Introduction to Software Development – CS 6010
Lecture 14 – Stack and Heap Memory

Master of Software Development (MSD) Program

Varun Shankar

Fall 2022

Miscellaneous

- I hope to have the midterms graded by Wednesday

Lecture 14 – Stack and Heap Memory

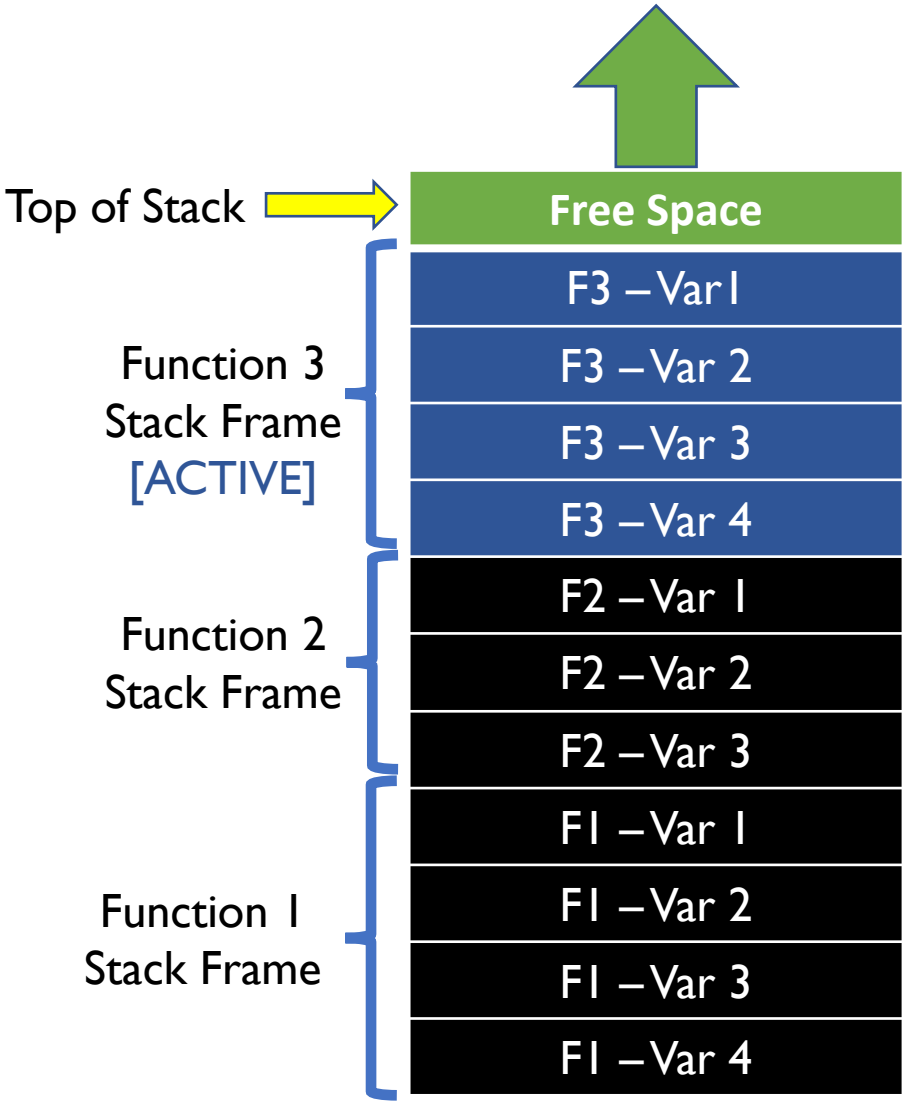
- Topics
  - (Call) Stack Memory
  - Heap Memory

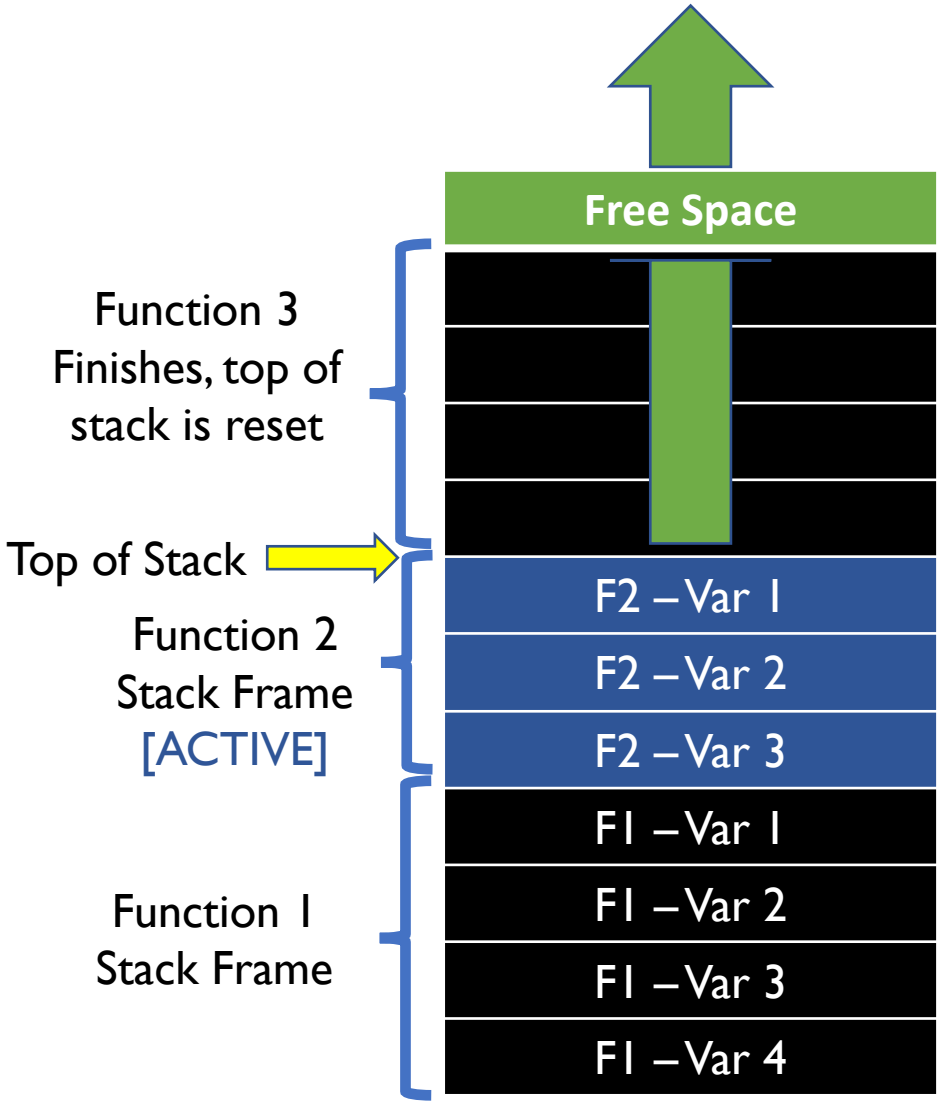
Managing Memory

- Managing memory is a “low level” operation, but very important to understand.
- However, it is also fairly error prone and complicated.
- Because of this, some high-level languages (such as Python and Java) don’t explicitly allow you to manage the computer (program’s) memory.
- MSD starts off with C++ because it does allow us to explore memory management.
  - A skill that is very important to possess when developing large scale applications.
- Let’s start by reviewing the type of memory that we have seen so far.

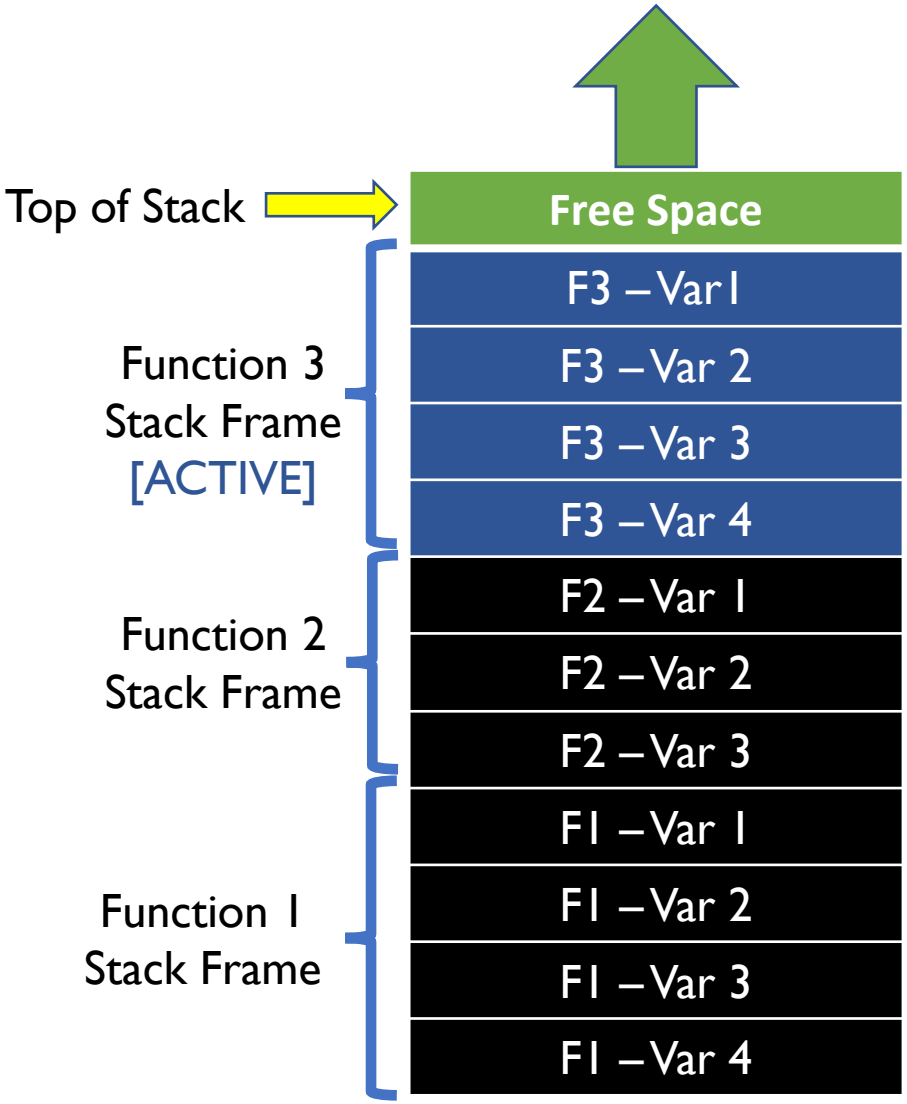
Stack Memory

- Specifically we have been calling this the *Call Stack* memory, but in general we just say *Stack* memory.
- When does memory become allocated on the stack?
  - When we call a function.
- When a function is called, memory is allocated on the stack to store all the variables / parameters used by that function.
  - Stack variables are automatically created/destroyed for you (the compiler takes care of this).
  - Creating / Destroying variables has not yet meant more than making the space for them / removing the space. In the future we will see that more complicated variables will require more work to be created / destroyed.
- When does a functions memory get deallocated?
  - When the function returns.
  - This is sometimes called “automatic memory management”.


Top of Stack
Free Space
F3 – Var 1
F3 – Var 2
F3 – Var 3
F3 – Var 4
Function 3
Stack Frame
[ACTIVE]
F2 – Var 1
F2 – Var 2
F2 – Var 3
Function 2
Stack Frame
F1 – Var 1
F1 – Var 2
F1 – Var 3
F1 – Var 4
Function 1
Stack Frame


Free Space
Function 3
Finishes, top of
stack is reset
Top of Stack
F2 – Var 1
F2 – Var 2
F2 – Var 3
Function 2
Stack Frame
[ACTIVE]
F1 – Var 1
F1 – Var 2
F1 – Var 3
F1 – Var 4
Function 1
Stack Frame

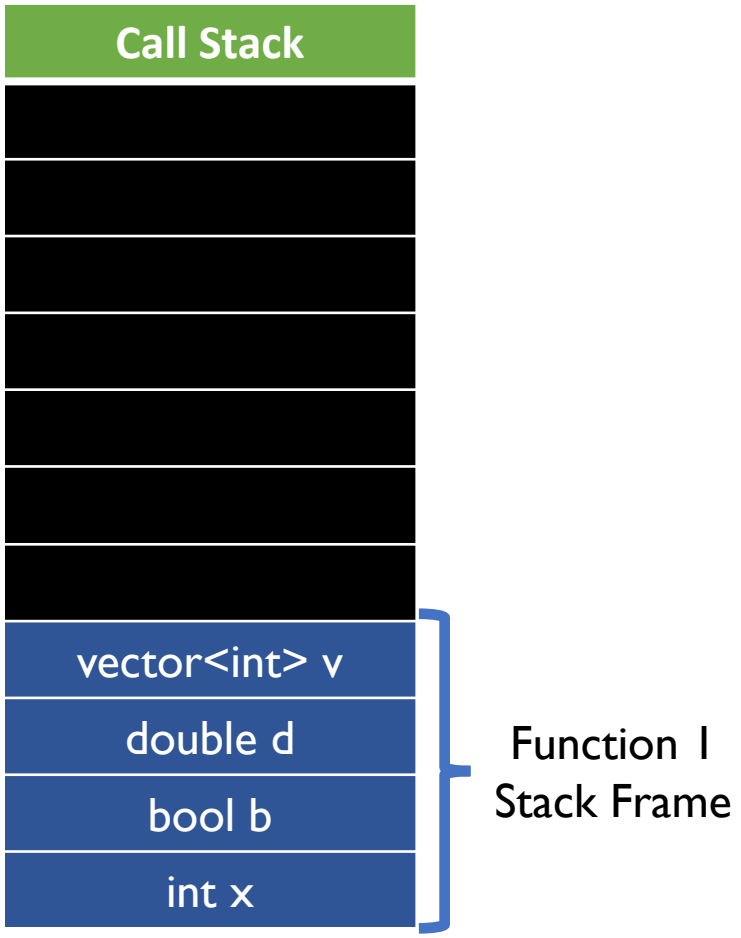
Global Variables

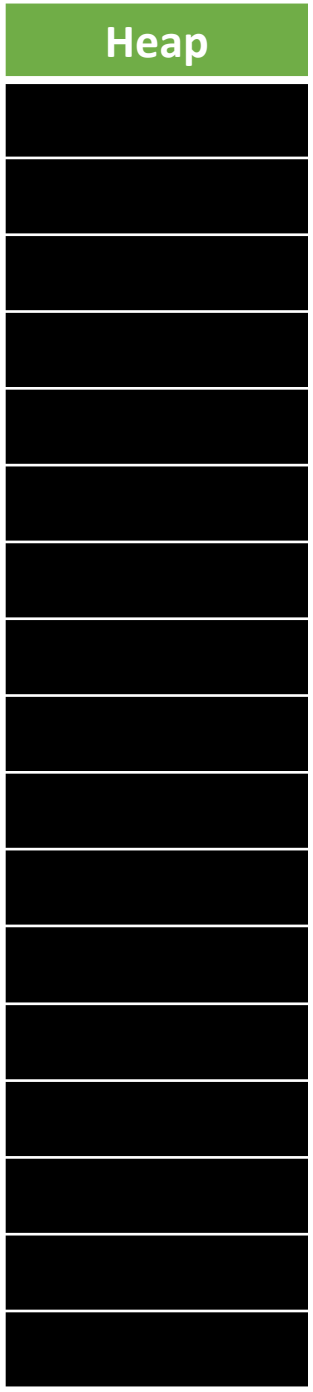



```
int globalX;

void doSomething()
{
    globalX = 20;
}


int main()
{
    globalX = 10;
    cout << globalX;
    doSomething();
    cout << globalX;
}
```

- Global variables are accessible from anywhere in your program.
- While they are an "easy" way to "pass" information between functions, they cause many problems because their data is not *protected* from being (accidentally) changed by a function
- Short story: Don't use them.

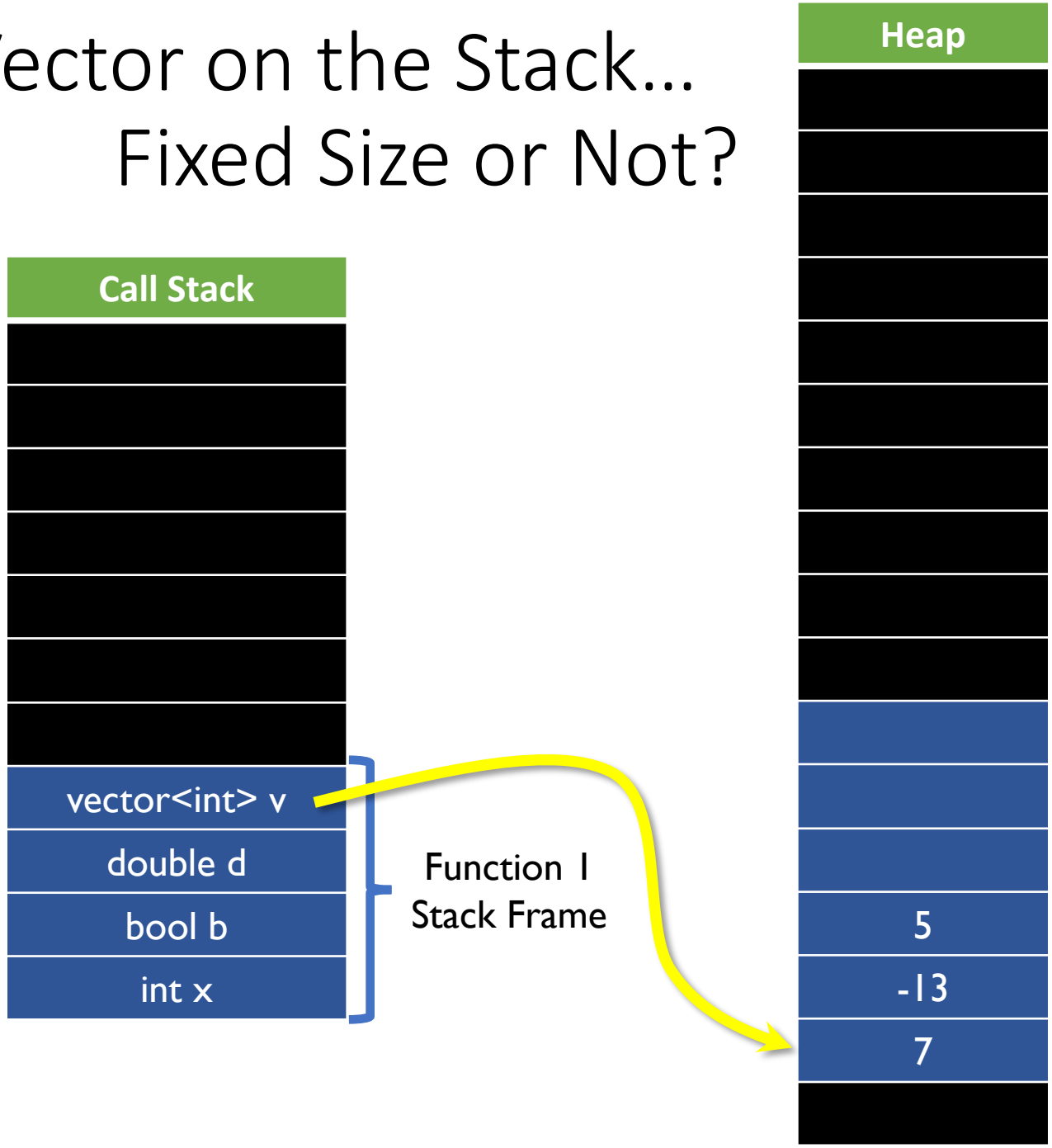
Vector on the Stack… Fixed Size or Not?





- But a vector can change size – how can it be on the stack?

Vector on the Stack… Fixed Size or Not?

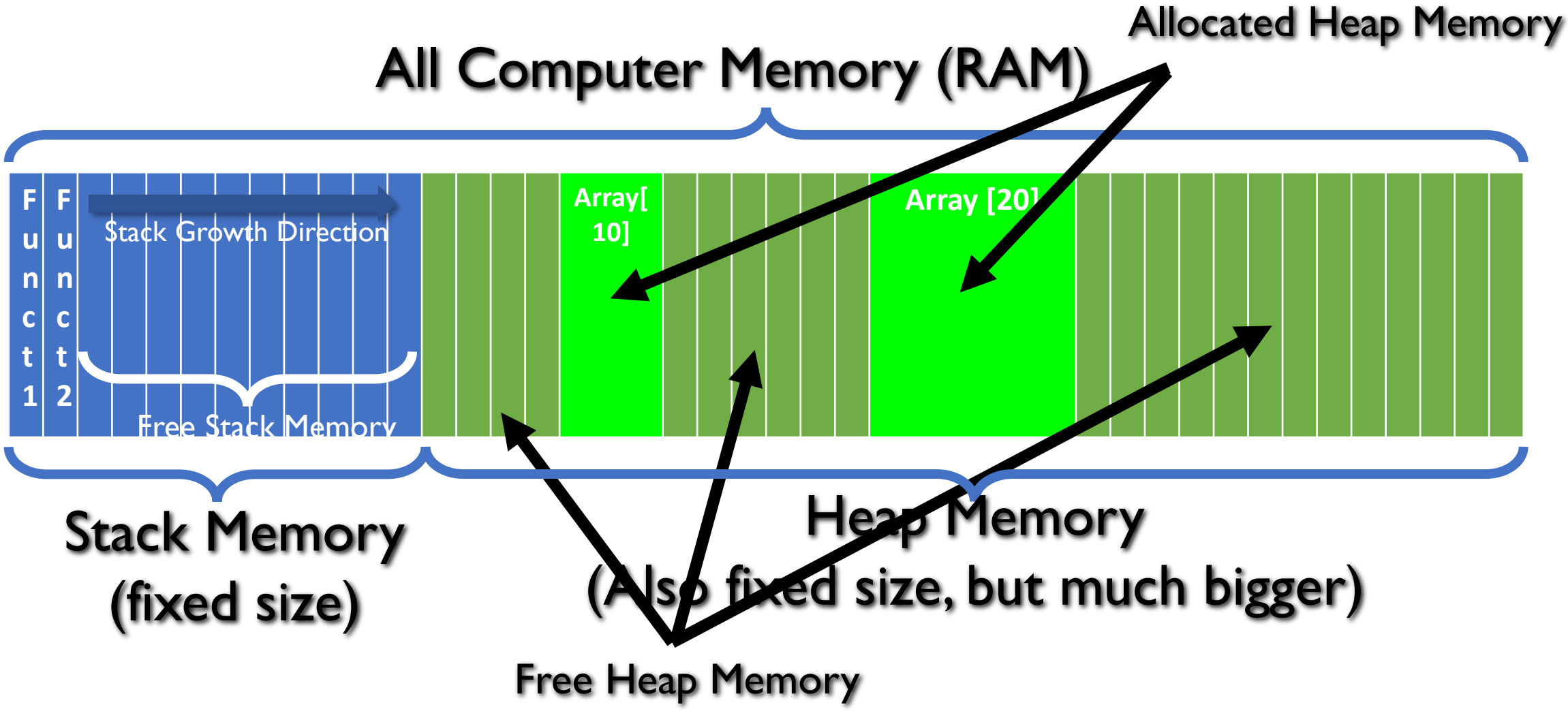


- But a vector can change size – how can it be on the stack?
  - The actual data is stored on the *heap*.
  - There is a pointer in the vector on the stack that points to the data on the heap.
- Why are there 3 empty locations for this vector?
  - Spots that can be used, but have not been yet.
- What information does the vector need to keep track of its data?
  - Pointer to the first piece of data.
  - Current number of data items (size)
  - Amount of memory available (capacity)

Heap Memory

- The *Heap* is basically the rest of your computer's RAM (and more)
  - In 6013, we will talk about *Virtual* memory, but not yet.
- Programmers must explicitly ask for heap memory when they want it.
- Programmers must also explicitly tell the compiler when they are finished using that heap memory.
- The heap is used for:
  - Large chunks of memory.
  - Data that must survive longer than a single function call.
- Memory stored in the heap can usually only be tracked using *pointers*, since the address is not known at compile time.

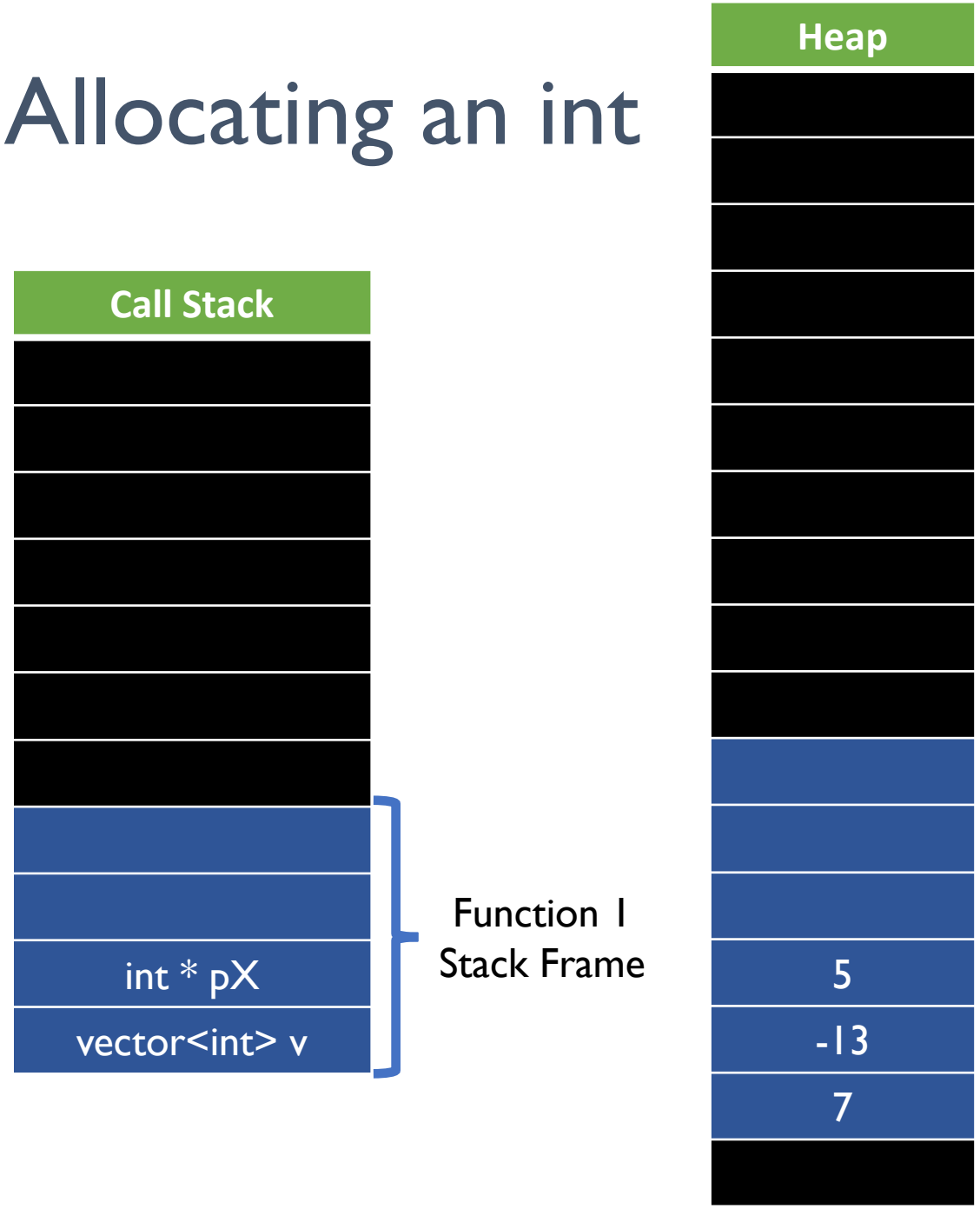
Stack and Heap Diagram



Getting Heap Memory – the *new* keyword

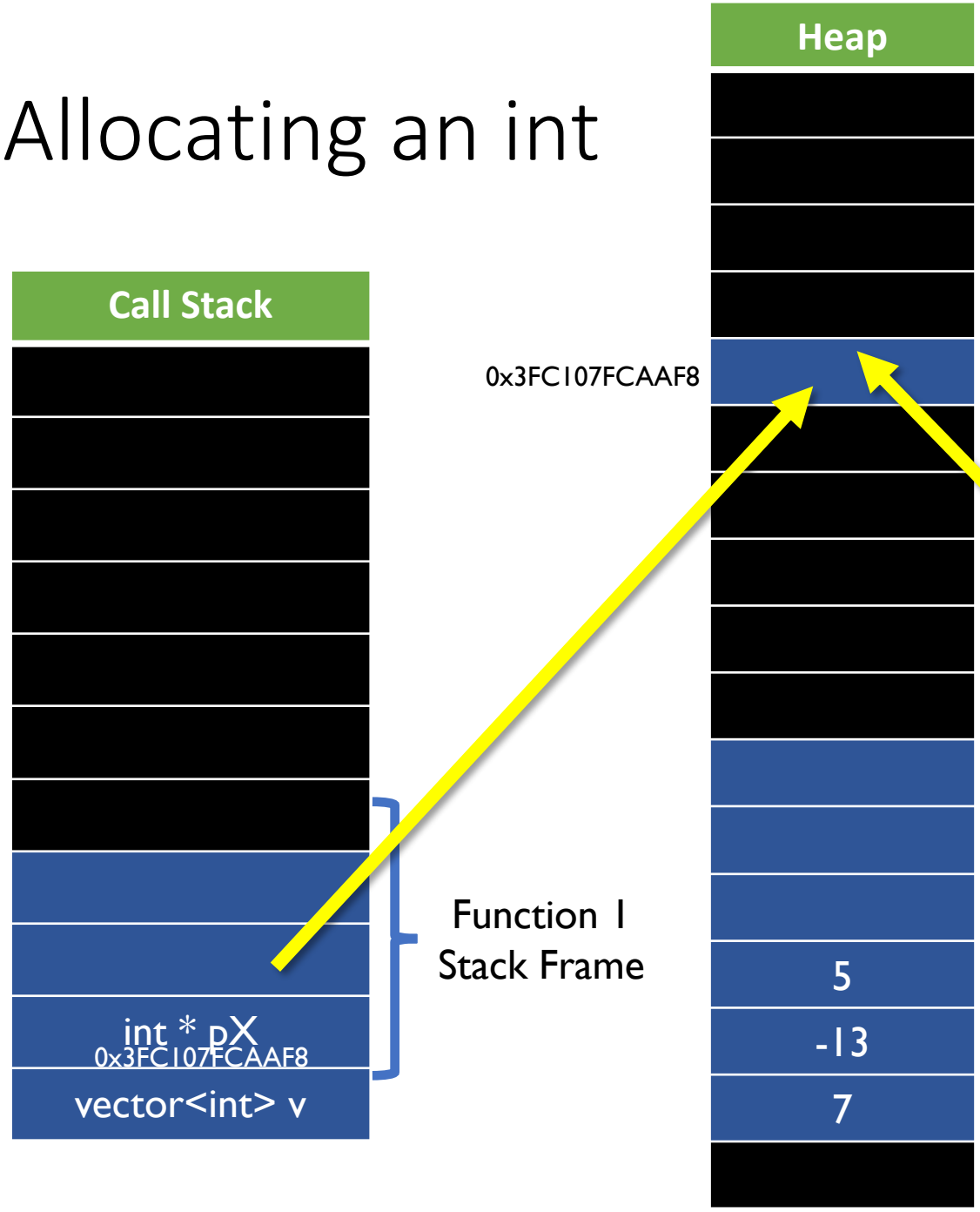
- *Request* memory from the heap using the *new* keyword and the data type / amount of memory you want.
  - int * pInt = new int; // Returns a pointer to a single int.
- The *runtime* (ie, the C++ standard library) find the appropriate amount of memory and returns a pointer to it.
  - double * myArray = new double[10]; // Returns a pointer to the 1st of 10 doubles (creates an array on the heap).
- The **pointers** pInt and myArray are on the stack.
- But the actual **data** "in" myArray is in the heap!

Allocating an int



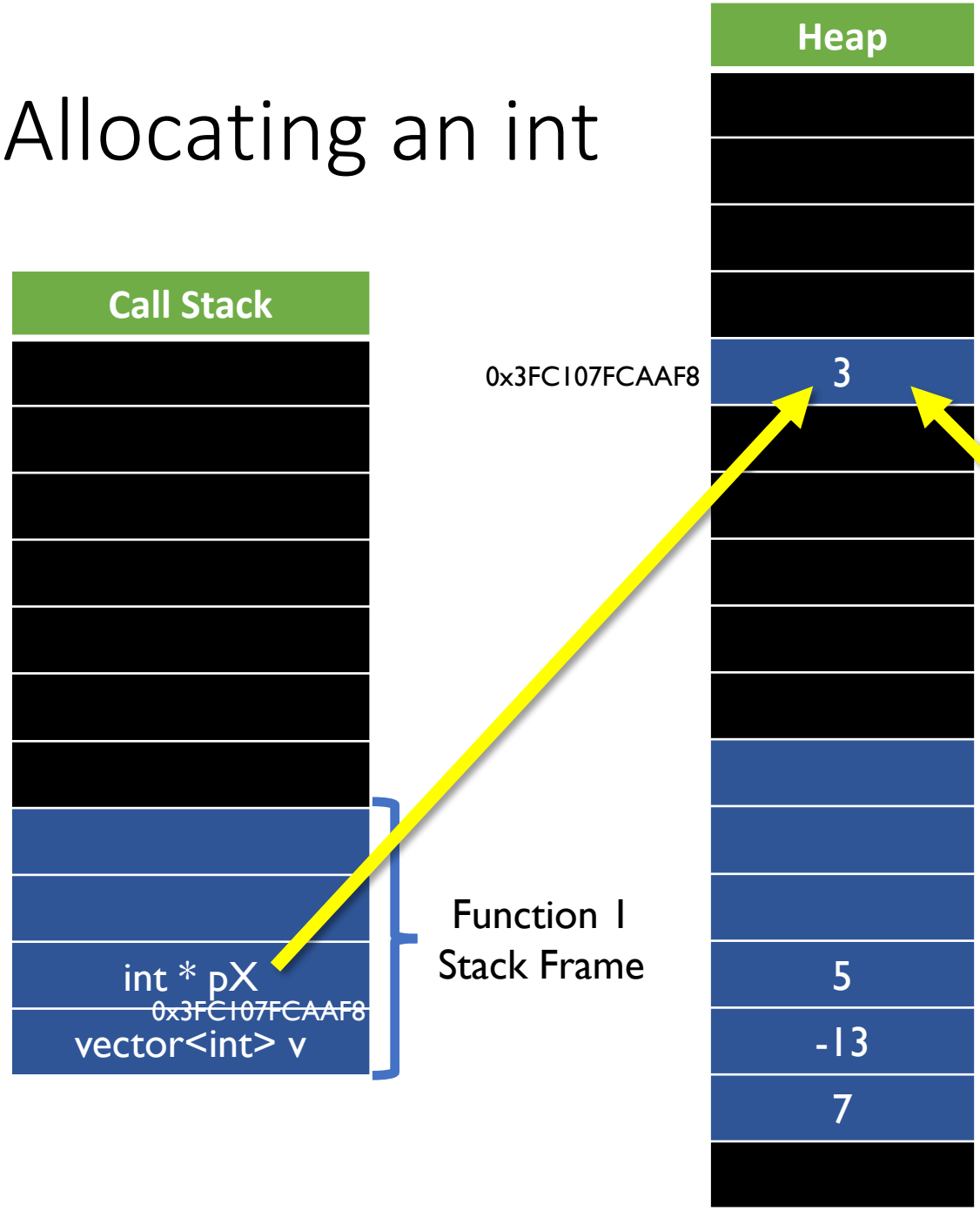
- `int * pX = new int;`
  - First: what does *`new int`* do?

Allocating an int



- `int * pX = new int;`
  - First: what does *`new int`* do?
    - Memory is allocated (by system)
  - Second: address to heap is put into pX's memory
  - What is the value of pX?
    - Some address: 0x3FC107FCAAF8
  - What is the value of *pX?
    - Nothing yet…
  - *pX = 3;
    - Now what is its value?

Allocating an int



- `int * pX = new int;`
  - First: what does *`new int`* do?
    - Memory is allocated (by system)
  - Second: arrow is put into pX's memory
  - What is the value of pX?
    - Some address: 0x3FC107FCAAF8
  - What is the value of *pX?
    - Nothing yet…
- *pX = 3;
  - Now what is its value?
  - 3
  - What is *pX called?
    - Dereferencing

Returning Heap Memory in C++ – the *delete* keyword

- To *deallocate* heap memory – which means to tell the system that we are done using that memory – we use the *delete* keyword.
- *delete*
  - Does NOT change our pointer to the data.
  - Does NOT even delete/remove the data itself.
  - It simply tells the system that our program is not going to use the memory anymore, and the system can now reallocate it for something else (which it may or may not do at this time!).
- In action:
  - delete pInt;
  - delete [ ] myArray; // Need the [ ] to delete arrays of data.

Choosing Between Stack and Heap

- Most standard variables will be automatically placed on the stack.
- For variables that will just be used in a function, and then will cease to exist when the function is done, the variable should be created on the stack.
  - Note, this is the default case.
- We use heap memory for variables when:
  - We don’t know how much memory will be needed until runtime.
  - We will need access to the data (variable) after the function finishes.
  - We need a huge amount of memory.
    - There is significantly more memory allocated to the heap than to the stack.

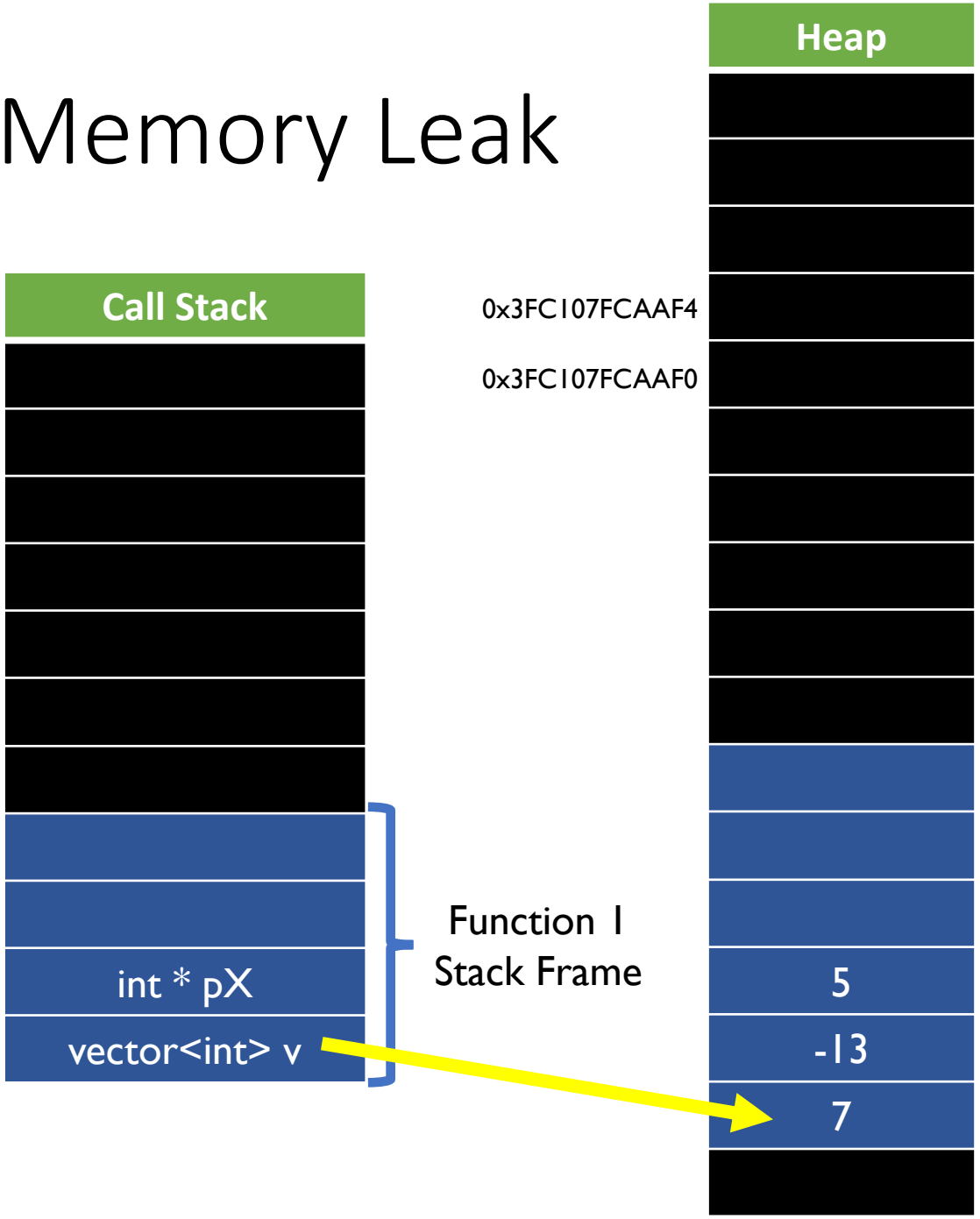
Memory (Pointer) Errors!

- C++ does not check to make sure that our memory accesses are safe!
- This can lead to strange errors.
  - Many times these errors don’t become apparent until much later in the code then when they happened – this can make it very hard to track down the error.
- The *address sanitizer* tool can help us track some of these errors down.

Memory Leaks

- The first (and probably least dangerous) memory error is called the *memory leak.*
- This occurs when we allocate memory (on the heap) and never deallocated it.
  - Note: If a programmer allocates memory (uses *new*), it is always on the heap.
- In other words, if we use *new* to get memory, but never use *delete* to get rid of it.
  - Note: “Getting rid of memory” just means returning it to the system so that it can be used for something else at a later point in the program.

Memory Leak

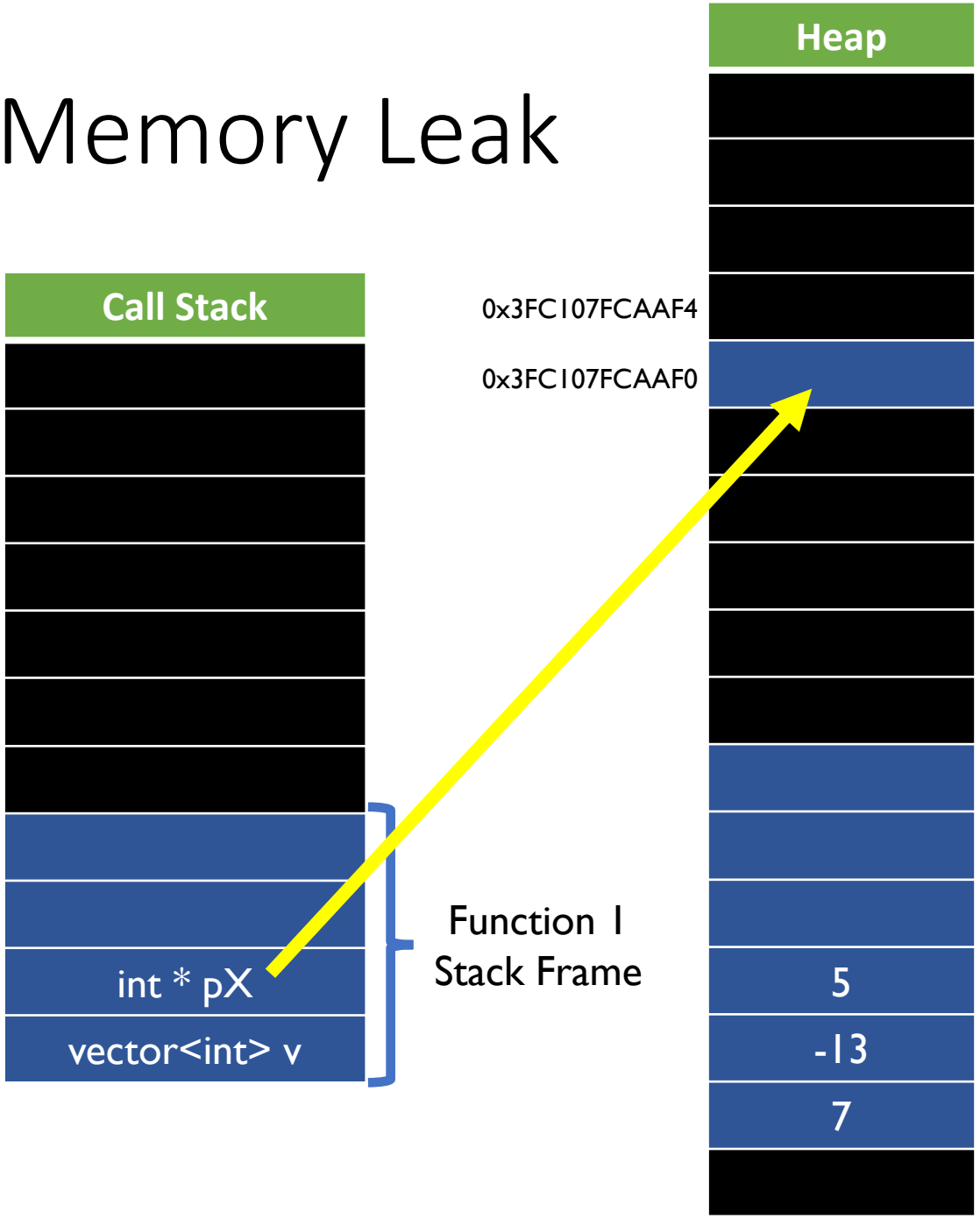



```
int * pX = new int;
```

- What does the above line do?

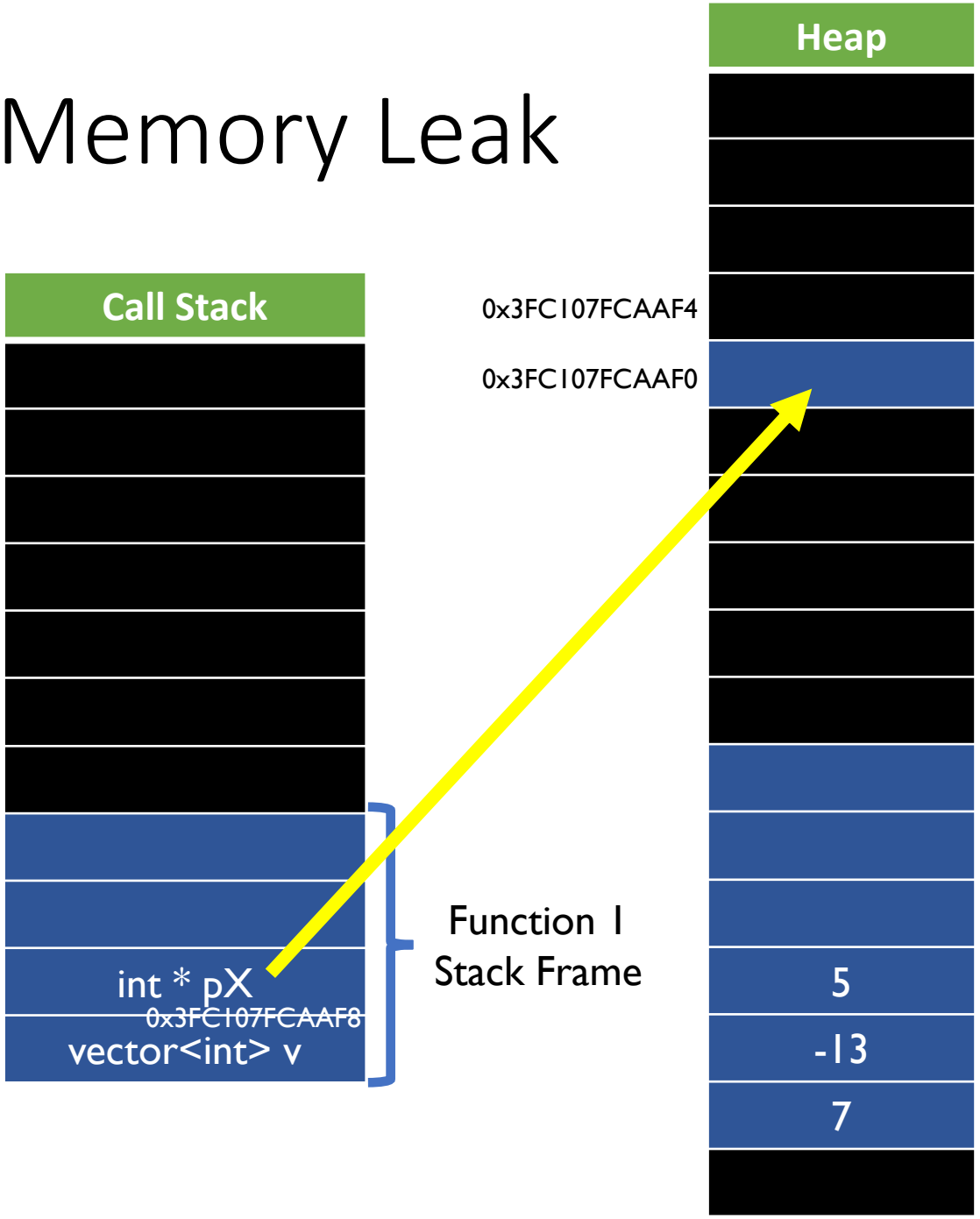
- Note: While the vector v pointer continues to exist, I am not going to display it for now as it is not relevant (for now) to the example of pX.

Memory Leak



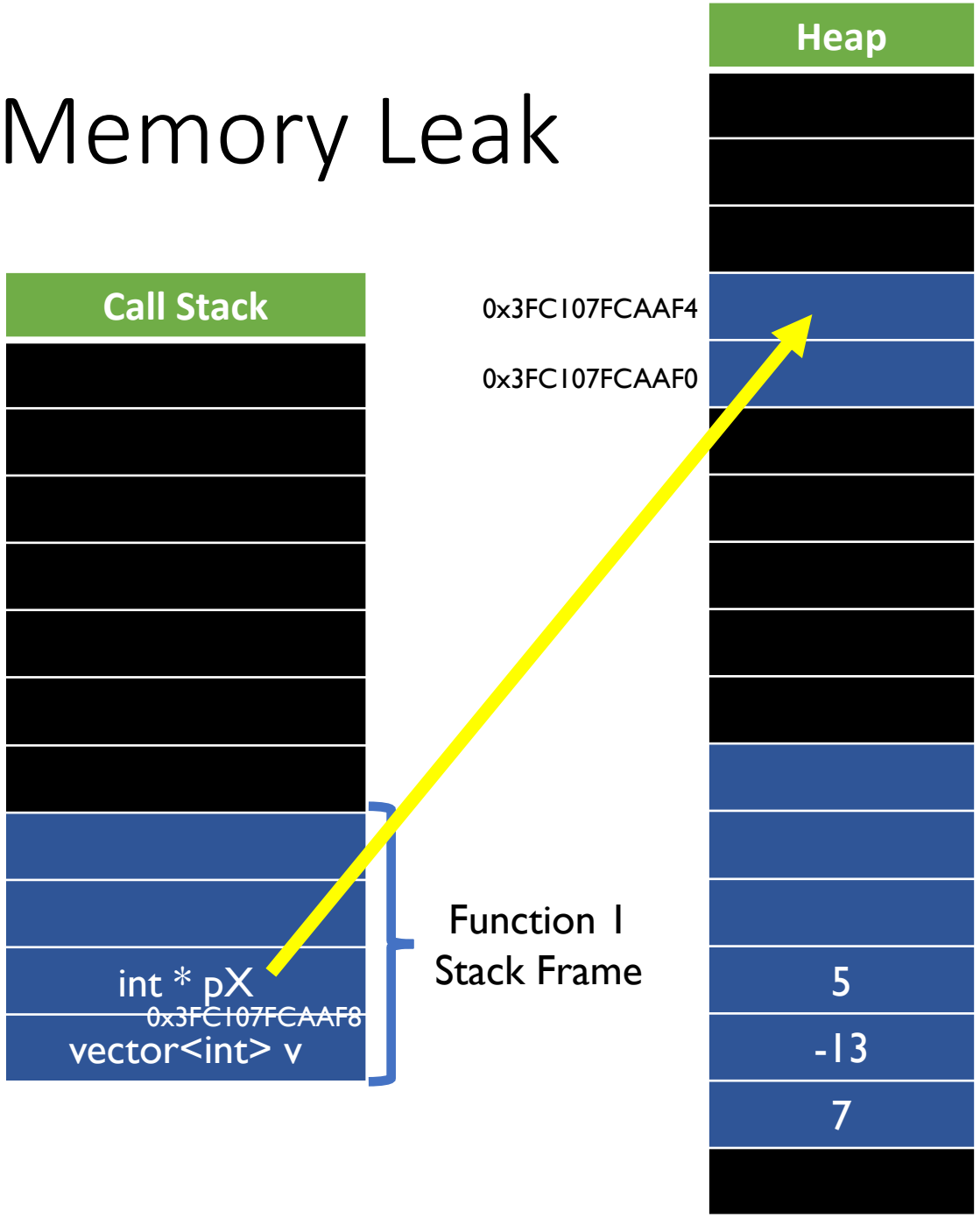

```
int * pX = new int;
```

Memory Leak




```
int * pX = new int;
pX = new int;
```

Memory Leak

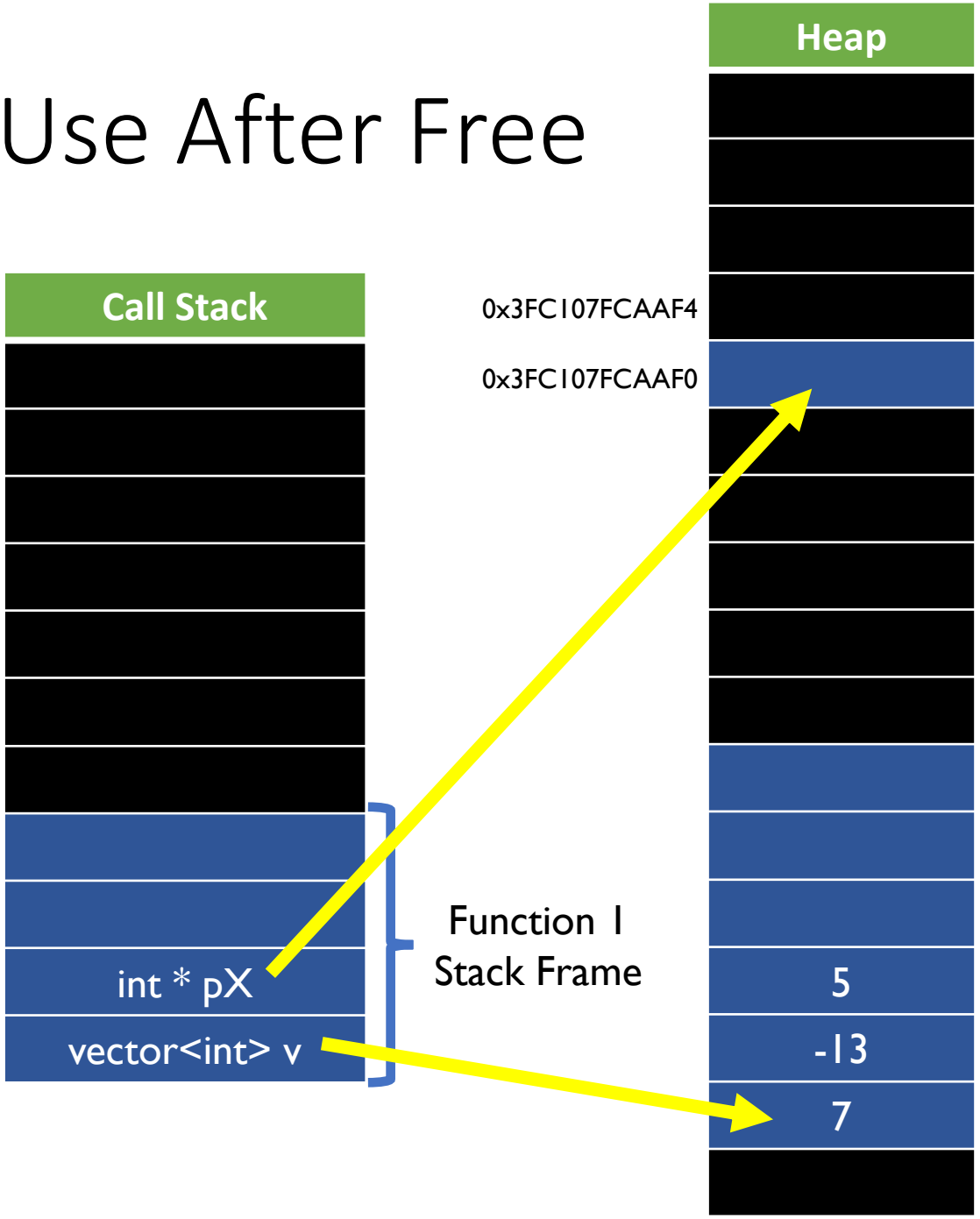


- `int * pX = new int;`
- `pX = new int;`
- The memory at `0x3FC107FCAAF0` is still allocated, but has been lost.
  - This is called a memory leak. While it does not cause the program to crash, if memory continues to leak throughout the life of the program, at some point the program will ask for more memory and no more will be available – and then the program will crash.

Use After Free Errors

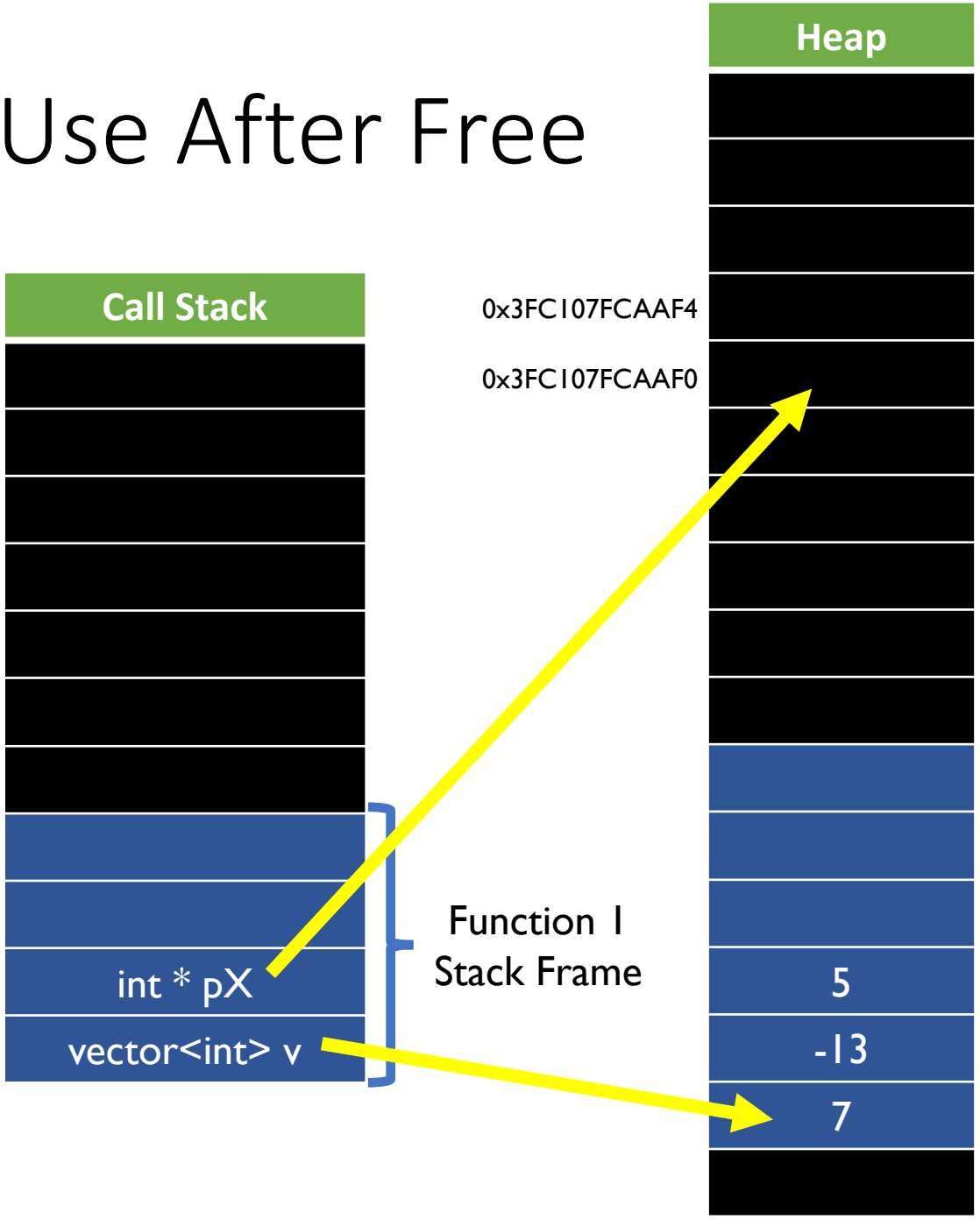
- Before *new/delete* came around in C++, C used *malloc/free.* This is were the *Use After Free* term comes from.
- In C++, we really have *Use After Delete* errors.
- Since delete simply marks the memory as “no longer in use” and does not invalidate our pointer, a programmer may accidentally use this (now invalid) pointer again – thus accessing “bad memory”.
- When we access “bad memory”, strange things happen!
  - Unfortunately, most of the time, everything will appear to continue to work correctly, and the error won’t appear for some time.
  - If the memory has been given (by the system) for another use, we could even corrupt the data in another part of our program.
- A good habit for avoiding these errors is to immediately set your pointer to *nullptr* (in older examples, you will see NULL) as soon as you delete it:
  - delete pInt;
  - pInt = nullptr; // If we use pInt again, the program will immediately crash (which is much better than crashing later)!
- C++ also has **smart** pointers that handle deletion for this (Week 5).

Use After Free



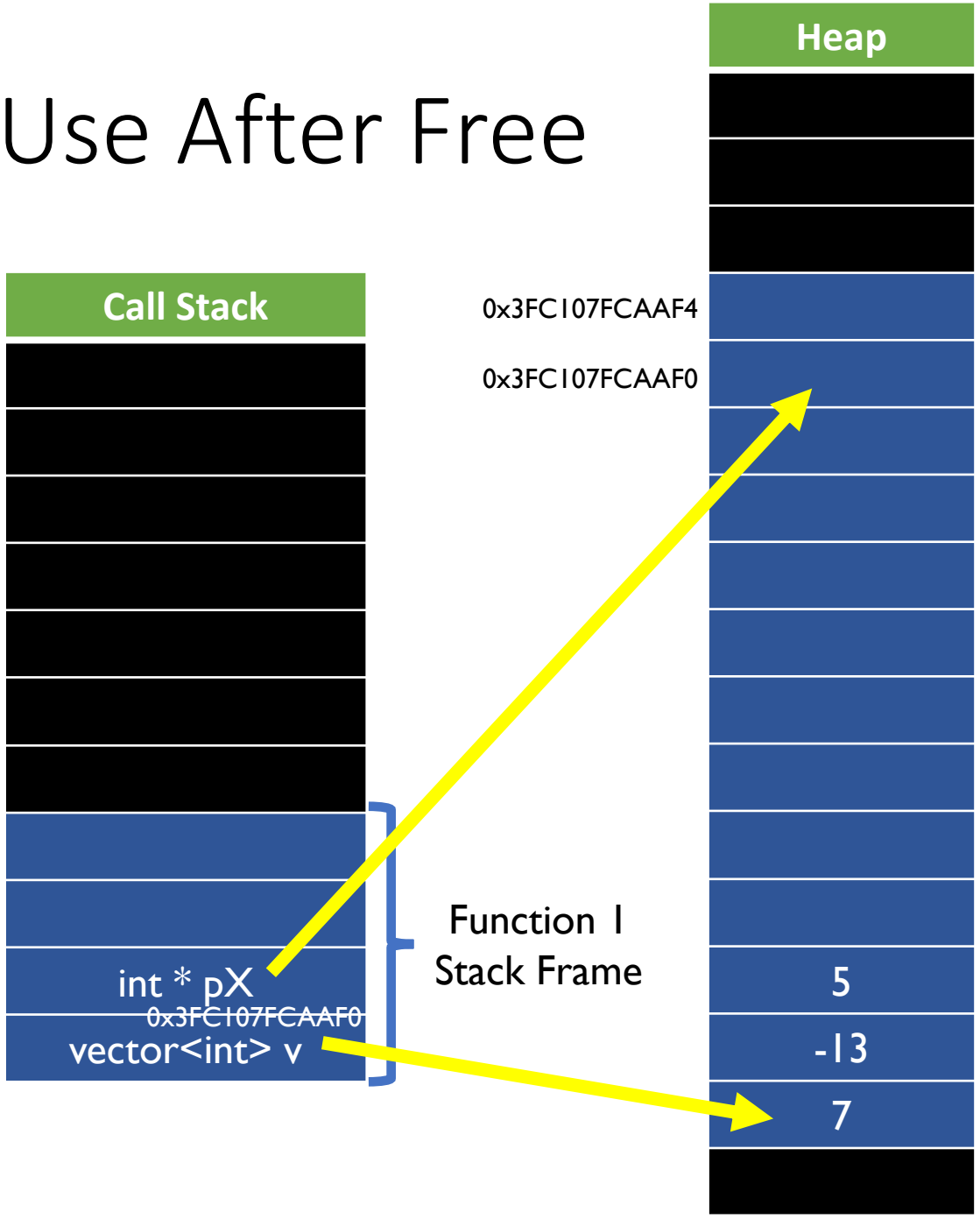
- `int * pX = new int;`
- `delete pX;`

Use After Free



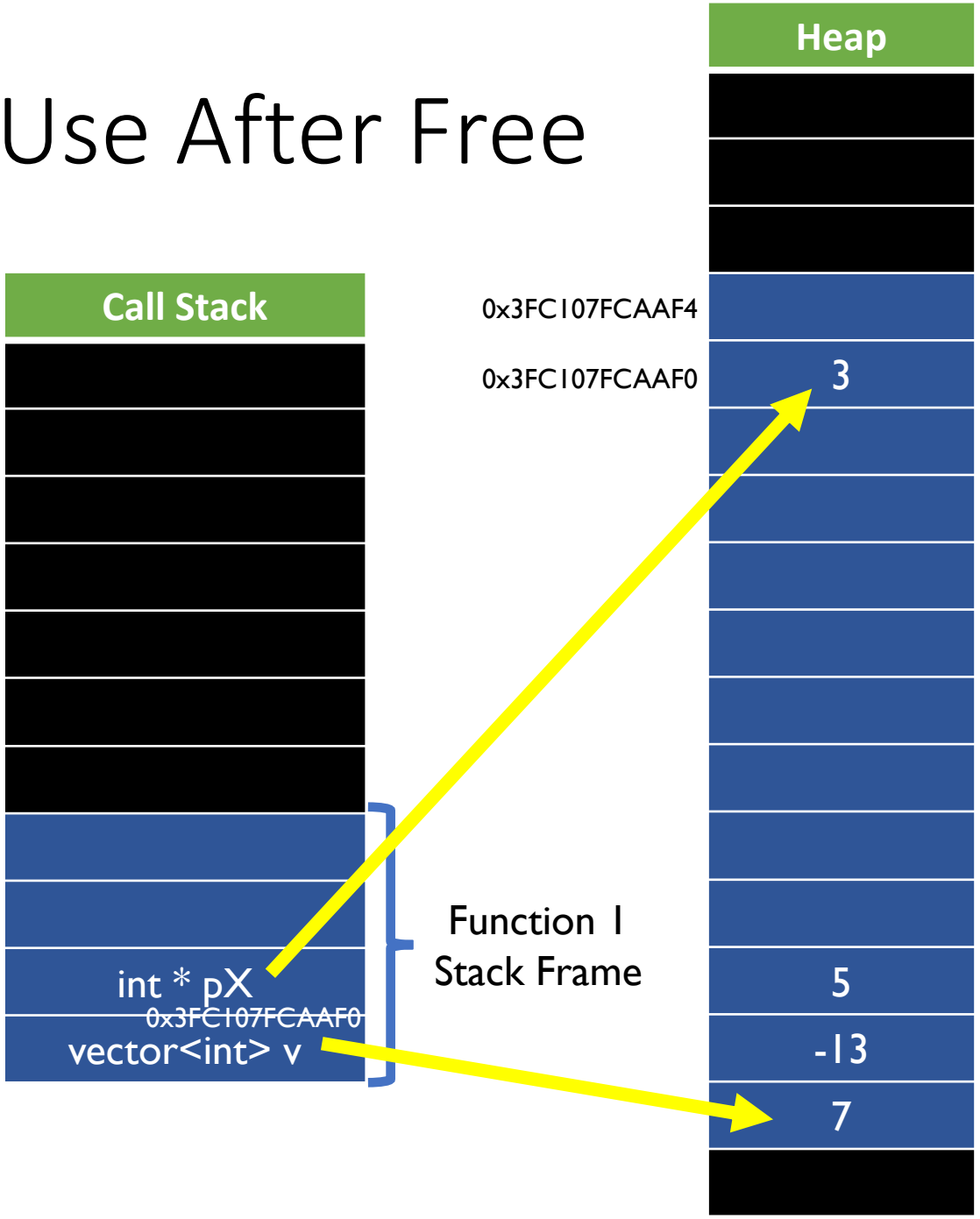
- `int * pX = new int;`
- `delete pX;`
  - `What about the address?`
  - `It is still stored in pX.`
- `vector.resize( 15 )`

Use After Free



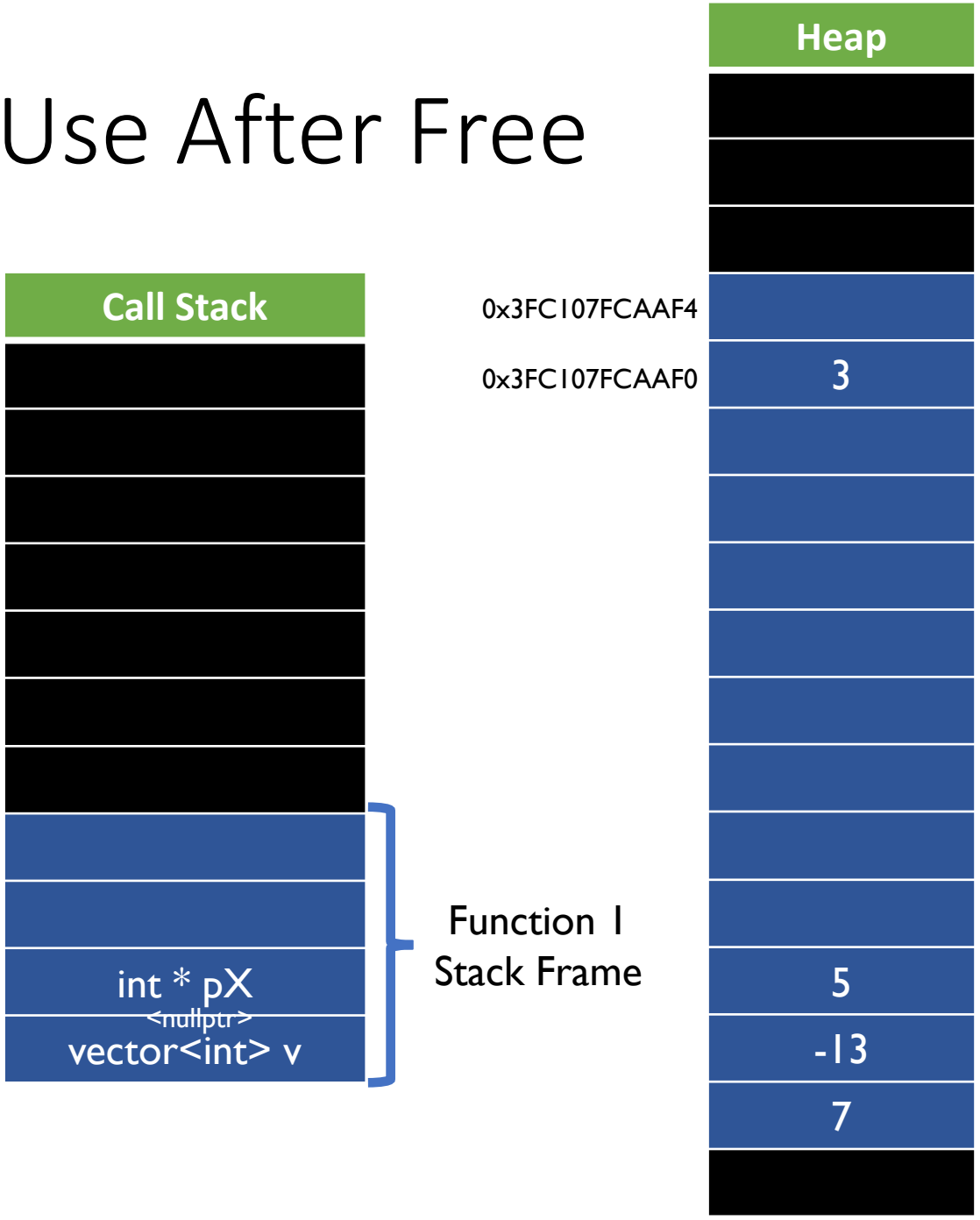
- `int * pX = new int;`
- `delete pX;`
  - `What about the "arrow"?`
  - `Well, it is still there.`
- `vector.resize( 15 )`
- `*pX = 3;`

Use After Free



- `int * pX = new int;`
- `delete pX;`
  - What about the "arrow"?
  - Well, it is still there.
- `vector.resize( 15 )`
- `*pX = 3;`
  - We just corrupted the memory in vector v.
  - Might not notice this for a long time…
- Can help fix this with:

Use After Free

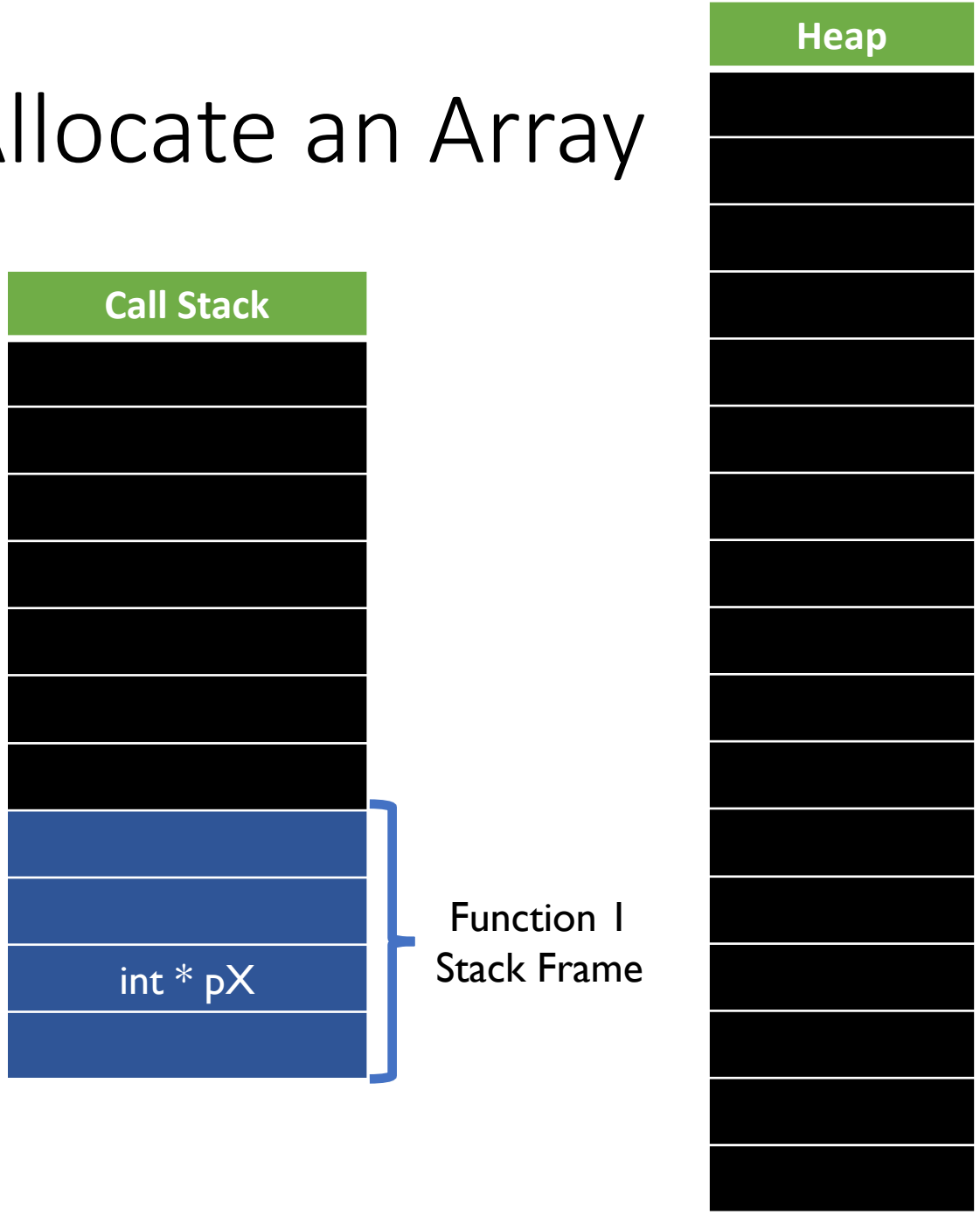


- `int * pX = new int;`
- `delete pX;`
  - What about the "arrow"?
  - Well, it is still there.
- `vector.resize( 15 )`
- `*pX = 3;`
  - We just corrupted the memory in vector v.
  - Might not notice this for a long time…
- Can help fix this with:
  - `pX = nullptr;`
  - `*pX = 3; // Crashes immediately`

Out of Bounds Access Error

- Since C++ does not check to make sure memory we are accessing is valid, it's easy to access memory that does not belong to us – to go *out of bounds.*
- myArray[-1] might appear to work, or it could crash the program
  - Note, while a program most likely would never use myArray[-1], they very easily could do myArray[myArray.size()]
  - And also very easily do: myArray[index], when index has a bad value (eg: was never initialized) in it.
- This issue exists with arrays both on the stack and on the heap.
- The address sanitizer can help with this (sometimes).

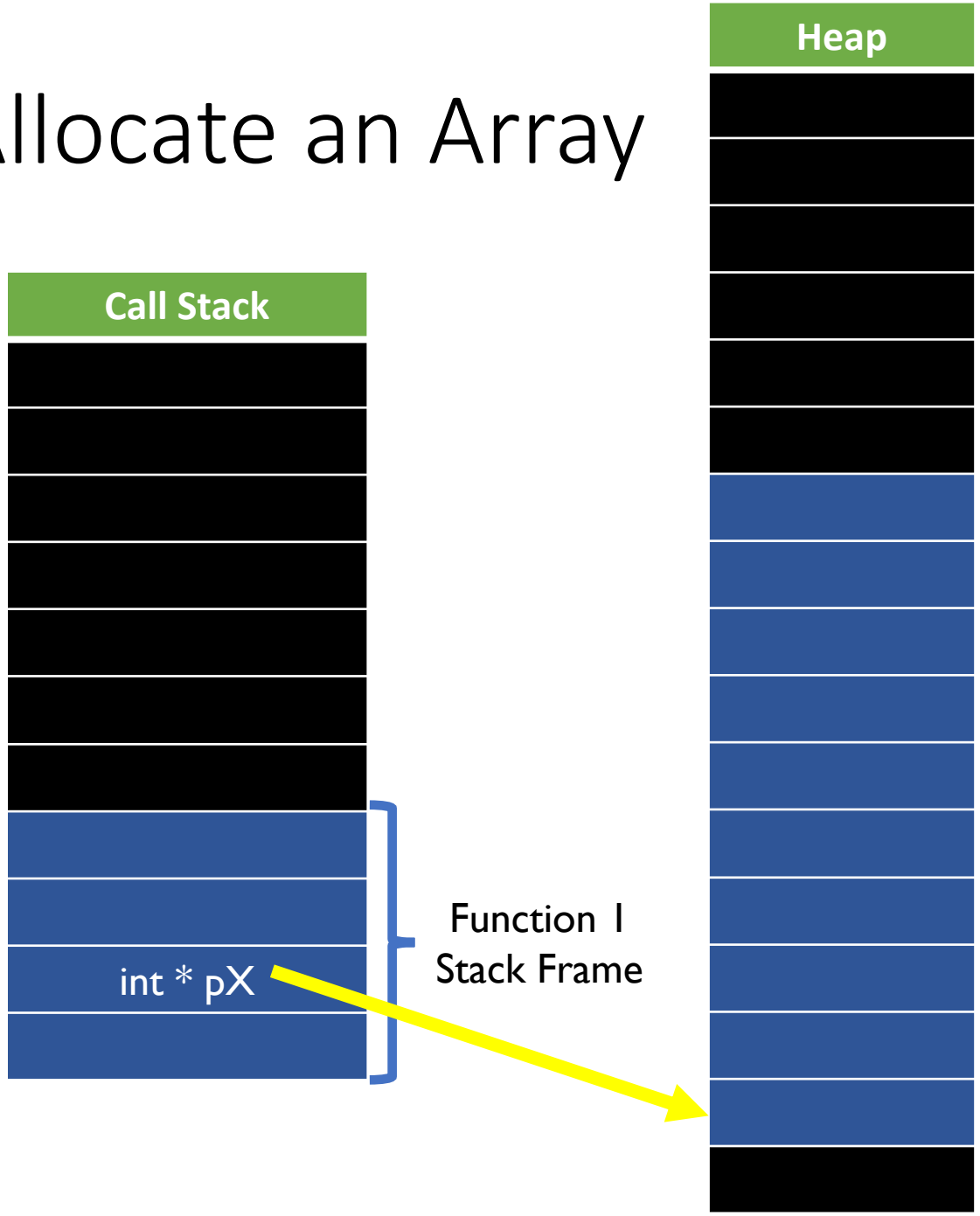
Allocate an Array




```
int * pX = new int[ 10 ];
```

- What does this do?

Allocate an Array



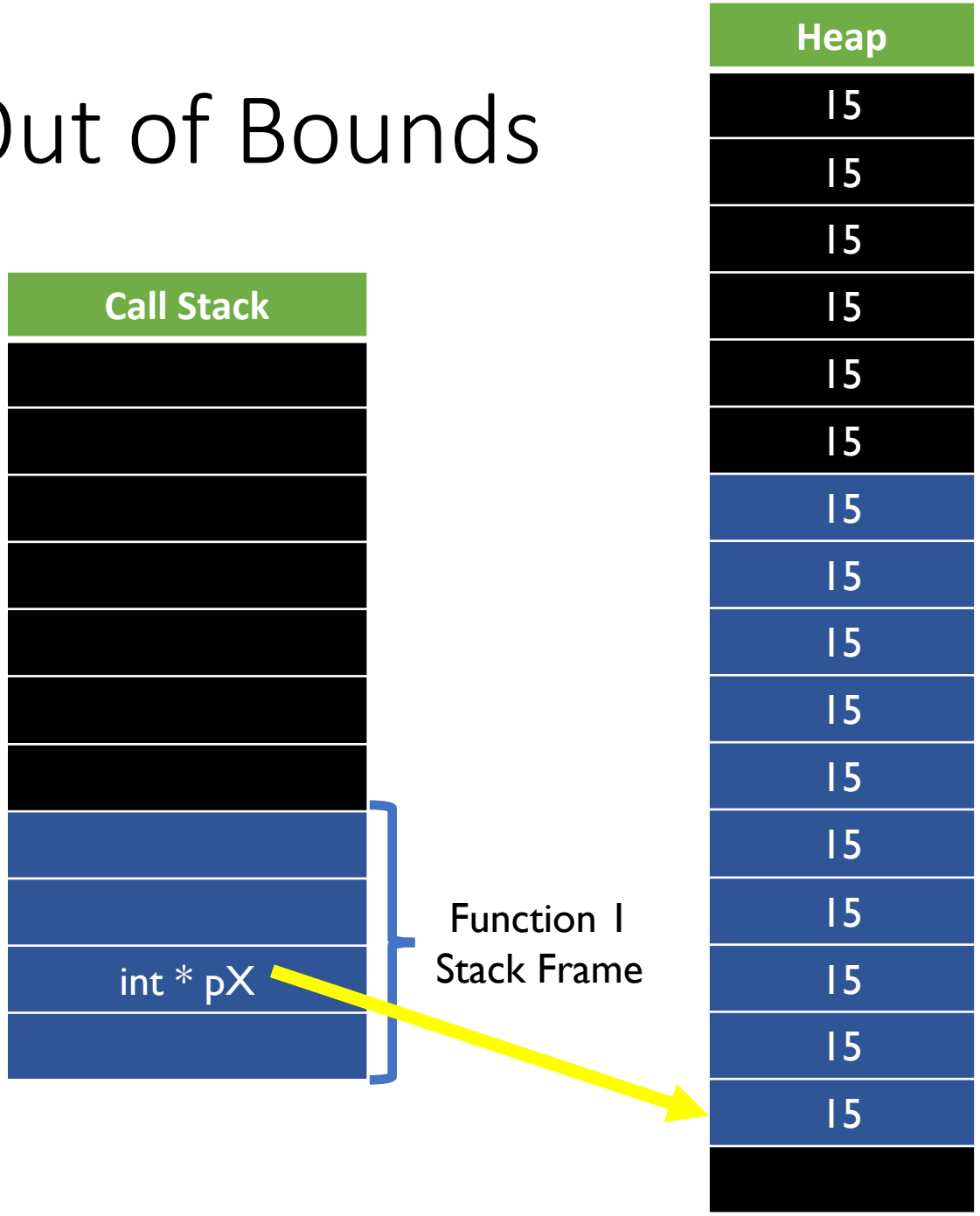

```
int * pX = new int[ 10 ];
```

- Memory has been allocated to us to use for pX.
- Now consider the following code:

```
for( int i = 0; i < 100; i++ ) {
   px[ i ] = 15;}
}
```

- See the problem?

Out of Bounds




```
int * pX = new int[ 10 ];
```

- Memory has been allocated to us to use for pX.
- Now consider the following code:

```
for( int i = 0; i < 100; i++ ) {
   px[ i ] = 15;}
}
```

- See the problem?
  - 15 is placed in memory we do not own.
  - (Memory that has not been allocated to us by the system.)

Returning Pointers to Stack Data (Don't!)

- Sometimes a programmer will accidentally return a pointer to data on the stack.
- When the function ends, the (call) stack memory for that function is deallocated, so we have a *dangling pointer* to invalid memory.
- This is a stack equivalent to the "use-after-free" error.
- Modern compilers are fairly good at catching these things and giving us warnings.
  - Pay attention to these warnings.

```
int * getInt () {
        int x;
        return &x;  // Returning address to a variable that no longer exists, but the
                       address may still exist!
}
```

Creating a String Object

- What information does a string need to know? How would we define this?

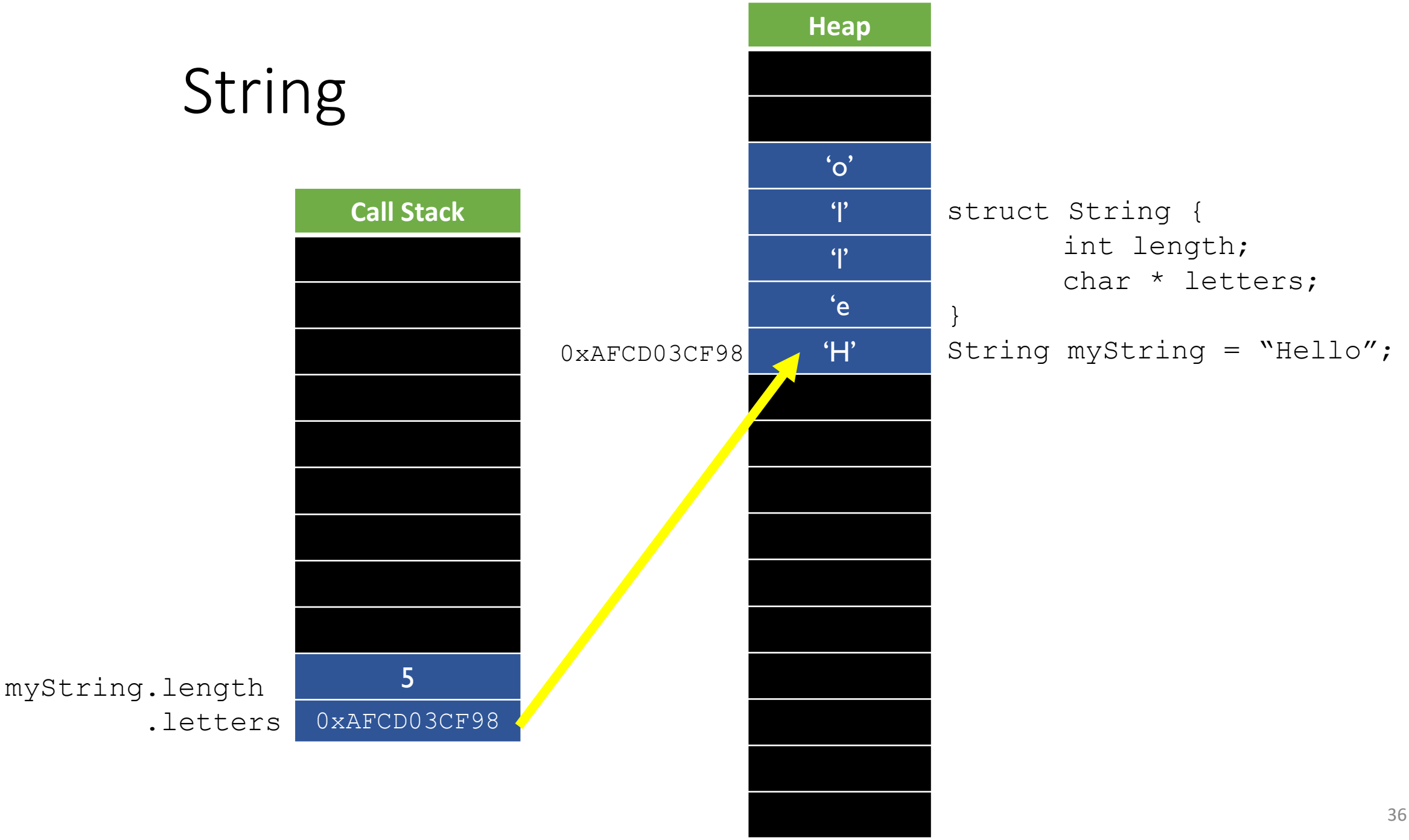
```
struct String {
        int length;
        char * letters;  // Start of the array on the heap.
}
```

- Now, how do we actually ***use*** this struct?

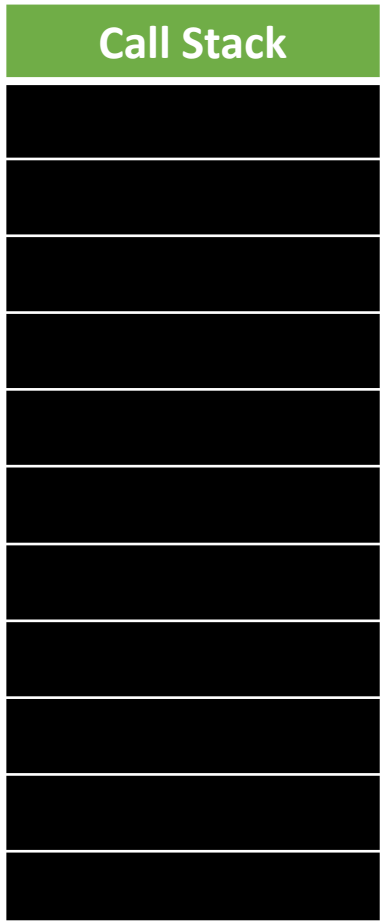
```
String myString = "Hello"; // Create a var of type String
```

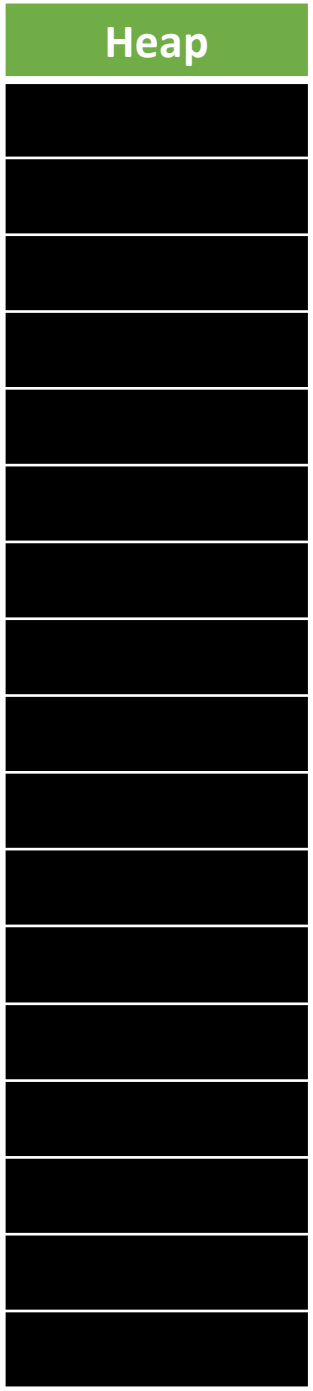
- Where is `myString` stored?
  - On the stack.
- What information is part of `myString` (and thus stored on the stack)?
  - length, and a pointer to a character array
- Where is the actual array of characters stored?
  - On the heap.

String


Heap
'o'
'l'
'l'
'e
0xAFCD03CF98
'H'
Call Stack
5
myString.length
0xAFCD03CF98
.letters
struct String {
int length;
char * letters;
}
String myString = "Hello";

Grow an Array

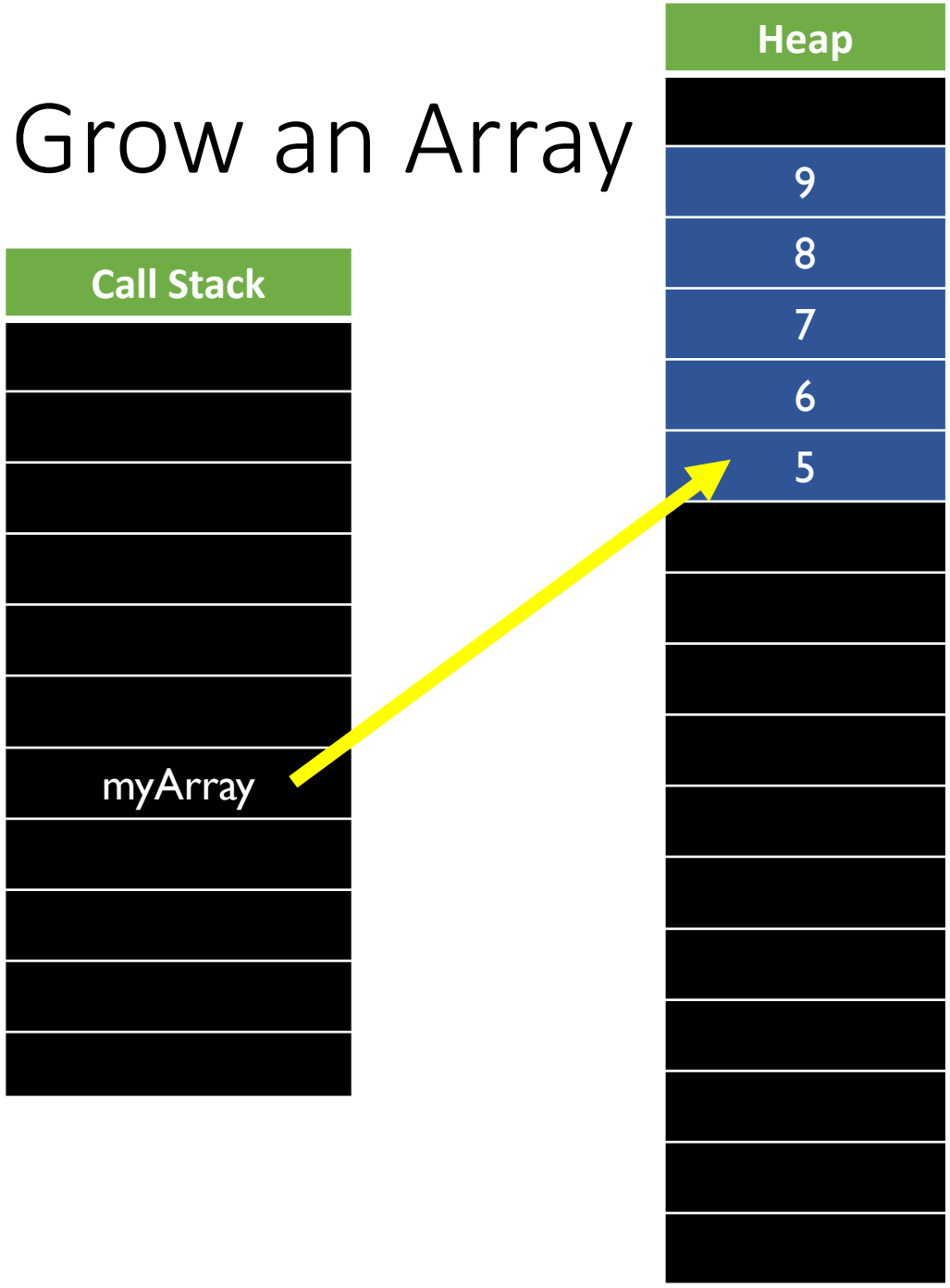




- Create an array of 5 integers

```
int * myArray = new int[ 5 ]{ 5,6,7,8,9 };
```

Grow an Array



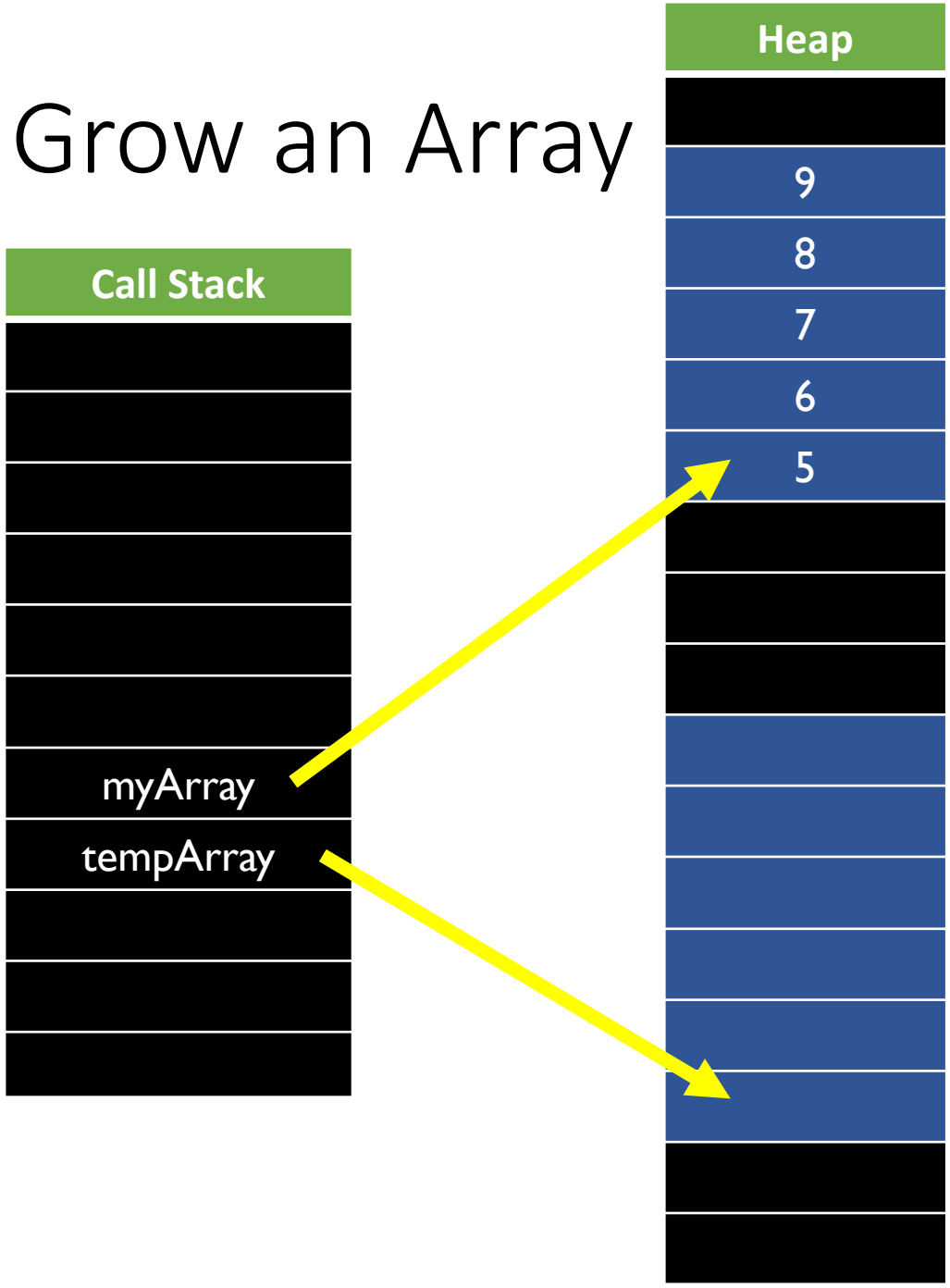
- Create an array of 5 integers

```
int * myArray = new int[ 5 ]{ 5,6,7,8,9 };
```

- Opps… I need to add a 6th number to myArray. How can I do this?
- Is there enough room allocated?
  - No – need to allocate more memory – how?

```
int * tempArray = new int[ 6 ];
```

Grow an Array



- Create an array of 5 integers

```
int * myArray =
            new int[ 5 ]{ 5, 6, 7, 8, 9 };
```

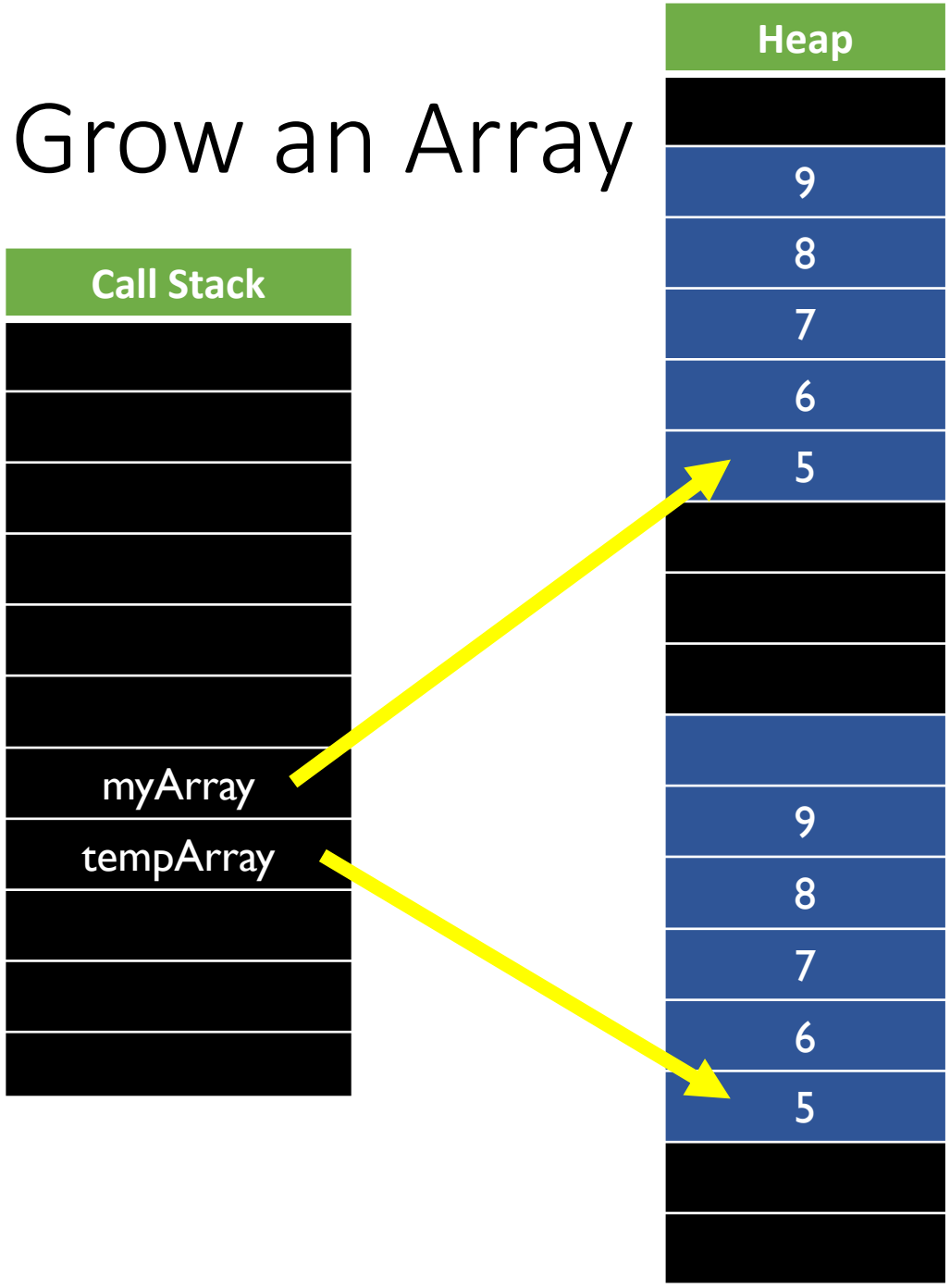
- Opps… I need to add a 6th number to myArray. How can I do this?
- Is there enough room allocated?
  - No – need to allocate more memory – how?

```
int * tempArray = new int[ 6 ];
```

- Now what?
  - Copy the old values into the new array. How?

```
for( int i = 0; i < 5; i++ ) {
        tempArray[ i ] = myArray[ i ];
}
```

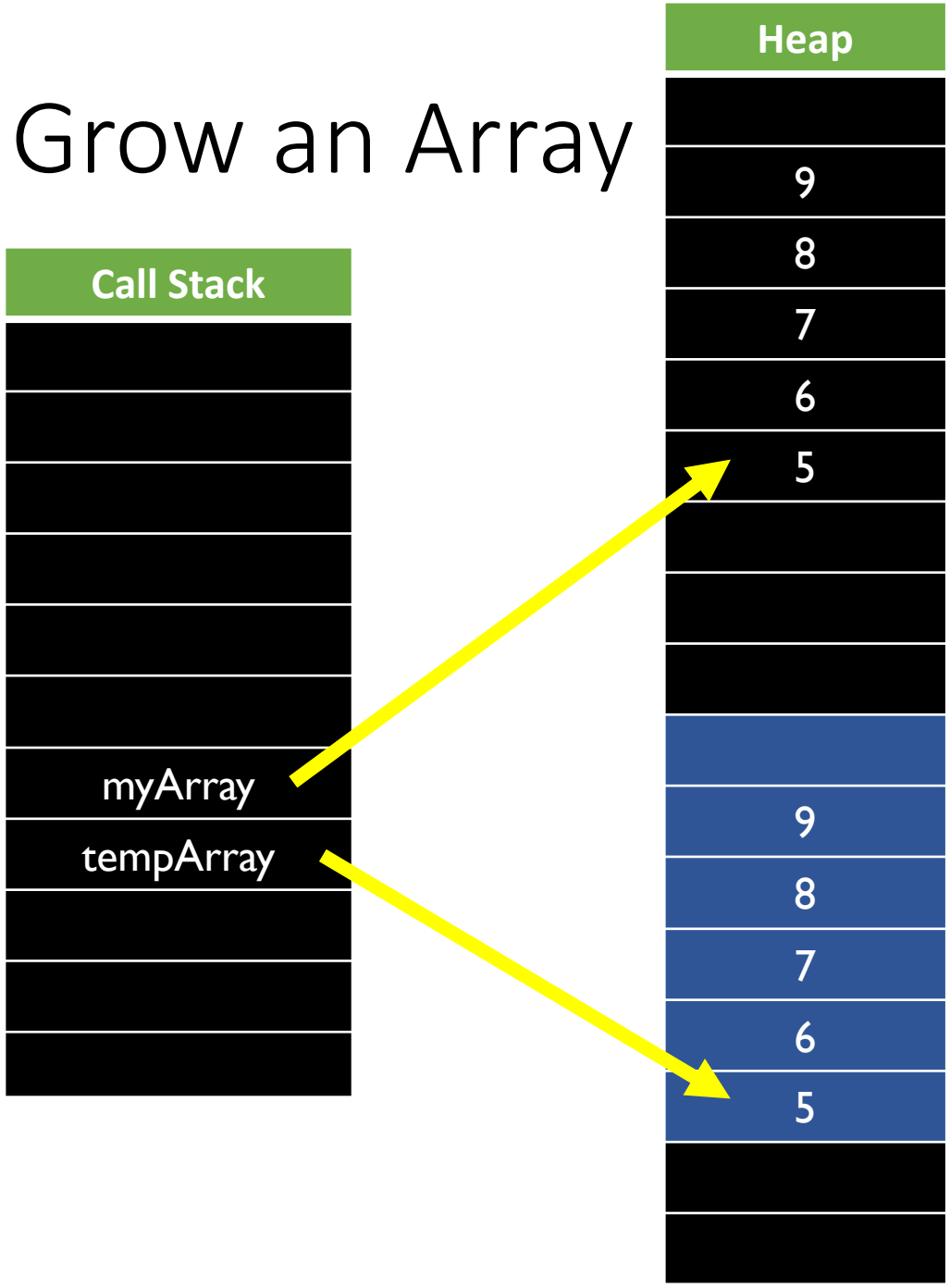
Grow an Array



- Now what about `tempArray` and `myArray`?
  - Do we need the memory that myArray currently uses?
    - No

```
delete [ ] myArray;
```

Grow an Array

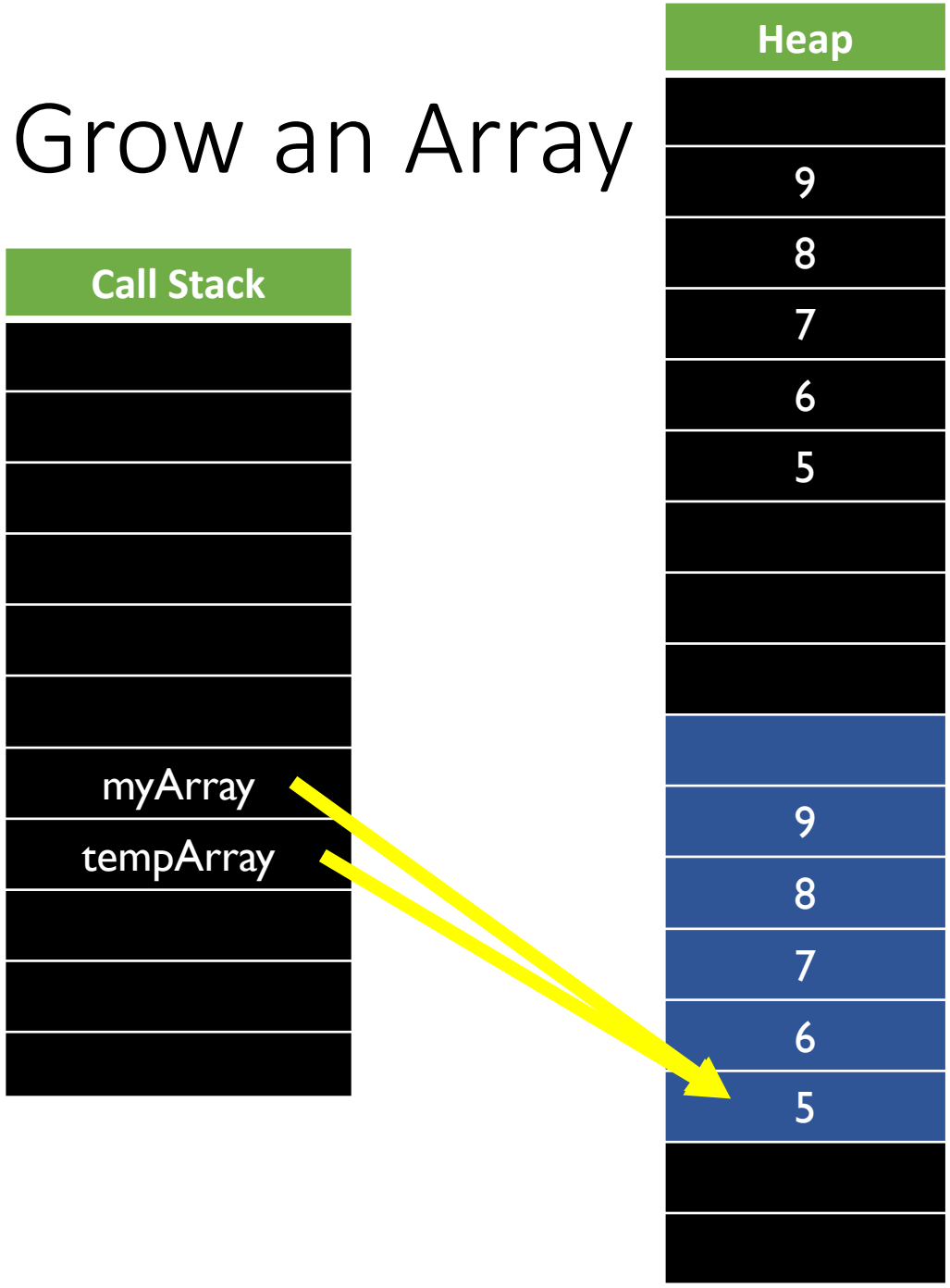


- Now what about `tempArray` and `myArray`?
  - Do we need the memory that myArray currently uses?
    - No

      ```
      delete [ ] myArray;
      ```
  - Why are the numbers still there?
    - Deleting does not remove / clear memory, just tells the system that we are no longer using it.
  - Why is myArray still pointing there?
    - Deleting does not change the address contained in `myArray`. What should we do to fix this?

      ```
      myArray = tempArray;
      ```

Grow an Array



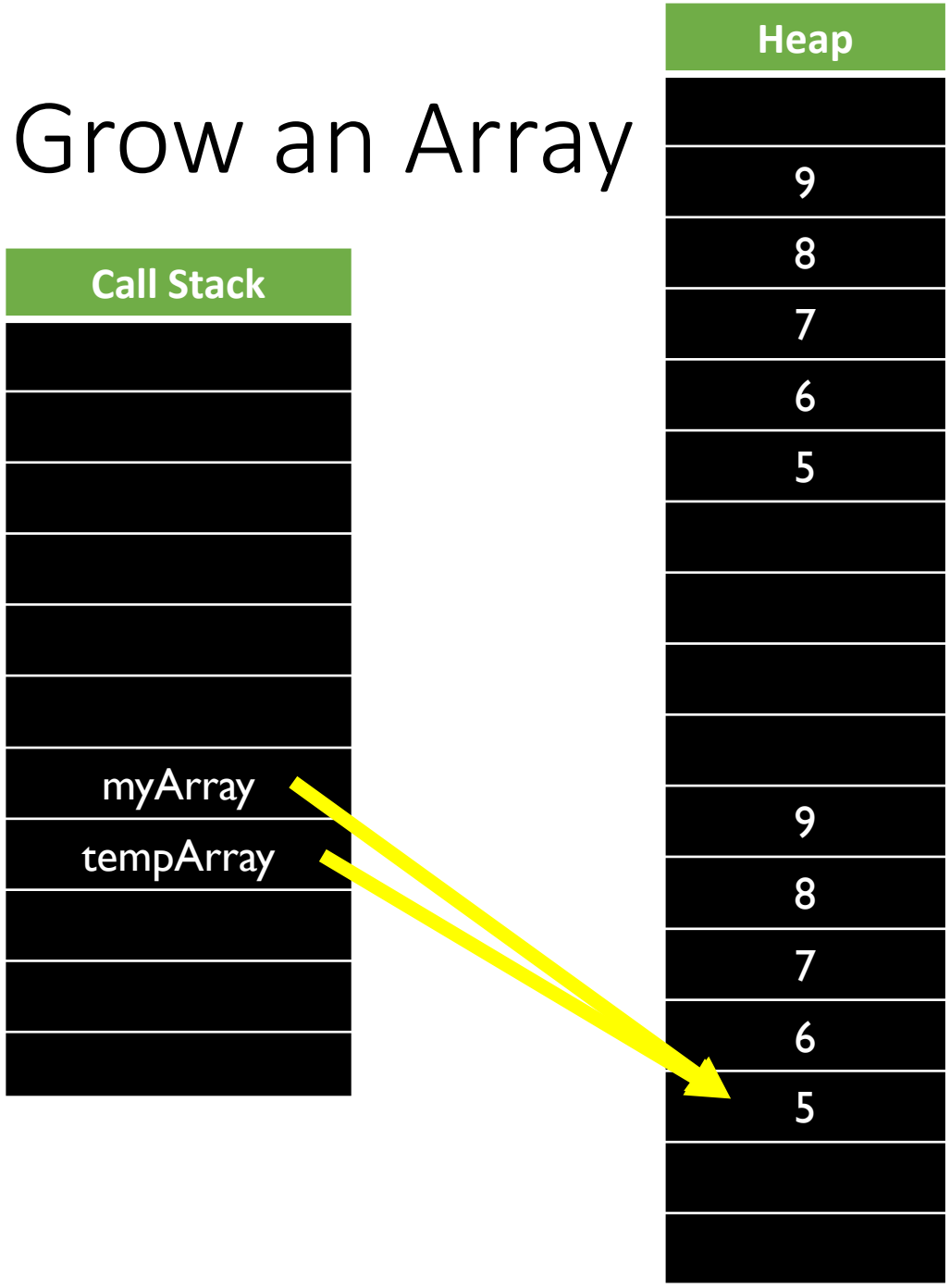
- Now what about `tempArray` and `myArray`?
  - Do we need the memory that myArray currently uses?
    - No

    ```
    delete [ ] myArray;
    ```
  - Why are the numbers still there?
    - Deleting does not remove / clear memory, just tells the system that we are no longer using it.
  - Why is myArray still pointing there?
    - Deleting does not change the address contained in `myArray`. What should we do to fix this?

    ```
    myArray = tempArray;
    ```
  - What about `tempArray`? What to do with it?

```
delete [ ] tempArray; //???
```

Grow an Array



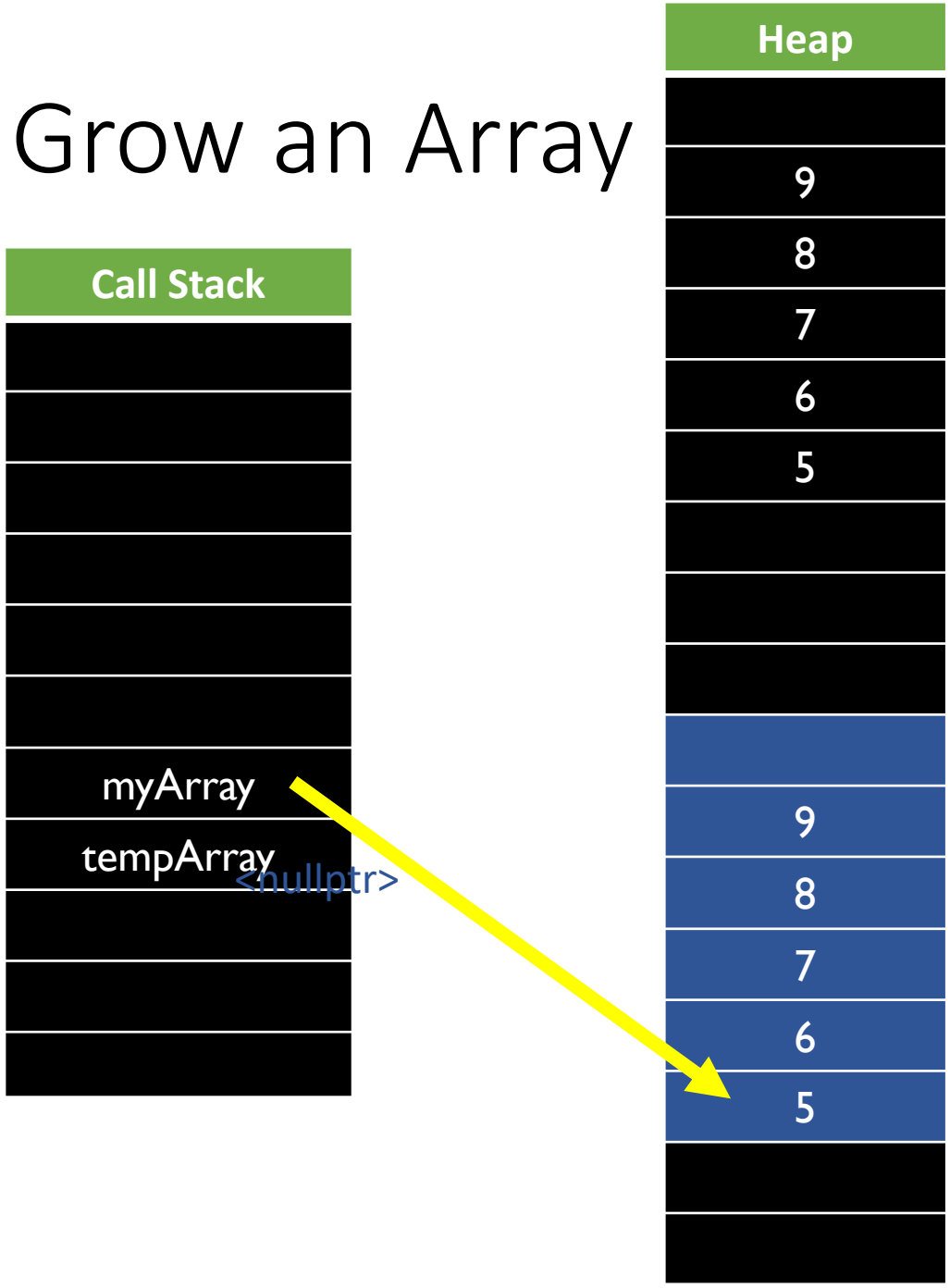
- Now what about `tempArray` and `myArray`?
  - Do we need the memory that myArray currently uses?
    - No

    `delete [ ] myArray;`
  - Why are the numbers still there?
    - Deleting does not remove / clear memory, just tells the system that we are no longer using it.
  - Why is myArray still pointing there?
    - Deleting does not change the address contained in `myArray`. What should we do to fix this?

    `myArray = tempArray;`
  - What about `tempArray`? What to do with it?

```
delete [ ] tempArray; // NO!!! we still need that memory!
```

Grow an Array



- Now what about `tempArray` and `myArray`?
  - Do we need the memory that myArray currently uses?
    - No

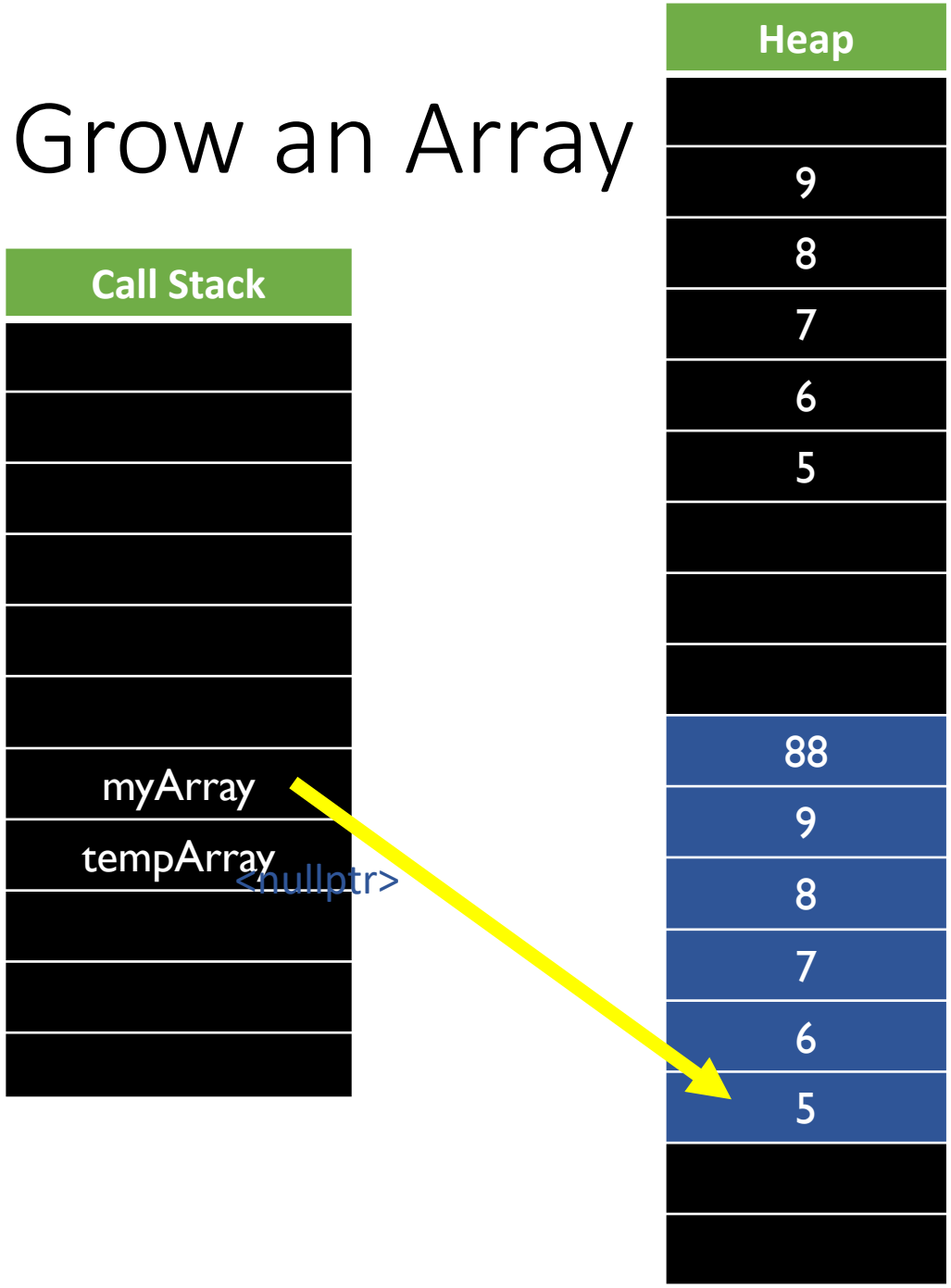
      `delete [ ] myArray;`
  - Why are the numbers still there?
    - Deleting does not remove / clear memory, just tells the system that we are no longer using it.
  - Why is myArray still pointing there?
    - Deleting does not change the address contained in `myArray`. What should we do to fix this?

      `myArray = tempArray;`
  - What about `tempArray`? What to do with it?

~~`delete [ ] tempArray; // NO!!! we still need that memory!`~~

```
tempArray = nullptr; //nullify the pointer
```

Grow an Array



- Now what about `tempArray` and `myArray`?
  - Do we need the memory that myArray currently uses?
    - No

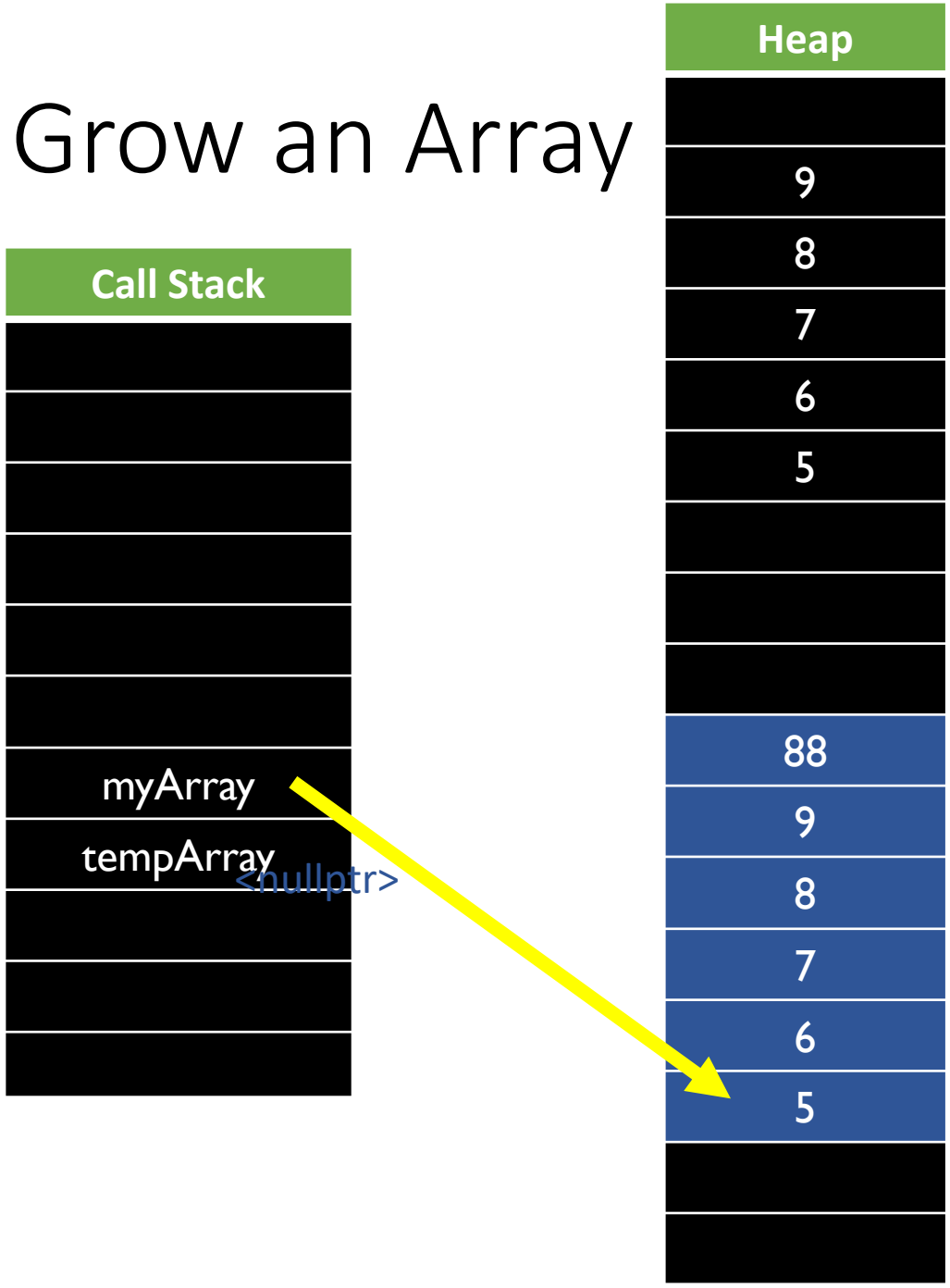
      `delete [ ] myArray;`
  - Why are the numbers still there?
    - Deleting does not remove / clear memory, just tells the system that we are no longer using it.
  - Why is myArray still pointing there?
    - Deleting does not change the address contained in `myArray`. What should we do to fix this?

      `myArray = tempArray;`
  - What about `tempArray`? What to do with it?

~~`delete [ ] tempArray; // NO!!! we still need that memory!`~~

```
tempArray = nullptr;
myArray[ 5 ] = 88; // Finally add the new value
```

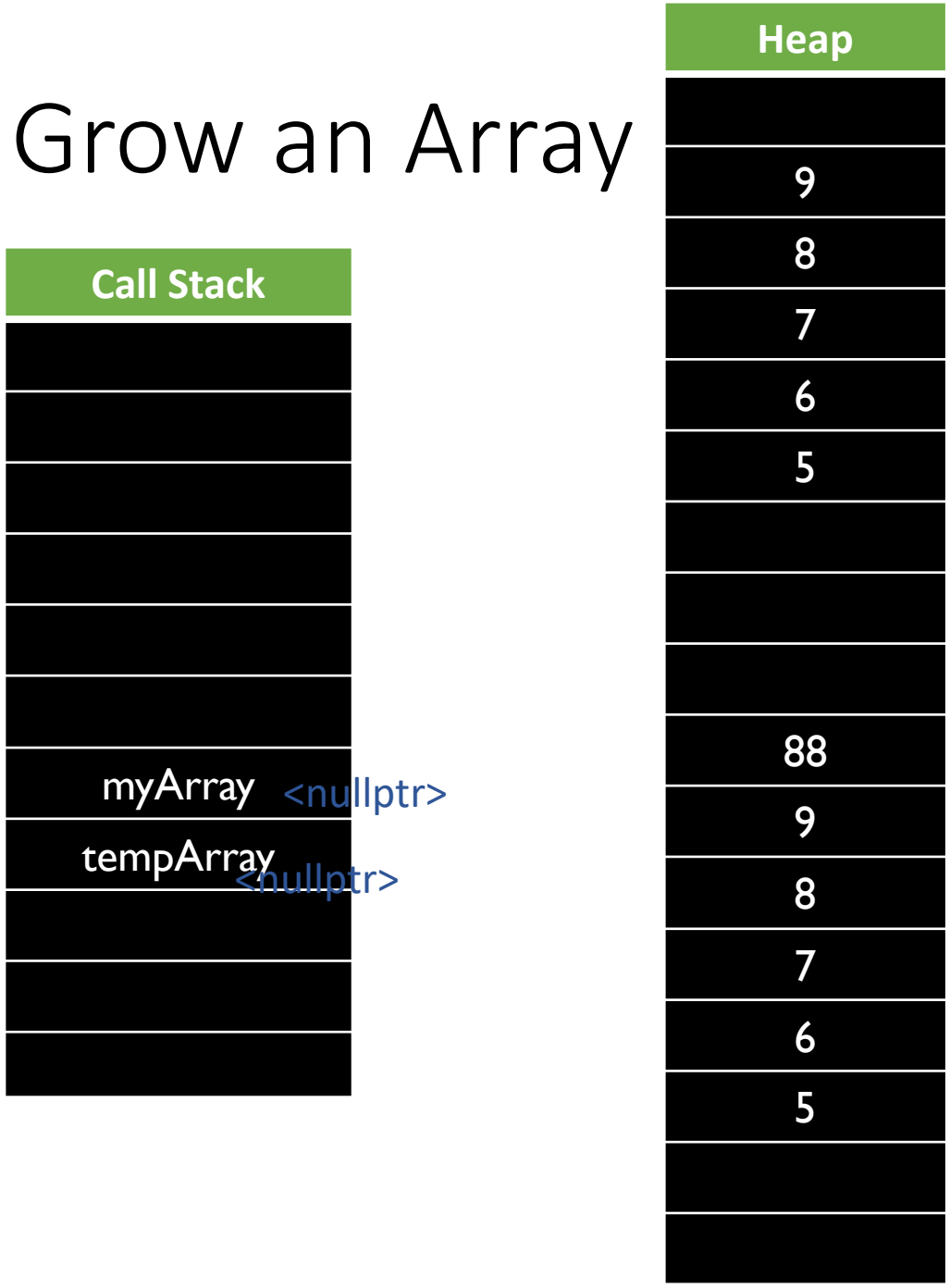
Grow an Array



- After we print out the values in myArray (or do whatever we want with the array), and are done with it, what do we do?

```
delete [ ] myArray; // These two lines go together quite
myArray = nullptr;  // frequently.
```

Grow an Array



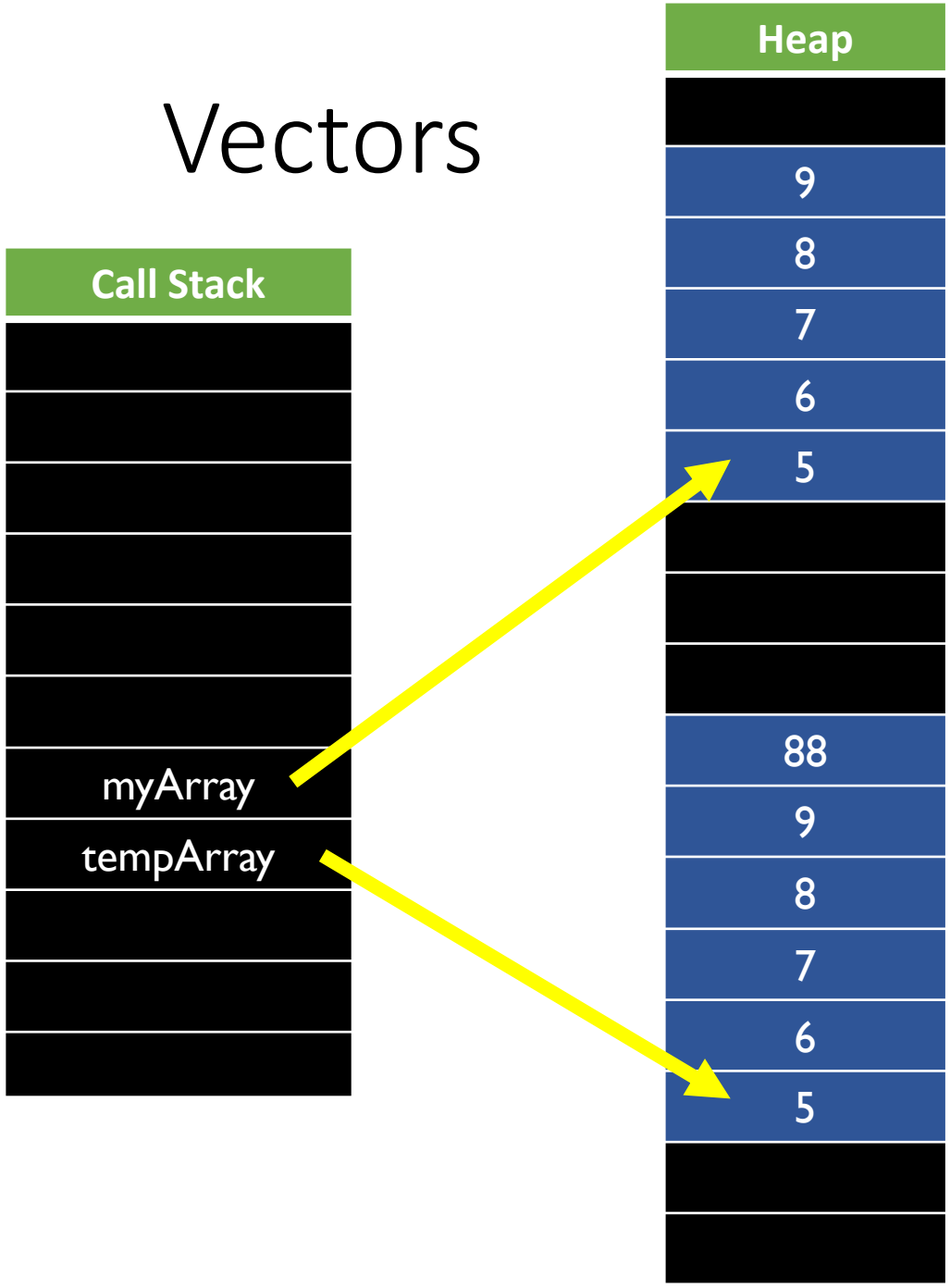
- After we print out the values in myArray (or do whatever we want with the array), and are done with it, what do we do?

```
delete [ ] myArray; // These two lines go together quite
myArray = nullptr;  // frequently.
```

Help Debugging Memory Errors

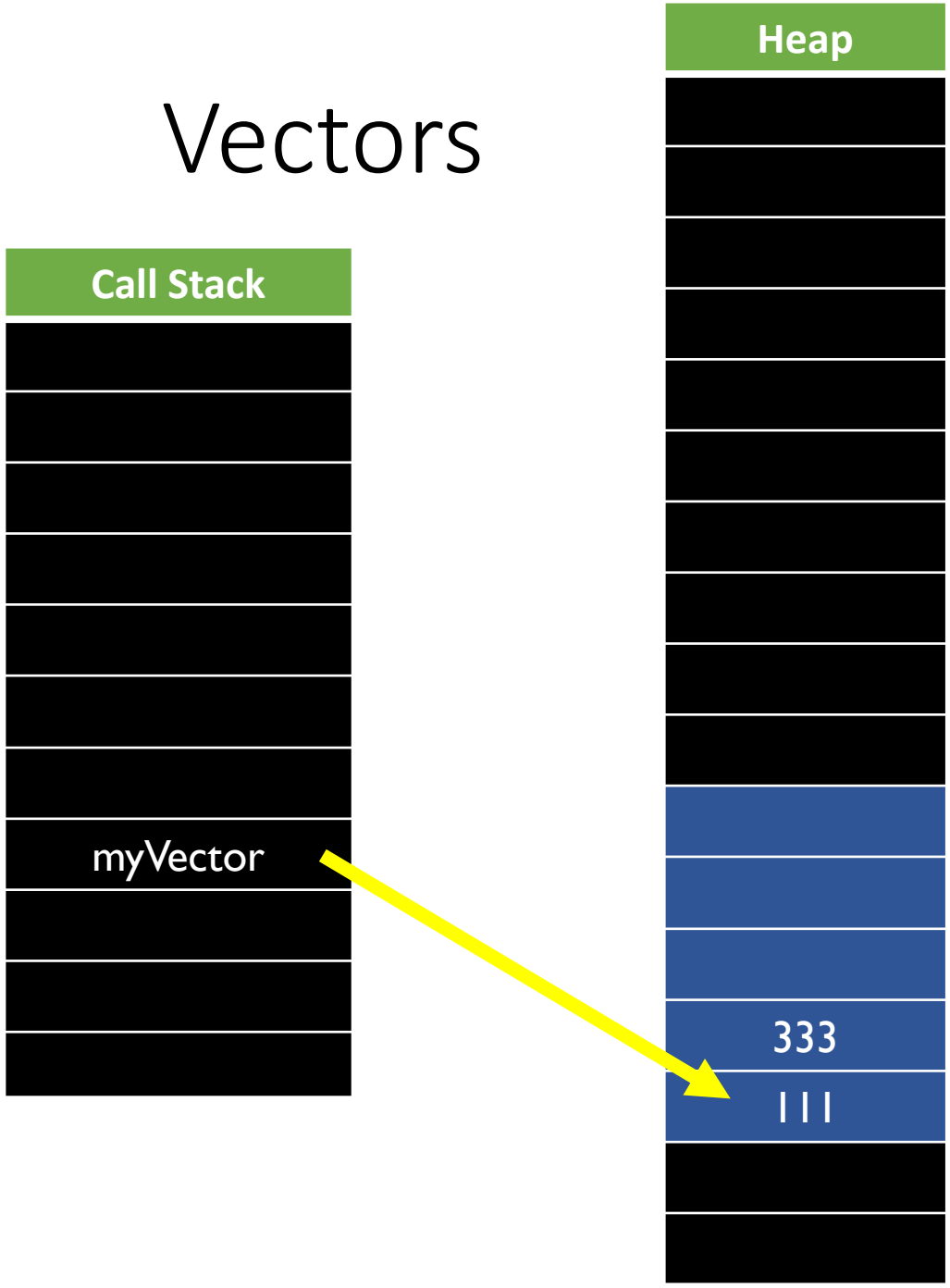
- Turn on Address Sanitizer (and Detect use of stack after return.)

Vectors



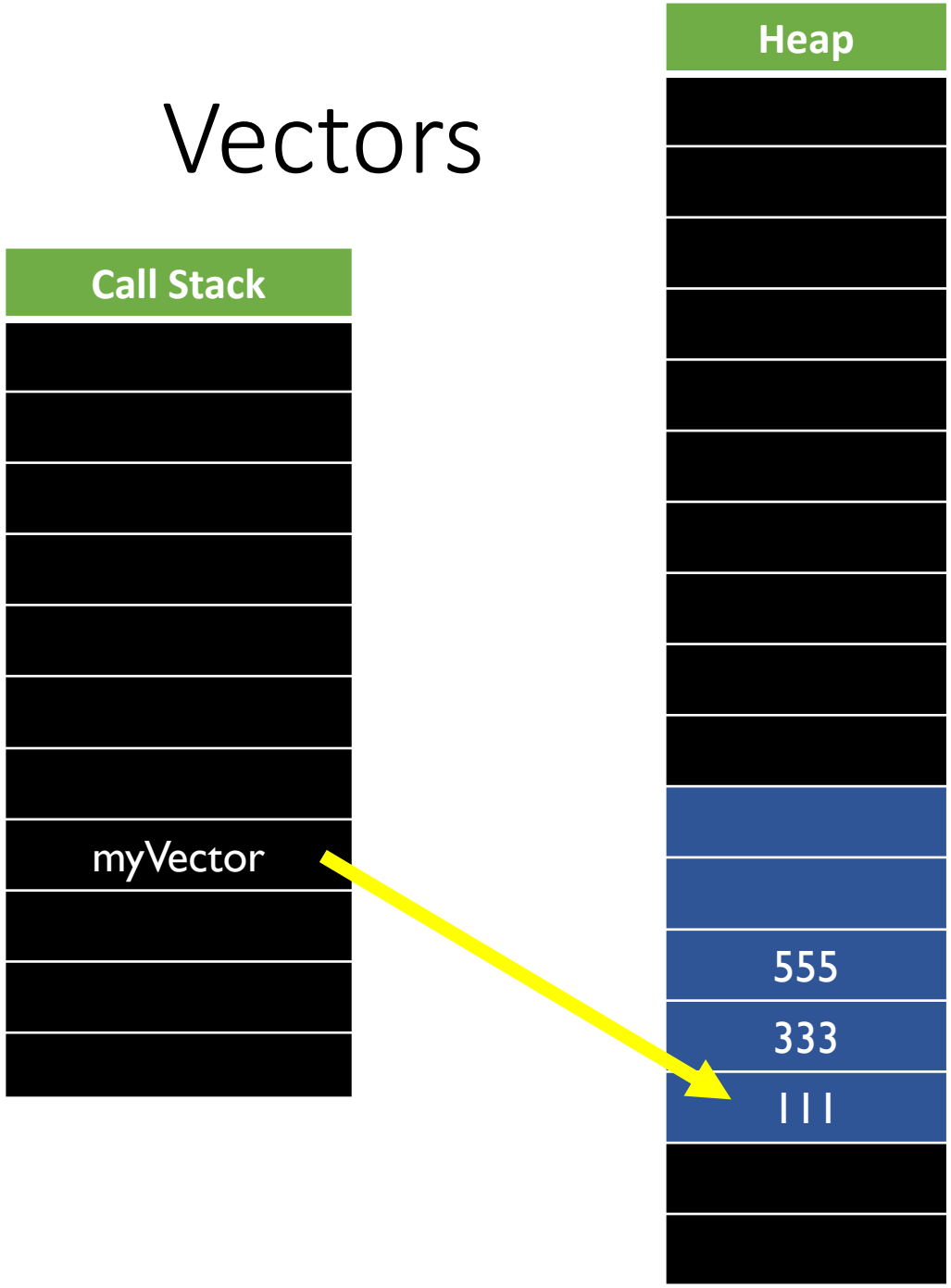
- Adding to the end of an array like this takes a lot of work.
  - Allocate new space
  - Copy old data
  - Delete old memory
- std::vectors are much smarter.

Vectors



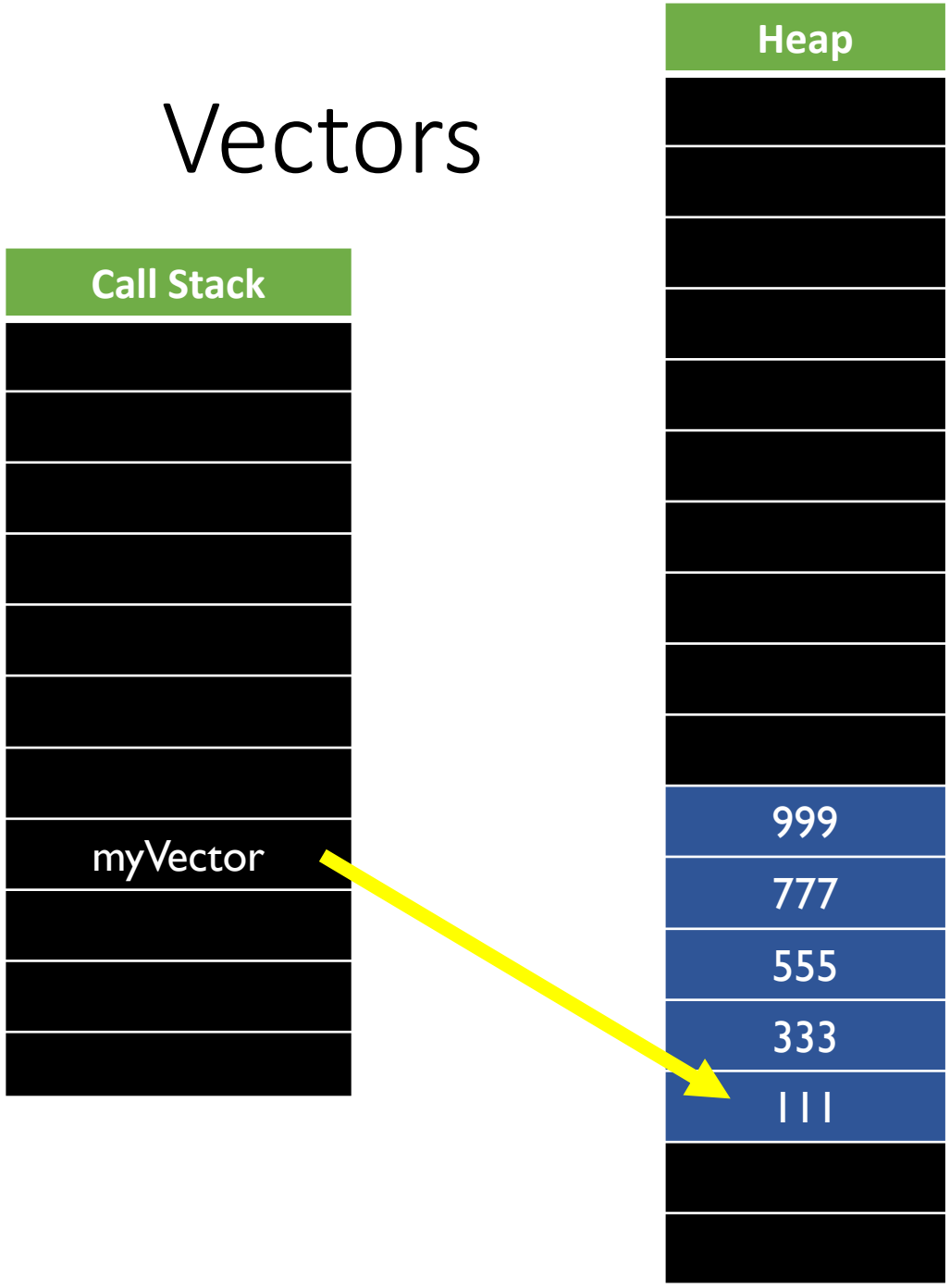
- Remind me, what 3 things do Vectors keep track of?
  - A pointer to the beginning of the data
  - The current size of the data
  - The allocated capacity
- In this case:
  - Size: 2 (Two elements)
  - Capacity 5 (Number of total elements we can hold)
  - So push_back(555) is very fast:

Vectors



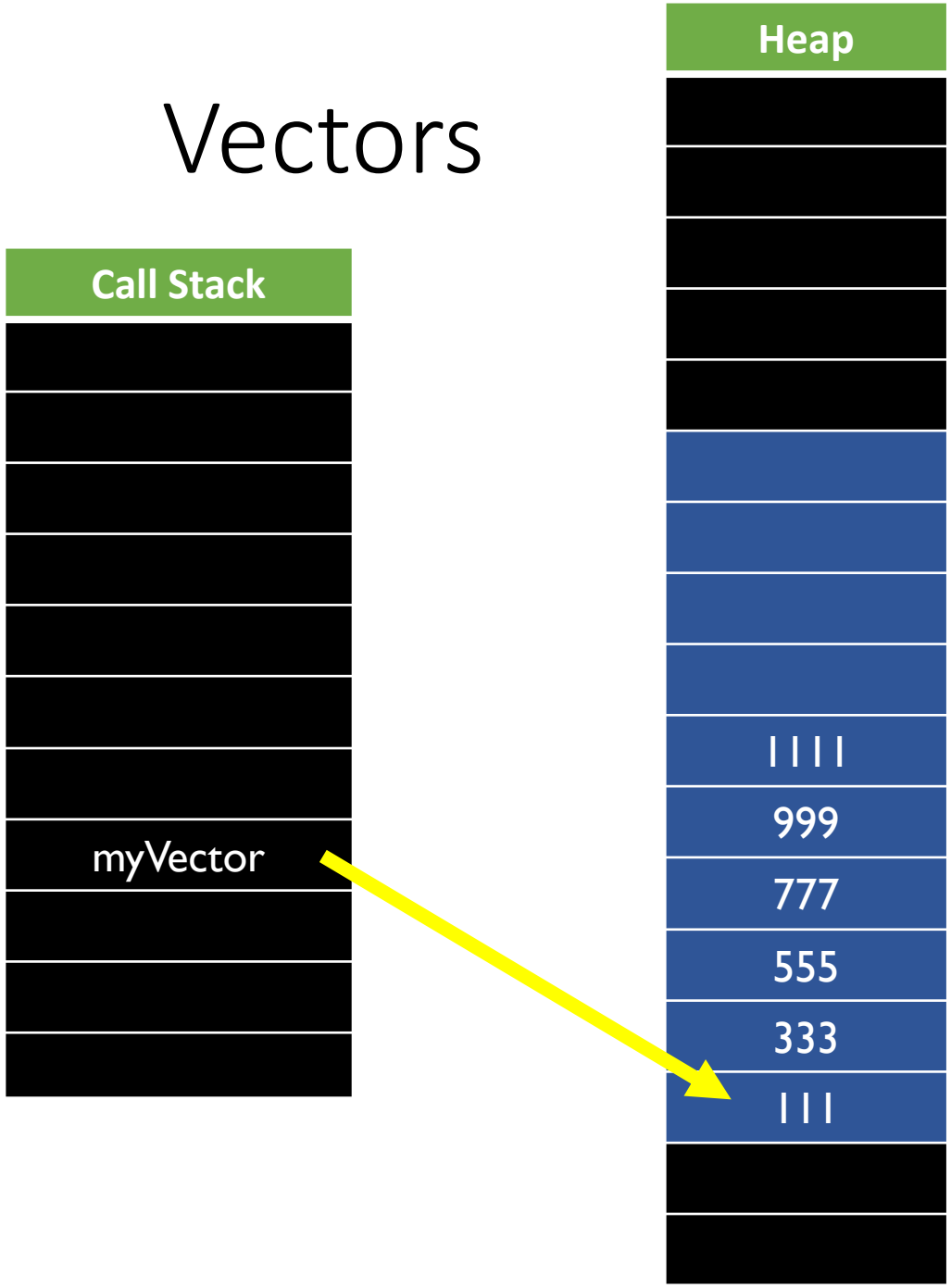
- Vectors keep track of 3 things.
  - A pointer to the beginning of the data
  - The current size of the data
  - The allocated capacity
- In this case:
  - Size: 2 (Two elements)
  - Capacity 5 (Number of total elements we can hold)
  - So push_back(555) is very fast:

Vectors



- push_back(777)
- push_back(999)
- Now what happens when we push_back(1111)?

Vectors



- Now what happens when we push_back(1111)?
  - The vector will double its allocated space and copy the old data into the new space.
  - Note: Despite the appearance of this illustration, the new space will not be on top of the old space.

Today’s Assignment(s)

- Code Review
- Finish up Bit manipulation assignment
- Lab on Pointers (due tomorrow)