CIS 190: C/C++ Programming

Assorted Topics
(and More on Pointers)

Outline

- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers

Input and Output Streams

- printf
  - stdout
  - output written to the terminal
- scanf
  - stdin
  - input read in from user
- redirection
  - executable < input.txt > output.txt

FILE I/O Basics

- read in from and print out to files
- use a file pointer (FILE*)

```
FILE * fopen (<filename>, <mode>)
```

- <filename> is a string
- <mode> is single-character string

FILE I/O Reading and Writing

```
ifp = fopen("input.txt", "r");
```

- opens input.txt for reading
  - file must already exist

```
ofp = fopen("output.txt", "w");
```

- opens output.txt for writing
  - if file exists, it will be overwritten

File I/O Opening and Closing

- before using file pointers, make sure they're valid

- if the file pointer is NULL, there was an error
  – need to deal with it – exit, re-prompt, etc.

- after you're done with a file, close it

```
fclose(ifp);
```

Using File Pointers

- fprintf

```
fprintf(ofp, "print: %s\n", textStr);
```

  – output written to where **ofp** points

- fscanf

```
fscanf(ifp, "%s %d", inputStr,
                     &inputInt);
```

  – input read in from where **ifp** points

Using stderr with fprintf

- three standard streams: stdin, stdout, stderr
- printing to stderr prints to the console
  - even when using redirection!

```
if (filePointer == NULL)
{
   fprintf(stderr, "The file %s
     could not be opened.\n", fileName);
   exit(-1); /* requires <stdlib.h> */
}
```

Reaching EOF with fscanf

- knowing when to stop reading in from a file
- EOF = End Of File (defined in a library)

```
while (fscanf(ifp, "%s", str) != EOF)
{
  /* do things */
}
/* while loop exited, EOF reached */
```

Outline

- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers

Command Line Arguments

- parameters to main() function

```
int main(int argc, char **argv)
```

- `int argc` – number of arguments
  - including name of executable
- `char **argv` – array of argument strings
  - `argv[0]` is string containing name of executable
  - `argv[1]` is first argument, etc.

Using argc

- before using command line arguments, double check that they exist using argc

- check the value of argc
  - if it's not correct, exit and prompt the correct args:

```
if (argc != NUM_ARGS) {
   fprintf(stderr,
           "Incorrect number of arguments.\n");
   fprintf(stderr,
           "Expected <exec> <input filename>.\n");
   exit(-1); }
```

Using argv

- **`char **argv`** is an array of strings
  - executable is **`argv[0]`**
  - arguments start at **`argv[1]`**

- to convert from a string to an integer:

```
intArg = atoi(argv[INT_ARG]);
```

  - **`atoi()`** converts alpha to int
    - need to #include <stdlib.h>

Outline

- File I/O
- Command Line Arguments
- Random Numbers
- Pointers Again

Random Numbers

- useful for many things:
  - cryptography, games of chance & probability, procedural generation, statistical sampling
- generated “random numbers” are PSUEDO random
- “Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” – *John von Neumann*

Seeding for Randomness

- you can **seed** the random number generator

- same seed means same “random” numbers
  - good for testing

```
void srand (unsigned int seed);
```

Seeding with Time

- can also give a "unique" seed with **`time()`**
  - need to #include <time.h> library

```
int timeSeed = (int) time(0);
srand(timeSeed);
```

- NOTE: if you want to use the time() function, do not have a variable called time

```
error: called object 'time' is not a function
```

Generating Random Numbers

```
int rand (void);
```

- returns an integer between 0 and RAND_MAX

- use % to get the range you want:

```
/* 0 to MAX - 1 */
int random = rand() % MAX;

/* returns MIN to MAX, inclusive */
int random = rand() % (MAX - MIN + 1) + MIN;
```

Outline

- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers

Why Pointers Again?

- important programming concept
- understand what's going on "inside"
- other languages use pointers heavily
  - you just don't see them!

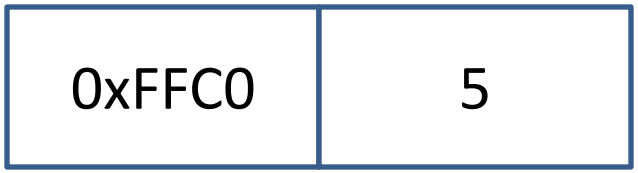
- but pointers can be difficult to understand
  - abstract concept
  - unlike what you've learned before

Memory Basics – Regular Variables

- all variables have two parts:
  - value

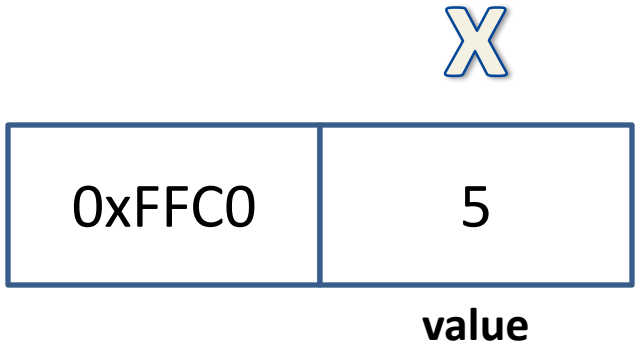
Memory Basics – Regular Variables

- all variables have two parts:
  - value
  - address where value is stored



Memory Basics – Regular Variables

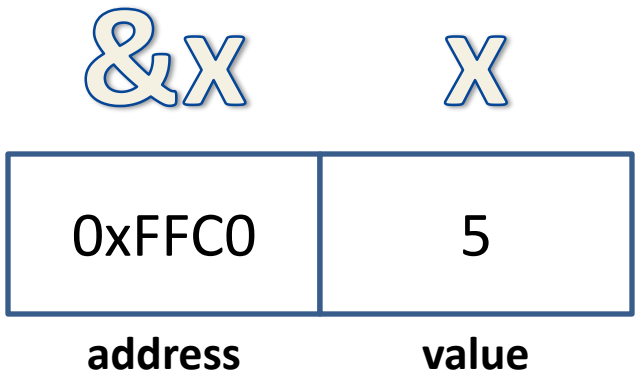
- all variables have two parts:
  - value
  - address where value is stored



- **x**'s **value** is 5

Memory Basics – Regular Variables

- all variables have two parts:
  - value
  - address where value is stored

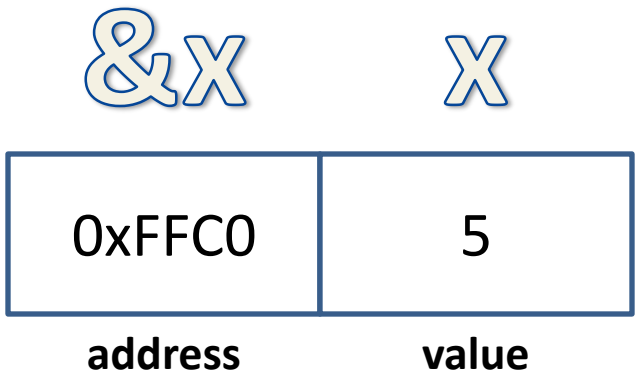


- **x**'s **value** is 5
- **x**'s **address** is 0xFFC0

Memory Basics – Regular Variables

- so the code to declare this is:

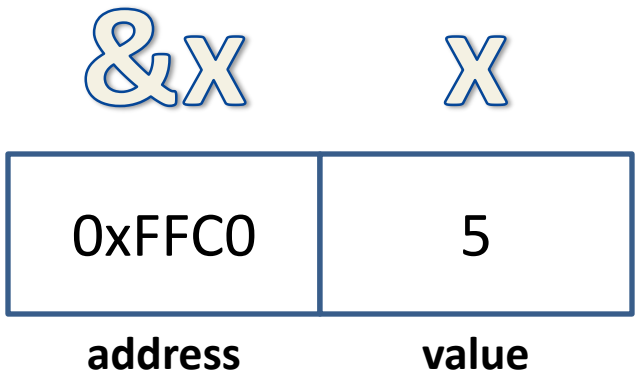
```
int x = 5;
```



Memory Basics – Regular Variables

- we can also declare a pointer:

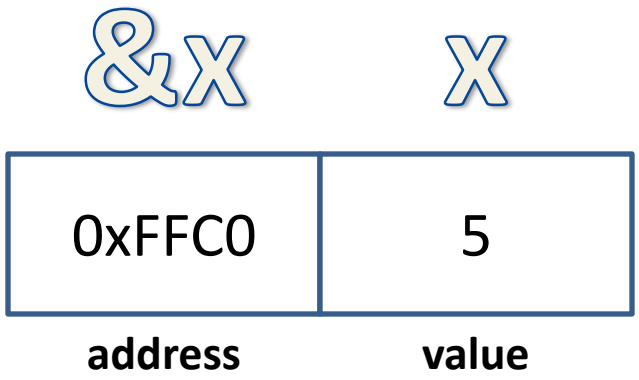
```
int x = 5;
int *ptr;
```



Memory Basics – Regular Variables

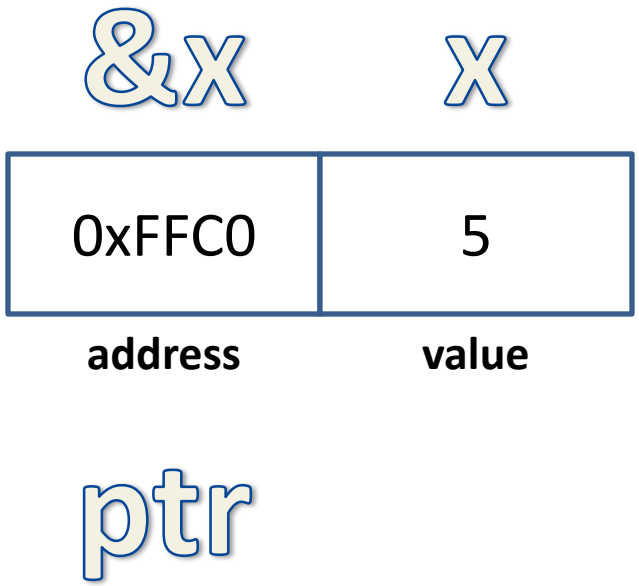
- and set it equal to the address of **x**:

```
int x = 5;
int *ptr;
ptr = &x;
```



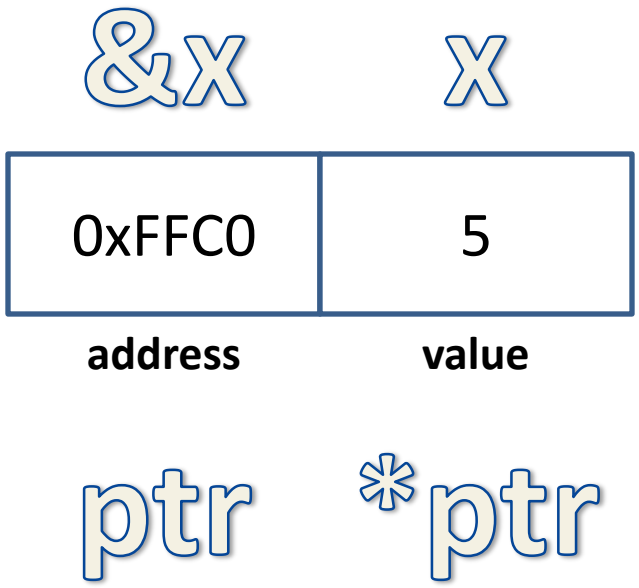
Memory Basics – Regular Variables

- `ptr = &x`



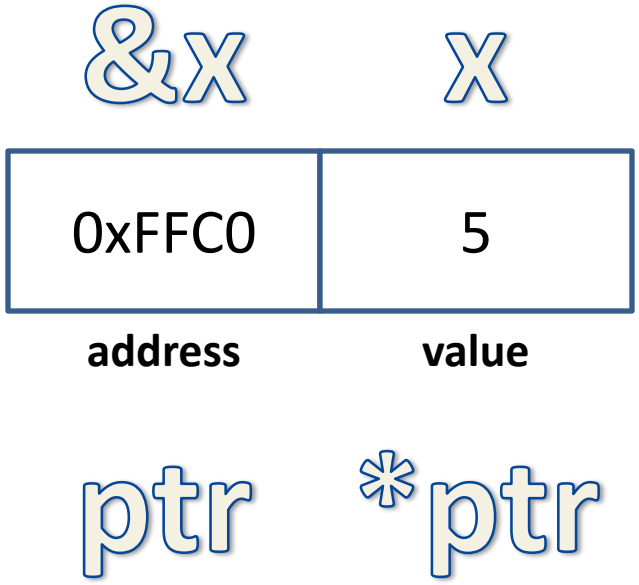
Memory Basics – Regular Variables

- `ptr = &x`
- `*ptr = x`



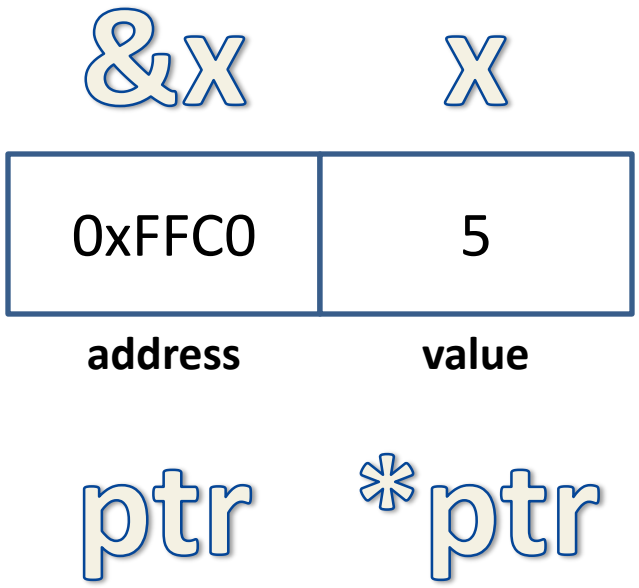
Memory Basics – Regular Variables

- **`ptr`** points to the address where **`x`** is stored
- **`*ptr`** gives us the value of **`x`**
  - (dereferencing ptr)



Memory Basics – Pointer Variables

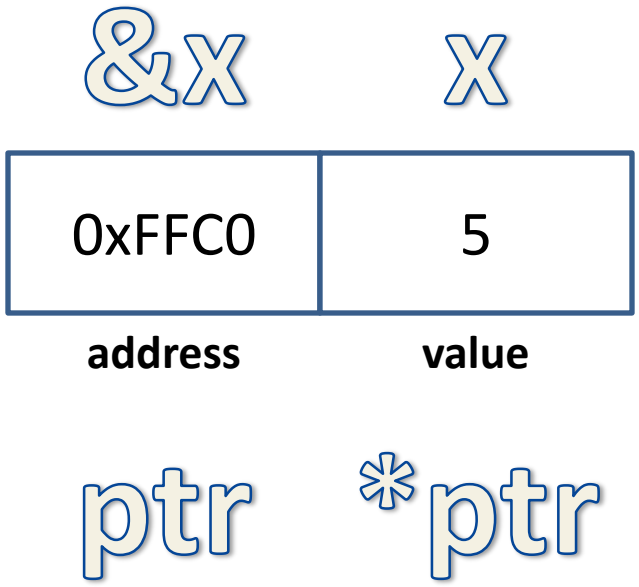
- but what about the variable **`ptr`**?
  - does it have a value and address too?



Memory Basics – Pointer Variables

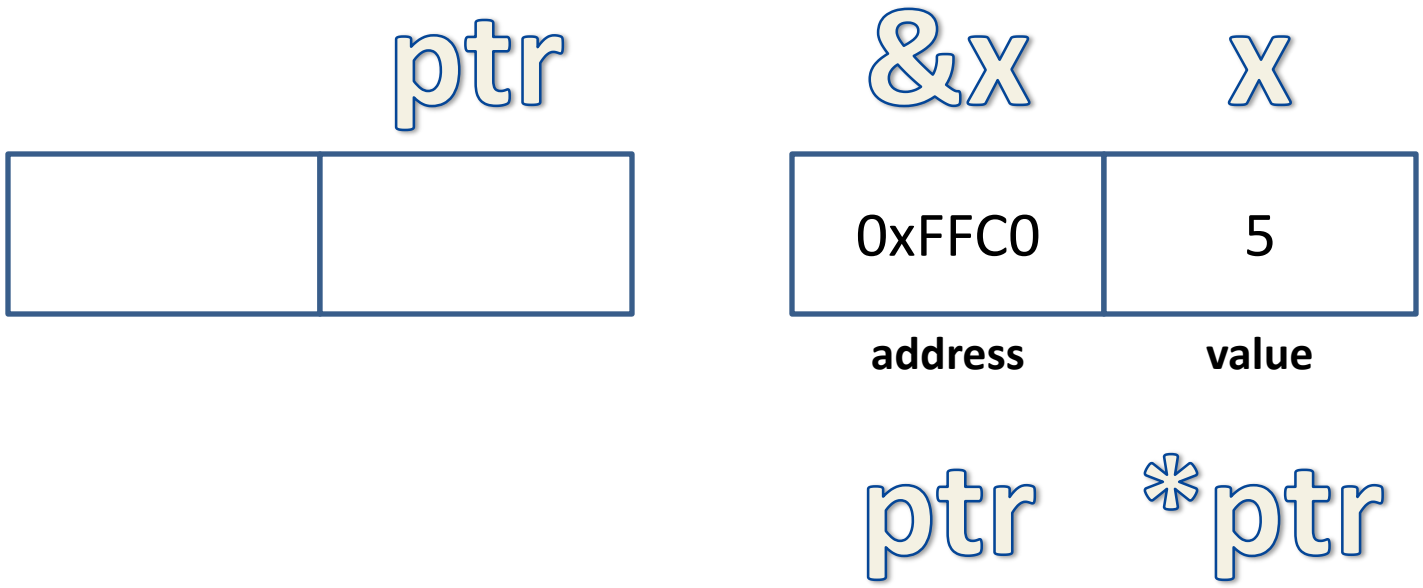
- but what about the variable **`ptr`**?
  - does it have a value and address too?

- YES!!!



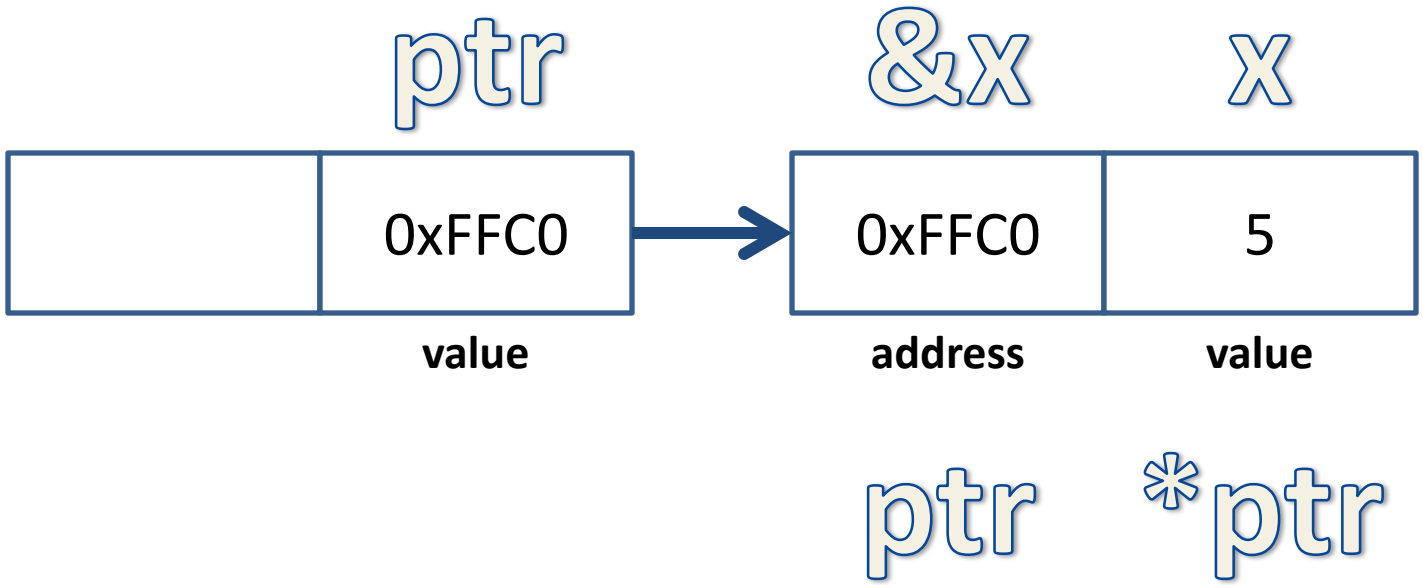
Memory Basics – Pointer Variables

- **`ptr`**'s value is just "**`ptr`**" – so it's 0xFFC0



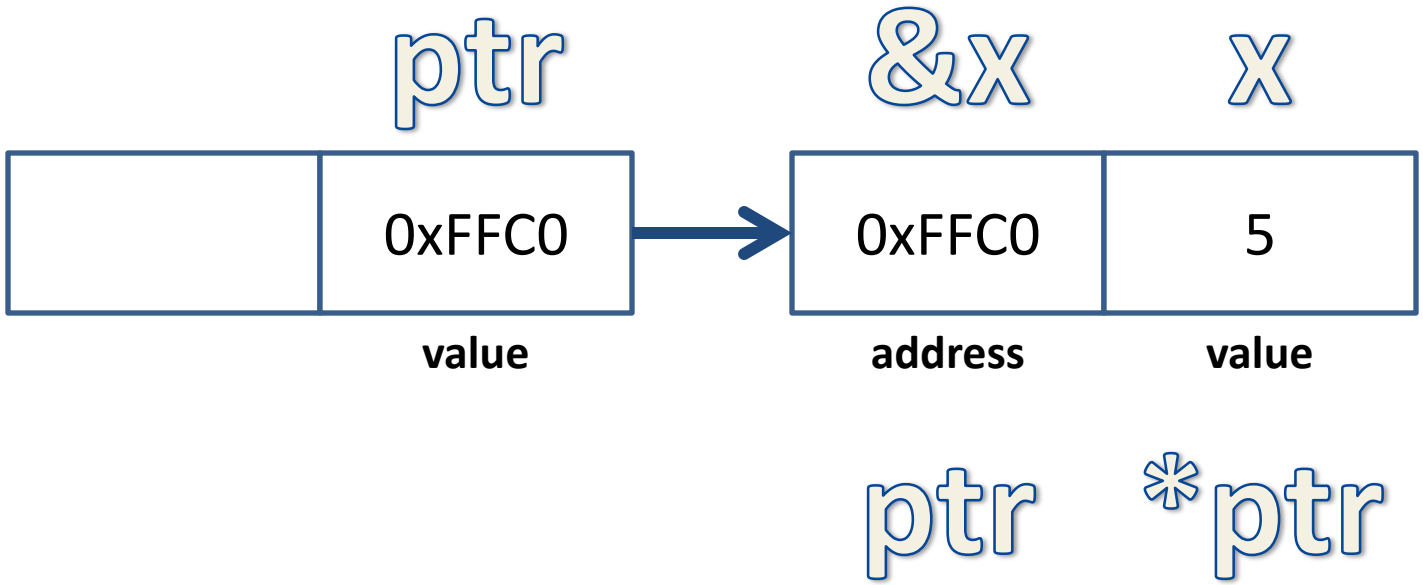
Memory Basics – Pointer Variables

- **`ptr`**'s value is just "**`ptr`**" – so it's 0xFFC0



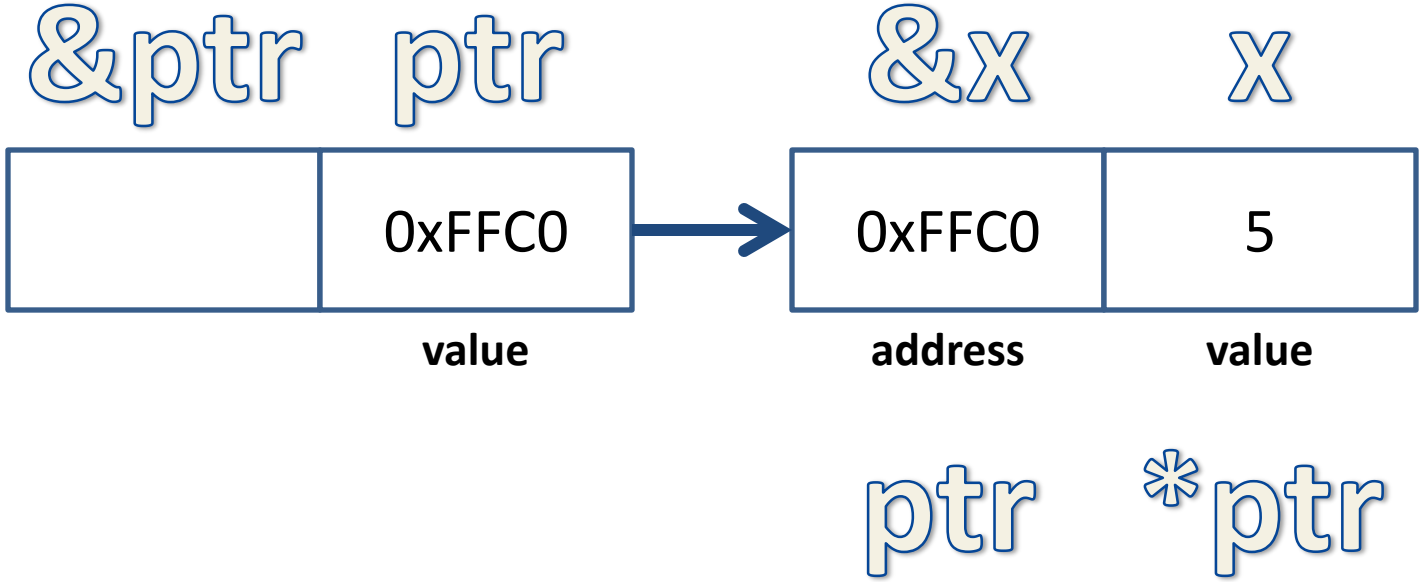
Memory Basics – Pointer Variables

- **`ptr`**'s value is just "**`ptr`**" – so it's 0xFFC0
- but what about its address?



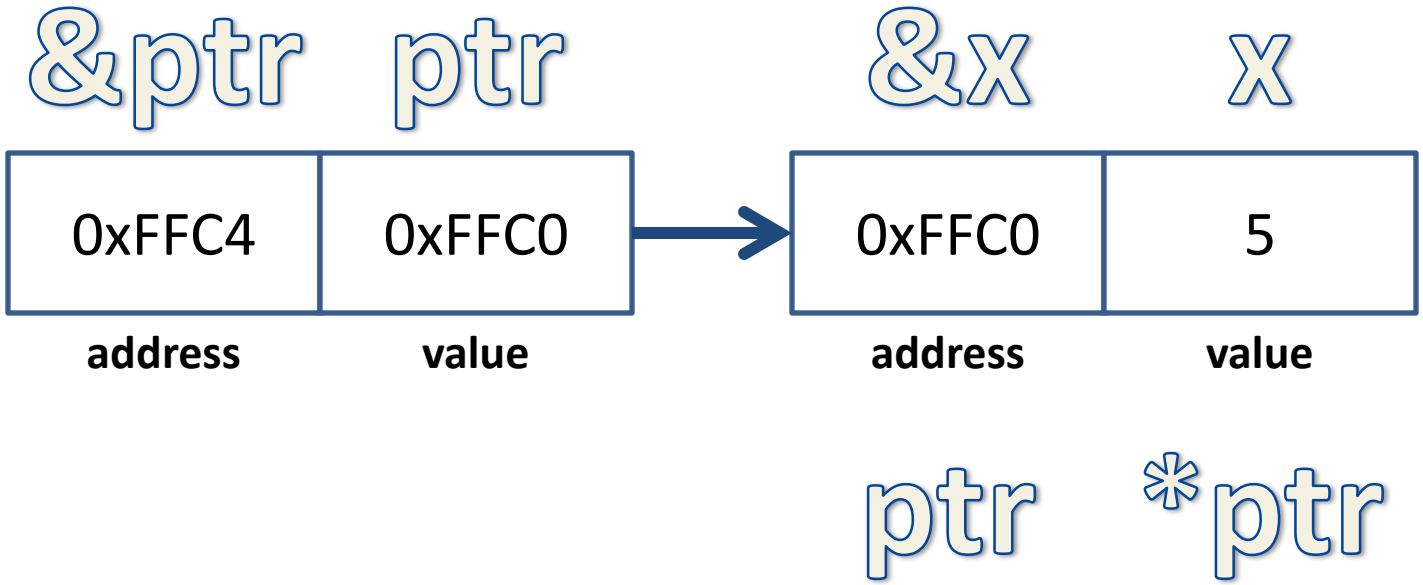
Memory Basics – Pointer Variables

- **`ptr`**'s value is just "**`ptr`**" – so it's 0xFFC0
- but what about its address?
  - its address is **`&ptr`**



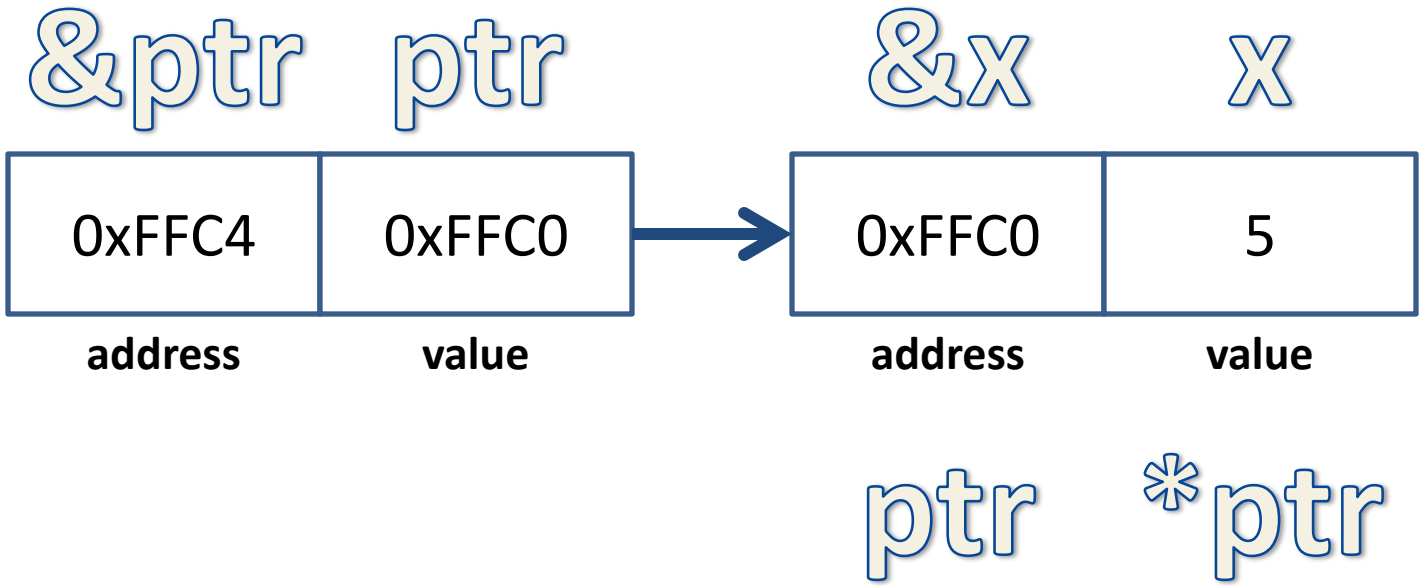
Memory Basics – Pointer Variables

- **`ptr`**'s value is just "**`ptr`**" – so it's 0xFFC0
- but what about its address?
  - its address is **`&ptr`**



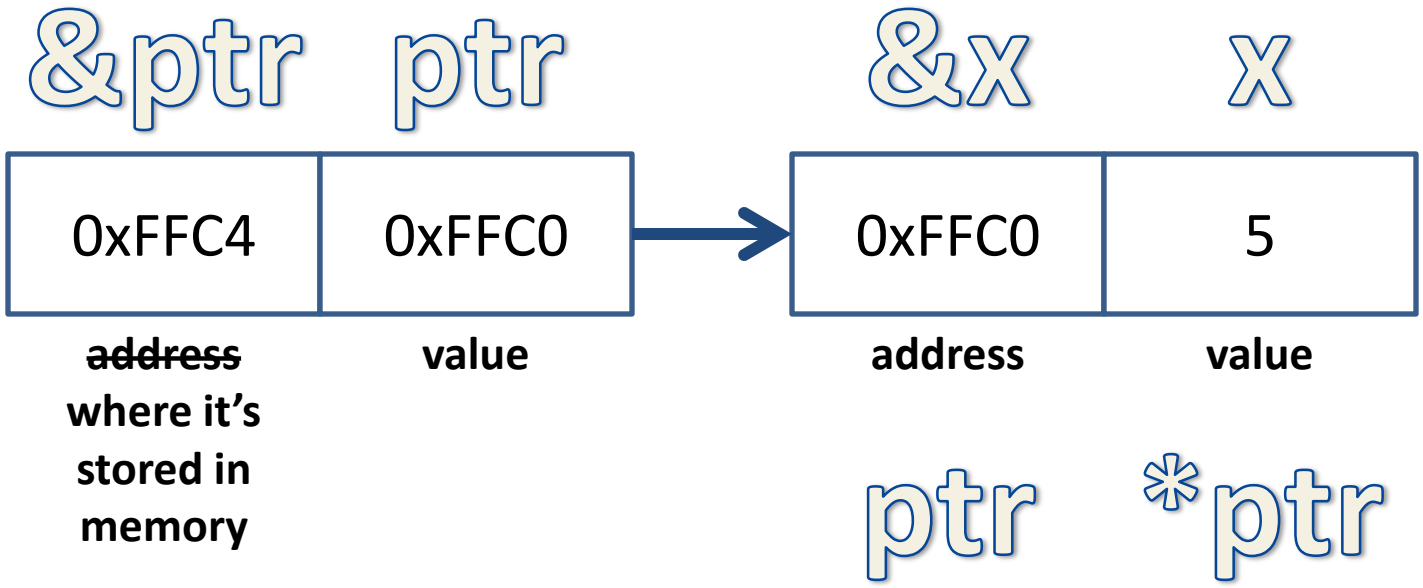
Memory Basics – Pointer Variables

- if you want, you can think of value and address for pointers as this instead…



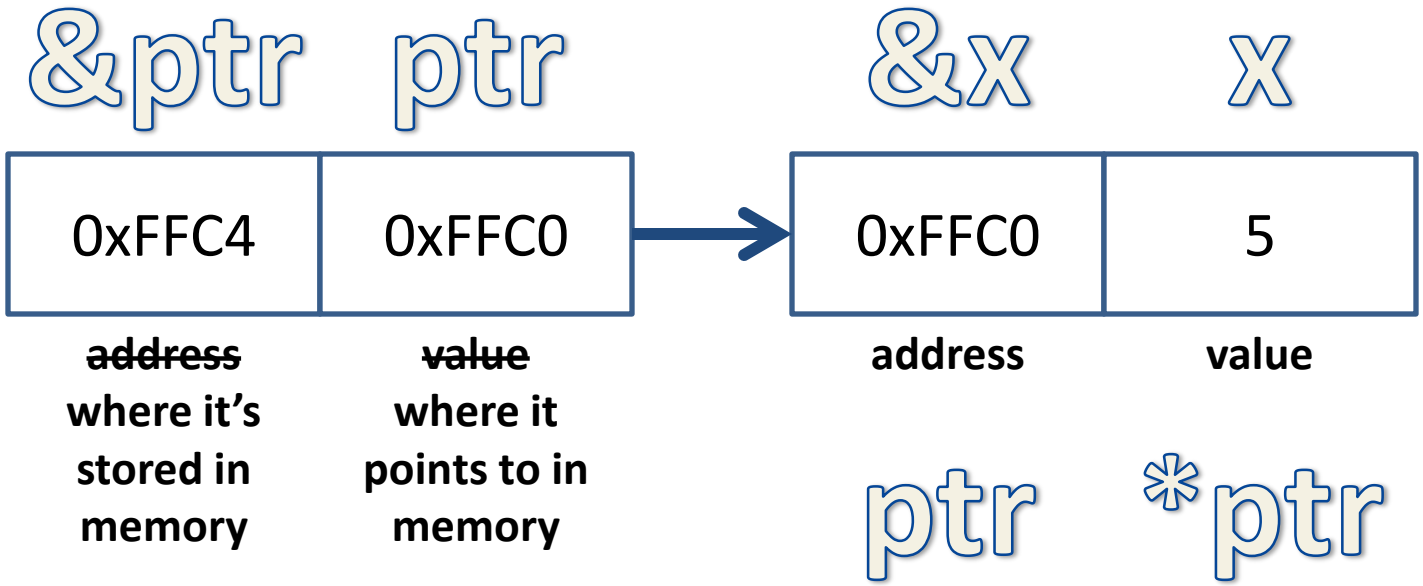
Memory Basics – Pointer Variables

- ~~address~~ where it's stored in memory



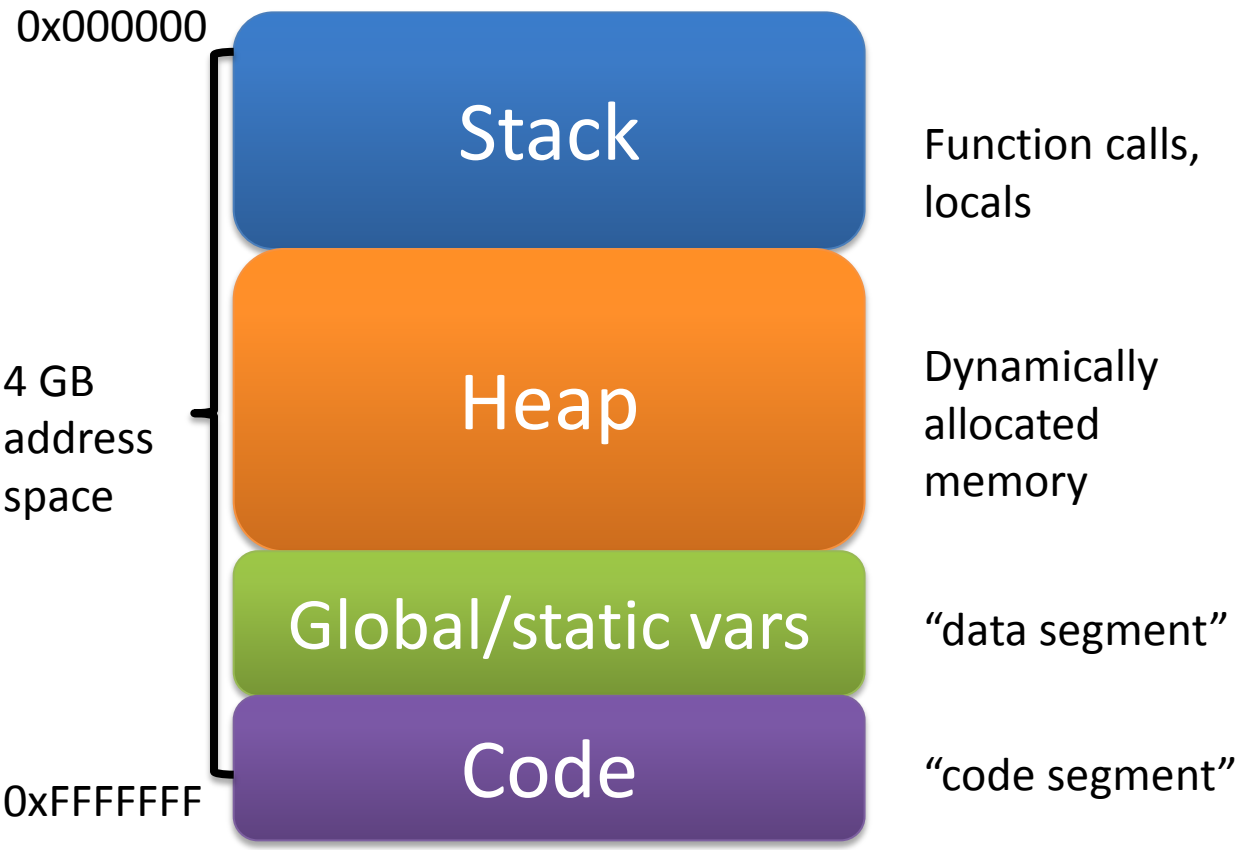
Memory Basics – Pointer Variables

- ~~address~~ where it's stored in memory
- ~~value~~ where it points to in memory



Memory Basics – "Owning" Memory

- each process gets its own memory chunk, or *address space*

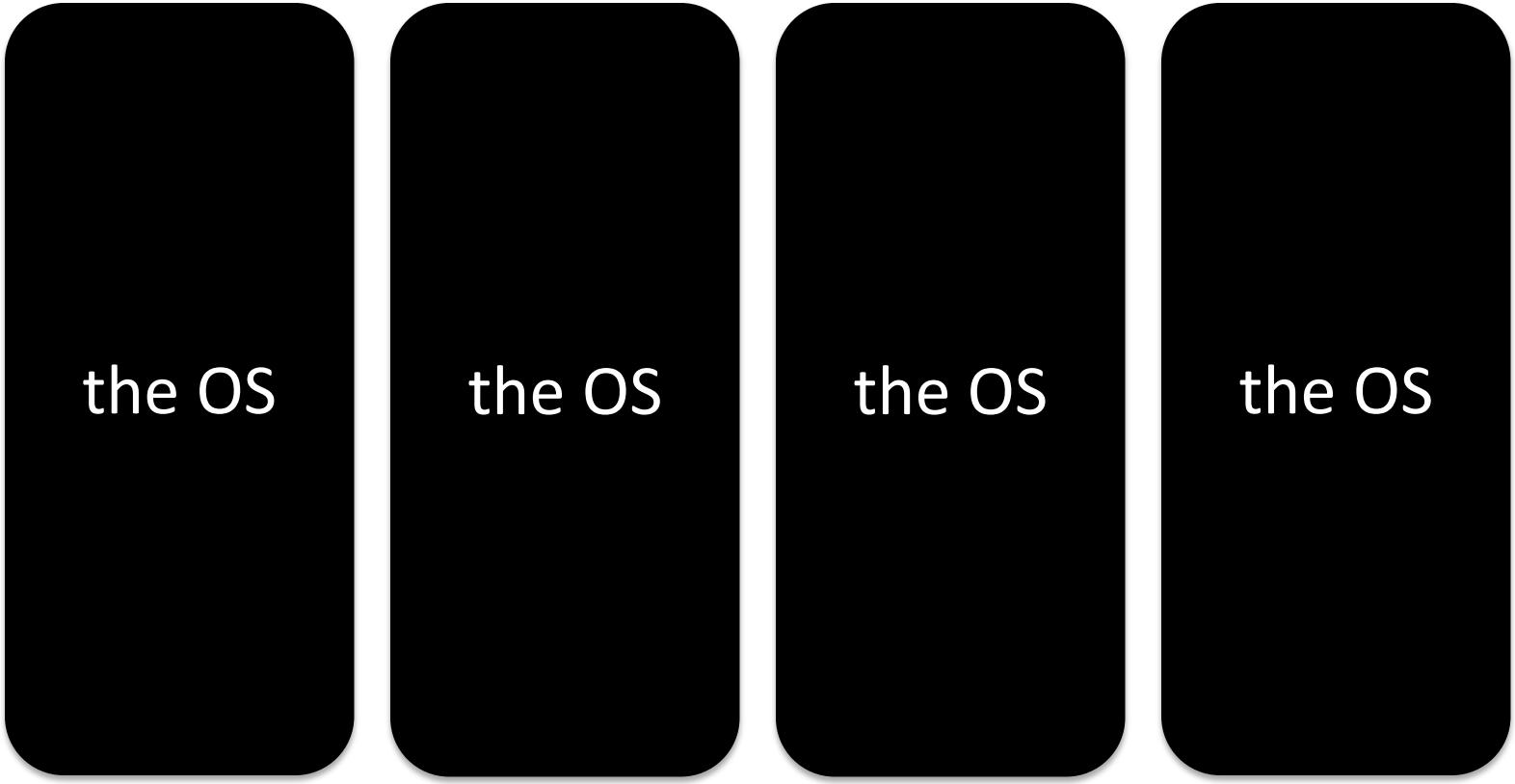


Memory Basics – “Owning” Memory

- you can think of memory as being “owned” by:
  - the OS
    - most of the memory the computer has
  - the process
    - a chunk of memory given by the OS – about 4 GB
  - the program
    - memory (on the stack) given to it by the process
  - you
    - when you dynamically allocate memory in the program (memory given to you by the process)

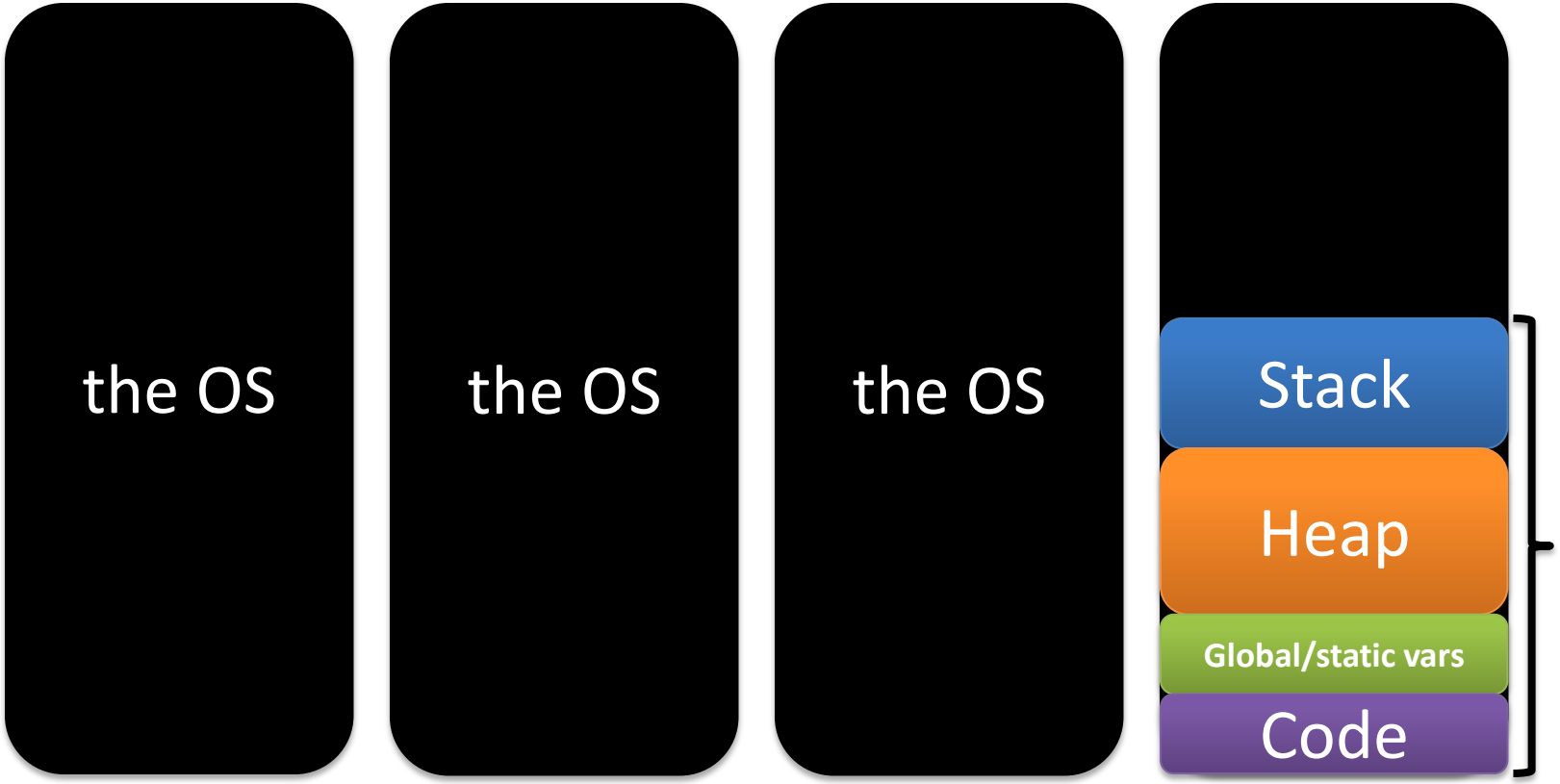
Memory Basics – “Owning” Memory

- the *Operating System* has a very large amount of memory available to it



Memory Basics – "Owning" Memory

- when ***the process*** begins, the Operating System gives it a chunk of that memory



Memory Basics – "Owning" Memory

- when ***the process*** begins, the Operating System gives it a chunk of that memory

Memory Basics – "Owning" Memory

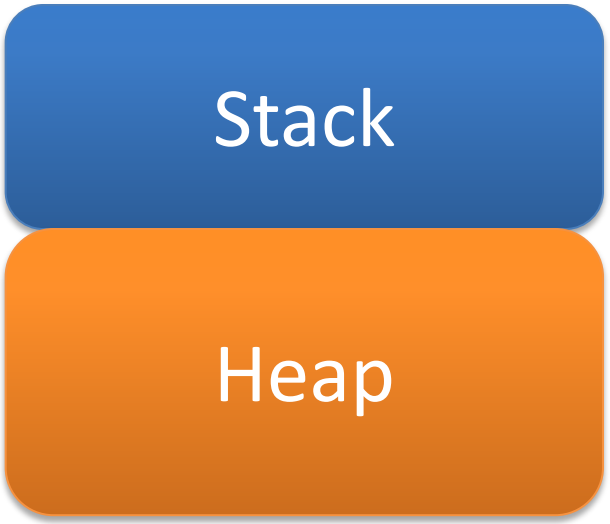
- when ***the process*** begins, the Operating System gives it a chunk of that memory

Memory Basics – "Owning" Memory

- within that chunk of memory, only the stack and the heap are available to ***you*** and ***the program***

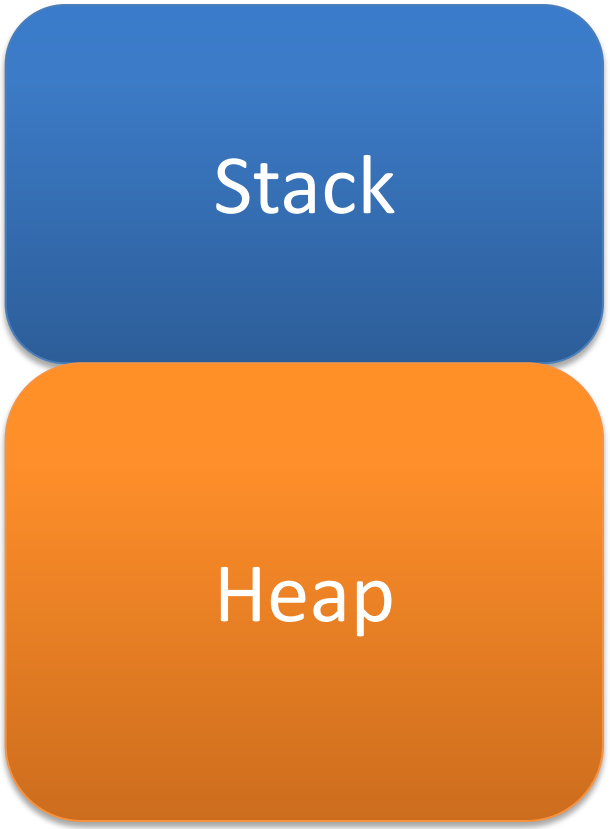
Memory Basics – “Owning” Memory

- within that chunk of memory, only the stack and the heap are available to ***you*** and ***the program***



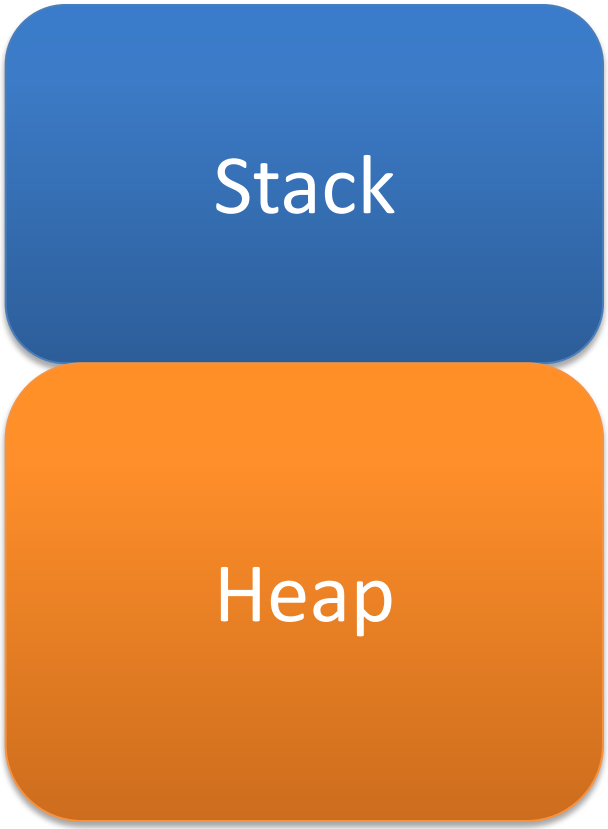
Memory Basics – "Owning" Memory

- within that chunk of memory, only the stack and the heap are available to ***you*** and ***the program***



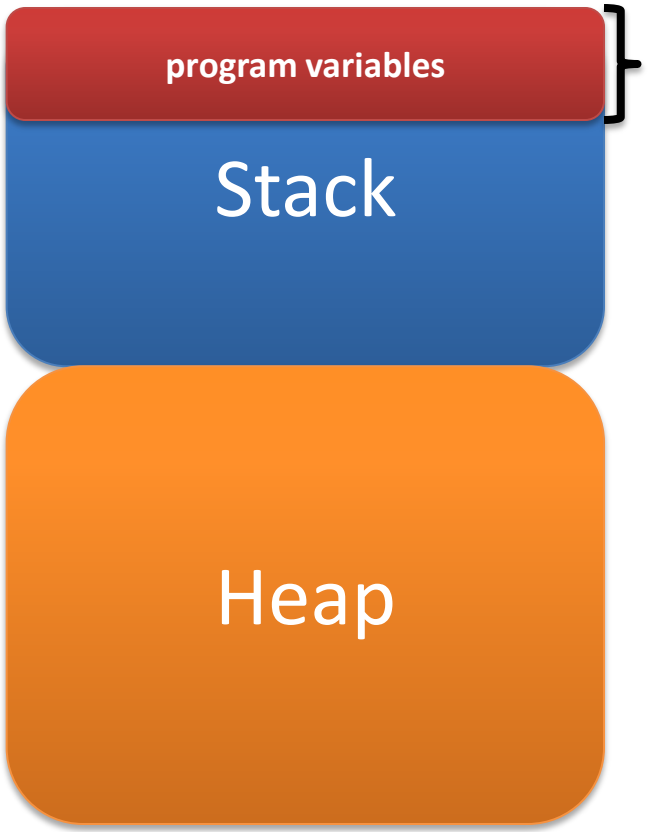
Memory Basics – “Owning” Memory

- some parts of the stack are given to ***the program*** for variables



Memory Basics – “Owning” Memory

- some parts of the stack are given to ***the program*** for variables



Memory Basics – "Owning" Memory

- and when a function is called, ***the program*** is given more space on the stack for the return address and in-function variables

Memory Basics – “Owning” Memory

- and every time ***you*** allocate memory, the process gives you space for it on the heap

Memory Basics – "Owning" Memory

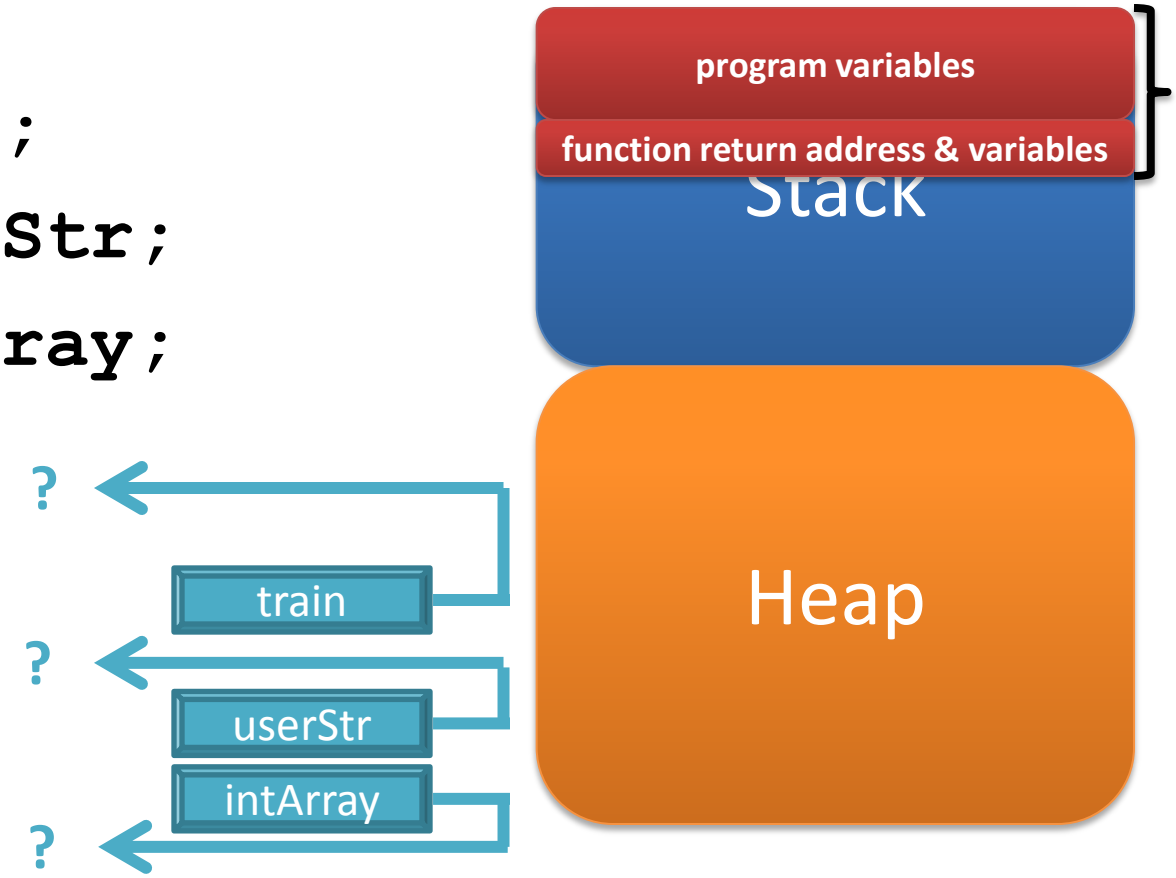
- and every time ***you*** allocate memory, the process gives you space for it on the heap

```
CAR* train;
char* userStr;
int* intArray;
```

Memory Basics – "Owning" Memory

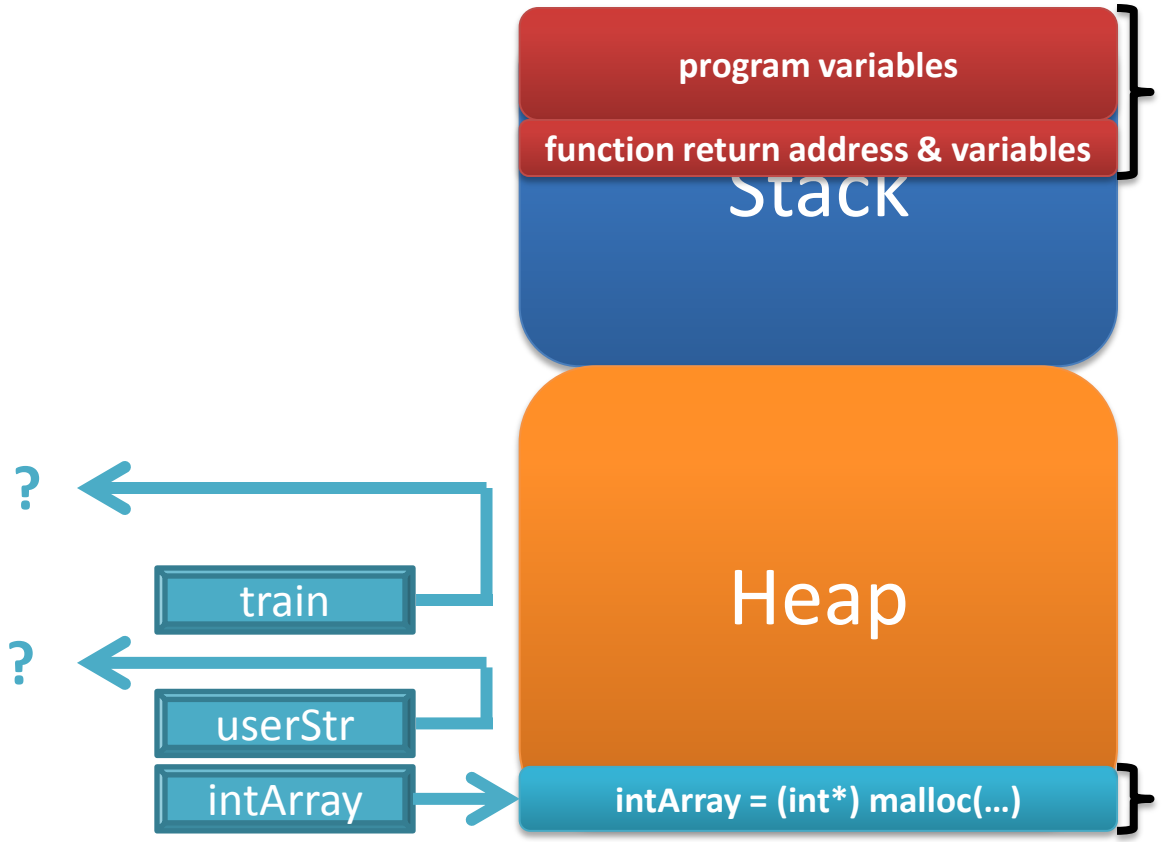
- and every time ***you*** allocate memory, the process gives you space for it on the heap

```
CAR* train;
char* userStr;
int* intArray;
```



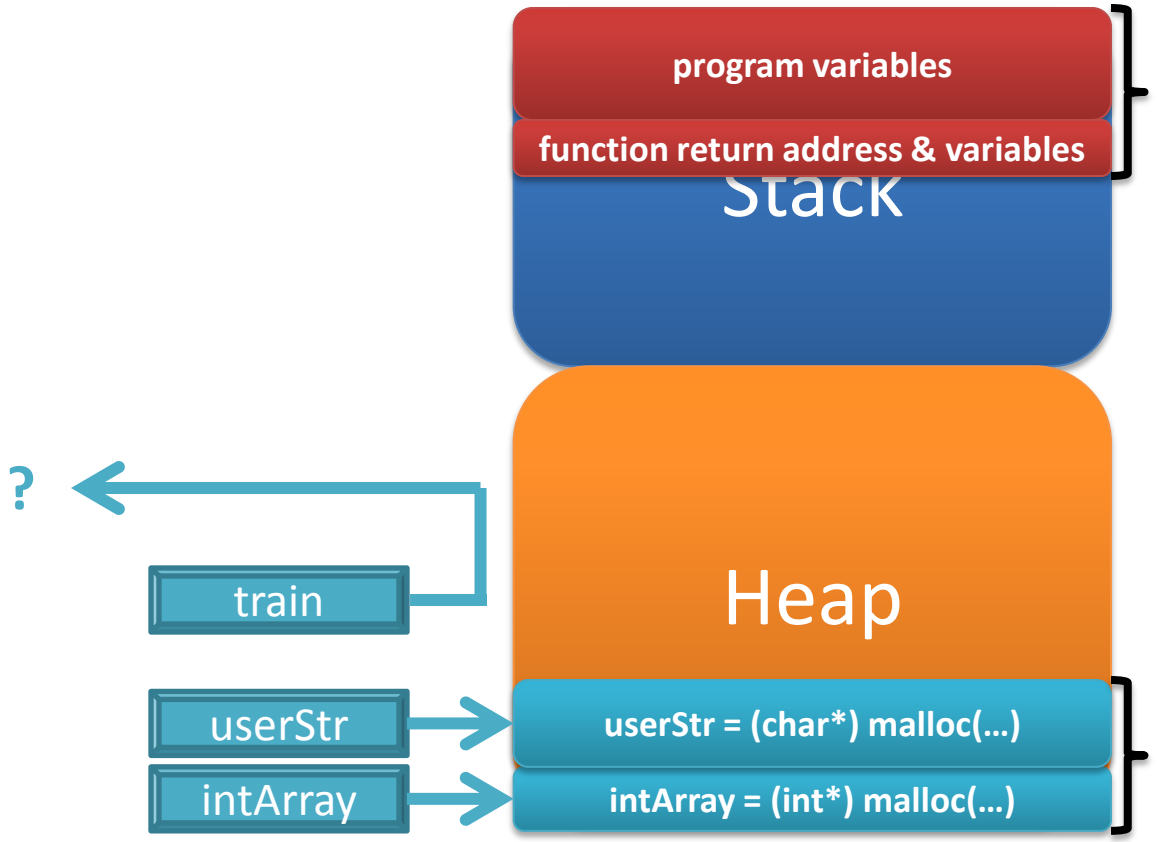
Memory Basics – "Owning" Memory

- and every time ***you*** allocate memory, the process gives you space for it on the heap



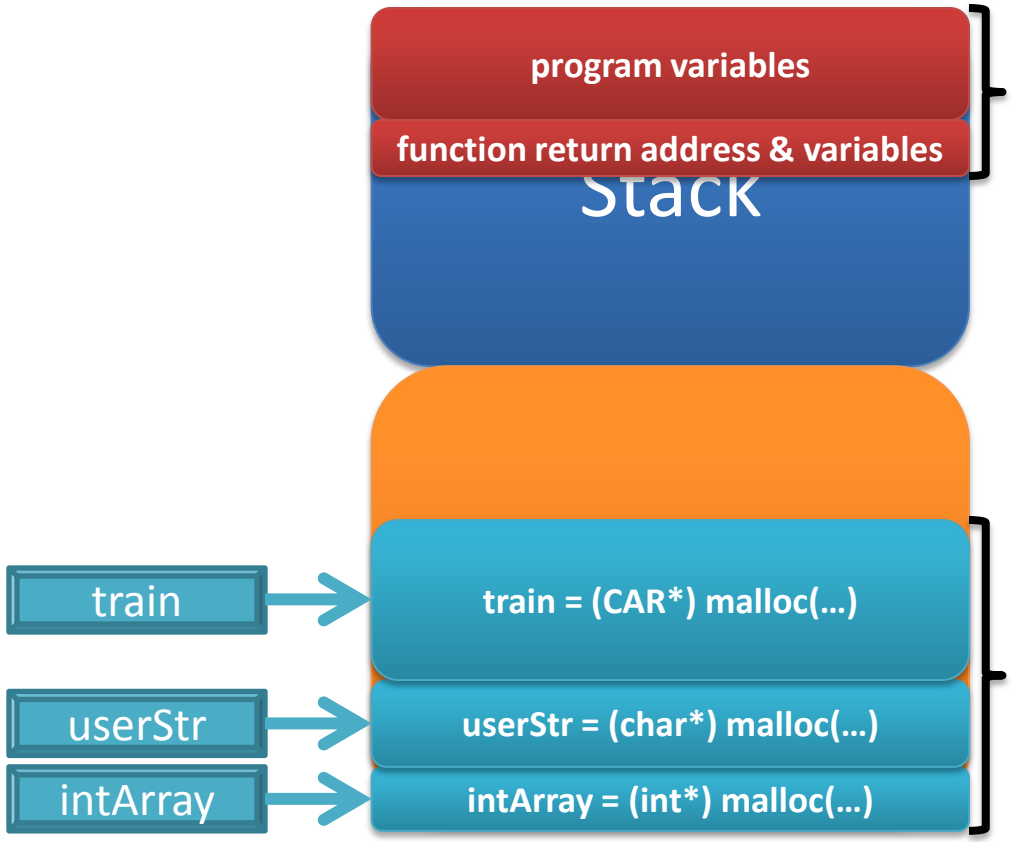
Memory Basics – "Owning" Memory

- and every time ***you*** allocate memory, the process gives you space for it on the heap



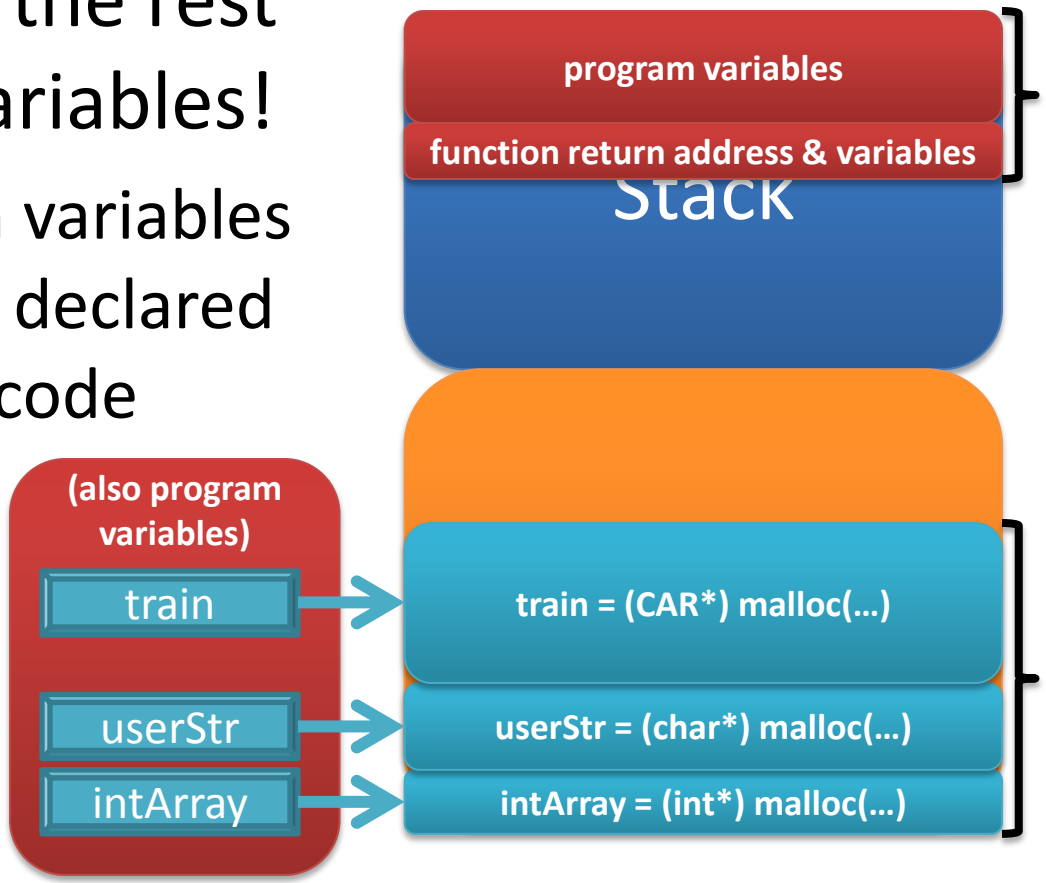
Memory Basics – "Owning" Memory

- and every time ***you*** allocate memory, the process gives you space for it on the heap



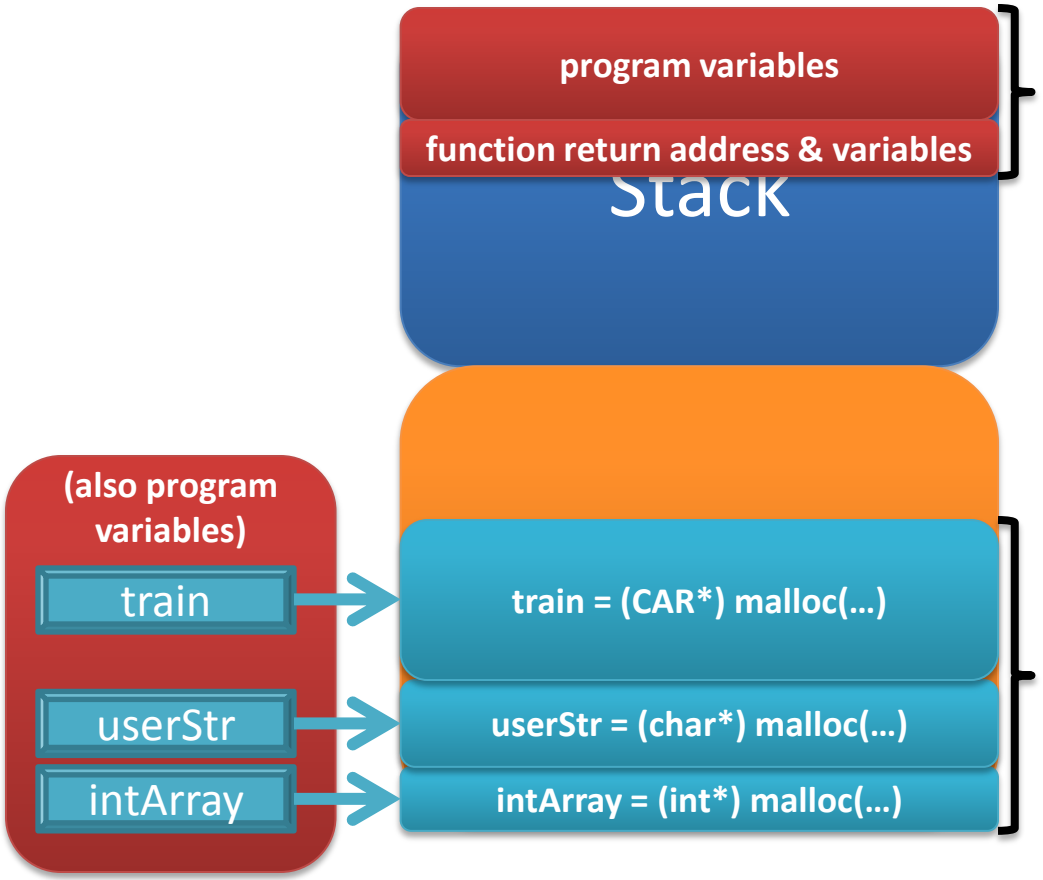
Memory Basics – "Owning" Memory

- don't forget – those pointers are ***program*** variables, so where they are stored is actually on the stack with the rest of the program variables!
  - they are program variables because they are declared in the program's code



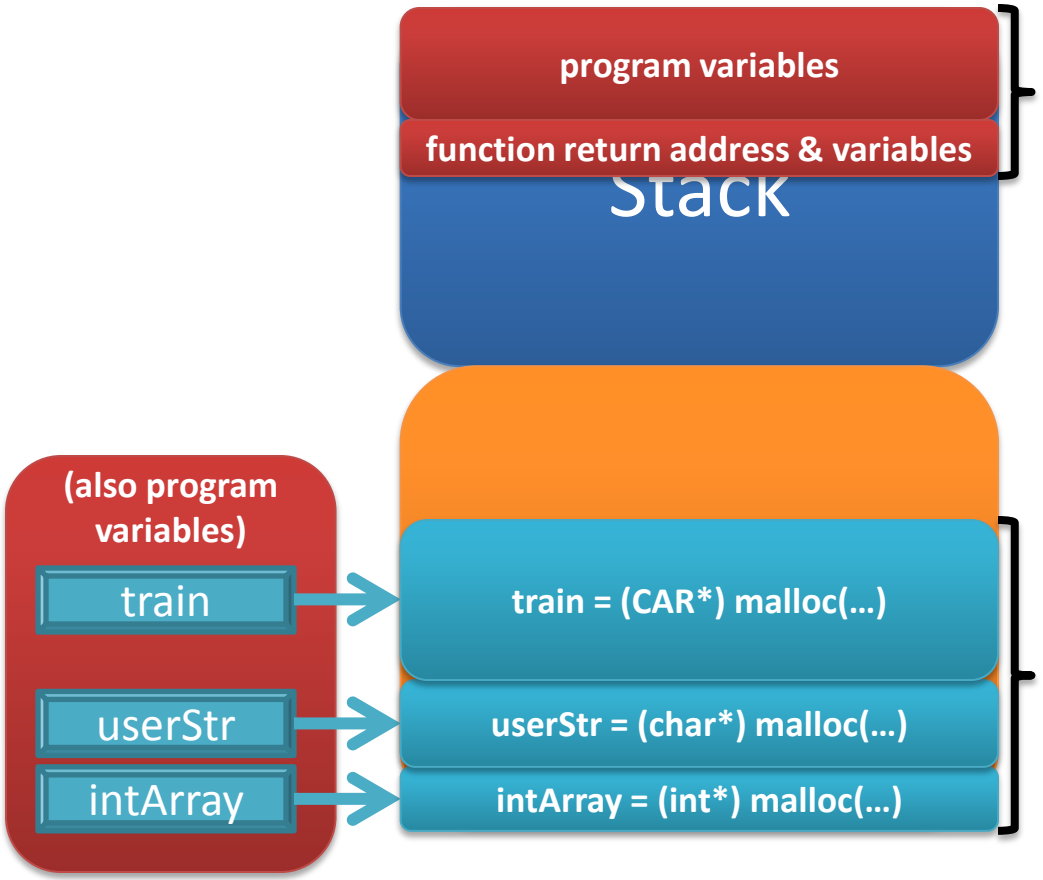
Memory Basics – "Returning" Memory

- but how does *the process* get any of that memory back?



Memory Basics – "Returning" Memory

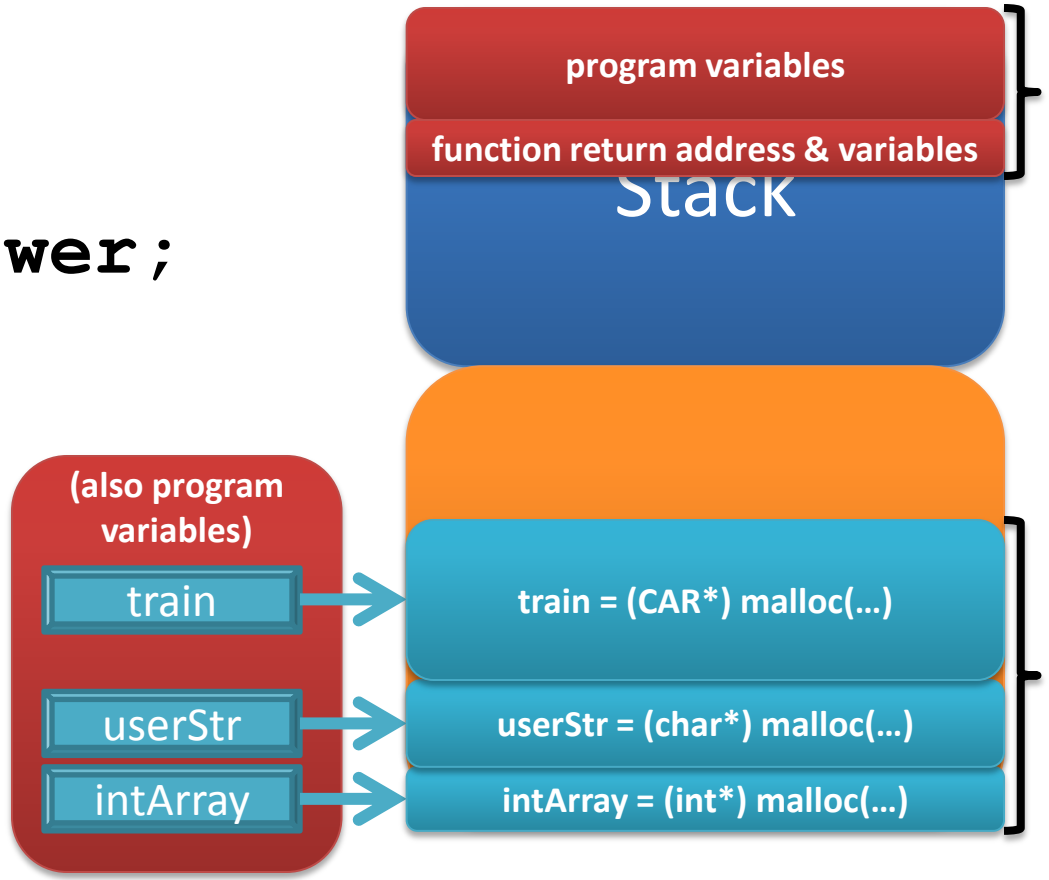
- when a function returns, the program gives that memory on the stack **back** to *the process*



Memory Basics – "Returning" Memory

- when a function returns, the program gives that memory on the stack **back** to *the process*

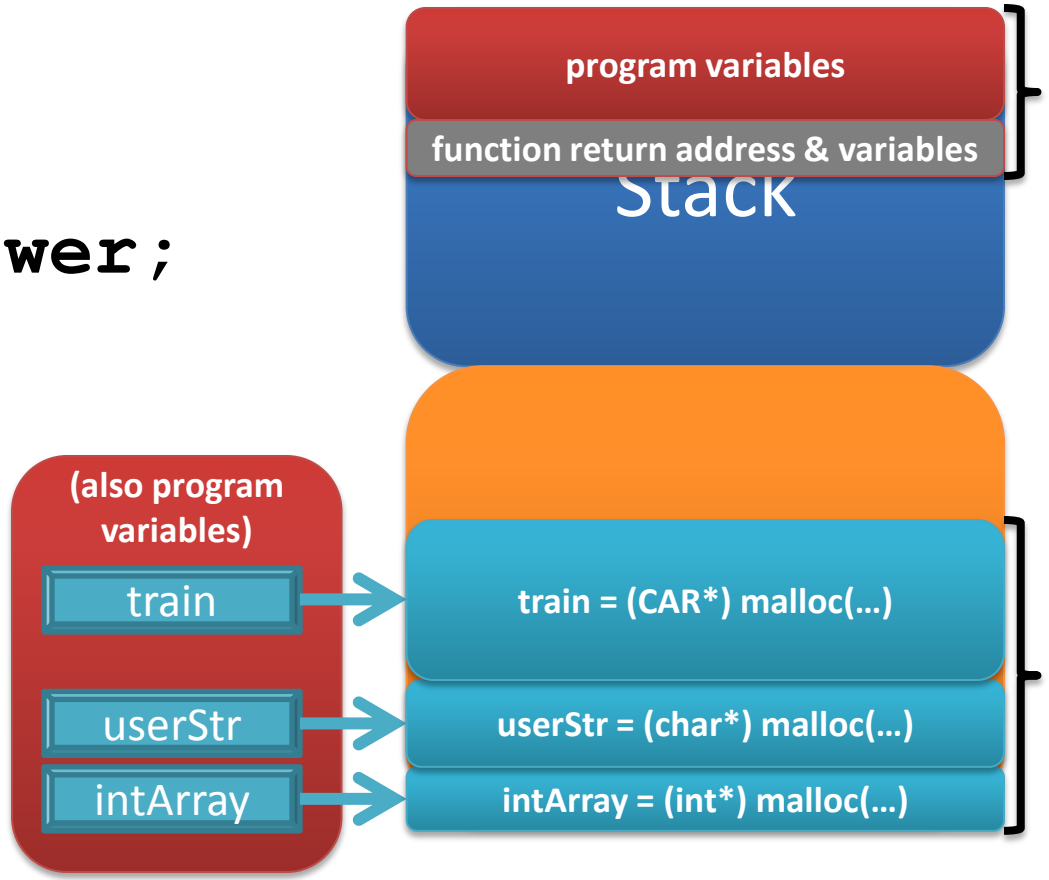
```
return fxnAnswer;
```



Memory Basics – "Returning" Memory

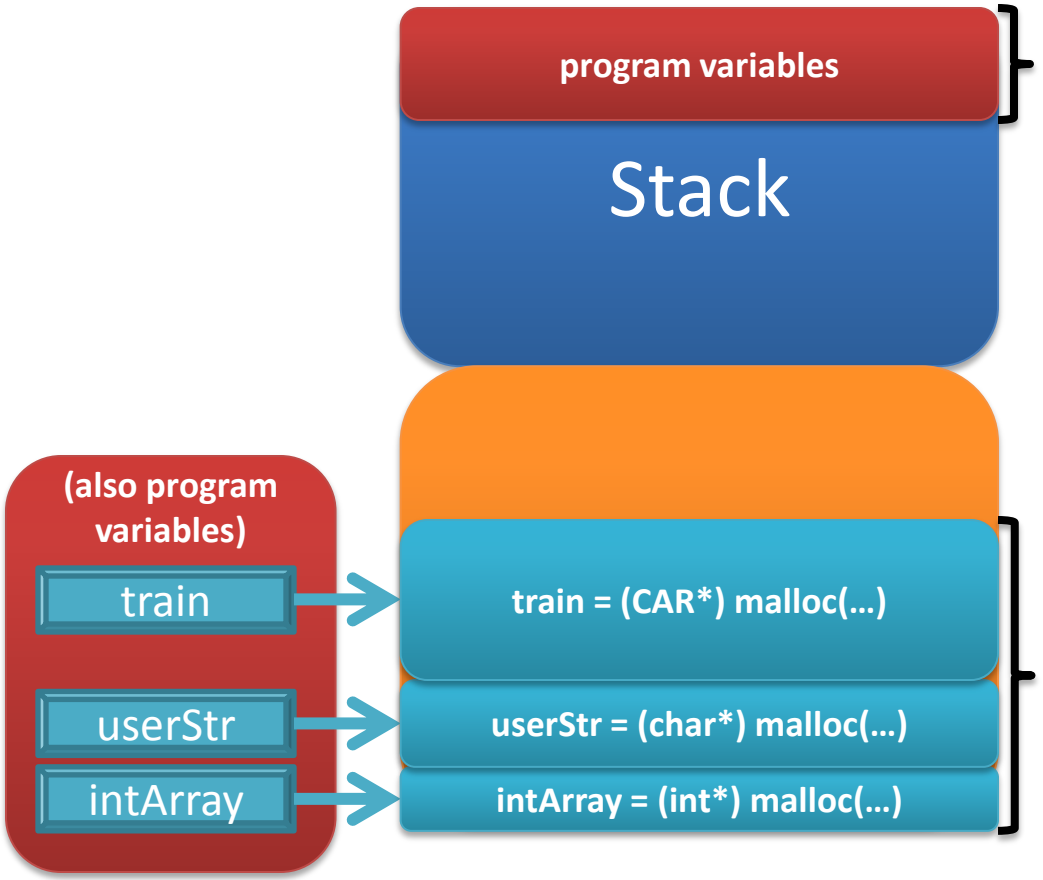
- when a function returns, the program gives that memory on the stack **back** to *the process*

```
return fxnAnswer;
```



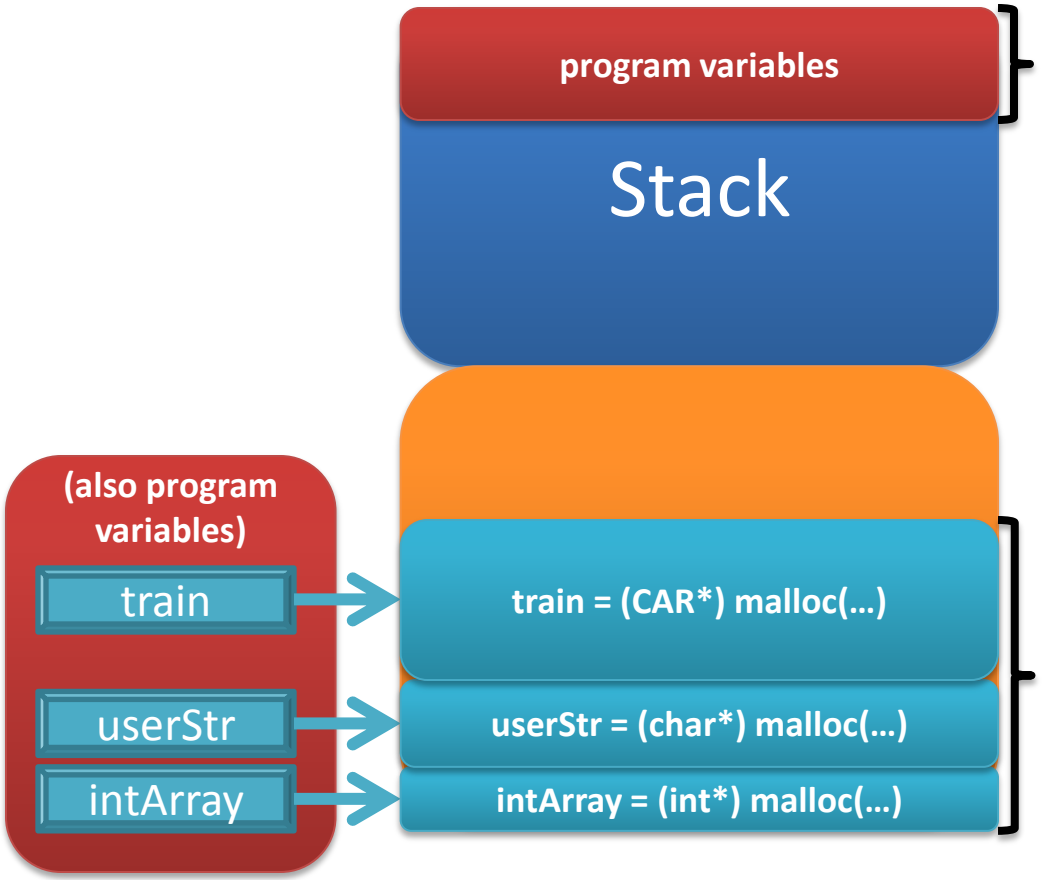
Memory Basics – "Returning" Memory

- when a function returns, the program gives that memory on the stack **back** to *the process*



Memory Basics – "Returning" Memory

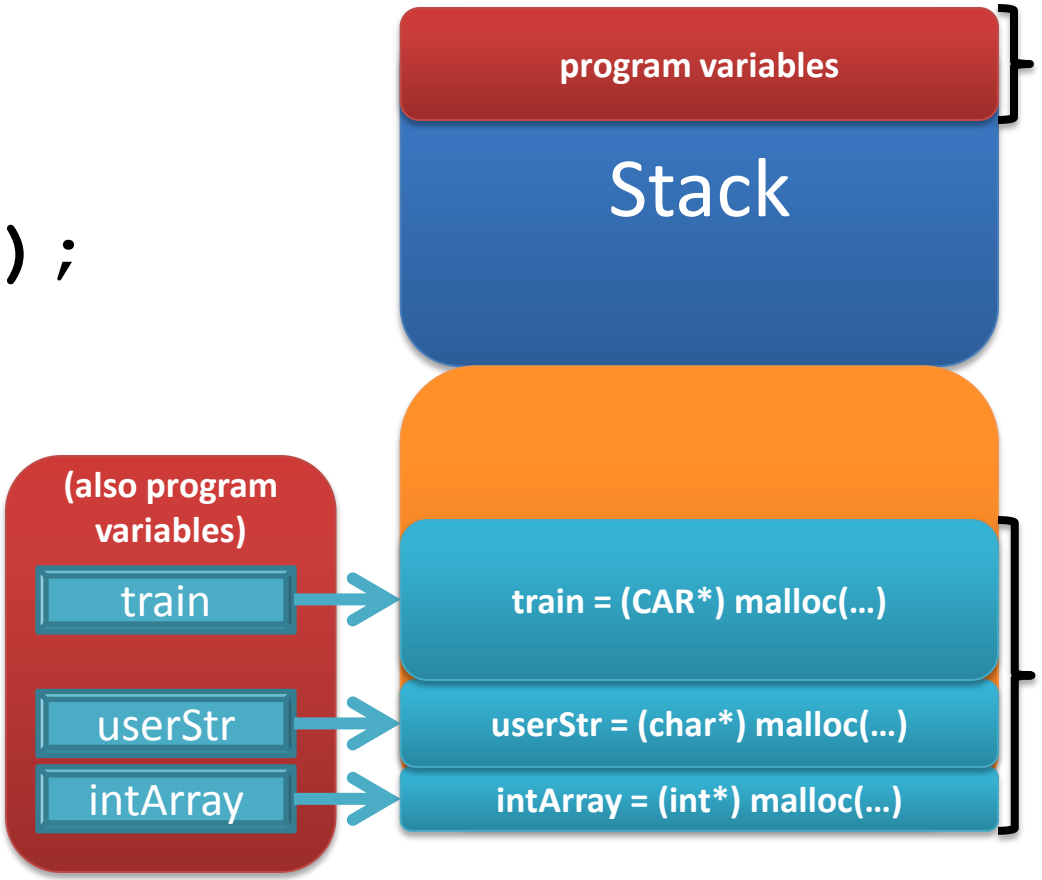
- and when you use free(), the memory you had on the heap is given **back** to *the process*



Memory Basics – "Returning" Memory

- and when you use free(), the memory you had on the heap is given **back** to *the process*

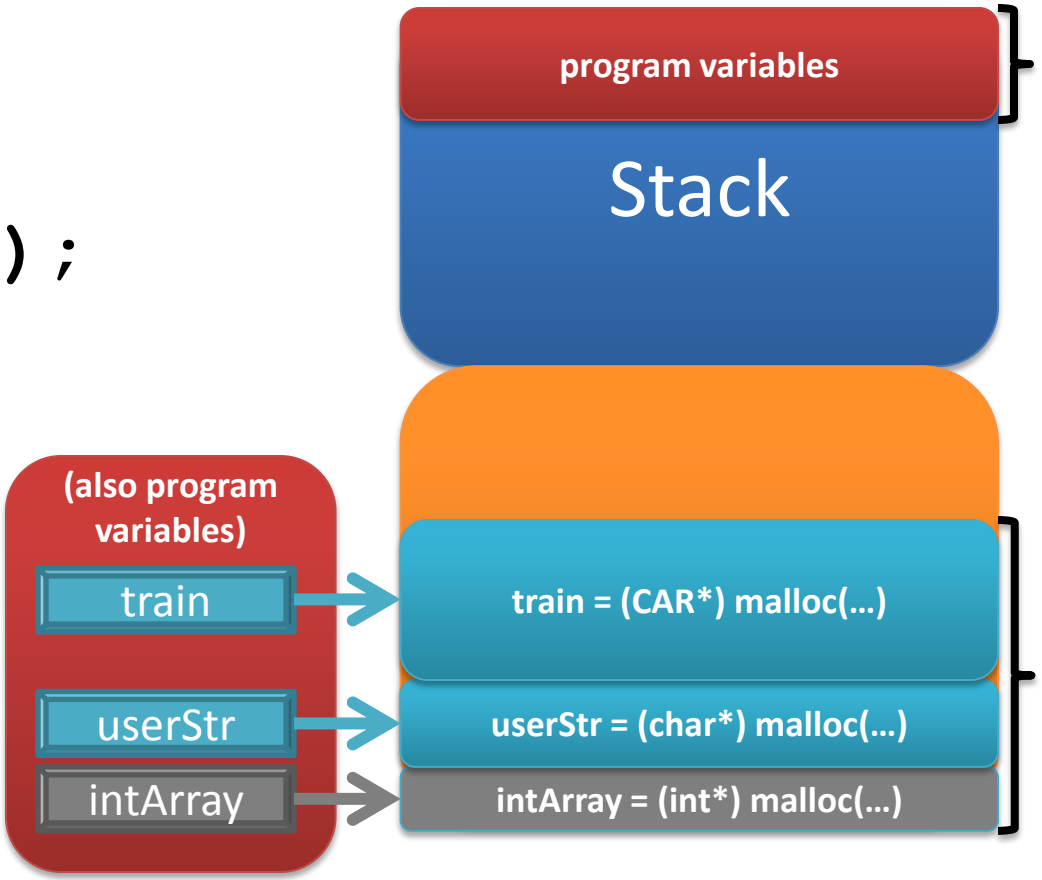
```
free(intArray);
```



Memory Basics – "Returning" Memory

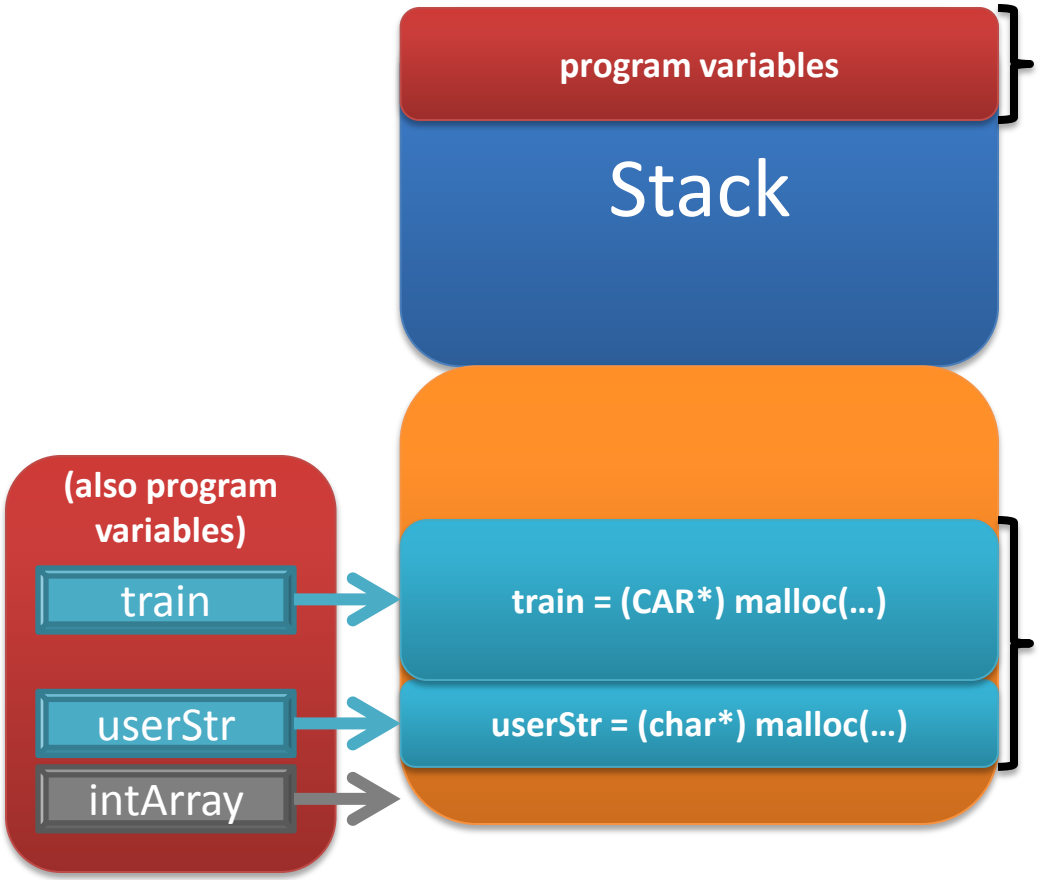
- and when you use free(), the memory you had on the heap is given **back** to *the process*

```
free(intArray);
```



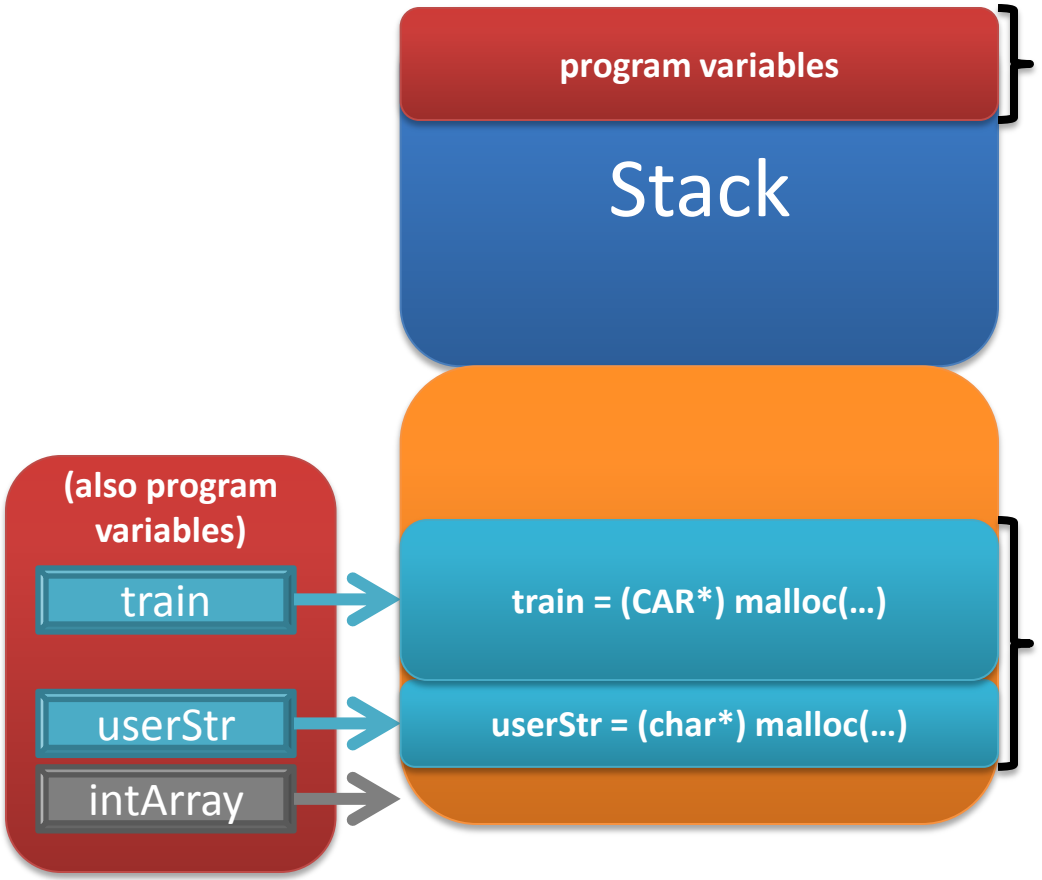
Memory Basics – "Returning" Memory

- and when you use free(), the memory you had on the heap is given **back** to *the process*



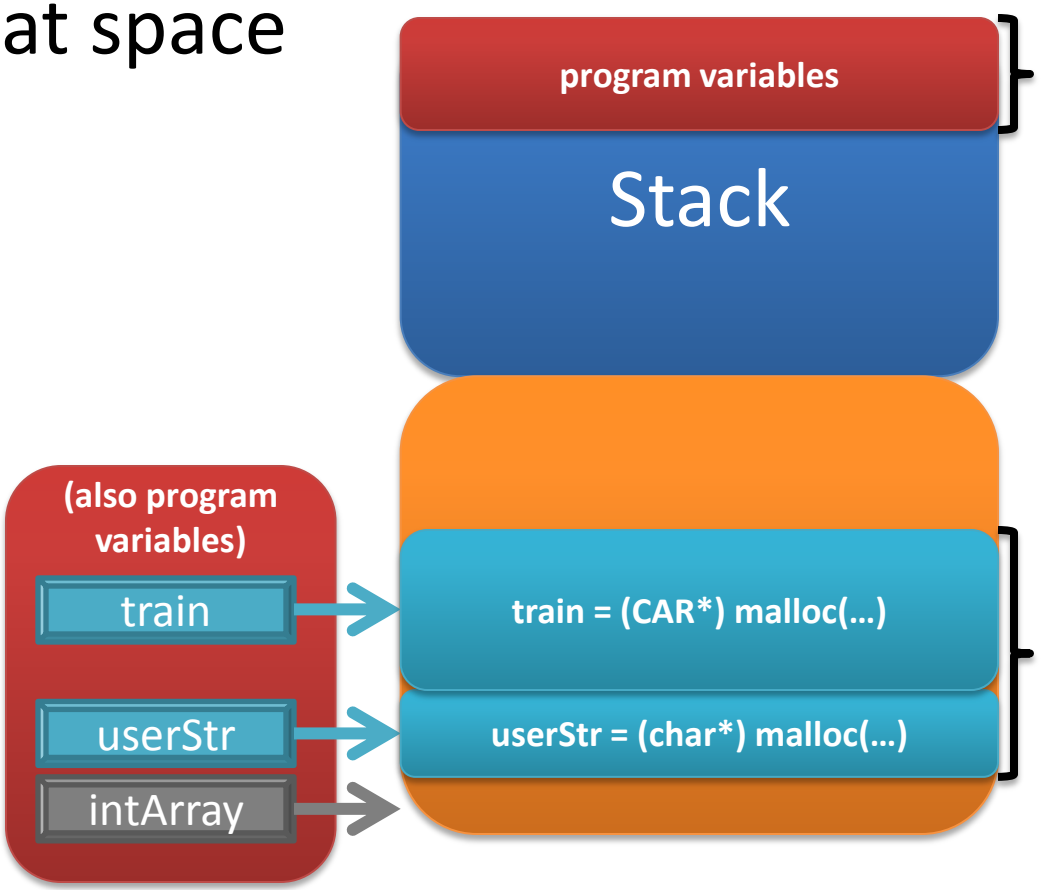
Memory Basics – Memory Errors

- but simply using free() doesn't change anything about the intArray variable



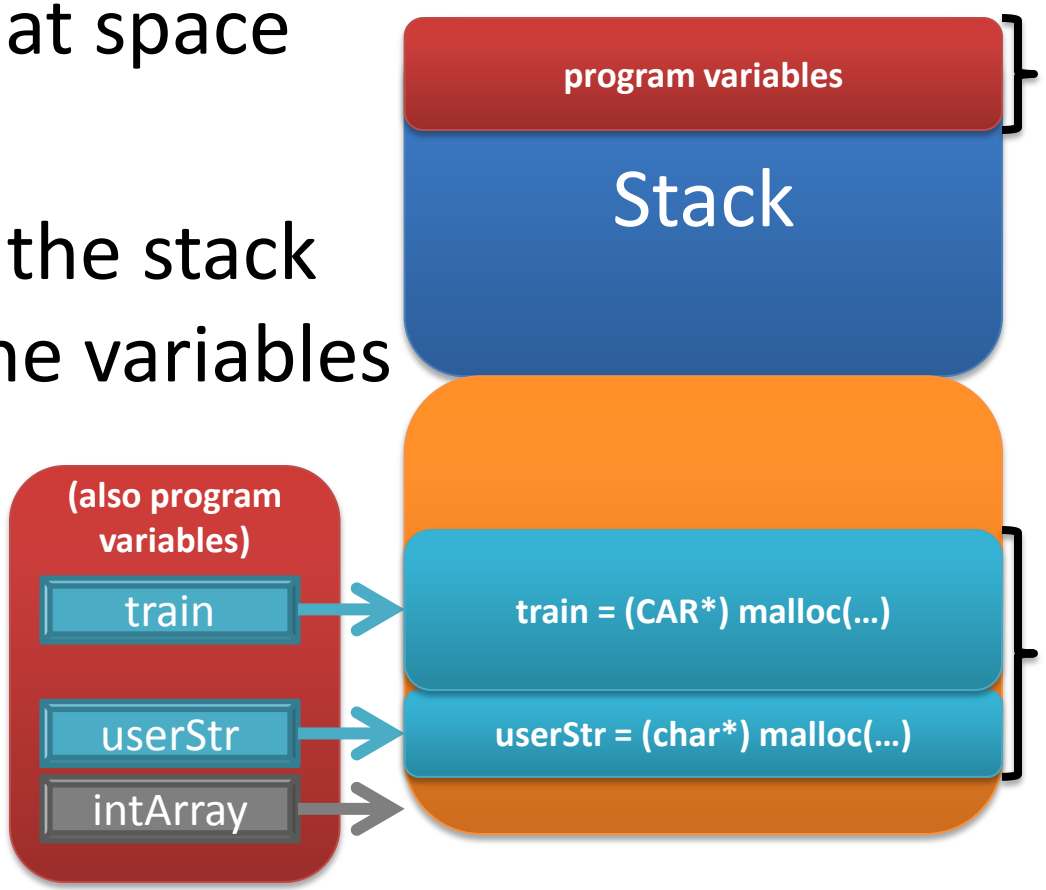
Memory Basics – Memory Errors

- but simply using free() doesn't change anything about the intArray variable
- it still points to that space in memory



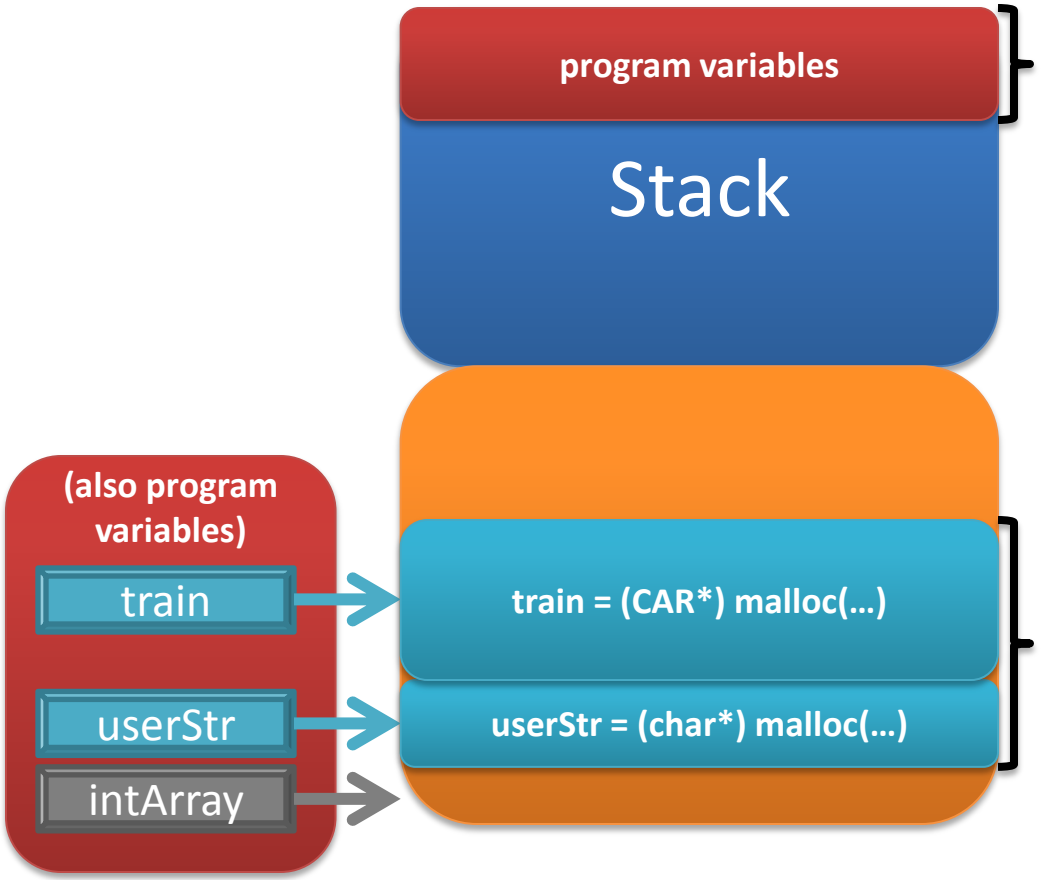
Memory Basics – Memory Errors

- but simply using free() doesn't change anything about the intArray variable
- it still points to that space in memory
- it's still stored on the stack with the rest of the variables



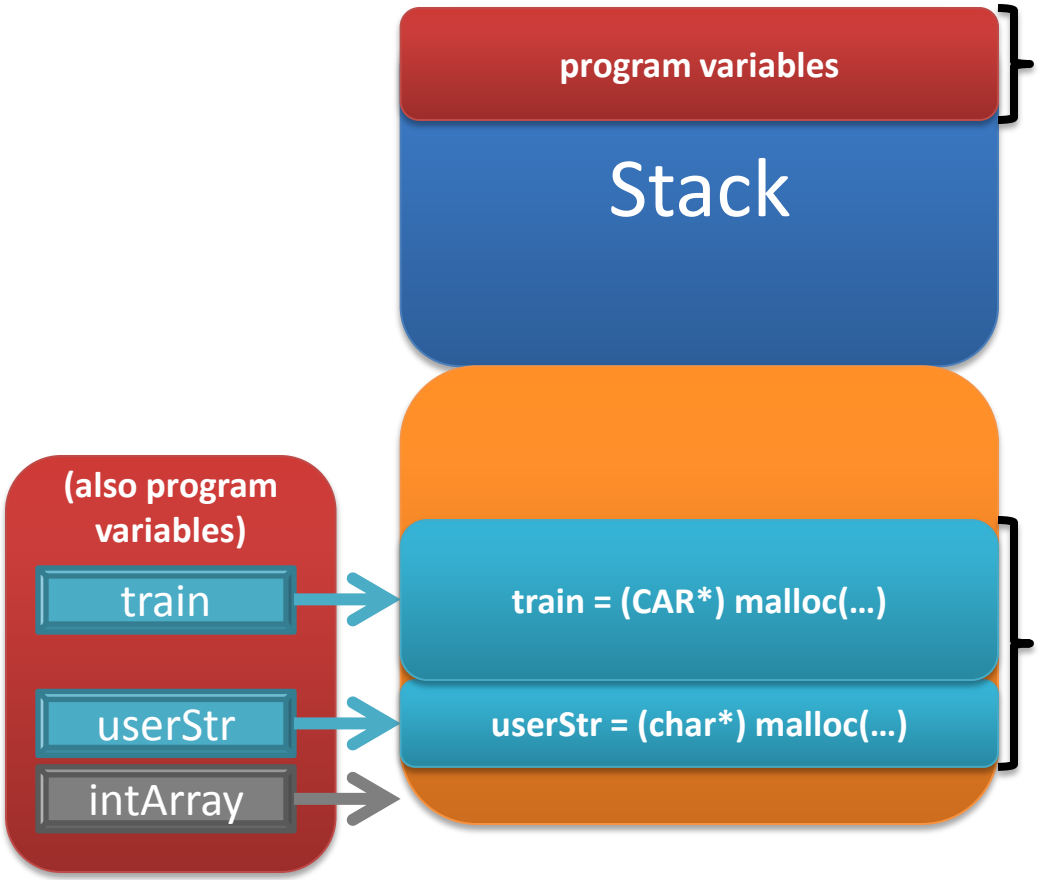
Memory Basics – Memory Errors

- intArray is now a **dangling pointer**



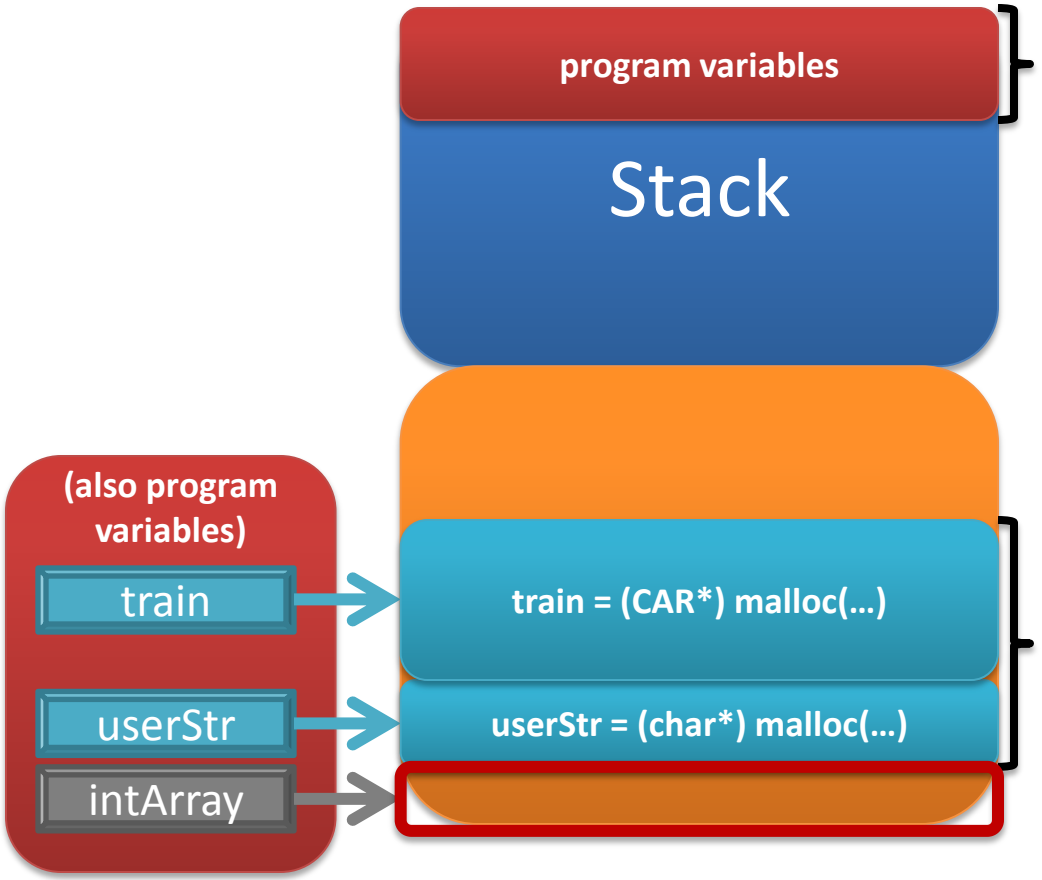
Memory Basics – Memory Errors

- intArray is now a **dangling pointer**
  - points to memory that has been freed



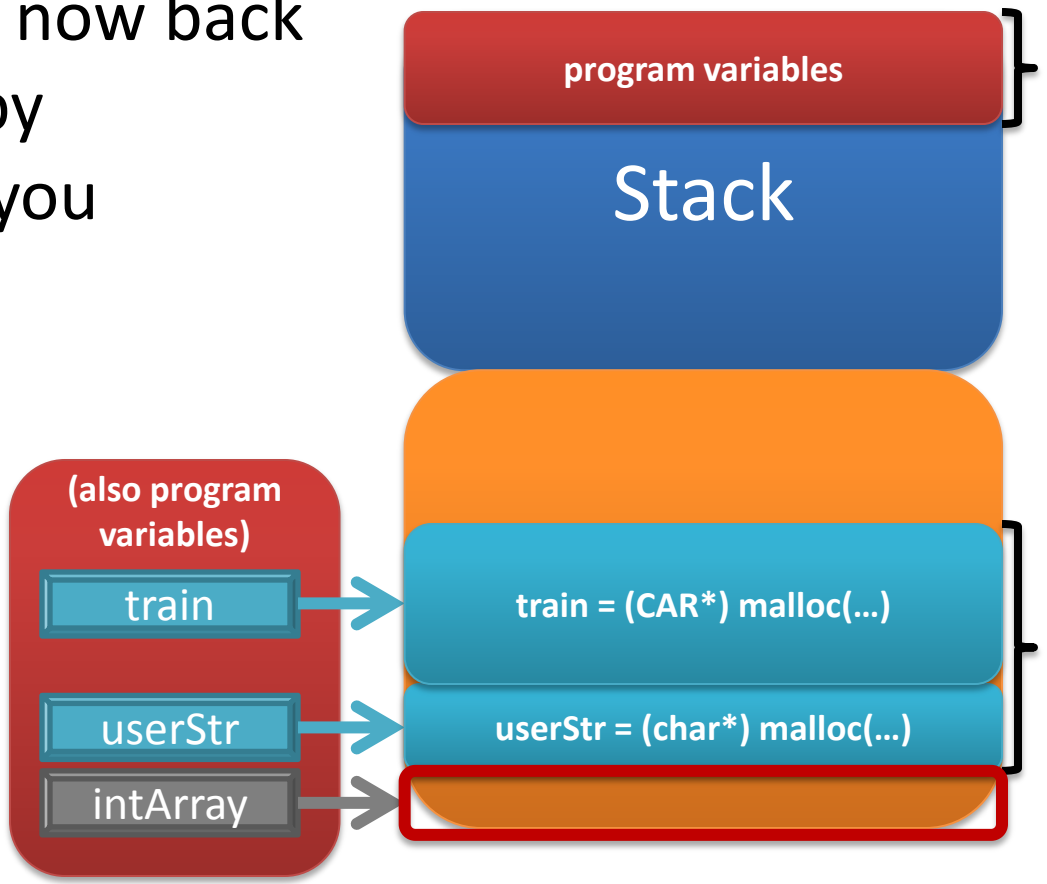
Memory Basics – Memory Errors

- intArray is now a **dangling pointer**
  - points to memory that has been freed



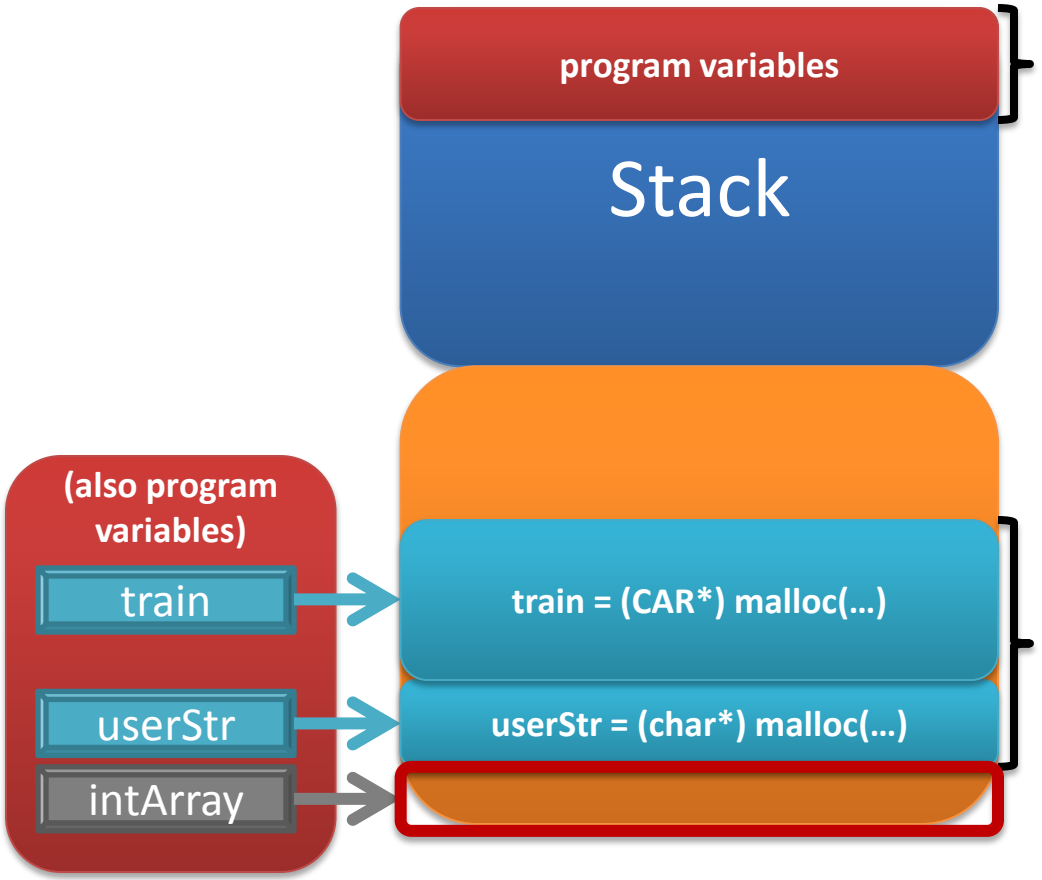
Memory Basics – Memory Errors

- intArray is now a **dangling pointer**
  - points to memory that has been freed
  - memory which is now back to being owned by ***the process***, not you



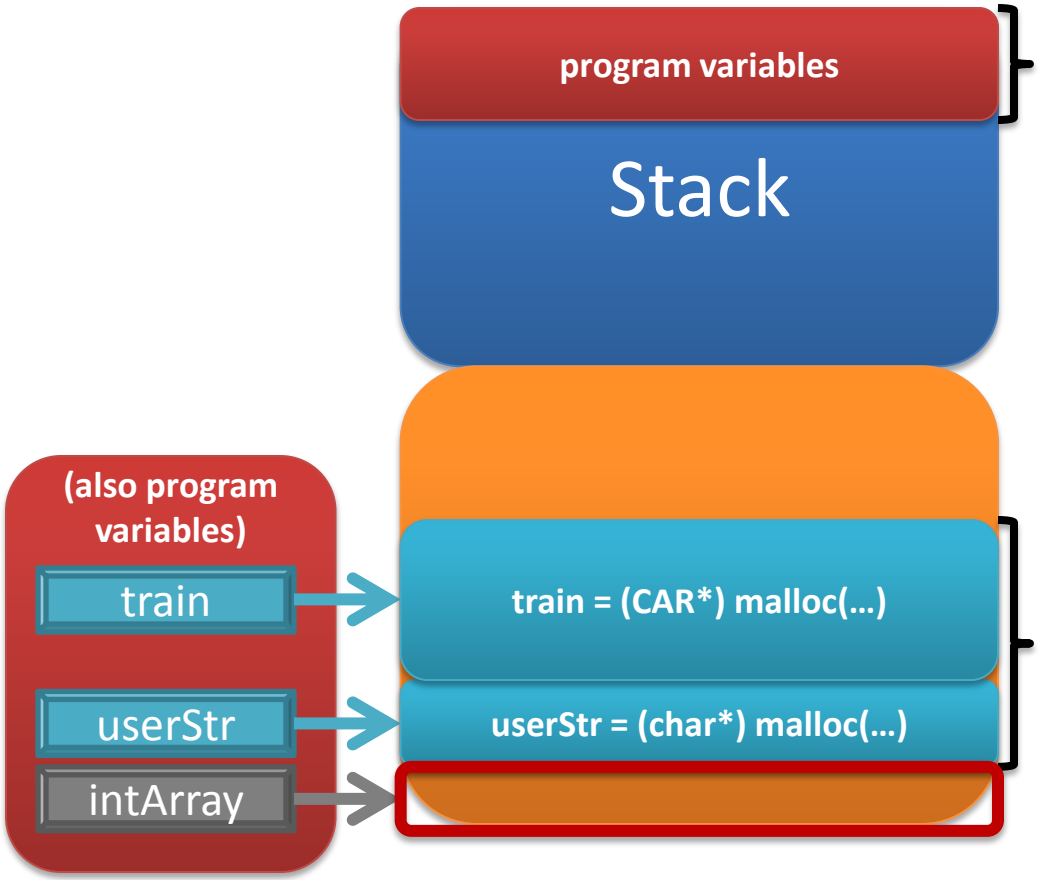
Memory Basics – Memory Errors

- if we tried to free() intArray's memory again
- we would get a



Memory Basics – Memory Errors

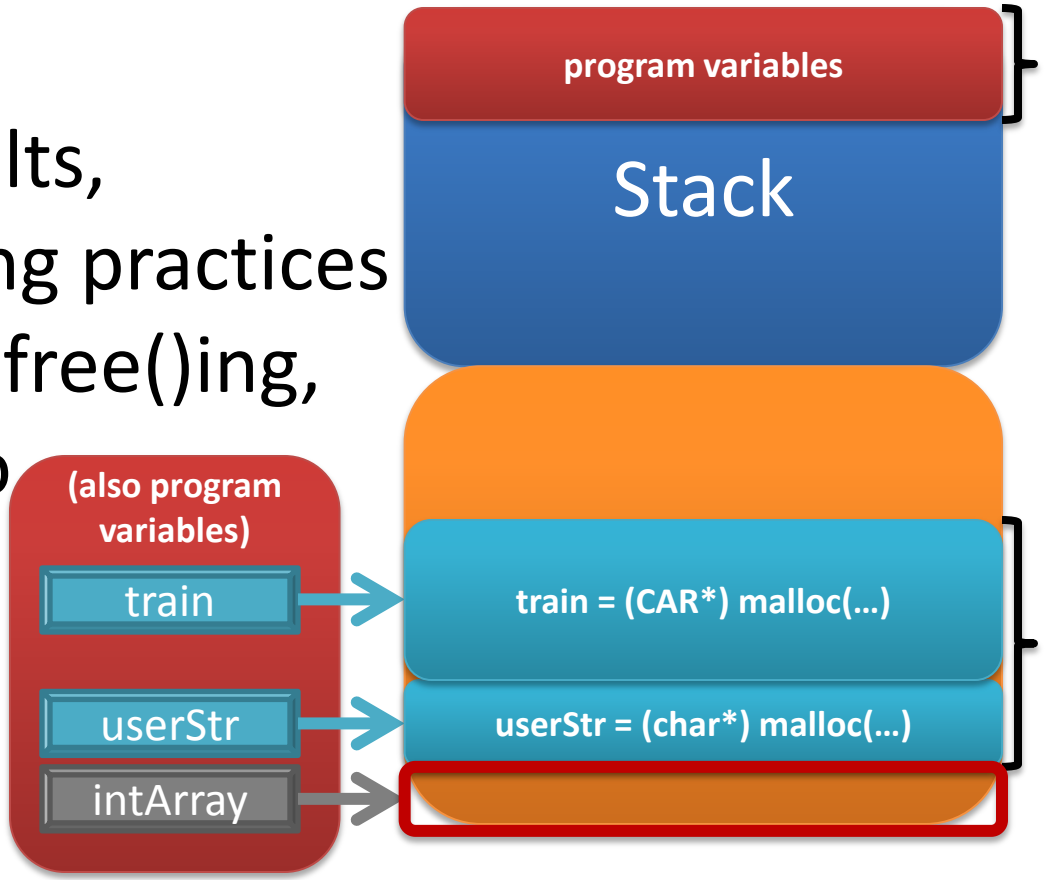
- if we tried to free() intArray's memory again
- we would get a SEGFAULT



Memory Basics – Memory Errors

- if we tried to free() intArray's memory again
- we would get a **SEGFAULT**

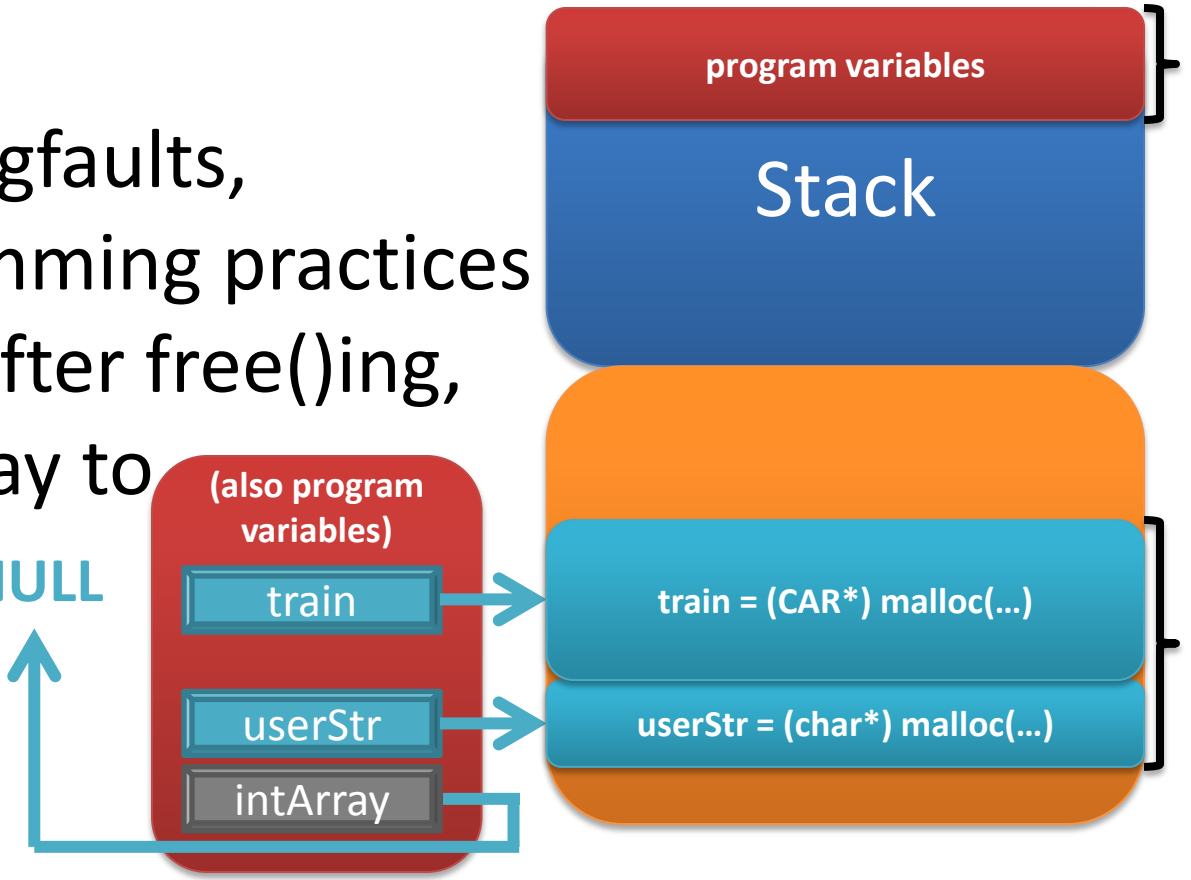
- to prevent segfaults, good programming practices dictate that after free()ing, we set intArray to be equal to



Memory Basics – Memory Errors

- if we tried to free() intArray's memory again
- we would get a **SEGFAULT**

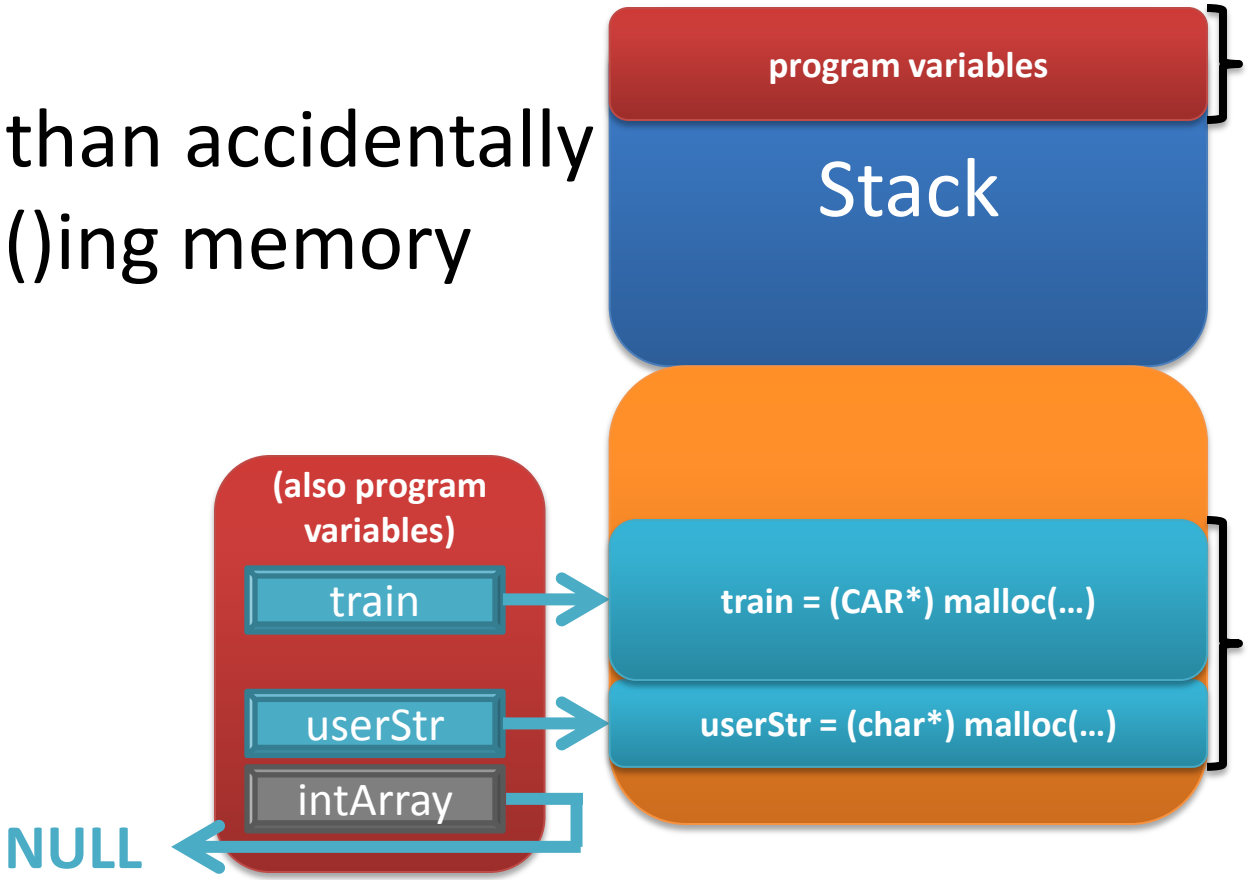
- to prevent segfaults, good programming practices dictate that after free()ing, we set intArray to be equal to **NULL**



Memory Basics – Memory Errors

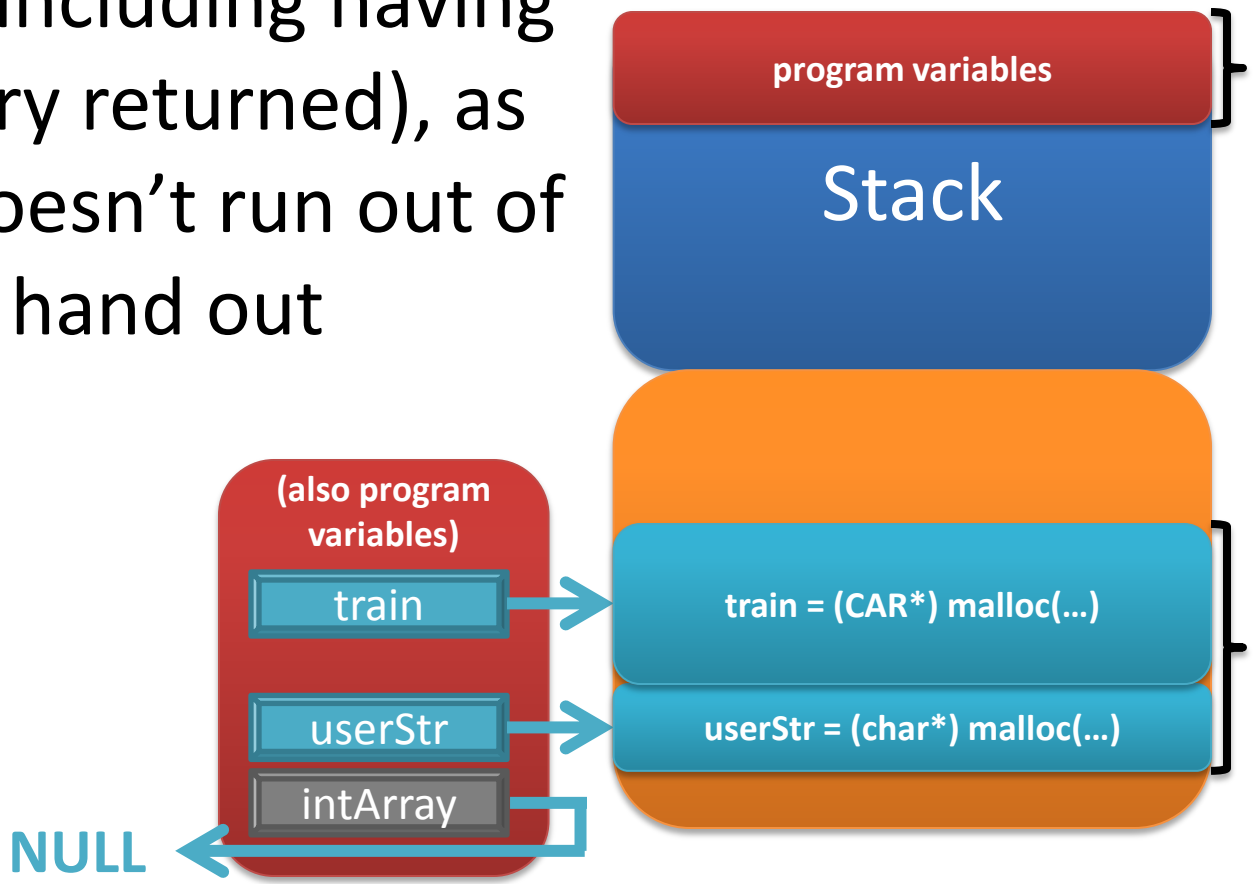
- NOTE: if you try to free a NULL pointer, no action occurs (and it doesn't segfault!)

- much safer than accidentally double free()ing memory



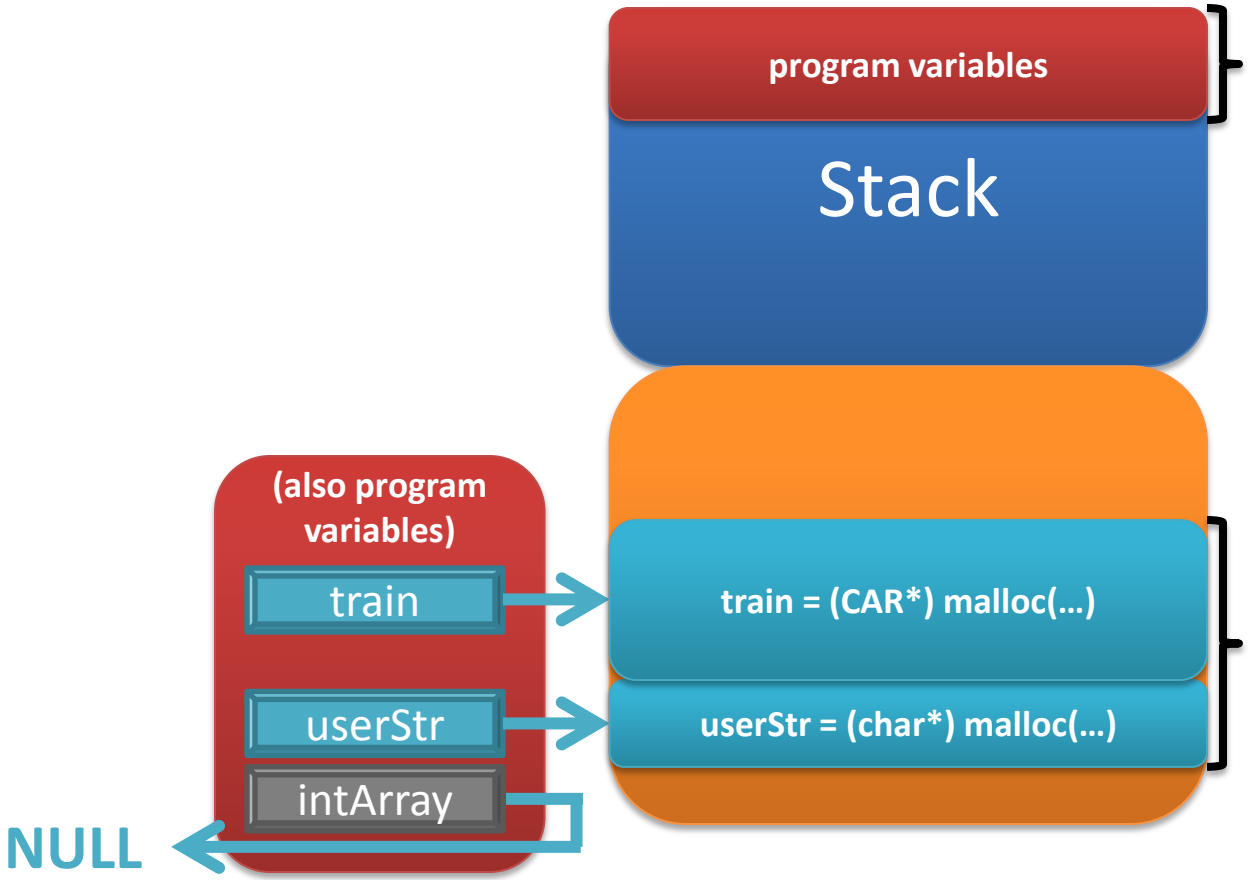
Memory Basics – Running Out

- the process is capable of giving memory to ***you*** and ***the program*** as many times as necessary (including having that memory returned), as long as it doesn't run out of memory to hand out



Memory Basics – Running Out

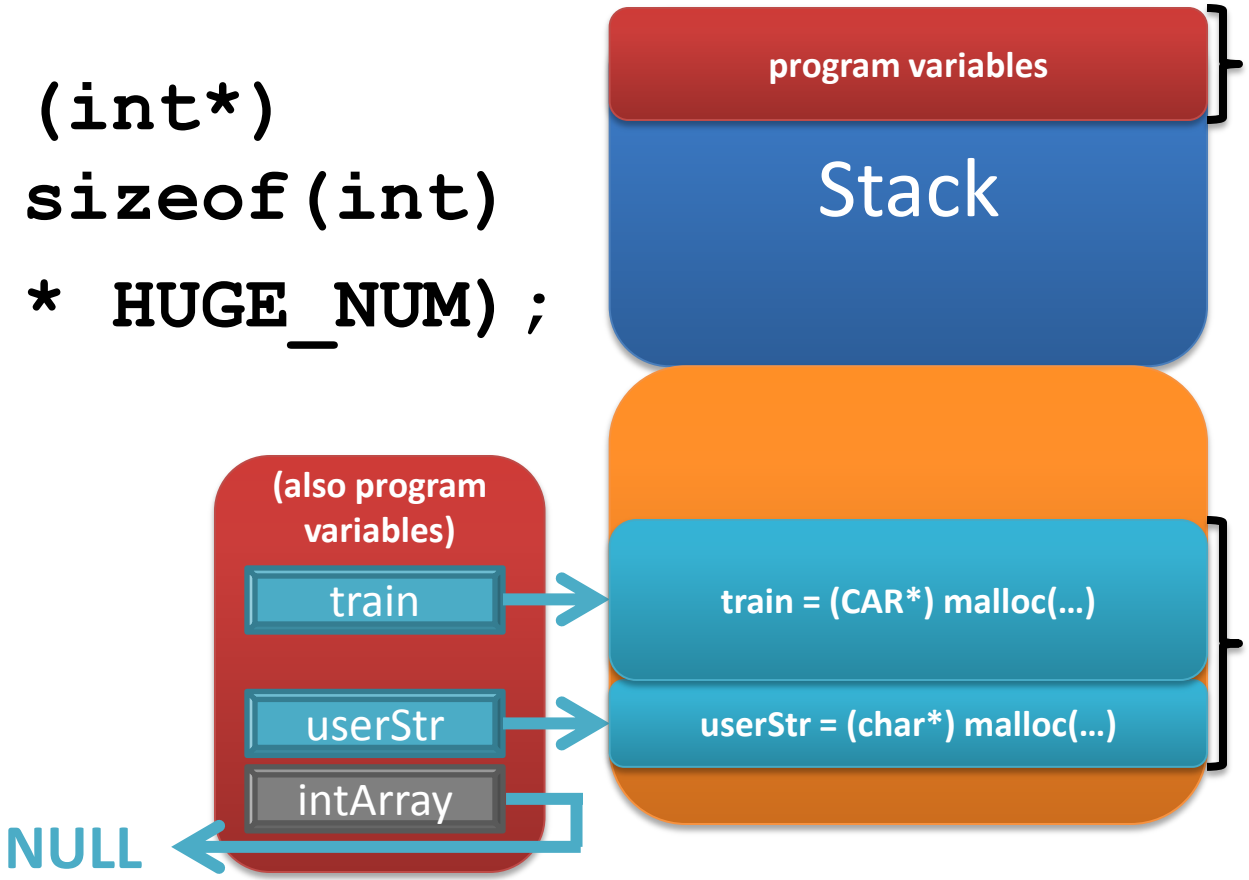
- if you try to allocate memory, but there's not enough contiguous space to handle your request



Memory Basics – Running Out

- if you try to allocate memory, but there's not enough contiguous space to handle your request

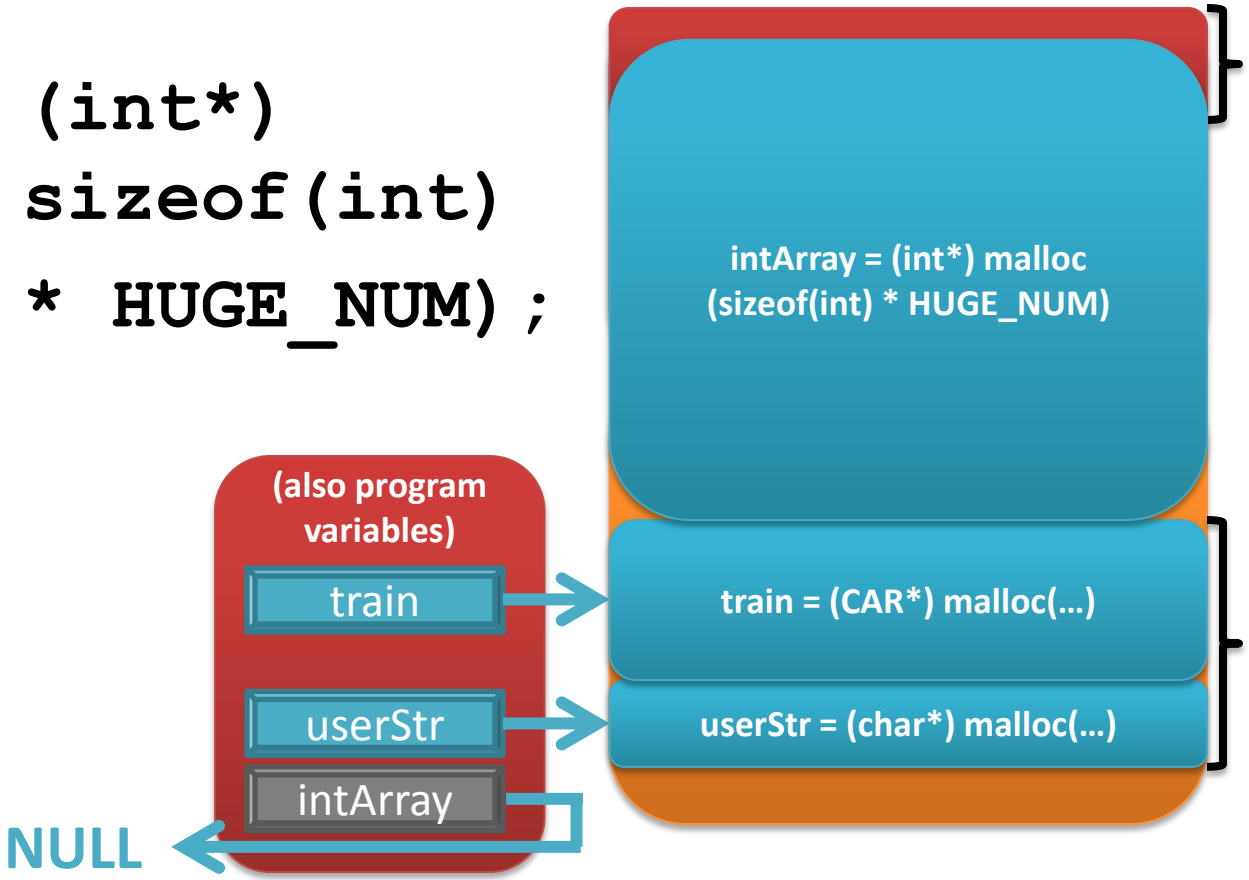
```
intArray = (int*)
   malloc ( sizeof(int)
            * HUGE_NUM);
```



Memory Basics – Running Out

- if you try to allocate memory, but there's not enough contiguous space to handle your request

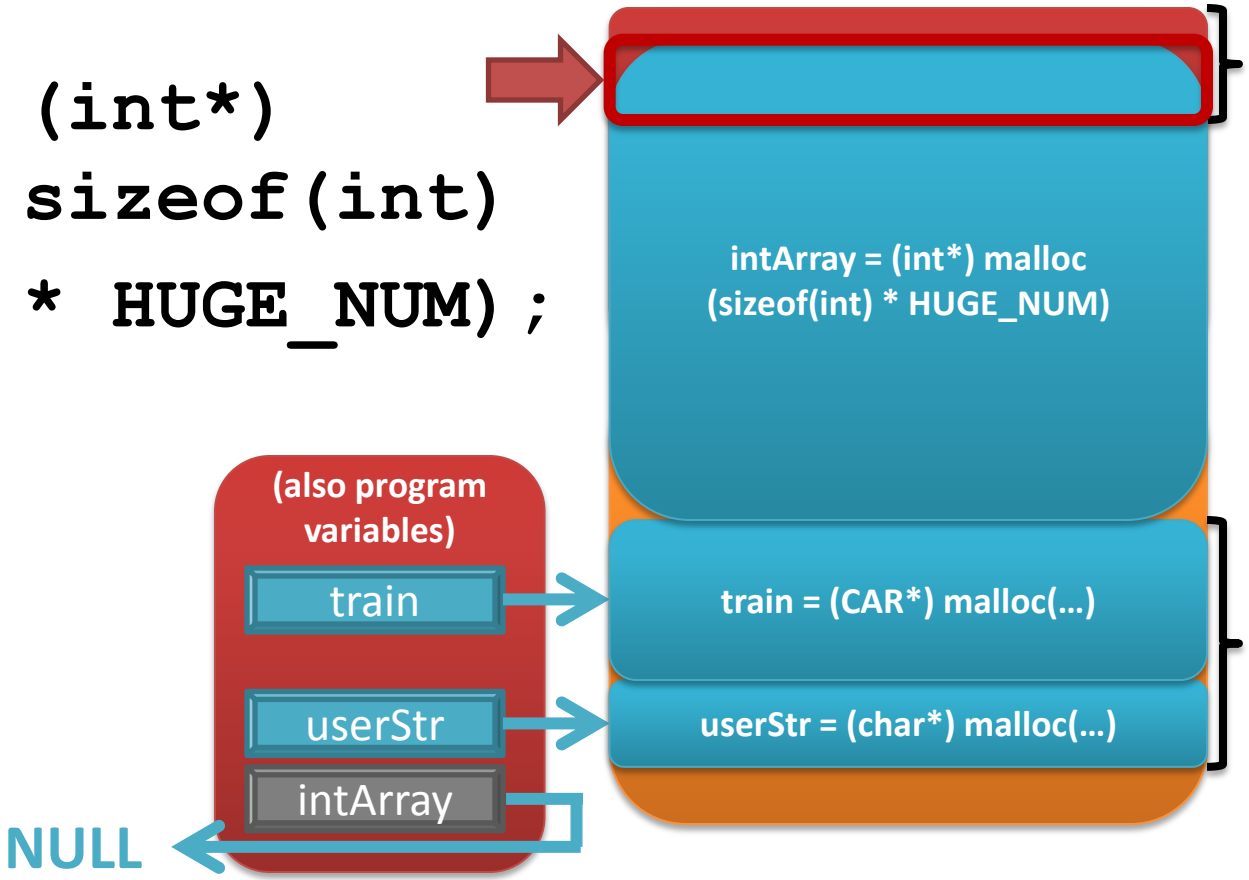
```
intArray = (int*)
   malloc ( sizeof(int)
            * HUGE_NUM);
```



Memory Basics – Running Out

- if you try to allocate memory, but there's not enough contiguous space to handle your request

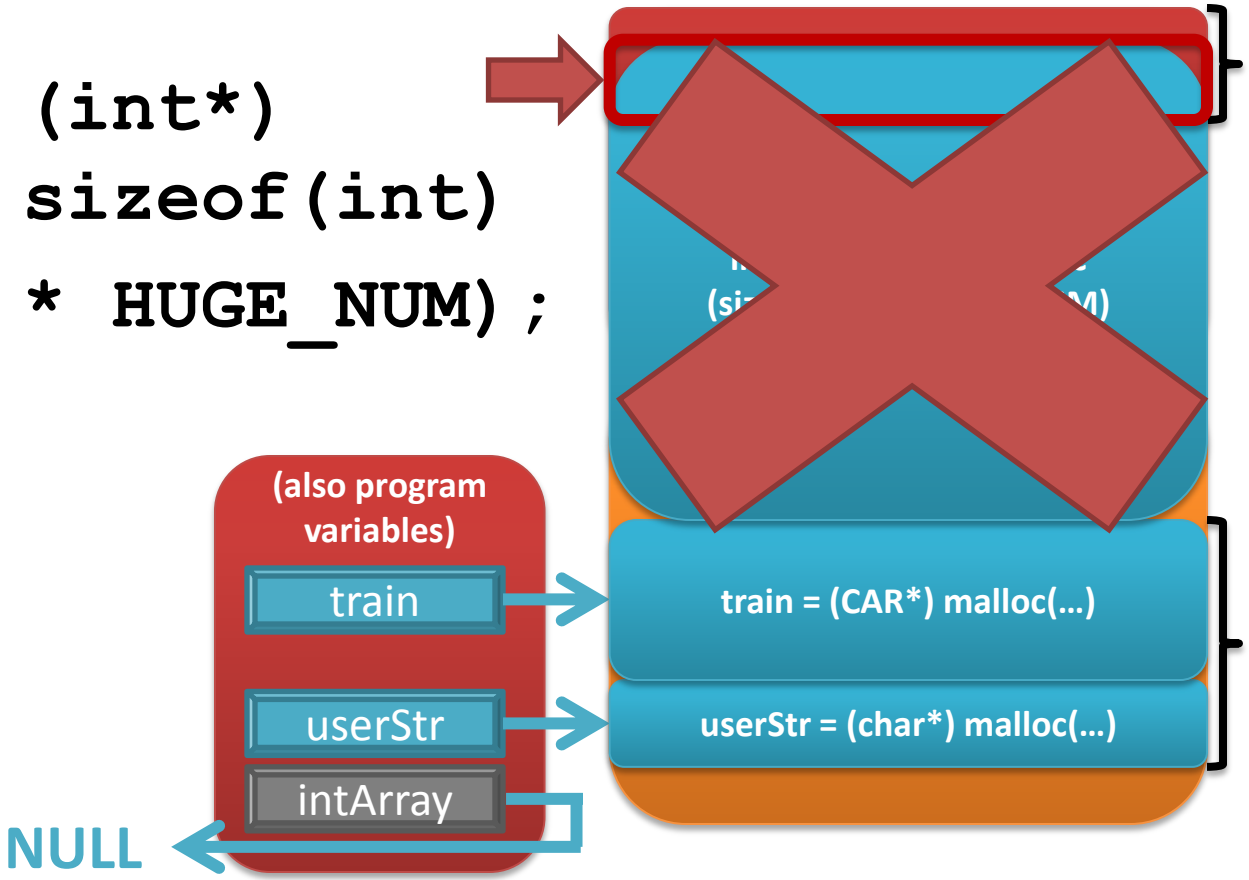
```
intArray = (int*)
   malloc ( sizeof(int)
            * HUGE_NUM);
```



Memory Basics – Running Out

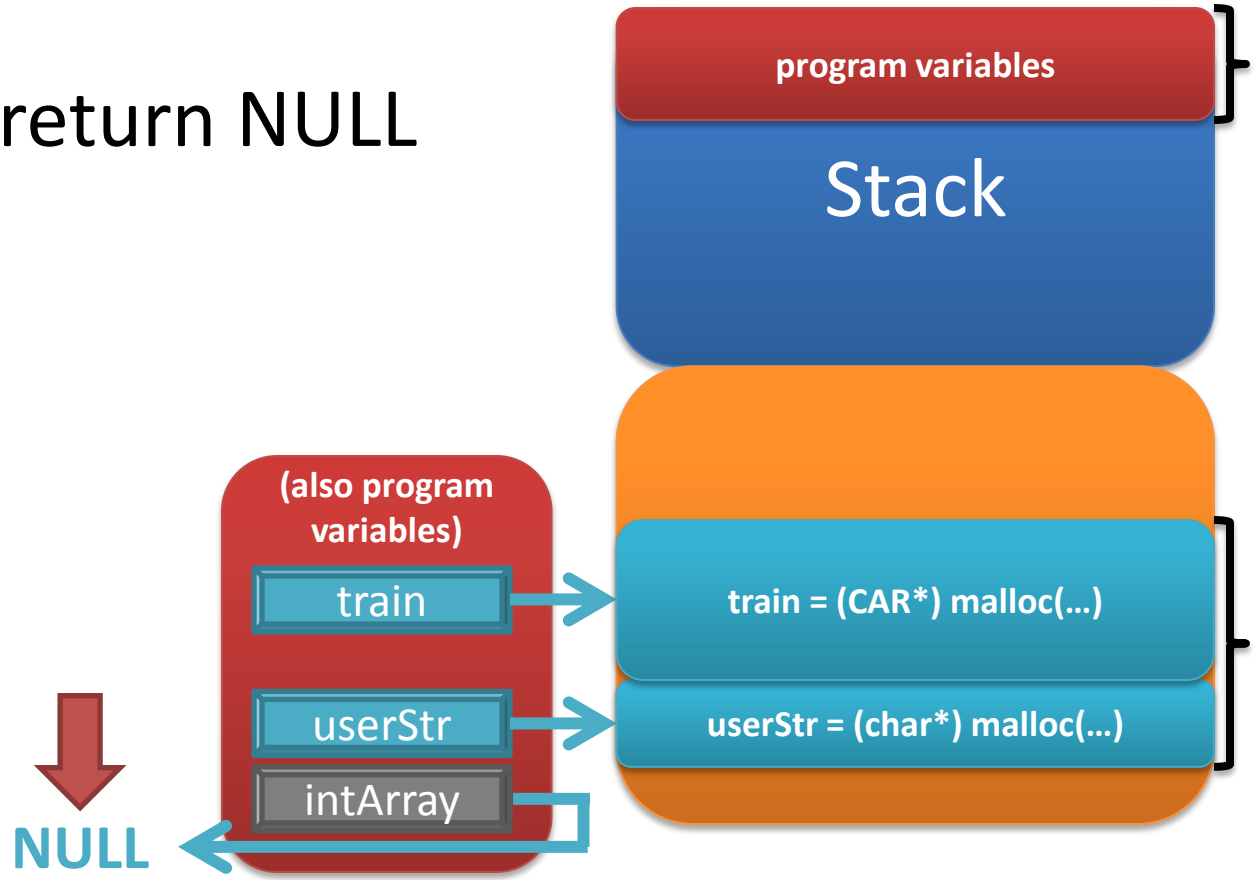
- if you try to allocate memory, but there's not enough contiguous space to handle your request

```
intArray = (int*)
   malloc ( sizeof(int)
            * HUGE_NUM);
```



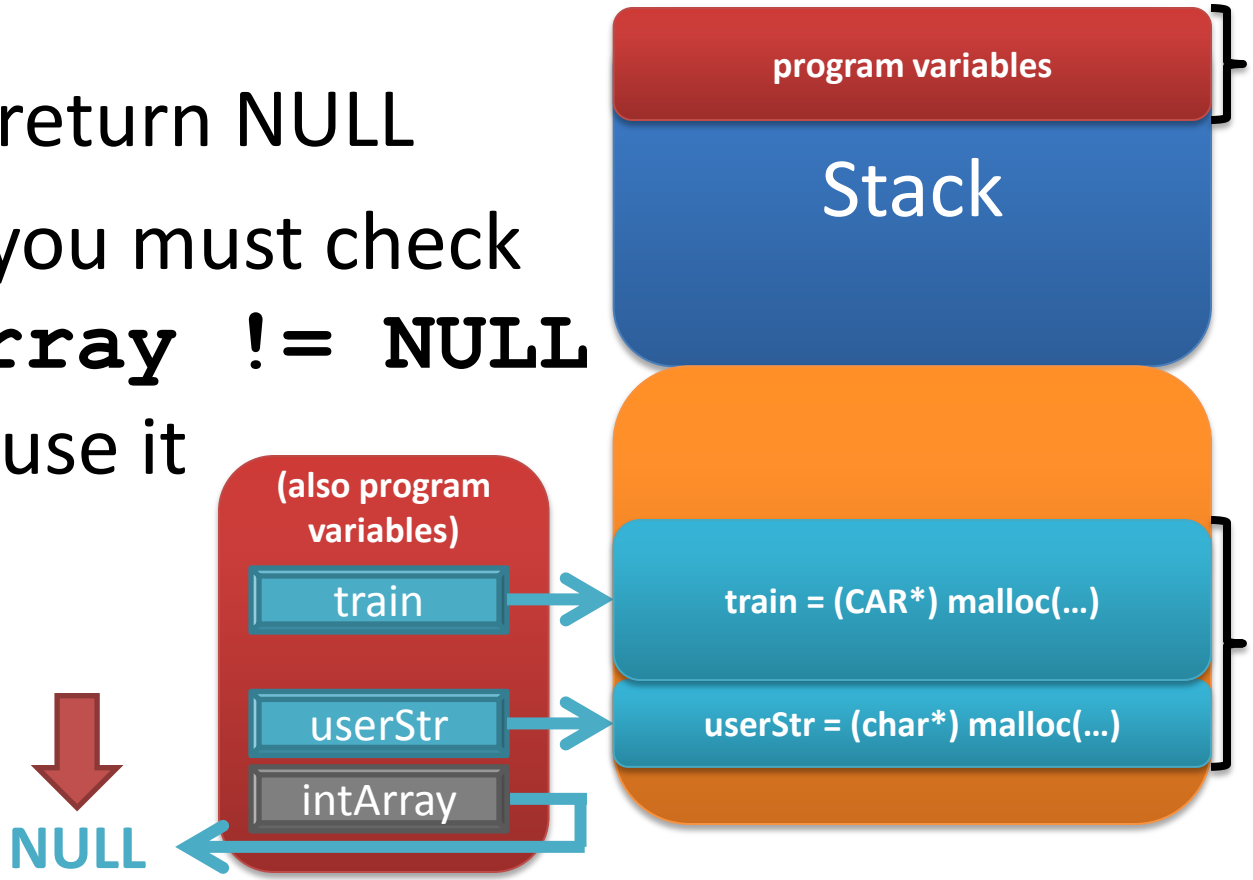
Memory Basics – Running Out

- if you try to allocate memory, but there's not enough contiguous space to handle your request
- malloc will return NULL



Memory Basics – Running Out

- if you try to allocate memory, but there's not enough contiguous space to handle your request
- malloc will return NULL
- that's why you must check that **`intArray != NULL`** before you use it



Quick Note on Segfaults

- segfaults are not consistent (unfortunately)
- even if something **should** result in a segfault, it might not (and then occasionally it will)
  - this doesn't mean there isn't an error!
  - C is trying to be "nice" to you when it can
- you have to be extra-super-duper-careful with your memory management!!!

Memory and Functions

- how do different types of variables get passed to and returned from functions?

- passing by value
- passing by reference
  - implicit: arrays, strings
  - explicit: pointers

Memory and Functions

- some simple examples:

```
int Add(int x, int y);
   int answer = Add(1, 2);
void PrintMenu(void);
   PrintMenu();
int GetAsciiValue(char c);
   int ascii = GetAsciiValue ('m');
```

- all passed by value

Memory and Functions

- passing arrays to functions

```
void TimesTwo(int array[], int size);

int arr [ARR_SIZE];
/* set values of arr */
TimesTwo(arr, ARR_SIZE);
```

- arrays of any type are passed by reference
  - changes made in-function persist

Memory and Functions

- passing arrays to functions

```
void TimesTwo(int array[], int size);
void TimesTwo(int * array, int size);
```

- both of these behave the same way
  - they either take a pointer to:
    - the beginning of an array
    - an int that we (can) treat like an array

Memory and Functions

- passing strings to functions

```
void PrintName(char name [NAME_SIZE]);

char myName [NAME_SIZE] = "Alice";
PrintName(myName);
```

- strings are arrays (of characters)
  - implicitly passed by reference

Memory and Functions

- passing pointers to int to functions

```
void Square(int *n);

int x = 9;

Square(&x);
```

- pass address of an integer (in this case, x)

Memory and Functions

- passing int pointers to function

```
void Square(int *n);

int x = 9;
int *xPtr = &x;
Square(???);
```

- pass ???

Memory and Functions

- passing int pointers to function

```
void Square(int *n);

int x = 9;
int *xPtr = &x;
Square(xPtr);
```

- pass xPtr, which is an address to an integer (x)

Memory and Functions

- returning pointers from functions

```
CAR* MakeCar(void) {
   CAR temp;

   return &temp;  }
```

- temp is on the stack – so what happens?

Memory and Functions

- returning pointers from functions

```
CAR* MakeCar(void) {
   CAR temp;

   return &temp;  }
```

- temp is on the stack – so it will be returned to ***the process*** when MakeCar() returns!

Memory and Functions

- returning pointers from functions

```
CAR* MakeCar(void) {
   CAR* temp;
   temp = (CAR*) malloc (sizeof(CAR));
   return  temp;  }
```

- temp is on the heap – so what happens?

Memory and Functions

- returning pointers from functions

```
CAR* MakeCar(void) {
   CAR* temp;
   temp = (CAR*) malloc (sizeof(CAR));
   return  temp;  }
```

- temp is on the heap – so it belongs to ***you*** and will remain on the heap until you free() it

Homework 4A

- Karaoke

- File I/O
- command line arguments
- allocating memory

- no grade for Homework 4A
- turn in working code or -10 points for HW 4B