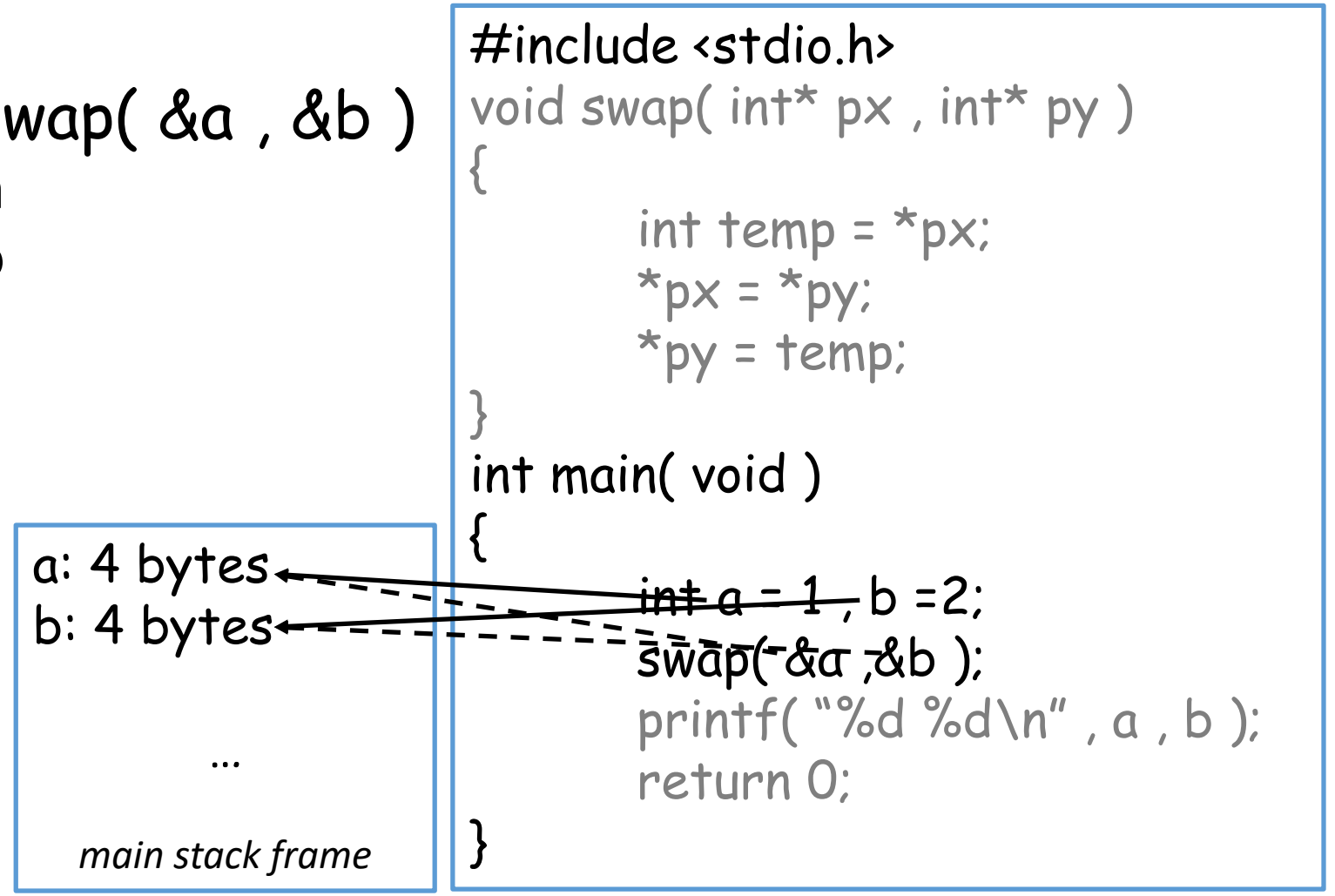
A working **swap** function

- The call in **main** is now **swap(&a , &b)** since we need to send in the addresses of **a** and **b**
- Pointer arguments allow a called function to access and modify values in the calling function



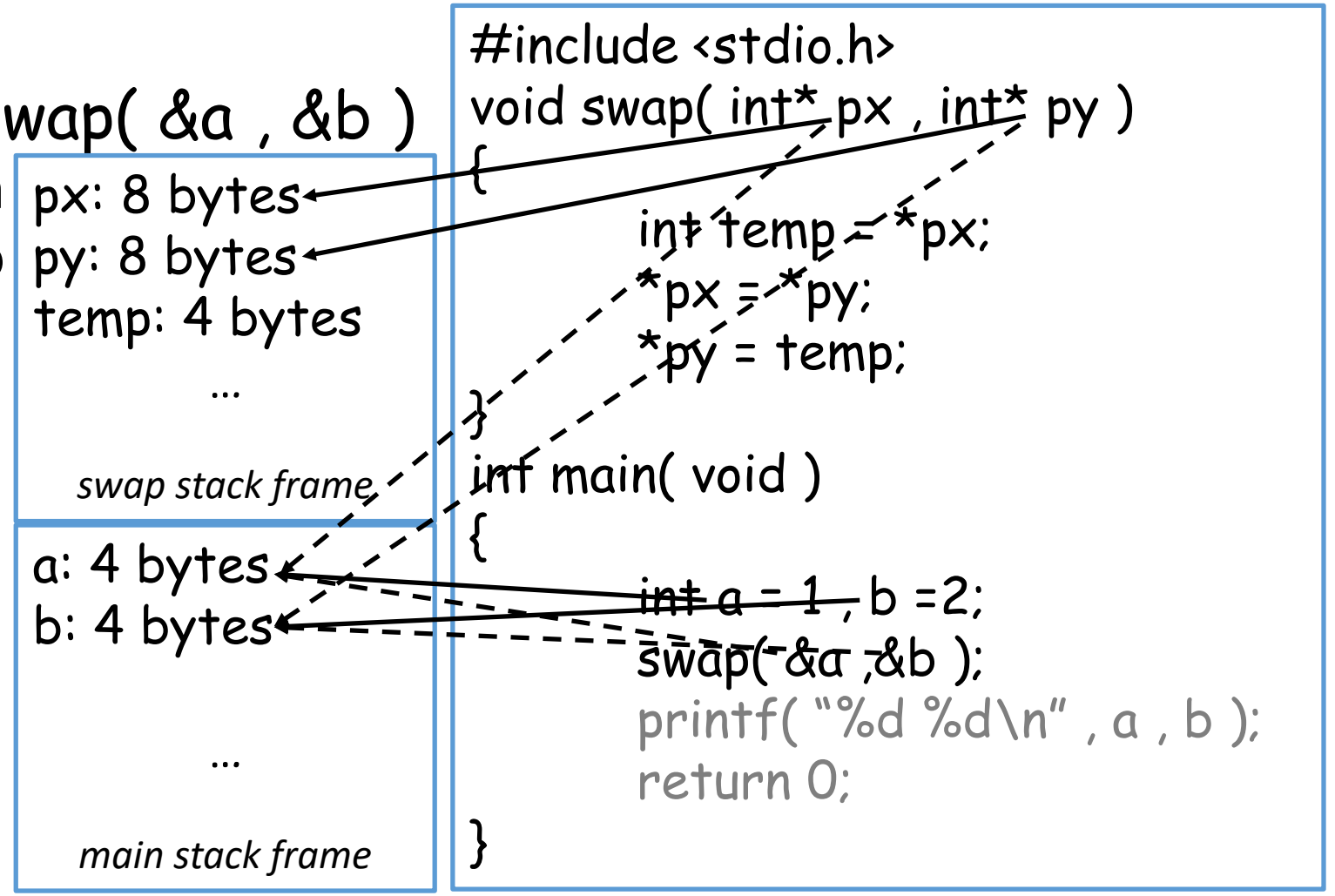
A working **swap** function

- The call in **main** is now **swap(&a , &b)** since we need to send in the addresses of **a** and **b**
- Pointer arguments allow a called function to access and modify values in the calling function



A working **swap** function

- The call in **main** is now **swap(&a , &b)** since we need to send in the addresses of **a** and **b**
- Pointer arguments allow a called function to access and modify values in the calling function



Using & and *

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

Pointer arithmetic

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

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

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 , 4 };
    int* b = a+2;

    return 0;
}
```

Pointer arithmetic

- A pointer is an 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[] = { 1 , 2 , 3 , 4 };
    int* b = a+2;

    return 0;
}
```

Pointer arithmetic

- A pointer is an 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 corresponds to 8 bytes

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

```
>> ./a.out
1 3
>>
```

Pointer arithmetic

- A pointer is an 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

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

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

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 corresponds to 8 bytes
- A pointer of type void* is treated as a raw memory address (w/o size information)

Pointer arithmetic

- A pointer is an 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 ip,
2. Return the old pointer's value,*
3. Dereference that
4. Set it to zero

```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 , 4 };
    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
1 2 0 4 : 4
>>
```

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

Pointer access

- 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 the same these are the same operations!

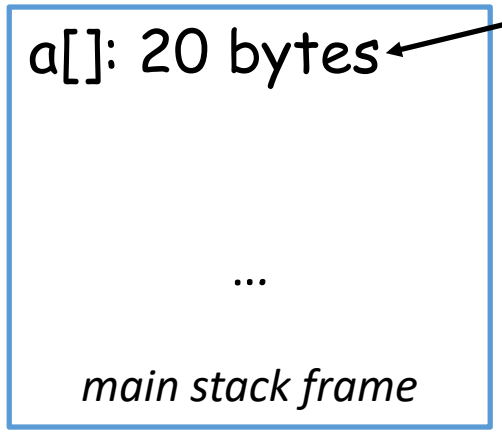
```
#include <stdio.h>
int main( void )
{
    int a[] = { 1 , 2 , 3 , 4 };
    int* b = a+2;
    printf( "%d\n" , *b );
    printf( "%d\n" , b[0] );
    return 0;
}
```

```
>> ./a.out
3
3
>>
```

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

Arrays within a stack frame

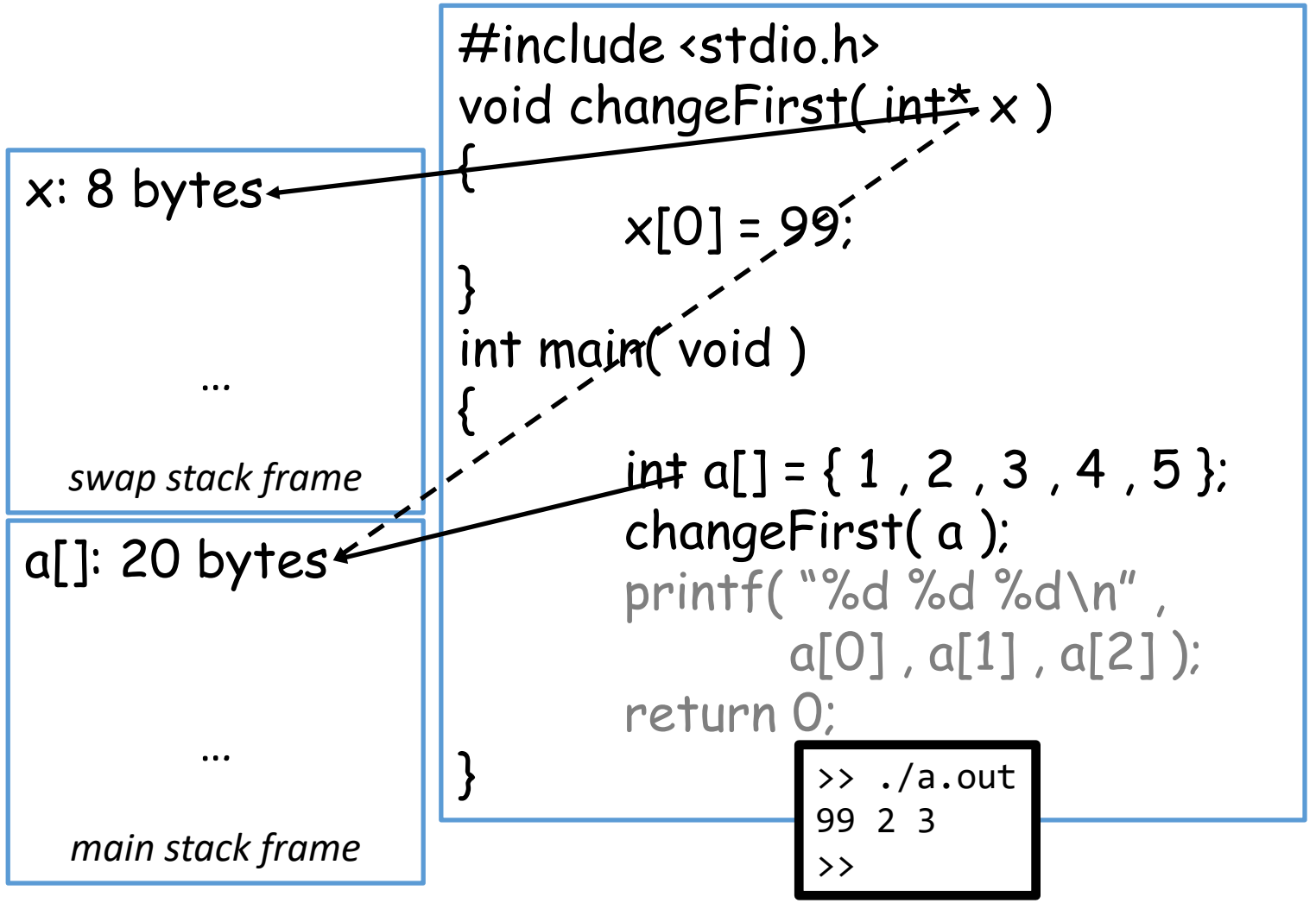
- Recall:
  When arrays are passed, the address of the array is copied, not the values
  - Since main's frame is on the stack while function changeFirst is running, the address is valid, and changeFirst can access it's values




```
#include <stdio.h>
void changeFirst( int* x )
{
        x[0] = 99;
}
int main( void )
{
        int a[] = { 1 , 2 , 3 , 4 , 5 };
        changeFirst( a );
        printf( "%d %d %d\n" ,
                a[0] , a[1] , a[2] );
        return 0;
}
```

Arrays within a stack frame

- Recall:
  When arrays are passed, the address of the array is copied, not the values
  - Since main's frame is on the stack while function changeFirst is running, the address is valid, and changeFirst can access it's values



Returning an array from a function

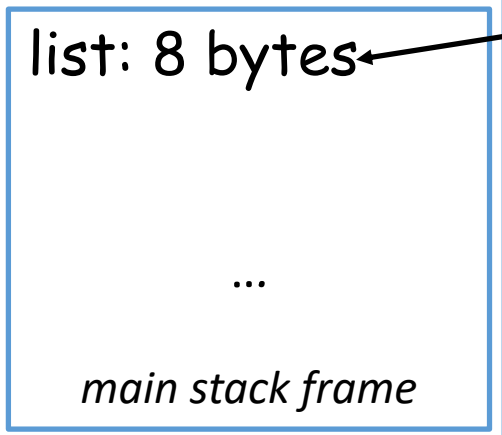
- However:
  When arrays are returned, the array's address is returned,
  not the array's values
  - If it's the address of an array on *getArray*'s stack frame, it won't mean anything after *getArray* returns

```
#include <stdio.h>
int* getArray( unsigned int sz )
{
        int a[sz];
        for( int i=0 ; i<sz ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int* list = NULL;
        list = getArray( 10 );
        for( int i=0 ; i<10 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
}
```

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

Returning an array from a function

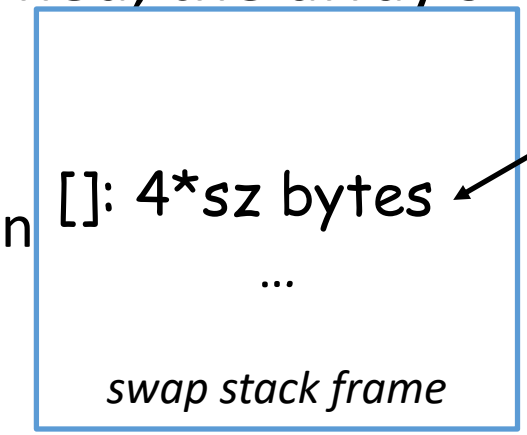
- However:
  When arrays are returned, the array's address is returned,
  not the array's values
  - If it's the address of an array on *getArray*'s stack frame, it won't mean anything after *getArray* returns

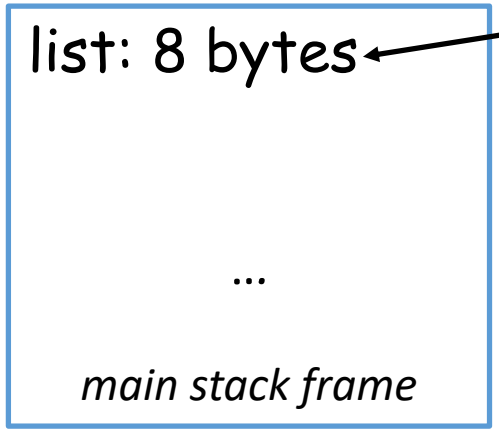



```
#include <stdio.h>
int* getArray( unsigned int sz )
{
        int a[sz];
        for( int i=0 ; i<sz ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int* list = NULL;
        list = getArray( 10 );
        for( int i=0 ; i<10 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

Returning an array from a function

- However:
  When arrays are returned, the array's address is returned, not the array's values
  - If it's the address of an array on *getArray*'s stack frame, it won't mean anything after *getArray* returns

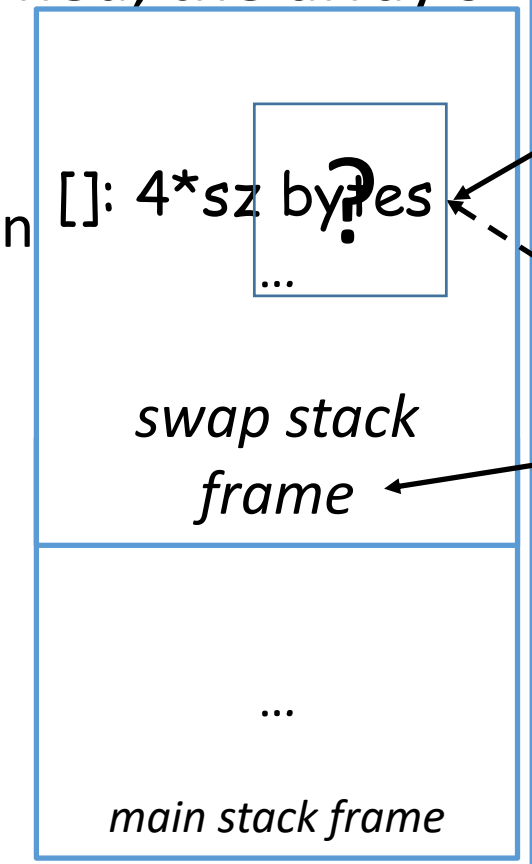





```
#include <stdio.h>
int* getArray( unsigned int sz )
{
        int a[sz];
        for( int i=0 ; i<sz ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int* list;
        list = getArray( 10 );
        for( int i=0 ; i<10 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

Returning an array from a function

- However:
  When arrays are returned, the array's address is returned, not the array's values
  - If it's the address of an array on *getArray*'s stack frame, it won't mean anything after *getArray* returns




```
#include <stdio.h>
int* getArray( unsigned int sz )
{
        int a[sz];
        for( int i=0 ; i<sz ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int* list;
        list = getArray( 10 );
        for( int i=0 ; i<10 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

Allocating a very large array

- Stack frames have a limited size

```
#include <stdio.h>
int main( void )
{
        int list[1<<30];   // ~10^9
        printf( "%d\n" , list[0] );
        return 0;
}
```

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

Limitations of arrays allocated in a stack frame

- Arrays (“statitcally”) allocated on the stack frame have limitations
  - Arrays created on a called function’s frame can’t be accessed by calling function
    - Returning a is meaningless

```
int* getArray( unsigned int sz )
{
        int a[sz];
        for( int i=0 ; i<sz ; i++ ) a[i] = 1;
        return a;
}
```

Limitations of arrays allocated in a stack frame

- Arrays (“statitcally”) allocated on the stack frame have limitations
  - Arrays created on a called function’s frame can’t be accessed by calling function
  - Size is limited by the size of stack frame (much less than available RAM)
    - `sz` cannot be too large

```
int* getArray( unsigned int sz )
{
        int a[sz];
        for( int i=0 ; i<sz ; i++ ) a[i] = 1;
        return a;
}
```

Limitations of arrays allocated in a stack frame

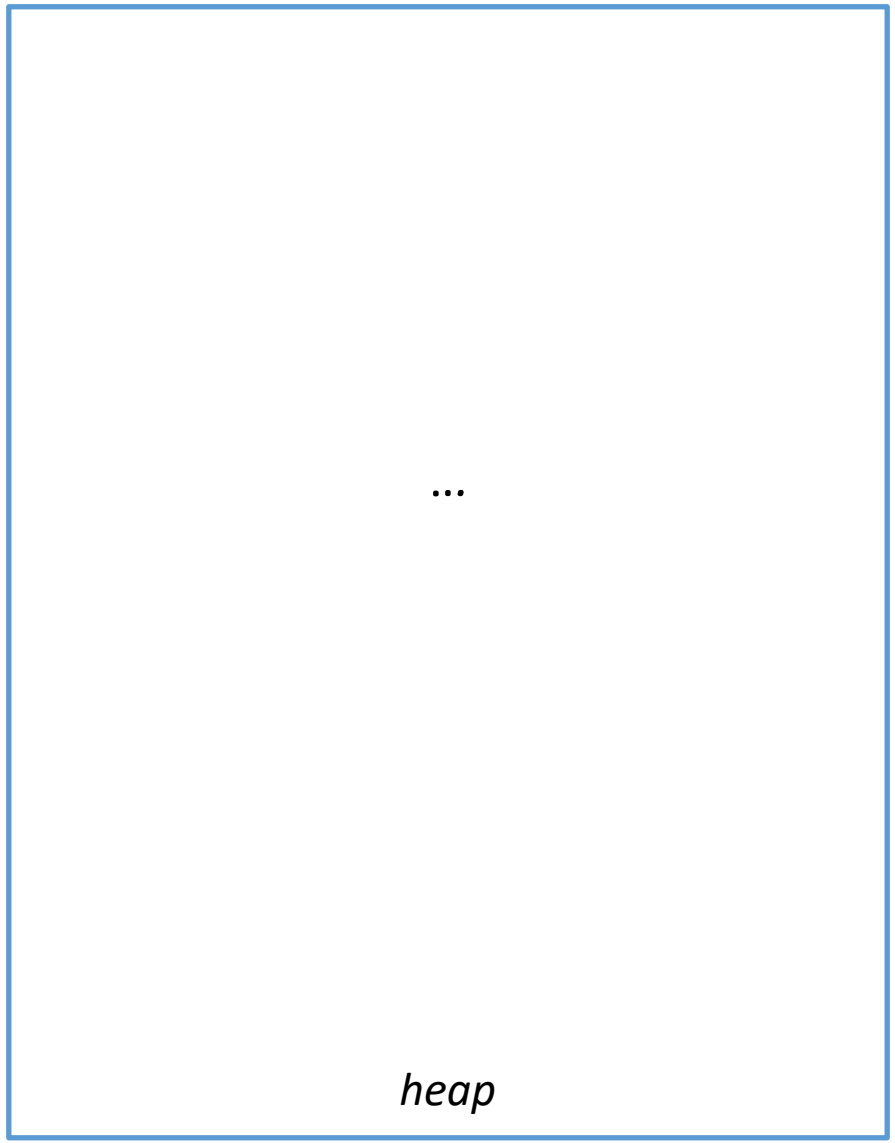
- Arrays ("statitcally") allocated on the stack frame have limitations
  - Arrays created on a called function's frame can't be accessed by calling function
  - Size is limited by the size of stack frame (much less than available RAM)
  - Prior to C99, could only allocate constant size arrays on the frame stack (Allocation on the stack was truly "static")
    - Declaring "`int a[sz];`" was illegal

```
int* getArray( unsigned int sz )
{
        int a[sz];
        for( int i=0 ; i<sz ; i++ ) a[i] = 1;
        return a;
}
```

Dynamically-allocating memory

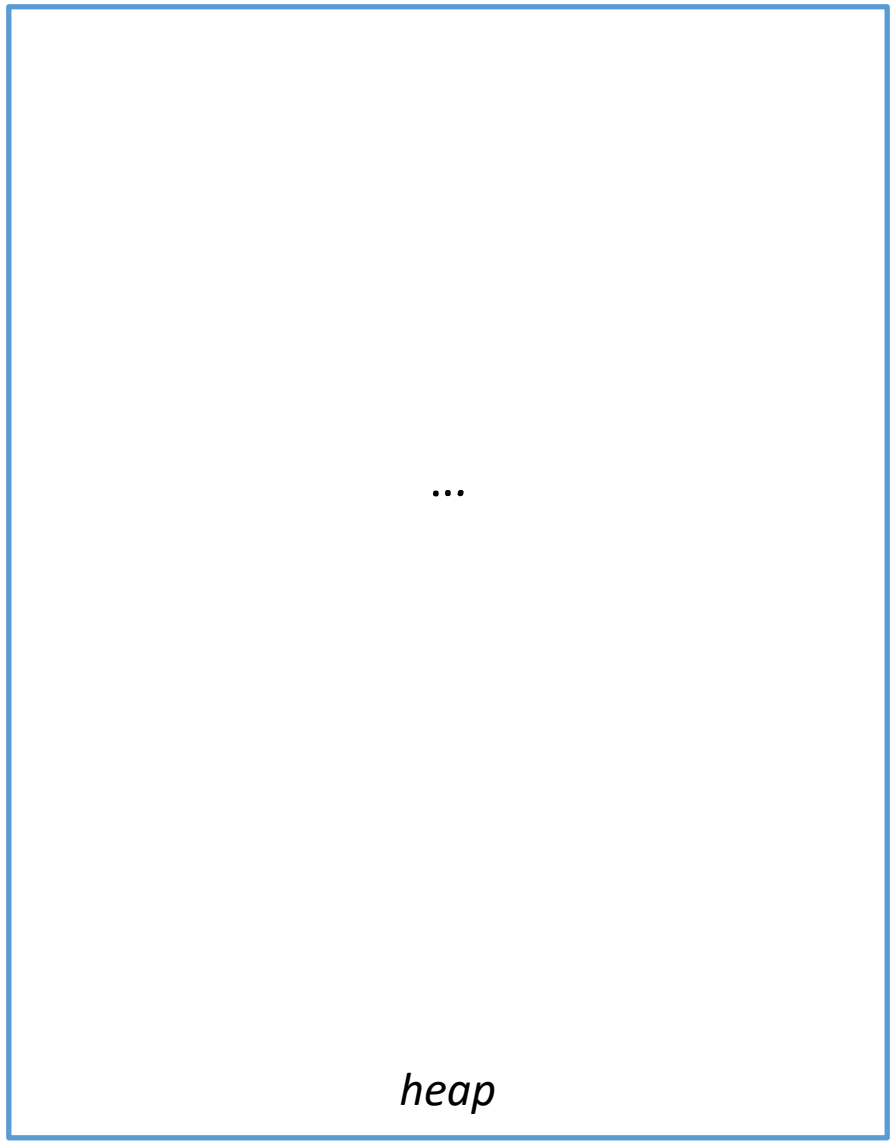
- Dynamically-allocated memory:
  - Is located on the *heap* – a part of memory separate from the stack
  - Lives as long as we like (until the entire program ends)
    - We don't lose access to it when function call returns
      - ✓ We can return it to a calling function!
      - ✗ We are responsible 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
  - Has a size that can be decided (dynamically) at run time

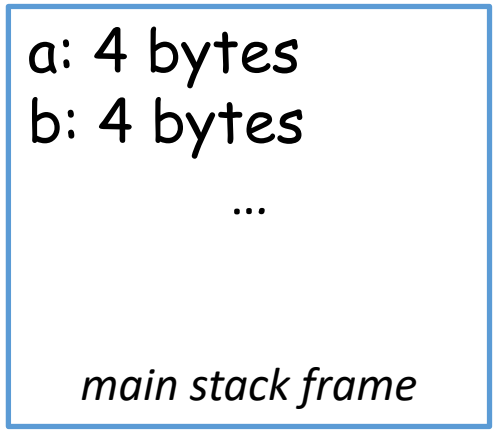
Dynamically-allocating memory




```
#include <stdio.h>
void foo( int x )
{
        // do something with x
}
int main( void )
{
        int a = 1;
        int b = foo( a );
        return 0;
}
```

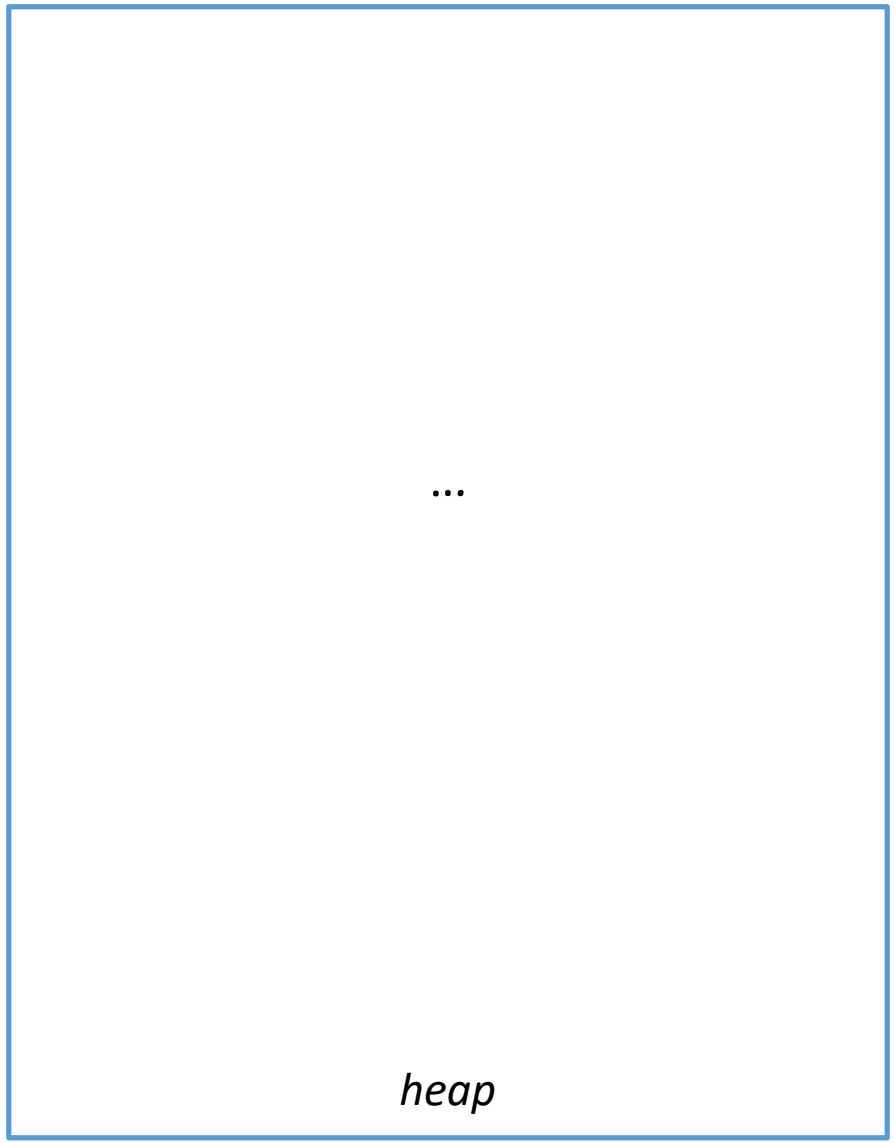
Dynamically-allocating memory

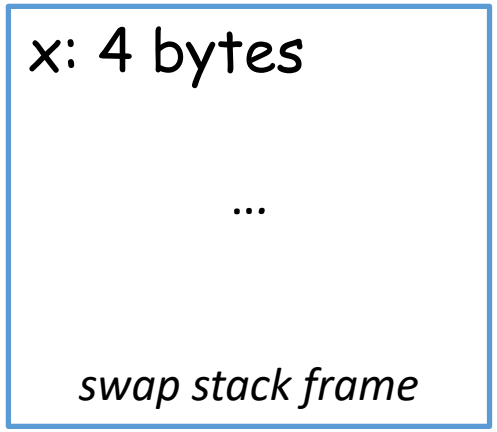


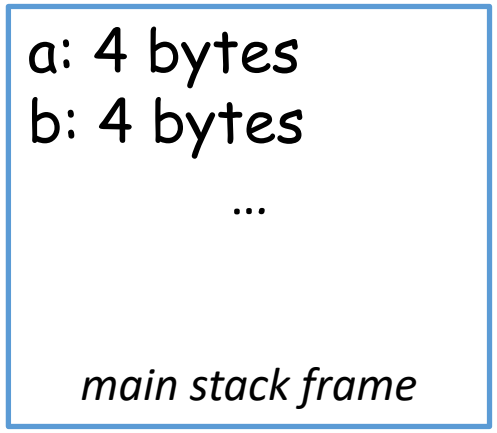



```
#include <stdio.h>
void foo( int x )
{
        // do something with x
}
int main( void )
{
        int a = 1;
        int b = foo( a );
        return 0;
}
```

Dynamically-allocating memory

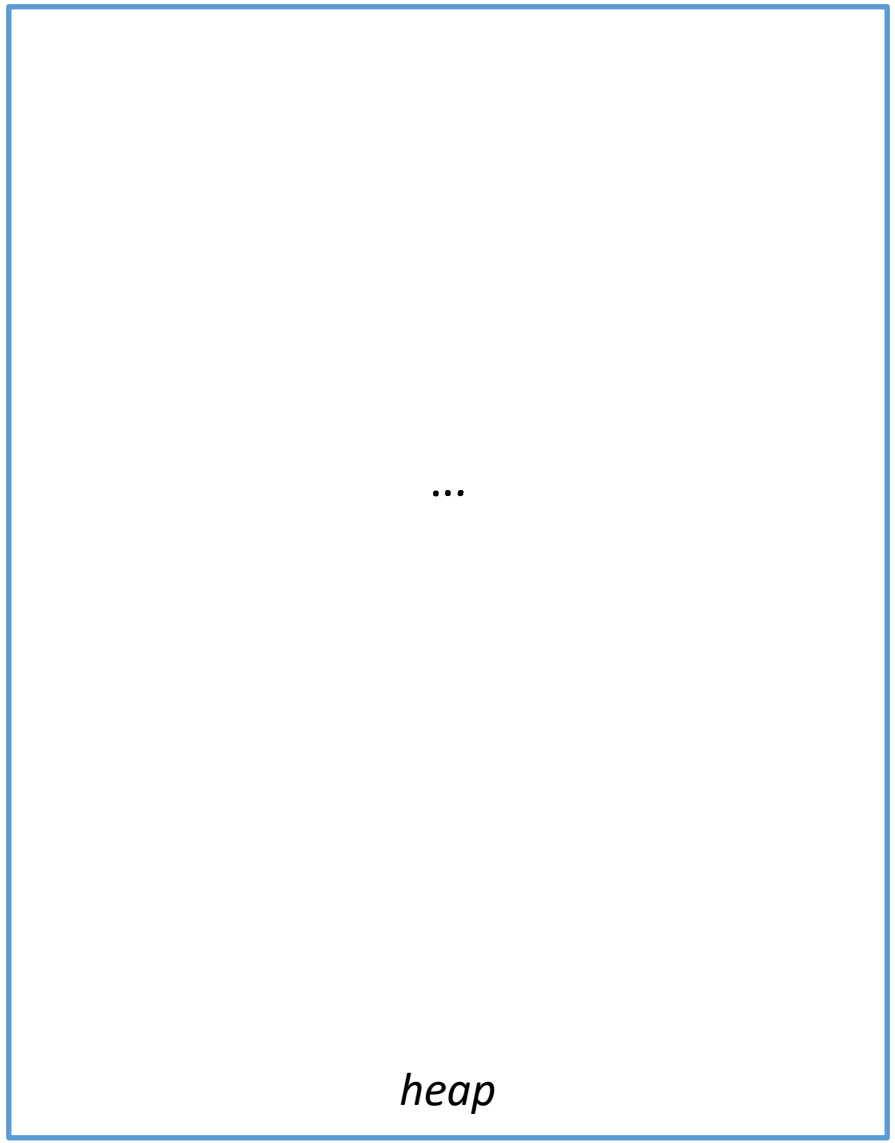


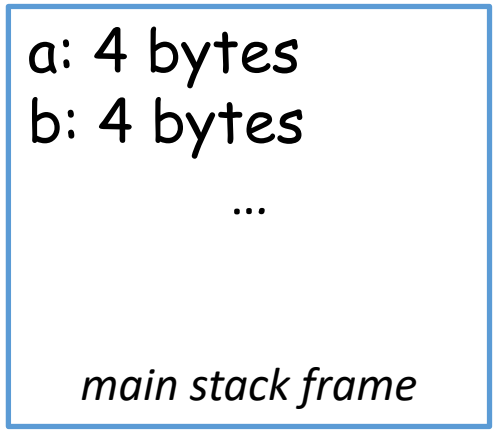





```
#include <stdio.h>
void foo( int x )
{
        // do something with x
}
int main( void )
{
        int a = 1;
        int b = foo( a );
        return 0;
}
```

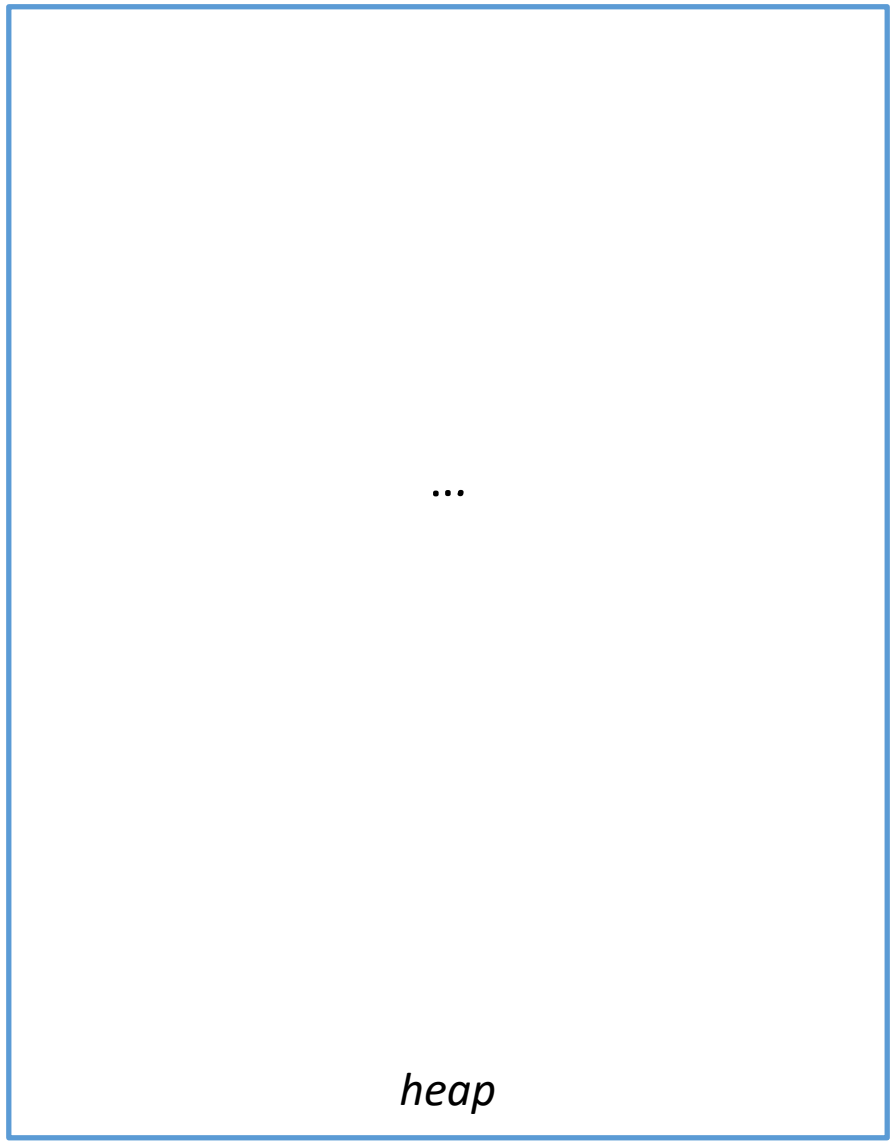
Dynamically-allocating memory






```
#include <stdio.h>
void foo( int x )
{
        // do something with x
}
int main( void )
{
        int a = 1;
        int b = foo( a );
        return 0;
}
```

Dynamically-allocating memory




```
#include <stdio.h>
void foo( int x )
{
        // do something with x
}
int main( void )
{
        int a = 1;
        int b = foo( a );
        return 0;
}
```

Dynamically-allocating memory

- 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”

Dynamically-allocating memory (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 will need

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

Dynamically-allocating memory (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
}
```

Dynamically-allocating memory (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;
…
```

Dynamically-allocating memory (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 usage of dynamically-allocated int is complete, 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 );
```

Dynamically-allocating memory (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 usage of dynamically-allocated int is complete, 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;
…
```

Dynamically-allocating memory (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 usage of dynamically-allocated int is complete, 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;
…
```

Deallocation

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

Only the last allocation was deallocated!

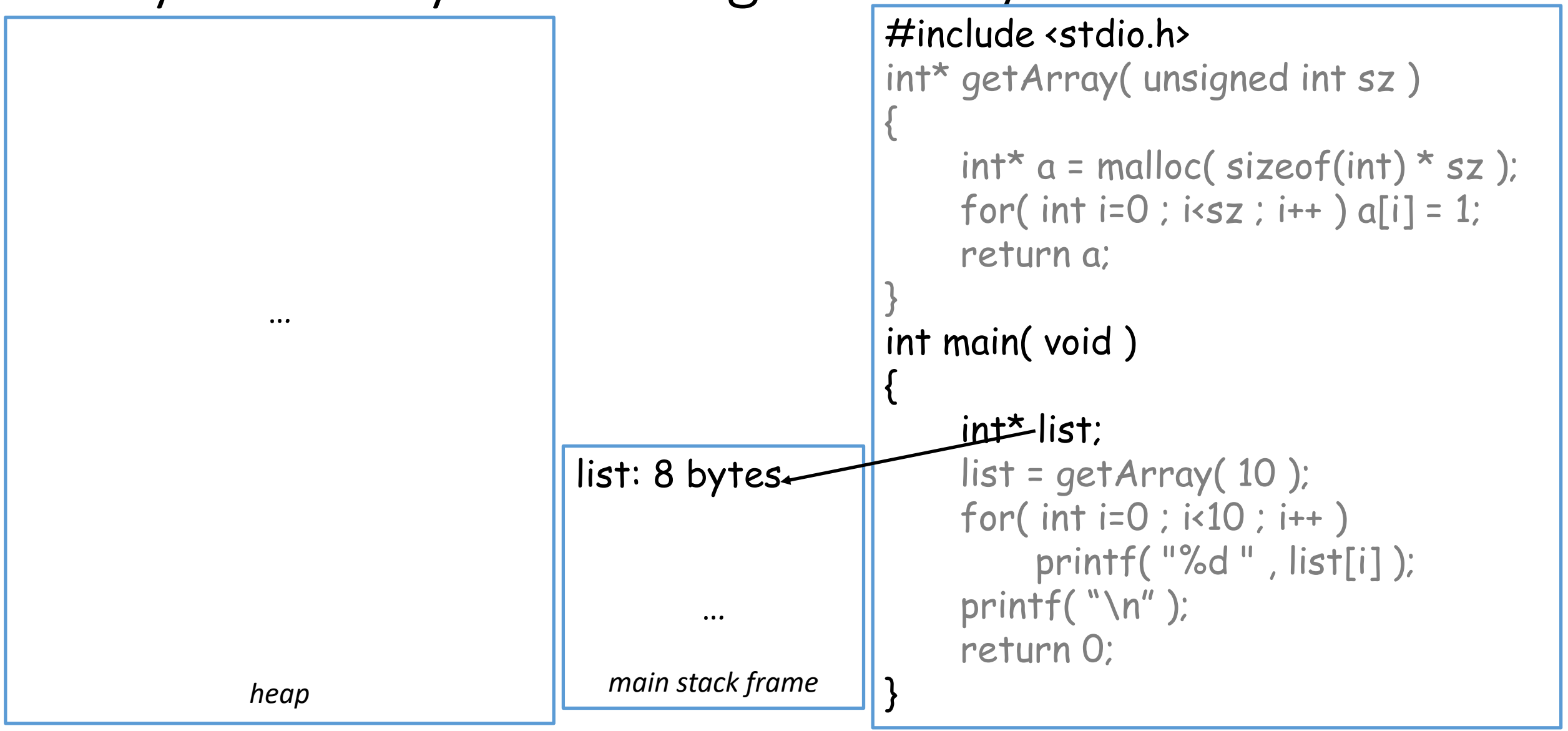
```
#include <stdio.h>
#include <stdlib.h>
int* getArray( unsigned int sz )
{
    int *a = malloc( sizeof(int) * sz );
    for( int i=0 ; i<sz ; i++ ) a[i] = 1;
    return a;
}
int main( void )
{
    int* list;
    for( int i=0 ; i<100000 ; i++ )
    {
        list = getArray( 100000 );
        // do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

Deallocation

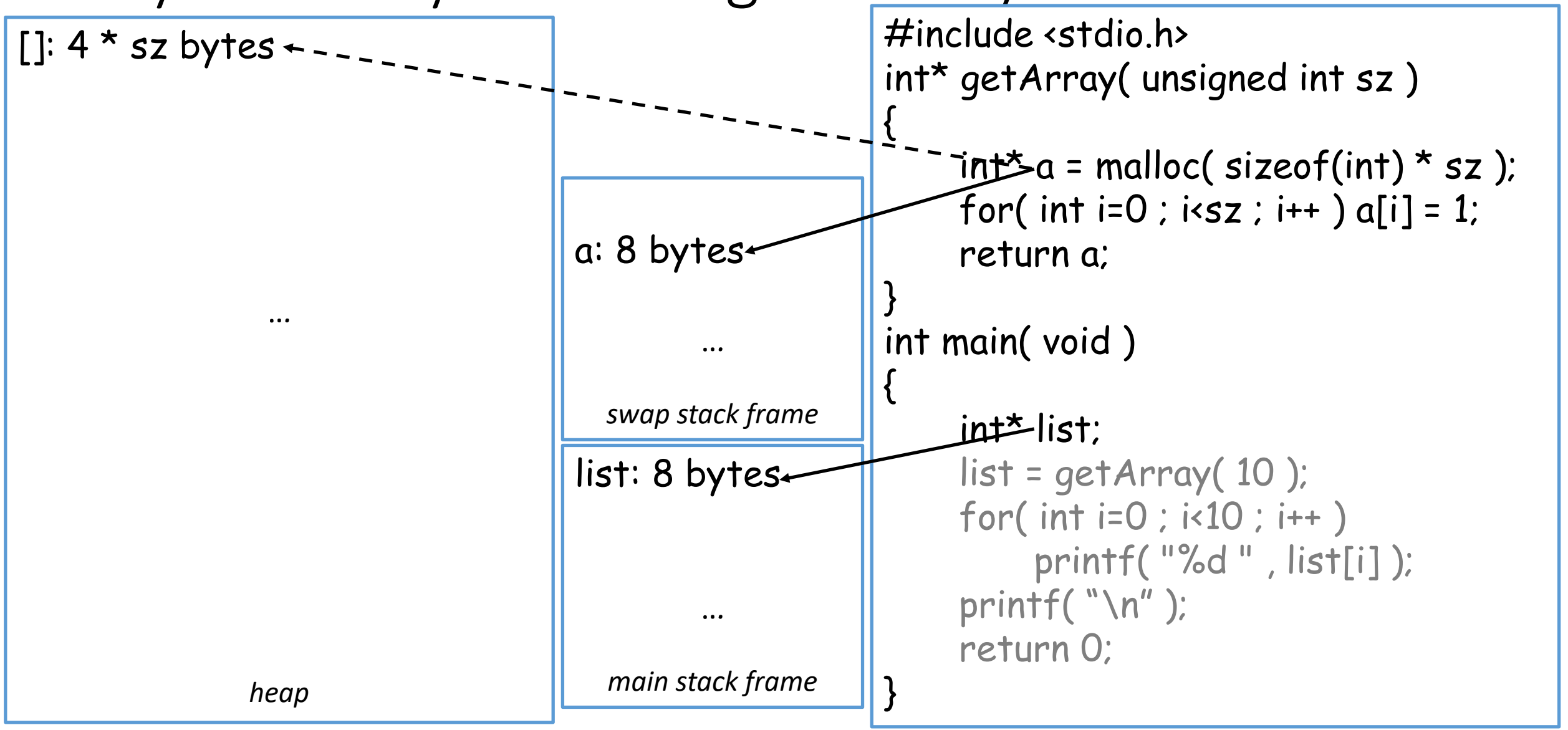
- Deallocation does not have to happen in the same function where allocation occurred. . .
  - But it does have to happen somewhere!
  - Otherwise you can get a *memory leak*

```
#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;
}
```

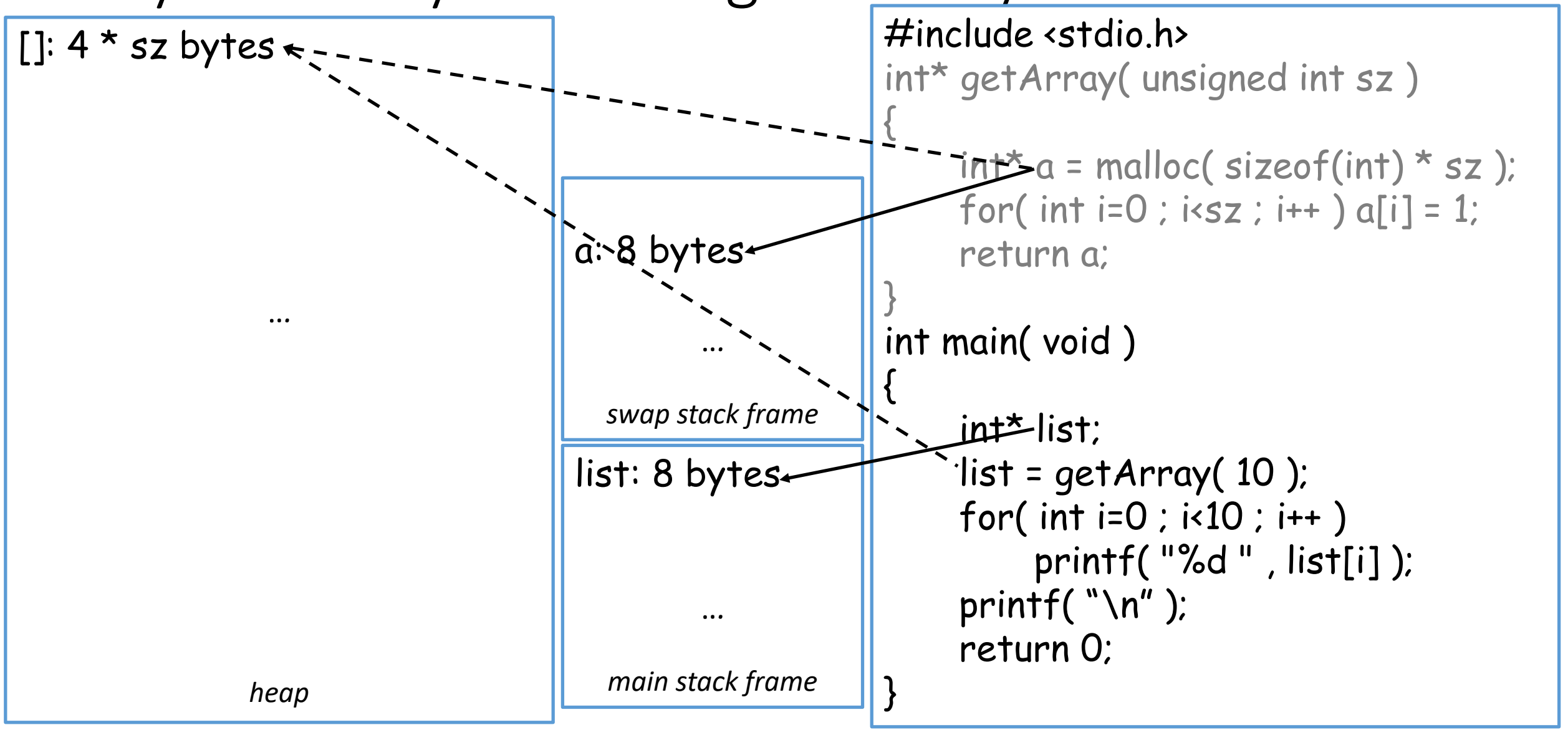
Dynamically-allocating memory



Dynamically-allocating memory



Dynamically-allocating memory



Dynamically-allocating memory

Q: What's wrong with this code?

A: We didn't deallocate

```
#include <stdio.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;
	list = getArray( 10 );
	for( int i=0 ; i<10 ; i++ )
		printf( "%d " , list[i] );
	printf( "\n" );
	ret
}
```

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