CIS 190: C/C++ Programming

Lecture 4

Assorted Topics
(and More on Pointers)

Outline

- Makefiles
- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers
- Memory and Functions
- Homework

Makefiles

- list of rules you can call from the terminal
  - **`make ruleTwo`** will call the ruleTwo
  - **`make`** will call first rule in the file

- basic formatting
  - use **#** at line beginning to denote comments
  - <u>must</u> use tab character, not 8 spaces

Rule Construction

```
target: dependencies (optional)
    command
    another command (optional)
```

- target (rule name)
- dependency (right side of colon)
- command (explicit commands)

Creating a Rule

- let's create a rule to compile and link the files for Homework 4A:

```
hw4a.c
karaoke.c
karaoke.h
```

- what commands will let us do this?

Creating a Rule

we need to, in order:

1. separately compile hw4a.c
2. separately compile karaoke.c
3. link hw4a.o and karaoke.o together

Creating a Rule

1. separately compile hw4a.c

- we’ll make a rule called **`hw4a.o`**
- what command would we run in the terminal?
- what files does it need to work?

Creating a Rule

1. separately compile hw4a.c

- we'll make a rule called **`hw4a.o`**
- what command would we run in the terminal?
- what files does it need to work?
  - hw4a.c
  - so it's dependent on hw4a.c

Creating a Rule

2. separately compile karaoke.c

- we'll call this rule **`karaoke.o`**
- what command will compile karaoke.c?
- what files does it need to work?

Creating a Rule

2. separately compile karaoke.c

- we'll call this rule **`karaoke.o`**
- what command will compile karaoke.c?
- what files does it need to work?
  - karaoke.c
  - so it's dependent on karaoke.c

Creating a Rule

3. link hw4a.o and karaoke.o together

- we'll call this rule **`hw4a`**
- what command will link the files together?
- what files does it depend on?

Creating a Rule

3. link hw4a.o and karaoke.o together

- we'll call this rule **`hw4a`**
- what command will link the files together?
- what files does it depend on?
  - hw4a.o
  - karaoke.o
  - so it's dependent on both of these files

Other Common Rules

- a rule to remove .o and executable files

```
clean:
        rm –f *.o hw4a
```

- a rule to remove garbage files

```
cleaner:
        rm –f *~
```

- a rule to run both

```
cleanest: clean cleaner
```

Why Use Makefiles

- makes compiling, linking, executing, etc
  - easier
  - quicker
  - less prone to human error

- allows use to create and run helper rules
  - clean up unneeded files (like hw2.c~ or trains.o)
  - open files for editing

Makefiles and Beyond

- there's much more you can do with Makefiles
  - variables
  - conditionals
  - system configuration
  - phony targets
- more information available here

http://www.chemie.fu-berlin.de/chemnet/use/info/make/make_toc.html

Outline

- Makefiles
- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers
- Memory and Functions
- Homework

Input and Output

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

FILE I/O Basics

- allow us to read in from and print out to files
  - instead of from and to the terminal

- use a ***file pointer*** (FILE*) to manage the file(s) we want to be handling

- naming conventions:

```
FILE* ofp; /* output file pointer */
FILE* ifp; /* input file pointer */
```

Opening a File

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

- fopen() returns a FILE pointer
  - hopefully to a successfully opened file

- <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

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 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

Dealing with FILE Pointers

- FILE pointers should be handled with the same care as allocated memory

1. check that it works before using
2. gracefully handle failure
3. free when finished

Handling FILE Pointers

1. check that it worked before using
   - if the FILE pointer is NULL, there was an error
2. gracefully handle failure
   - print out an error message
   - exit or re-prompt the user, as appropriate
3. free the pointer when finished
   - use fclose() and pass in the file pointer

Standard Streams in C

- three standard ***streams***: stdin, stdout, stderr

- printf() and scanf() automatically access stdout and stdin, respectively

- printing to stderr prints to the terminal
  - even if we use redirection

Using File Pointers

- fprintf

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

  – output written to where **`ofp`** points

- fscanf

  `fscanf(ifp, “%d”, &inputInt);`

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

LIVECODING

Using stderr with fprintf

```
/* if an error occurs */
if (error)
{
  fprintf(stderr,
          “An error occurred!”);
  exit(-1);
  /* exit() requires <stdlib.h> */
}
```

LIVECODING

Reaching EOF with fscanf

- fscanf() returns an integer
  - number of items in argument list that were filled
- if no data is read in, it returns EOF
  - EOF = End Of File (pre-defined)
- once EOF is returned, we have reached the end of the file
  - handle appropriately (e.g., close)

LIVECODING

Reaching EOF Example

- example usage:

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

- to use fscanf() effectively, it helps to know basic information about the layout of the file

LIVECODING

Outline

- Makefiles
- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers
- Memory and Functions
- Homework

Giving Command Line Arguments

- ***command line arguments*** are given after the executable name on the command line
  - allows user to change parameters at run time without recompiling or needing access to code
  - also sometimes called CLAs

- for example, the following might allow a user to set the maximum number of train cars:

```
> ./hw2 25
```

Handling Command Line Arguments

- handled as 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

More About argc/argv

- names are by convention, not required

- `char **argv` can also be written as
  `char *argv[]`

- argv is just an array of strings (the arguments)
- for example, **`argv[0]`** is the executable
  - since that is the first argument passed in

Command Line Argument Example

- **`> ./hw2 25 Savannah`**
  – set max # of cars and a departure city

Command Line Argument Example

`> ./hw2 25 Savannah`

– set max # of cars and a departure city

- in this example:
  - **`argc`** = **`???`**
  - **`argv[0]`** is **`???`**
  - **`argv[1]`** is **`???`**
  - **`argv[2]`** is **`???`**

Command Line Argument Example

`> ./hw2 25 Savannah`

- set max # of cars and a departure city

- in this example:
  - **`argc`** = 3 (executable, number, and city)

Command Line Argument Example

`> ./hw2 25 Savannah`

- set max # of cars and a departure city

- in this example:
  - **`argc`** = 3 (executable, number, and city)
  - **`argv[0]`** is "**`./hw2`**"

Command Line Argument Example

```
> ./hw2 25 Savannah
```

  - set max # of cars and a departure city

- in this example:
  - **`argc`** = 3 (executable, number, and city)
  - **`argv[0]`** is "**`./hw2`**"
  - **`argv[1]`** is "**`25`**"

Command Line Argument Example

> ./hw2 25 Savannah

- set max # of cars and a departure city

- in this example:
  - **argc** = 3 (executable, number, and city)
  - **argv[0]** is "**./hw2**"
  - **argv[1]** is "**25**"
  - **argv[2]** is "**Savannah**"

How to Use argc

- before we begin using CLAs, we need to make sure that we have been given what we expect
- check that the value of argc is correct
  - that the number of arguments is correct
- if it’s not correct, exit and prompt user with expected program usage

LIVECODING

How to Use argv

- `char **argv` is an array of strings

- if an argument needs to be an integer, we must convert it from a string
  - using the `atoi()` function (from <stdlib.h>)

```
intArg = atoi("5");
intArg = atoi( argv[2] );
```

LIVECODING

Optional Command Line Arguments

- argument(s) can optional
  - e.g., default train to size 20 if max size not given

- number of acceptable CLAs is now a range, or at least a minimum number

- should only use the CLAs you actually have

Handling Optional CLAs

```
if (argc > MAX_ARGS) {
  /* print out error message */
  exit(-1);
}
if (argc >= SIZE_ARG+1) {
  trainSize = argv[SIZE_ARG];
} else {
  trainSize = DEFAULT_TRAIN_SIZE;
}
```

Outline

- Makefiles
- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers
- Memory and Functions
- Homework

Random Numbers

- useful for many things:
  - cryptography, games of chance & probability, procedural generation, statistical sampling

- random numbers generated via computer can only be ***pseudorandom***

Pseudo Randomness

- "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." – *John von Neumann*

- pseudorandom
  – appears to be random, but actually isn't
  – mathematically generated, so it can't be

Seeding for Randomness

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

- same seed means same “random” numbers
  - good for testing, allow identical runs

```
void srand (unsigned int seed);
srand(1);
srand(seedValue);
```

Seeding with User Input

- can allow the user to choose the seed
  - gives user more control over how program runs

```
srand(userSeedChoice);
```

- obtain user seed choice via
  - in-program prompt (“Please enter seed: ”)
  - as a command line argument
    - can make this an optional CLA

Seeding with Time

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

- time() returns the seconds since the "epoch"
  - normally since 00:00 hours, Jan 1, 1970 UTC

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

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

Example of Seeding with time()

- get the seconds since epoch

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

  - `time()` wants a pointer, so just give it 0
  - returns a `time_t` object, so we cast as `int`
- use timeSeed to seed the rand() function

```
srand(timeSeed);
```

- NOTE: running again within a second will return the same value from time()

Generating Random Numbers

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

- call the rand() function each time you want a random number

```
int randomNum = rand();
```

- integer returned is between 0 and RAND_MAX
  - RAND_MAX guaranteed to be at least 32767

Getting a Usable Random Number

- if we want a smaller range than 0 - 32767?
- use % (mod) to get the range you want

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

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

Outline

- Makefiles
- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers
- Memory and Functions
- Homework

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

5

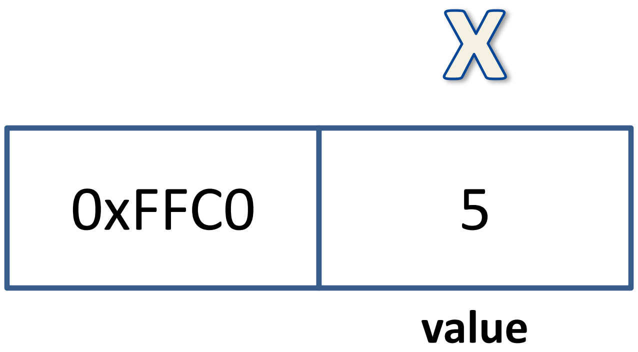
Memory Basics – Regular Variables

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

0xFFC0	5

Memory Basics – Regular Variables

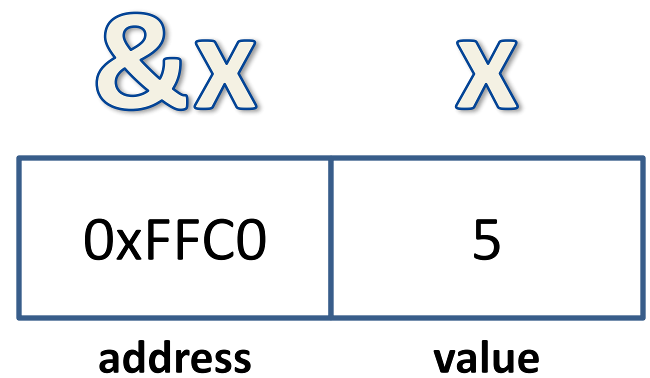
- all variables have two parts:
  - value
  - address where value is stored



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

Memory Basics – Regular Variables

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

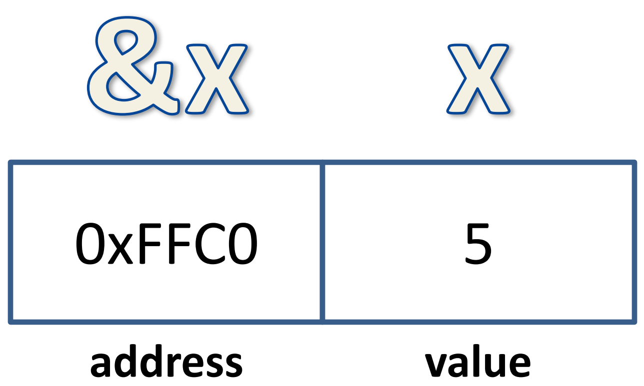


- **x**'s **value** is 5
- **x**'s **address** is 0xFFC0

Memory Basics – Regular Variables

- so the code to declare this is:

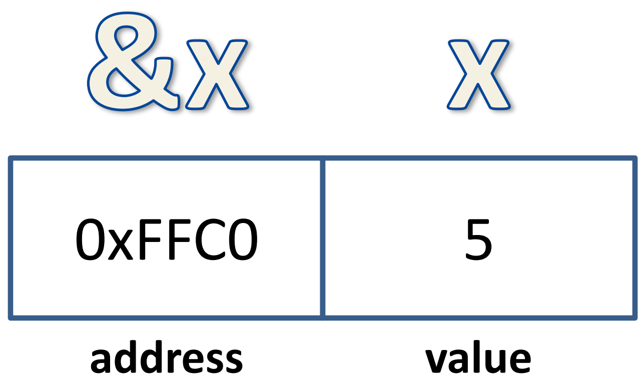
```
int x = 5;
```



Memory Basics – Regular Variables

- we can also declare a pointer:

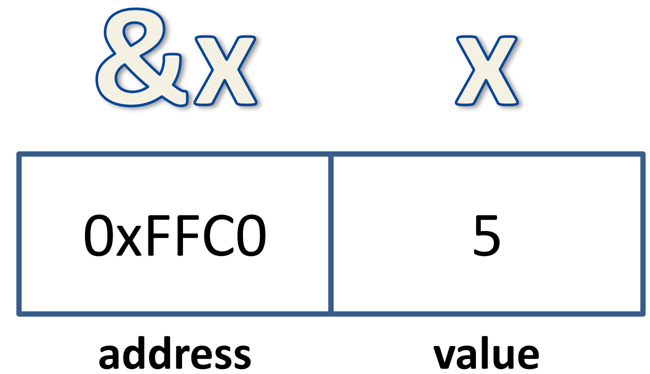
```
int x = 5;
int *ptr;
```



Memory Basics – Regular Variables

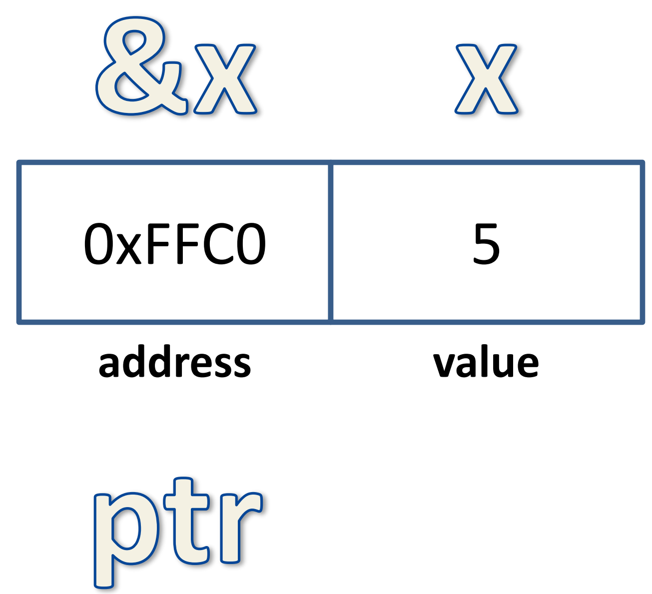
- and set it equal to the address of **x**:

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



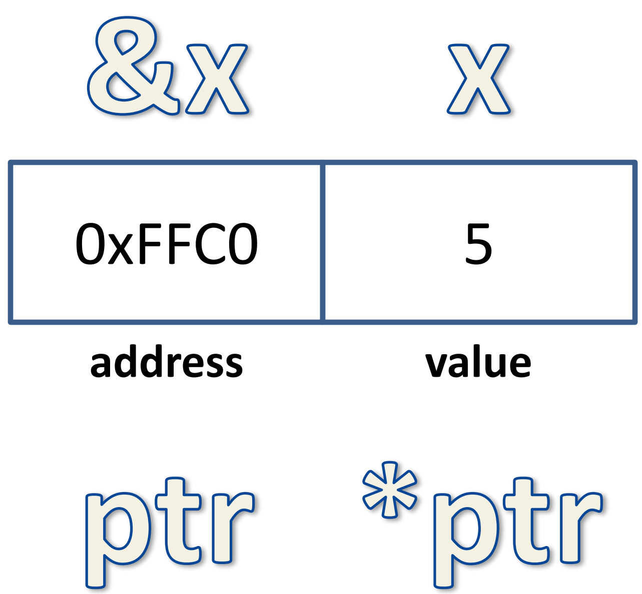
Memory Basics – Regular Variables

- **`ptr = &x`**



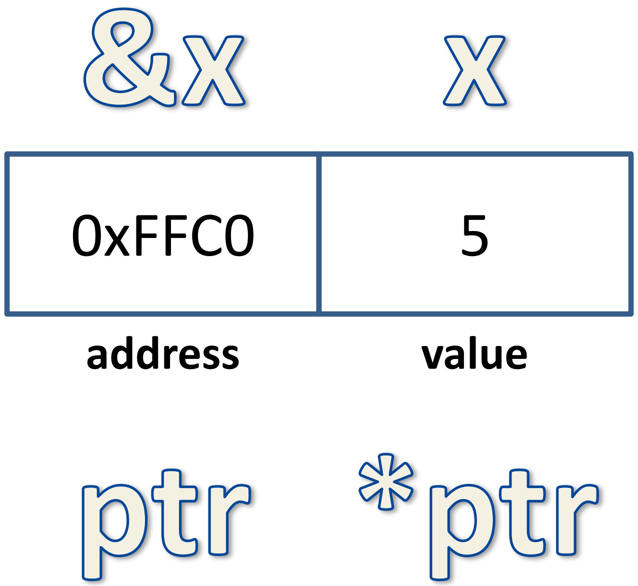
Memory Basics – Regular Variables

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



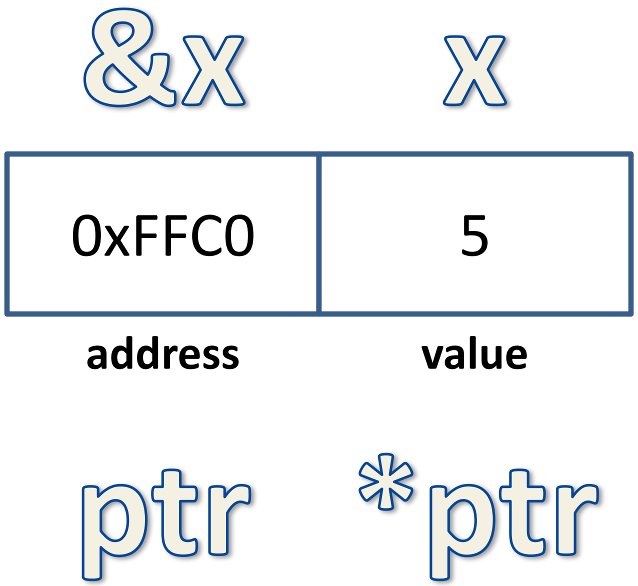
Memory Basics – Regular Variables

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



Memory Basics – Pointer Variables

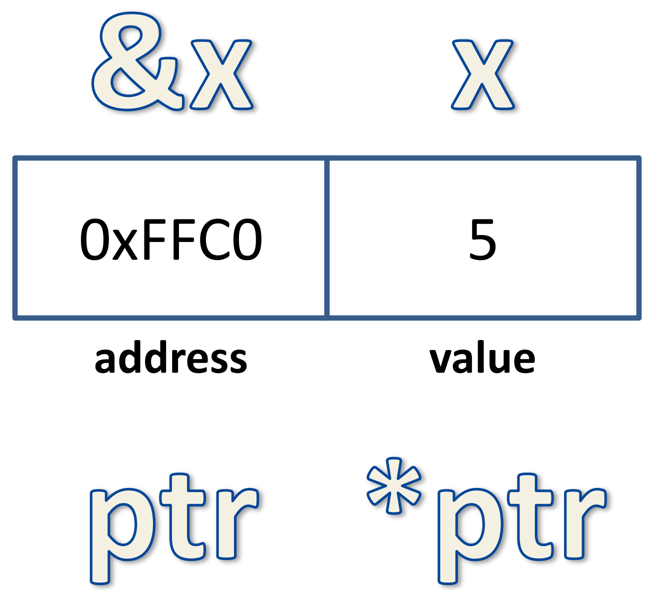
- but what about the variable **`ptr`**?
  - does it have a value and address too?



Memory Basics – Pointer Variables

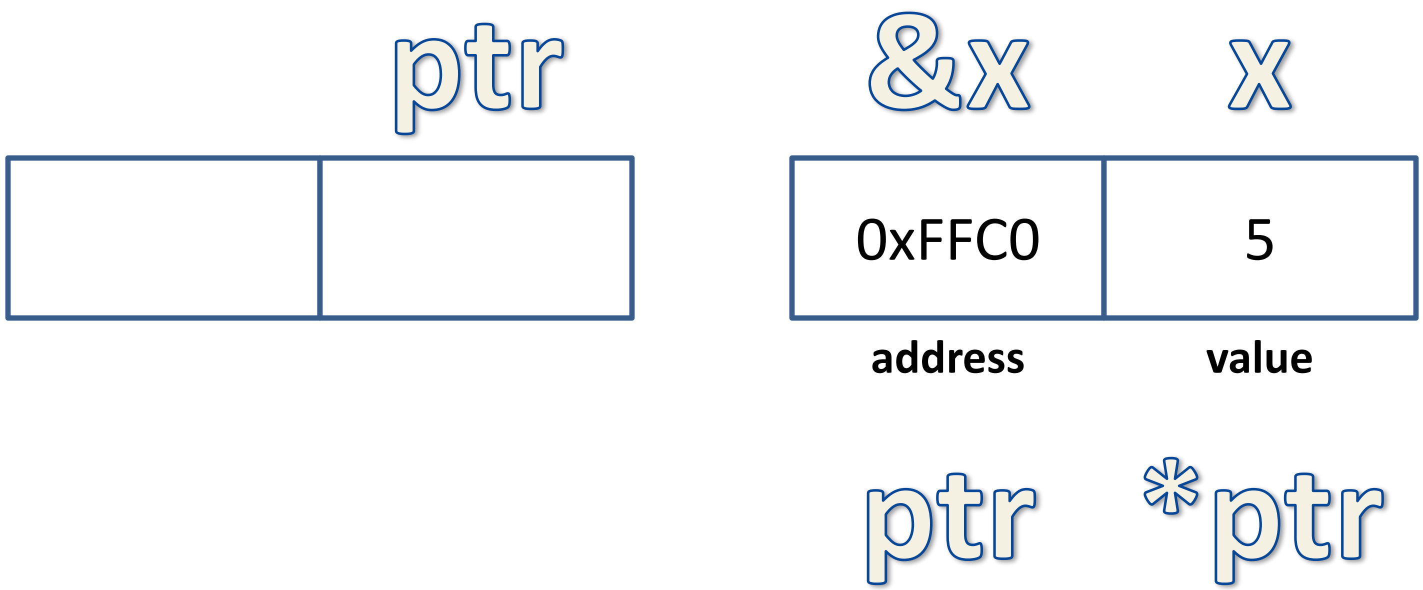
- but what about the variable **`ptr`**?
  - does it have a value and address too?

- YES!!!



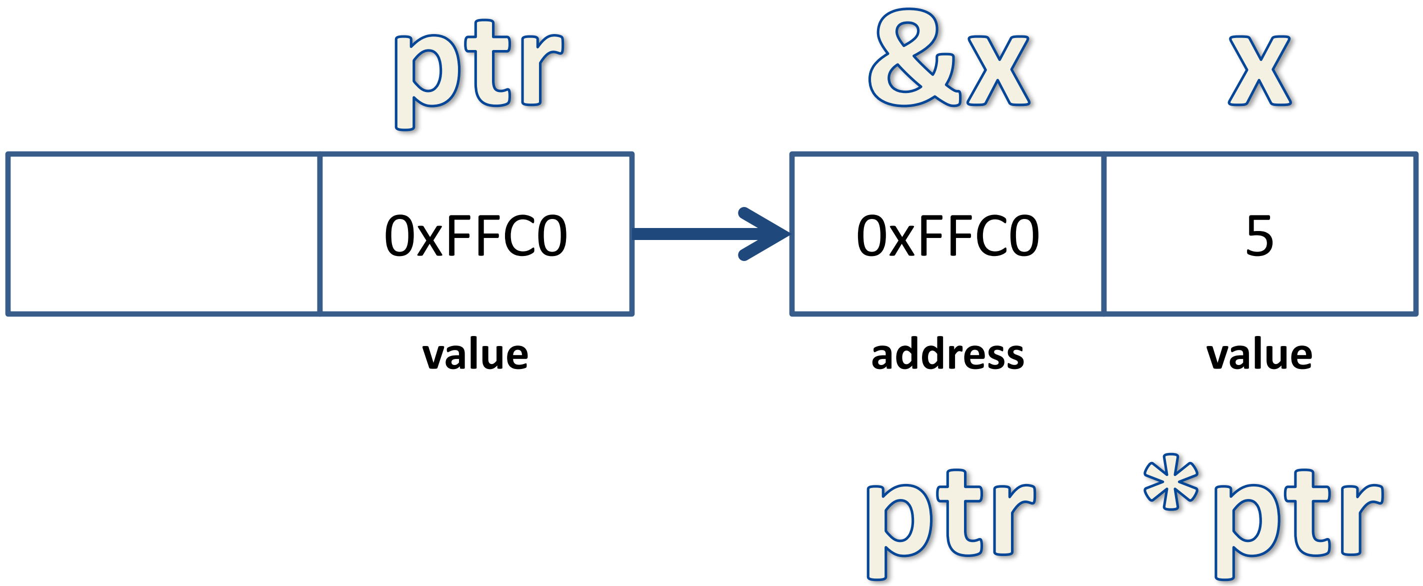
Memory Basics – Pointer Variables

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



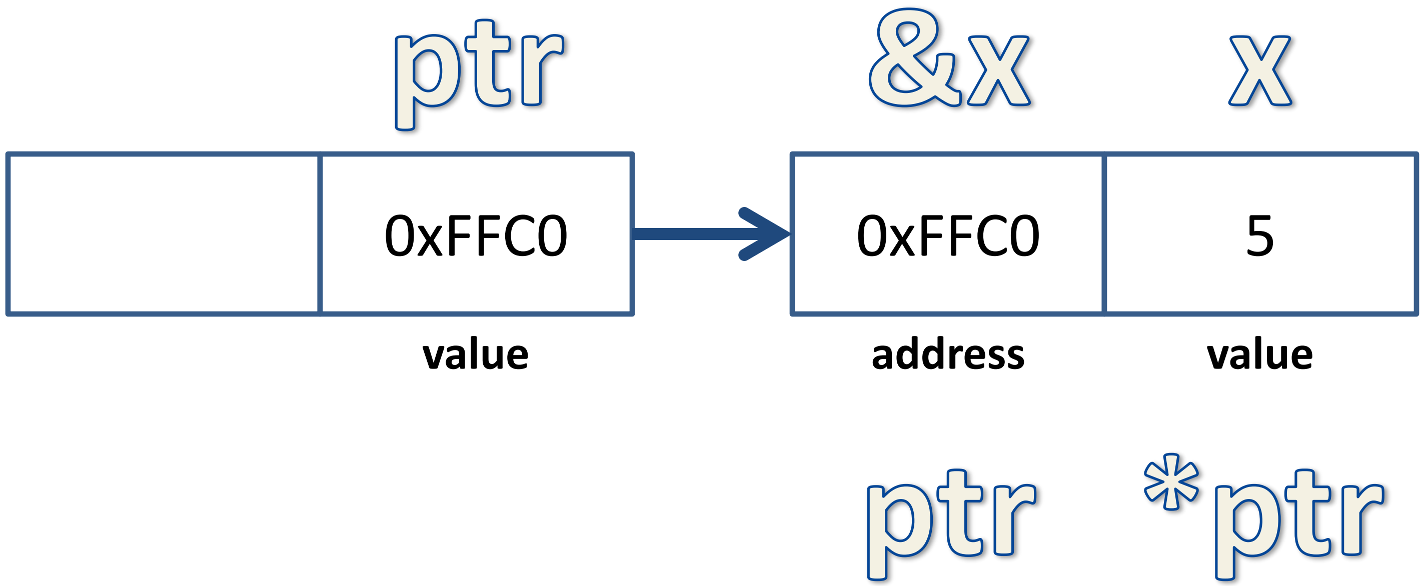
Memory Basics – Pointer Variables

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



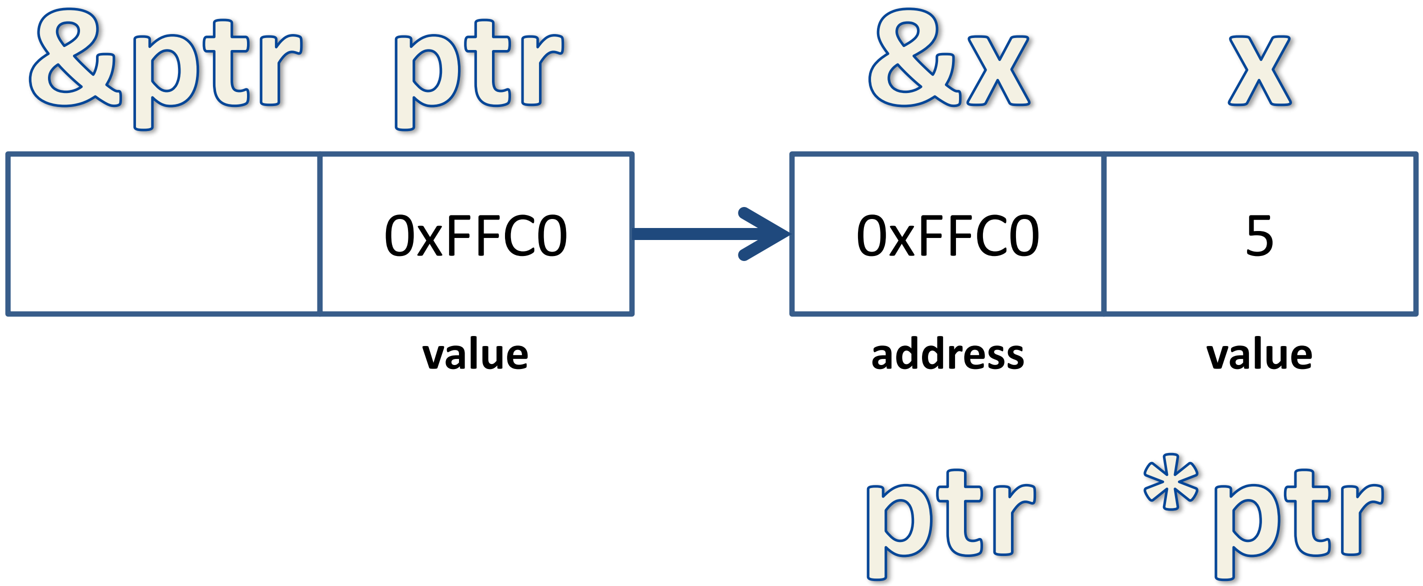
Memory Basics – Pointer Variables

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



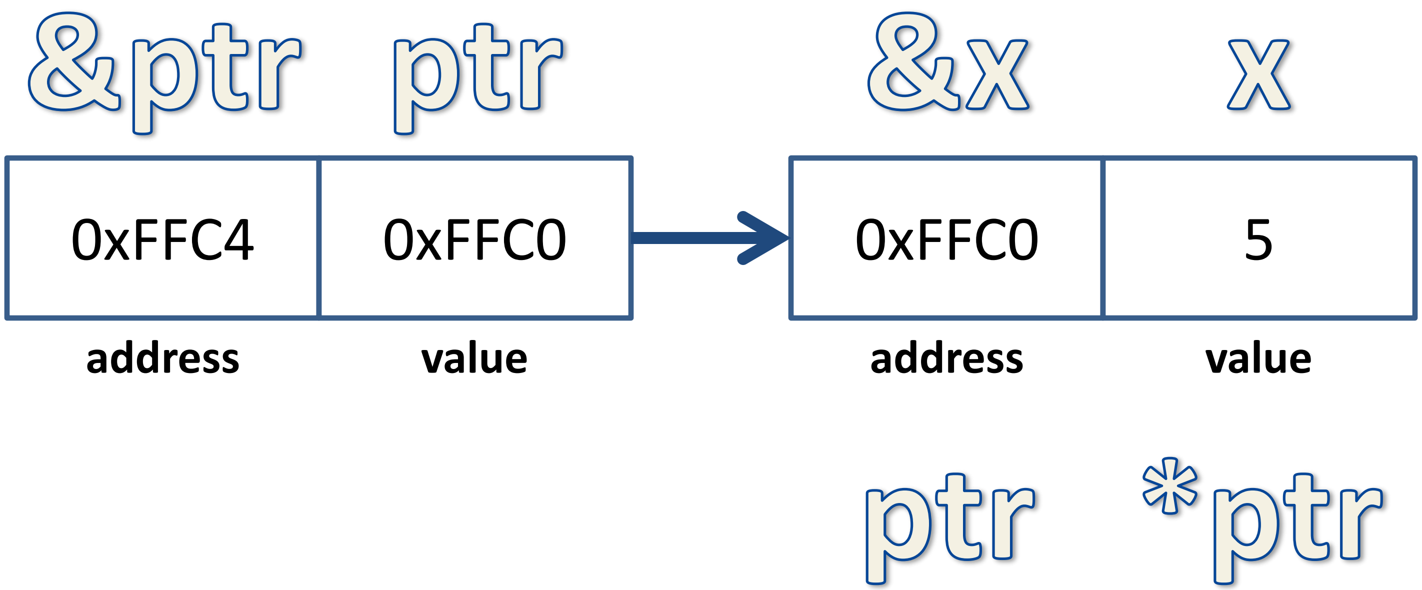
Memory Basics – Pointer Variables

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



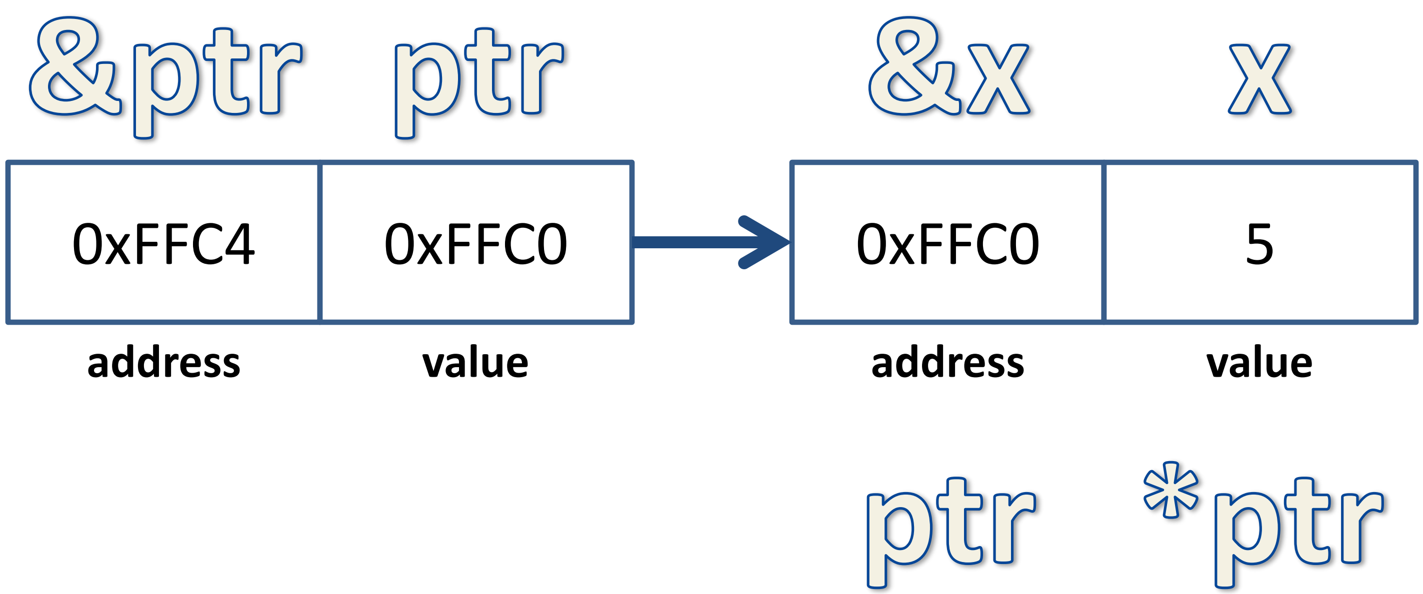
Memory Basics – Pointer Variables

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



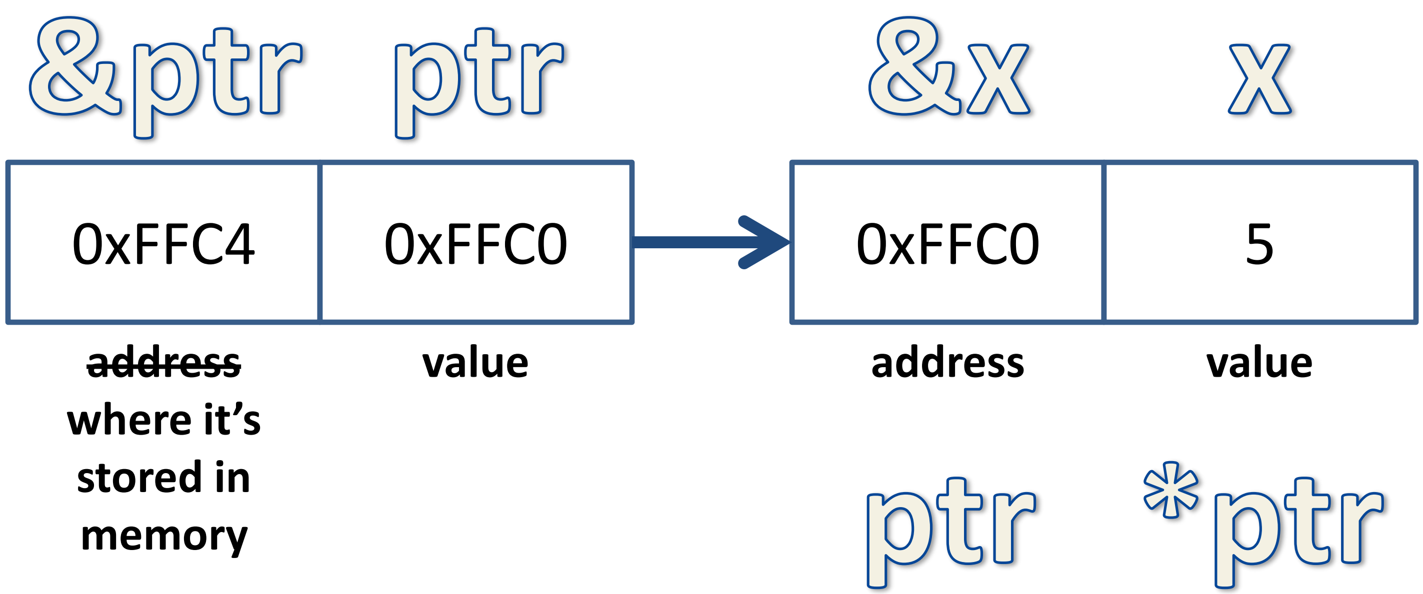
Memory Basics – Pointer Variables

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



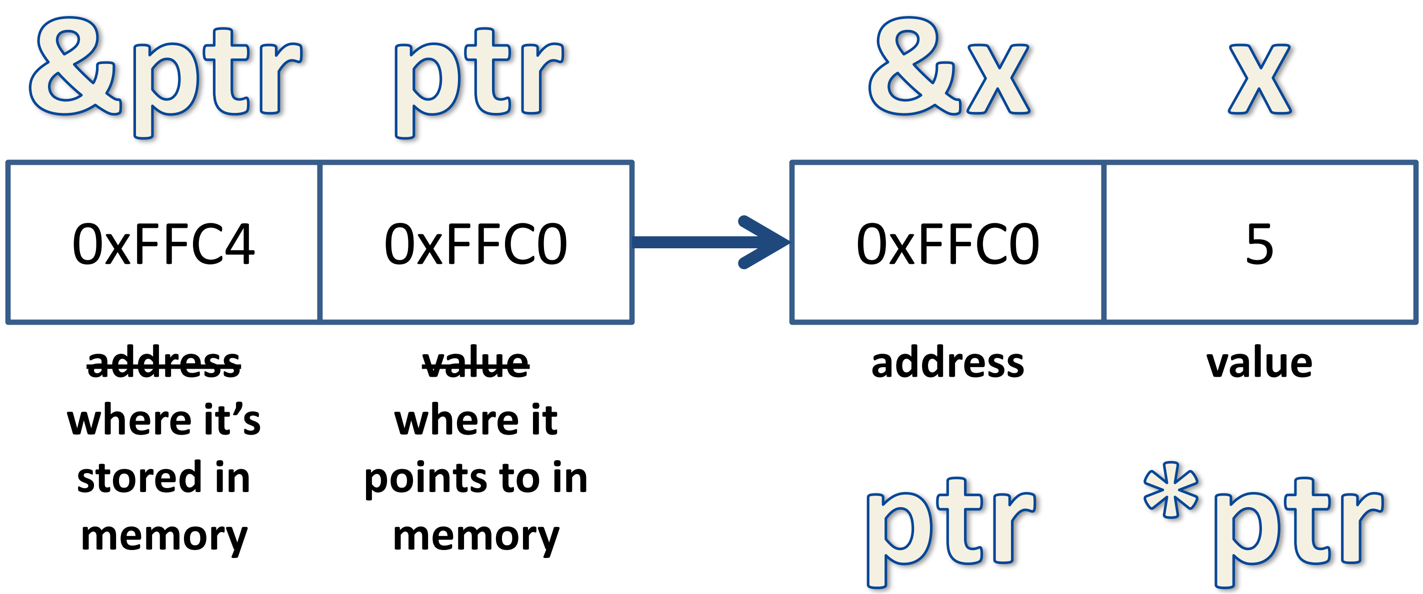
Memory Basics – Pointer Variables

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



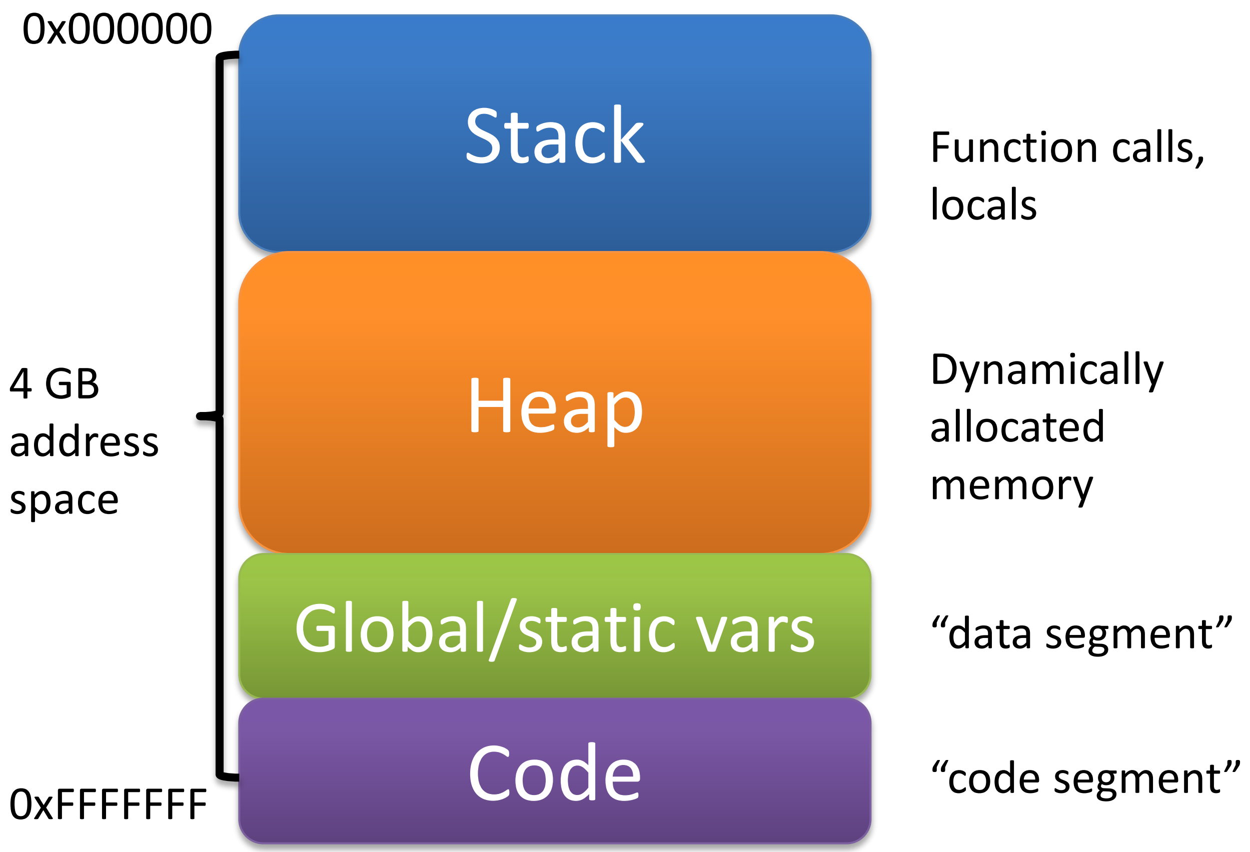
Memory Basics – Pointer Variables

- ~~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*

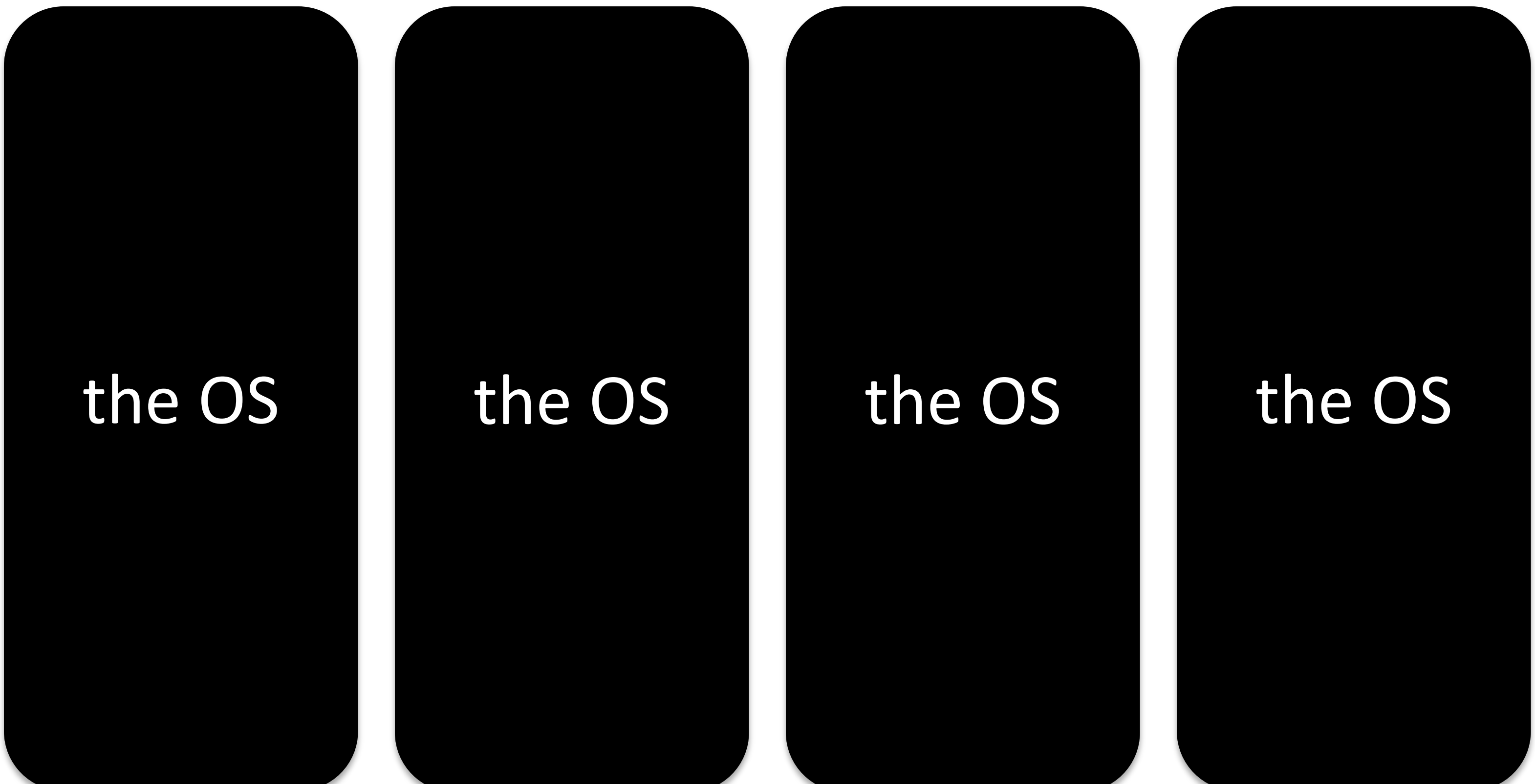


Memory Basics – "Owning" Memory

- 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)

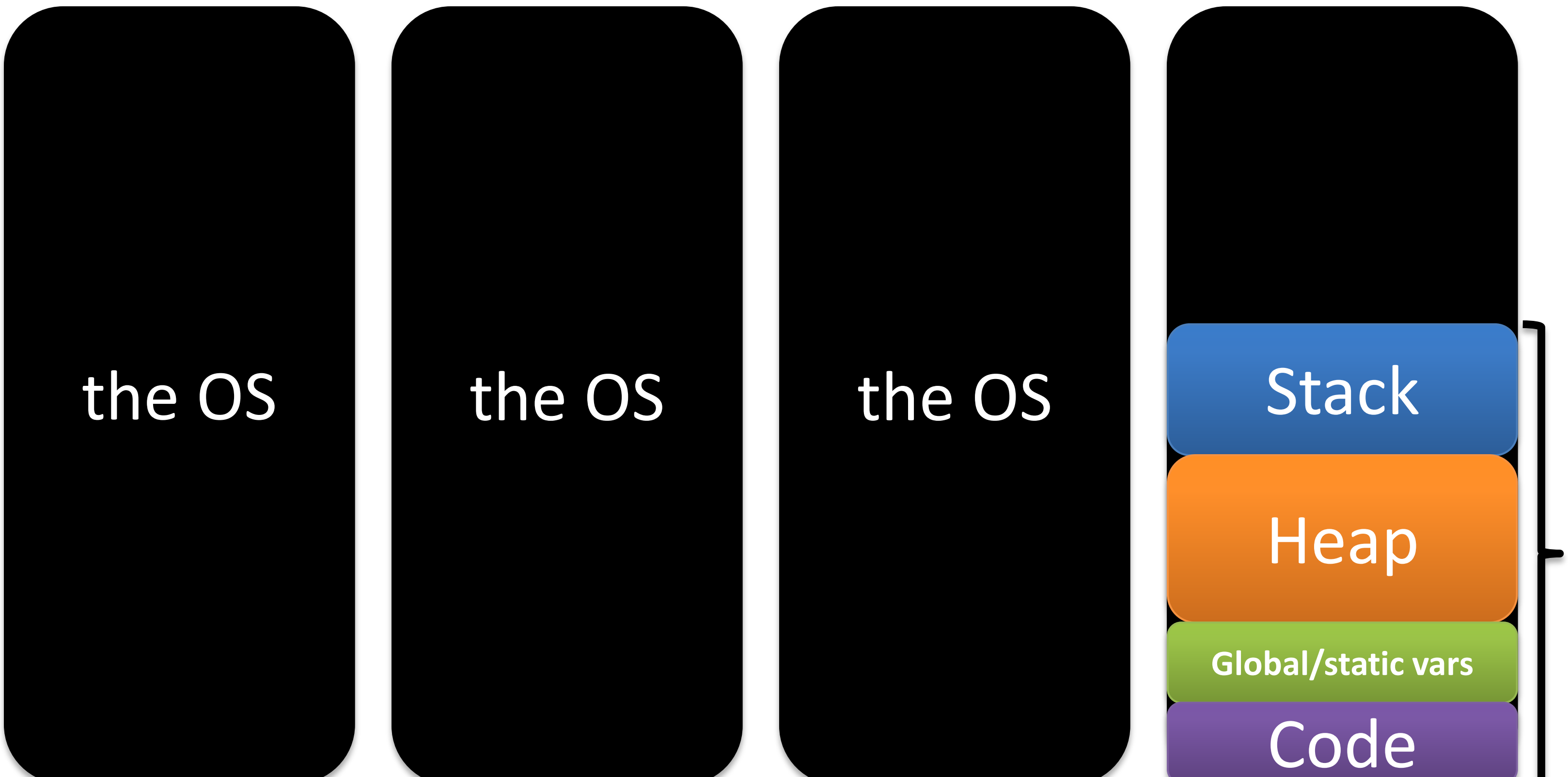
Memory Basics – "Owning" Memory

- 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

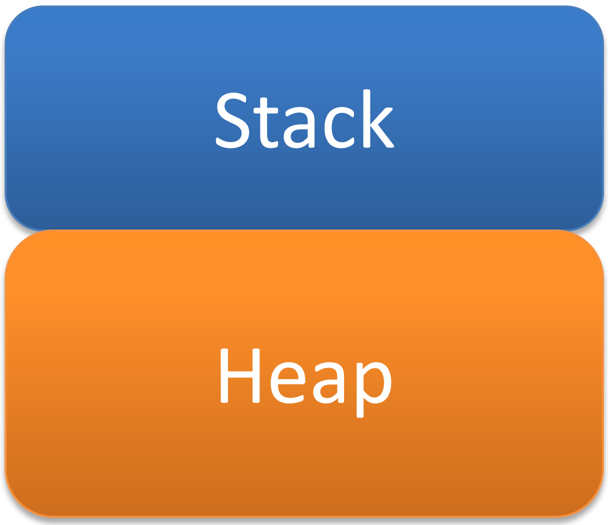
- when ***the process*** begins, the Operating System gives it a chunk of that memory

Memory Basics – “Owning” Memory

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

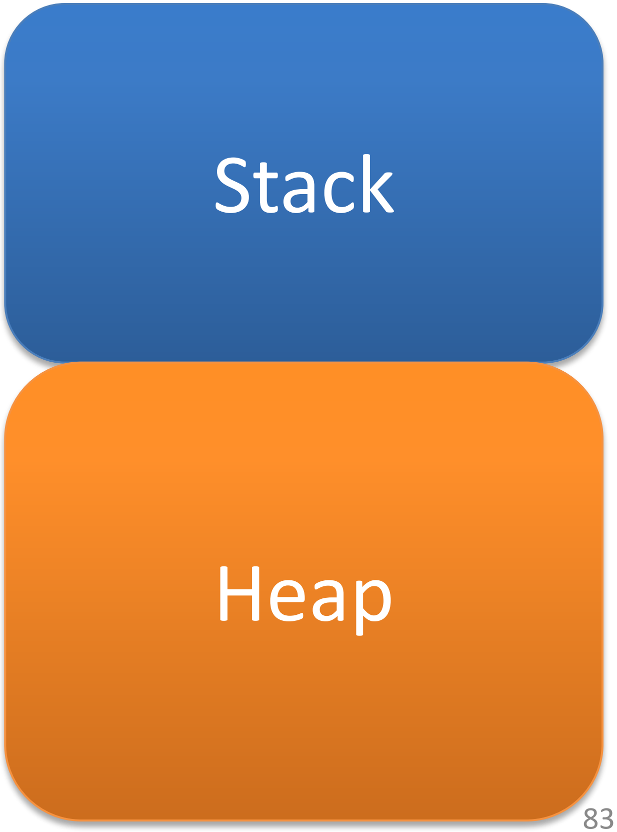
Memory Basics – "Owning" Memory

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



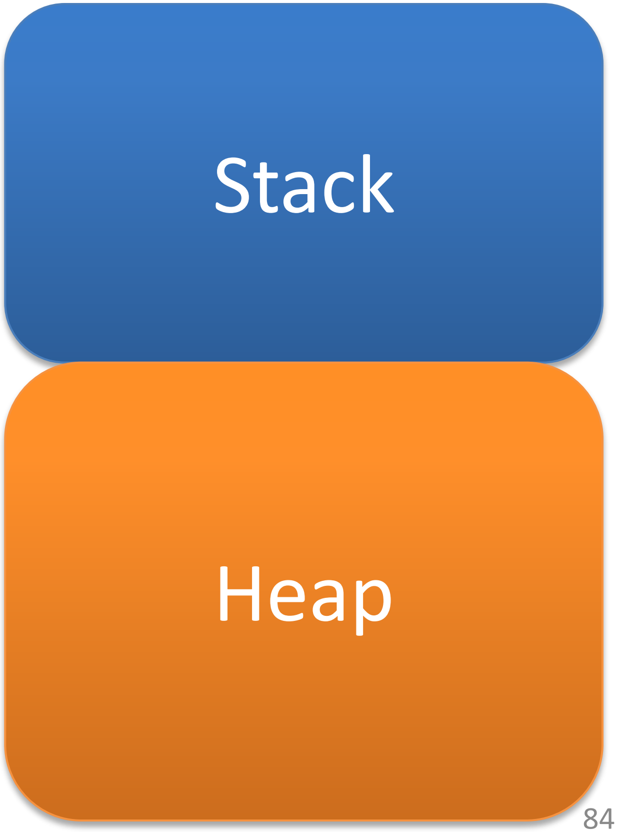
Memory Basics – "Owning" Memory

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



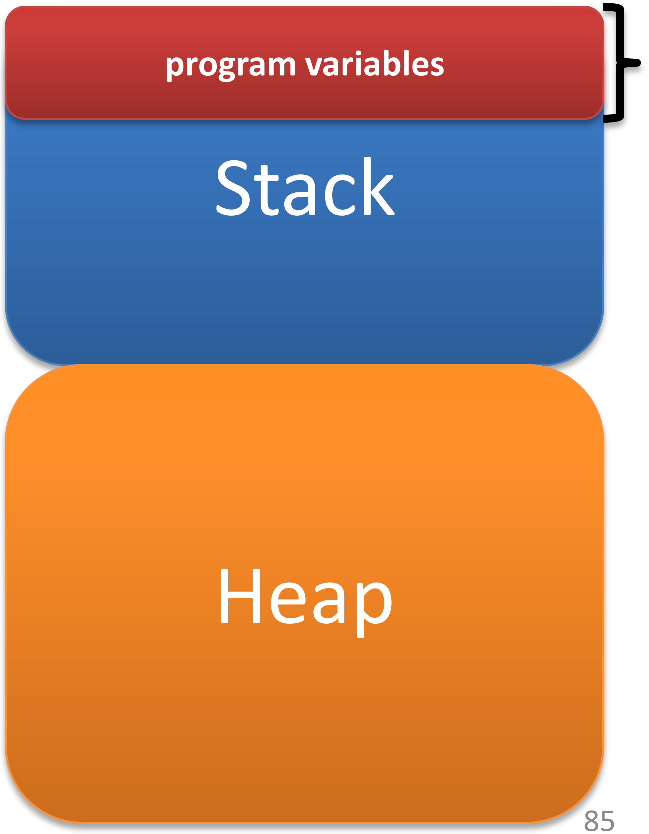
Memory Basics – "Owning" Memory

- 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

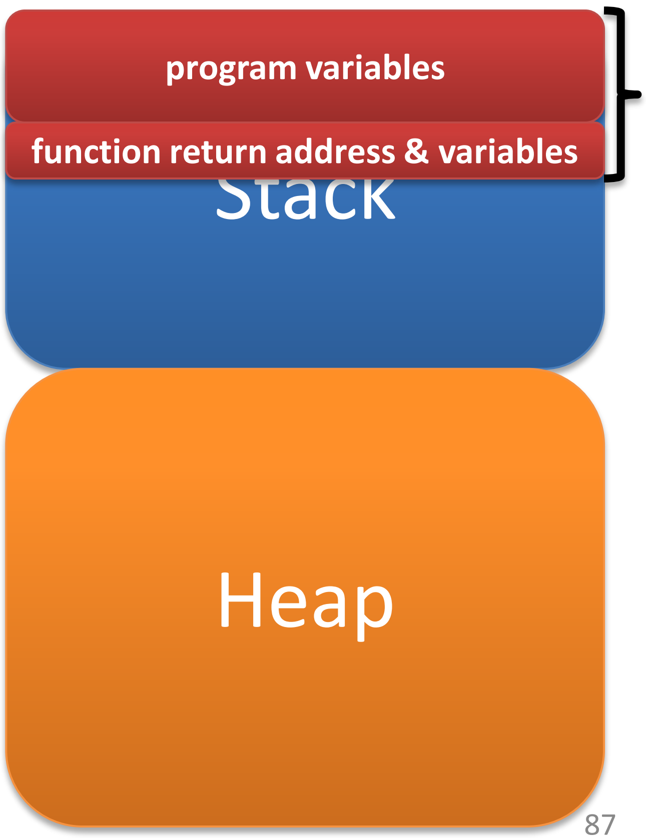


Memory Basics – "Owning" Memory

- 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

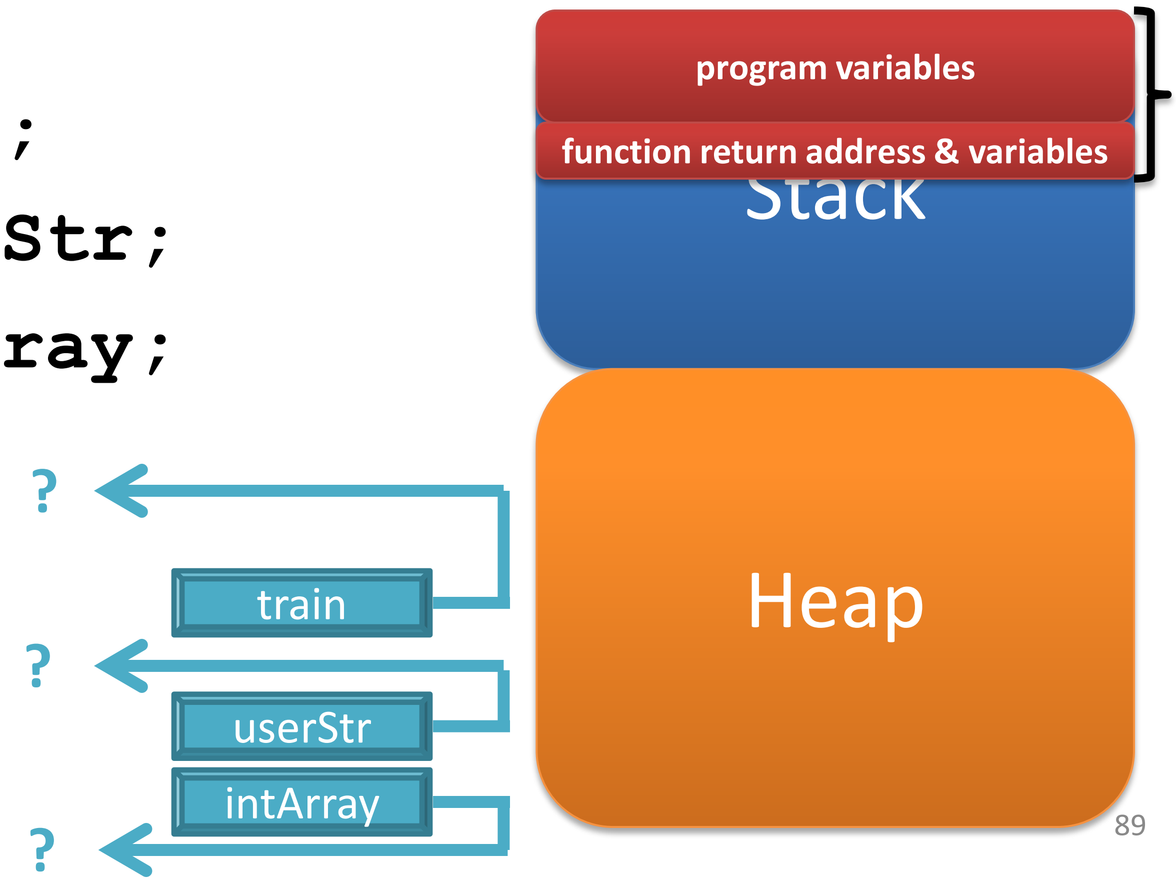
- and every time ***you*** allocate memory, the process gives you space for it on the heap

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

Memory Basics – "Owning" Memory

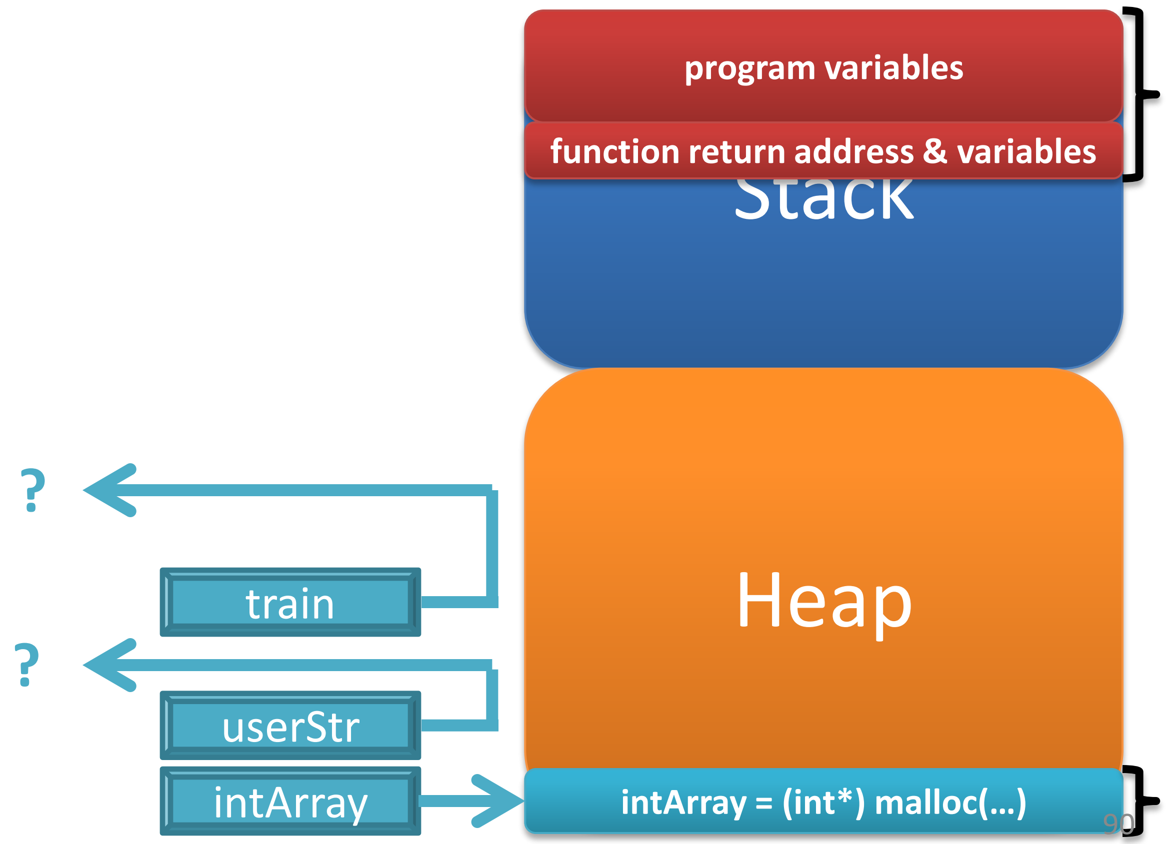
- and every time ***you*** allocate memory, the process gives you space for it on the heap

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



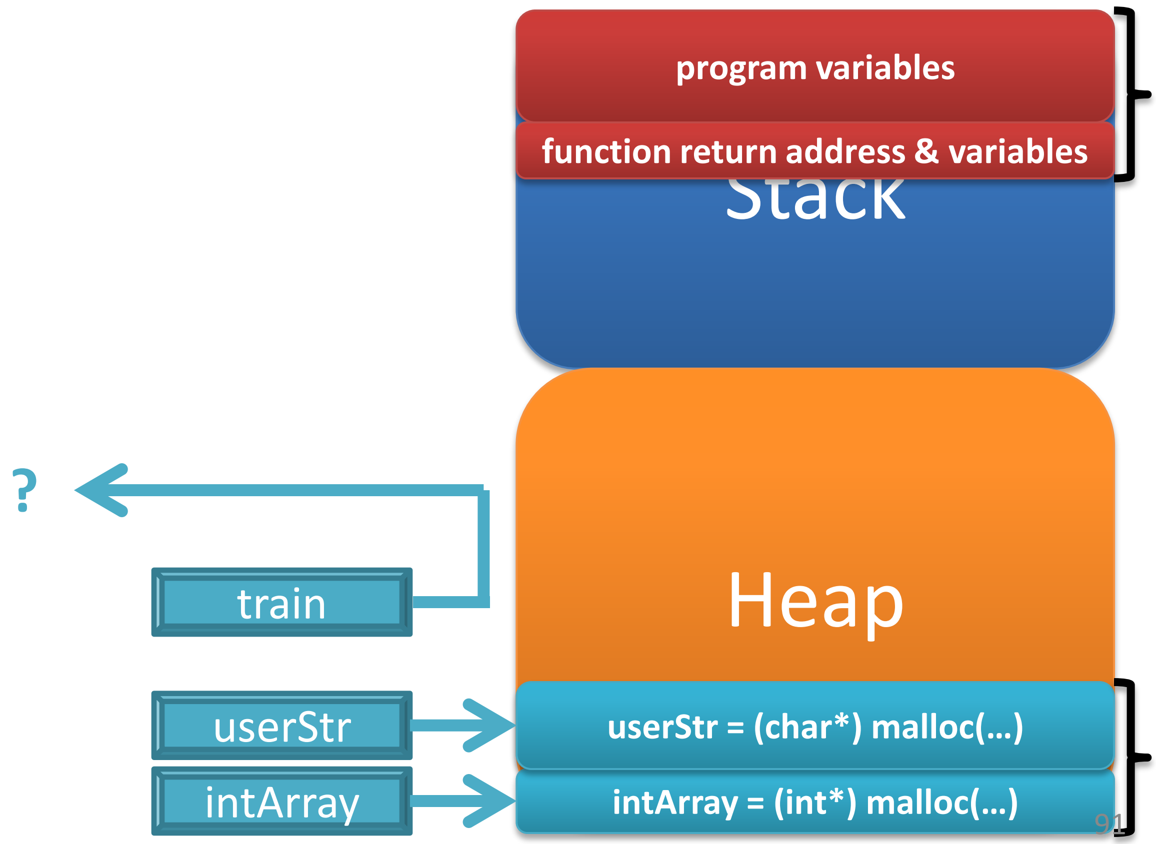
Memory Basics – “Owning” Memory

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



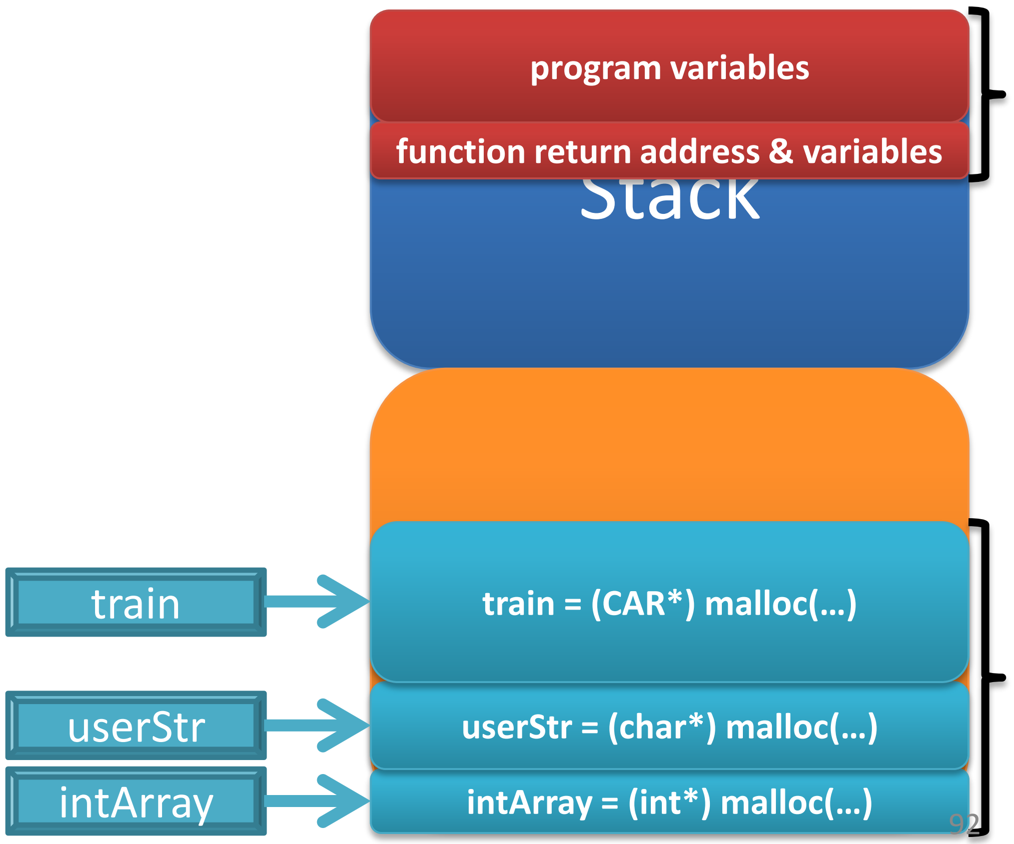
Memory Basics – “Owning” Memory

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



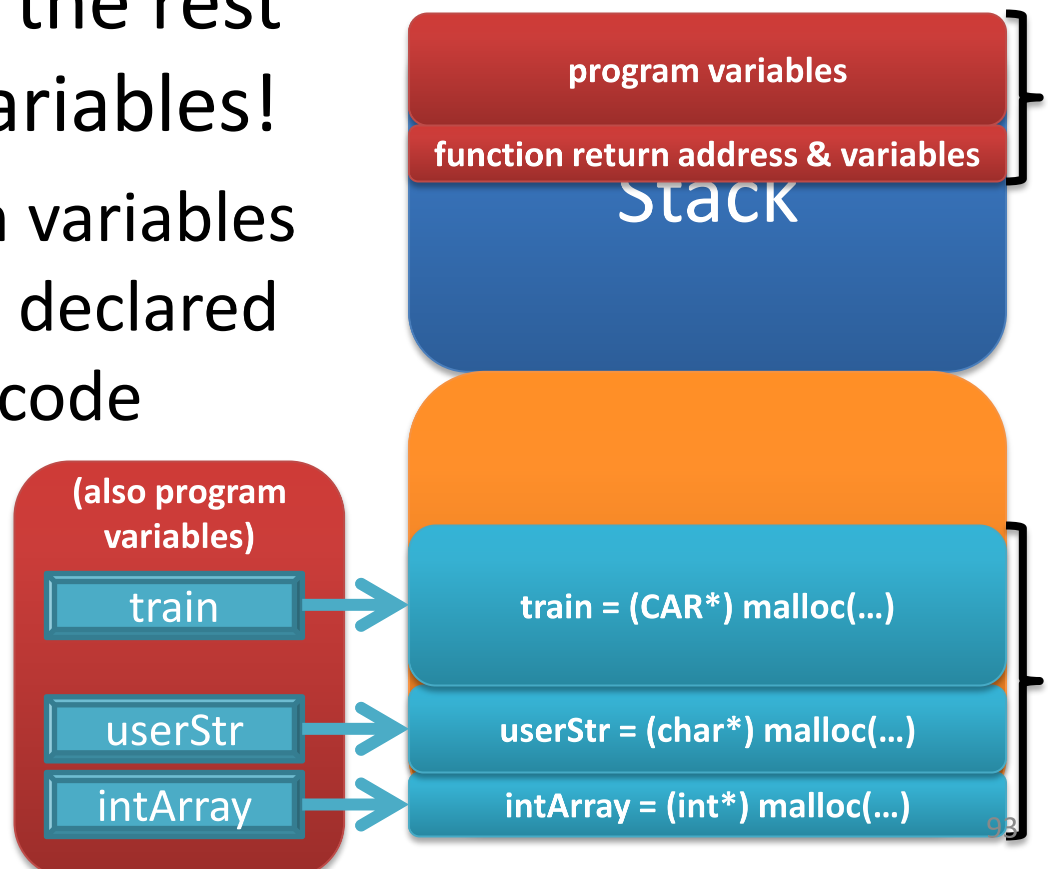
Memory Basics – "Owning" Memory

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



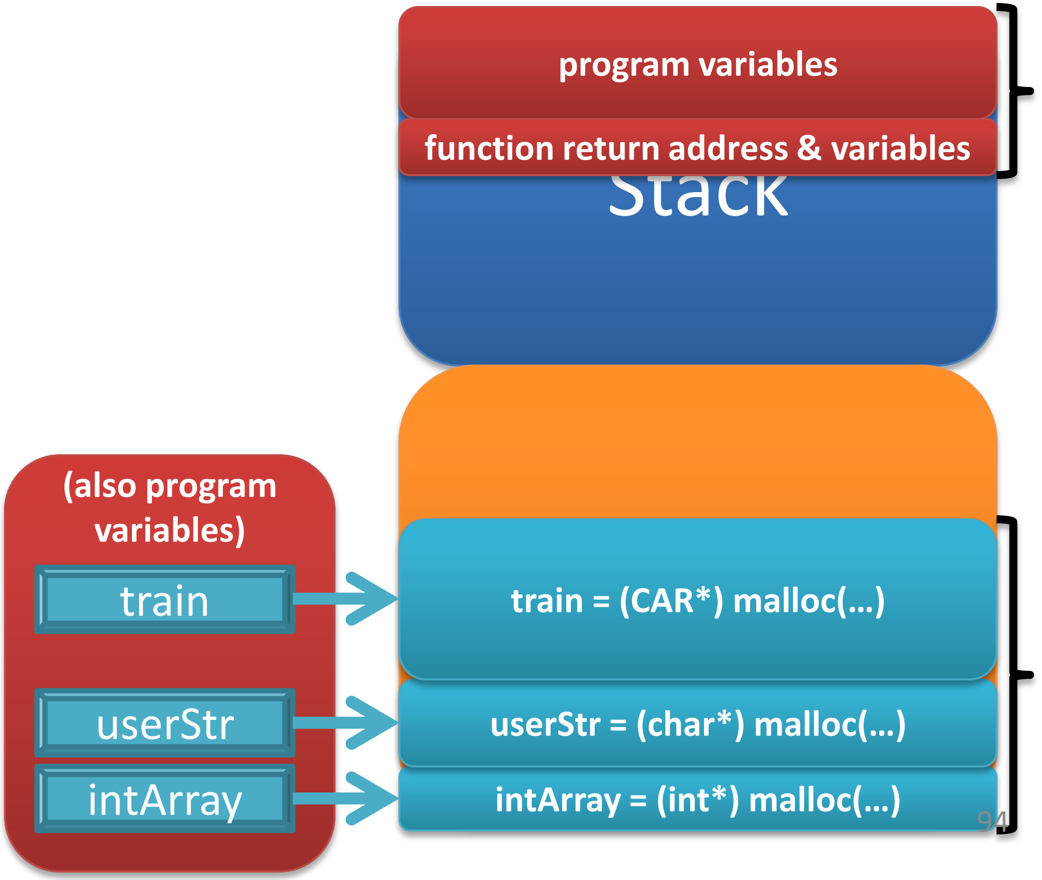
Memory Basics – "Owning" Memory

- 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



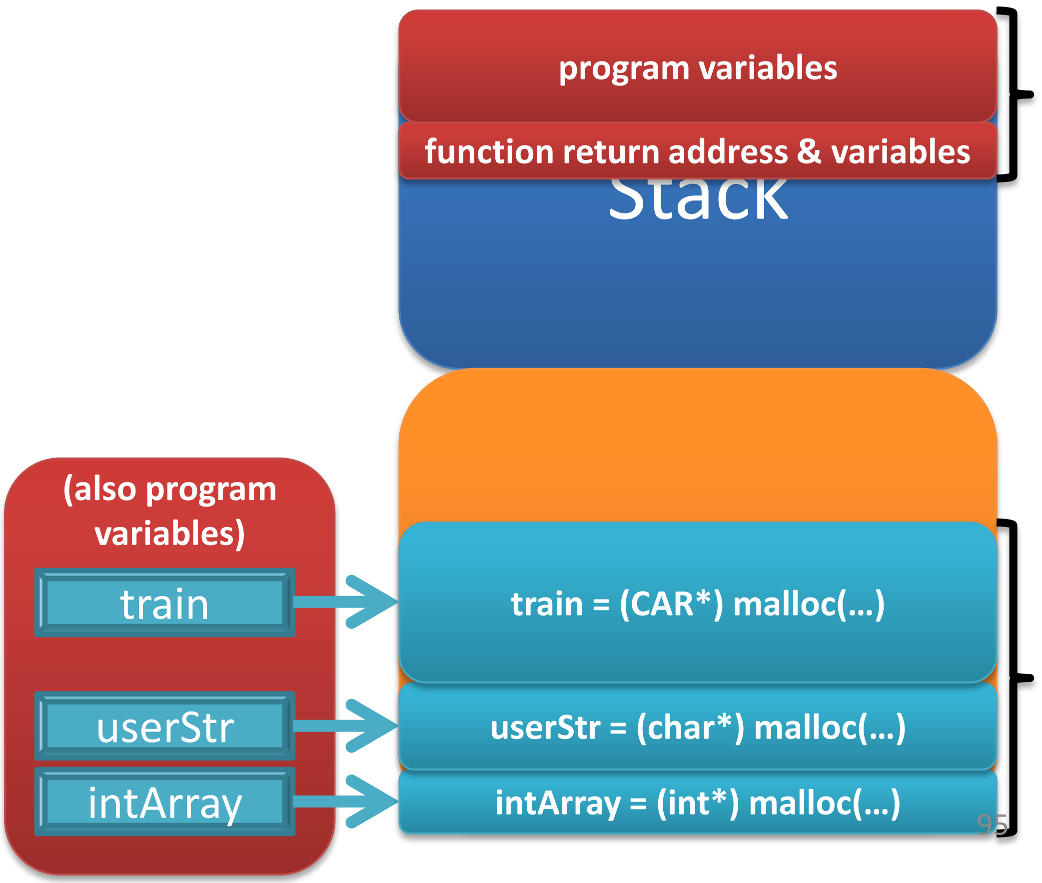
Memory Basics – "Returning" Memory

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



Memory Basics – "Returning" Memory

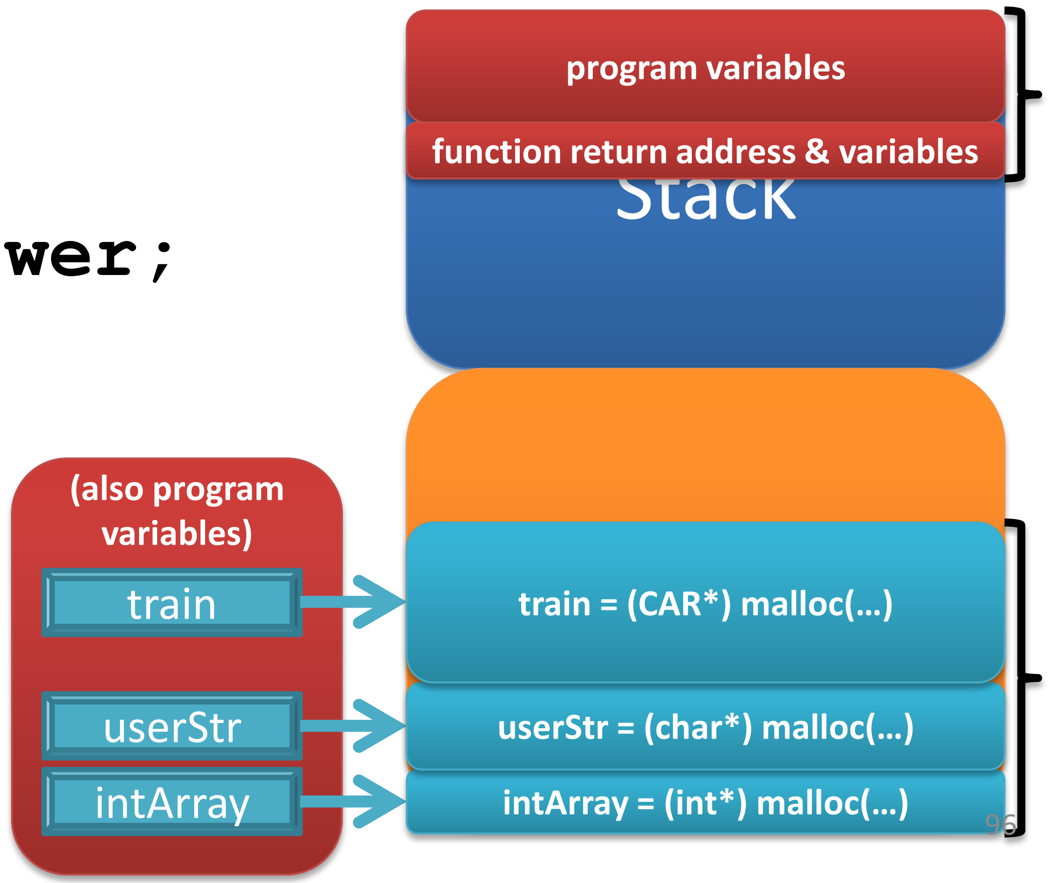
- when a function returns, the program gives that memory on the <u>stack</u> **back** to *the process*



Memory Basics – “Returning” Memory

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

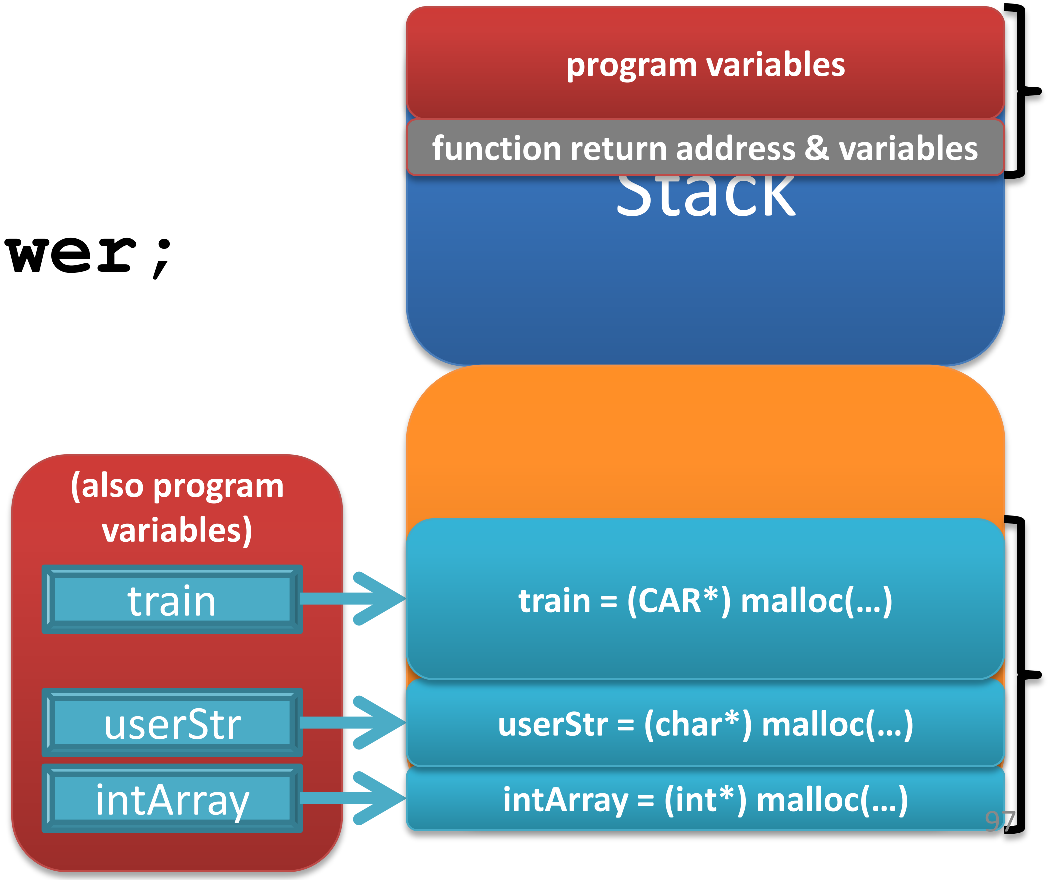
```
return fxnAnswer;
```



Memory Basics – “Returning” Memory

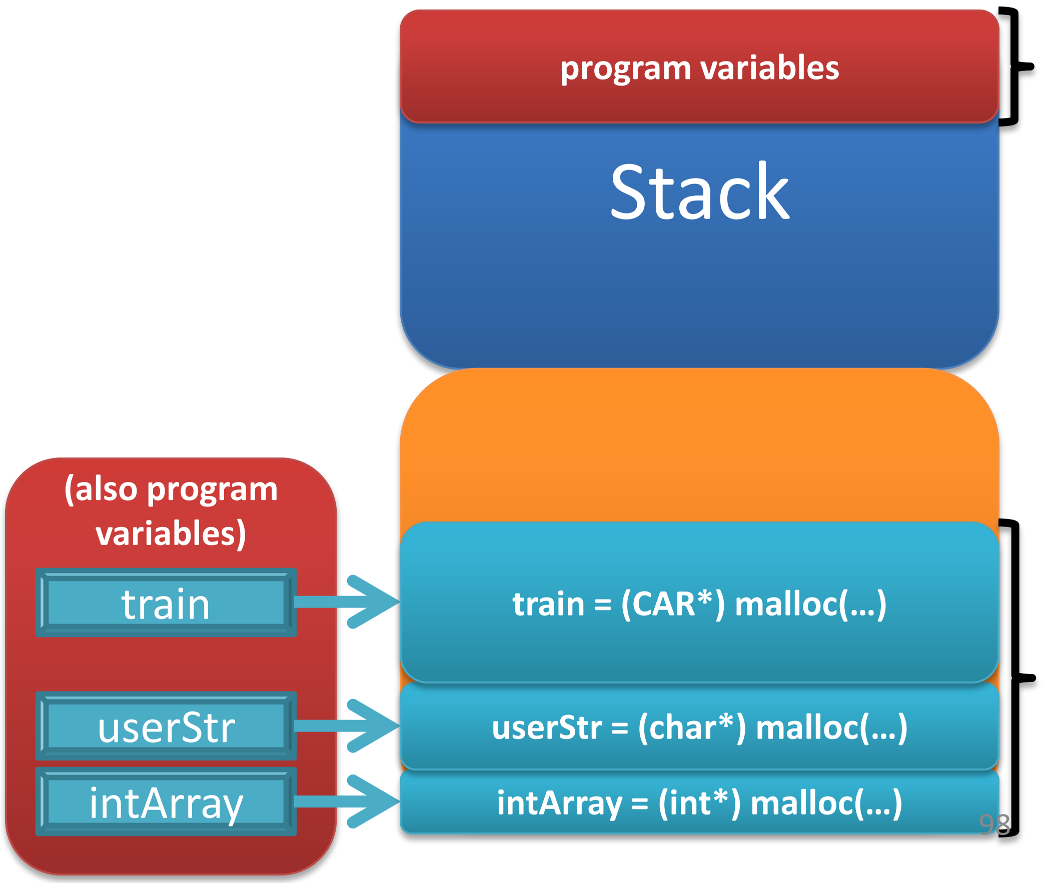
- when a function returns, the program gives that memory on the stack **back** to *the process*

```
return fxnAnswer;
```



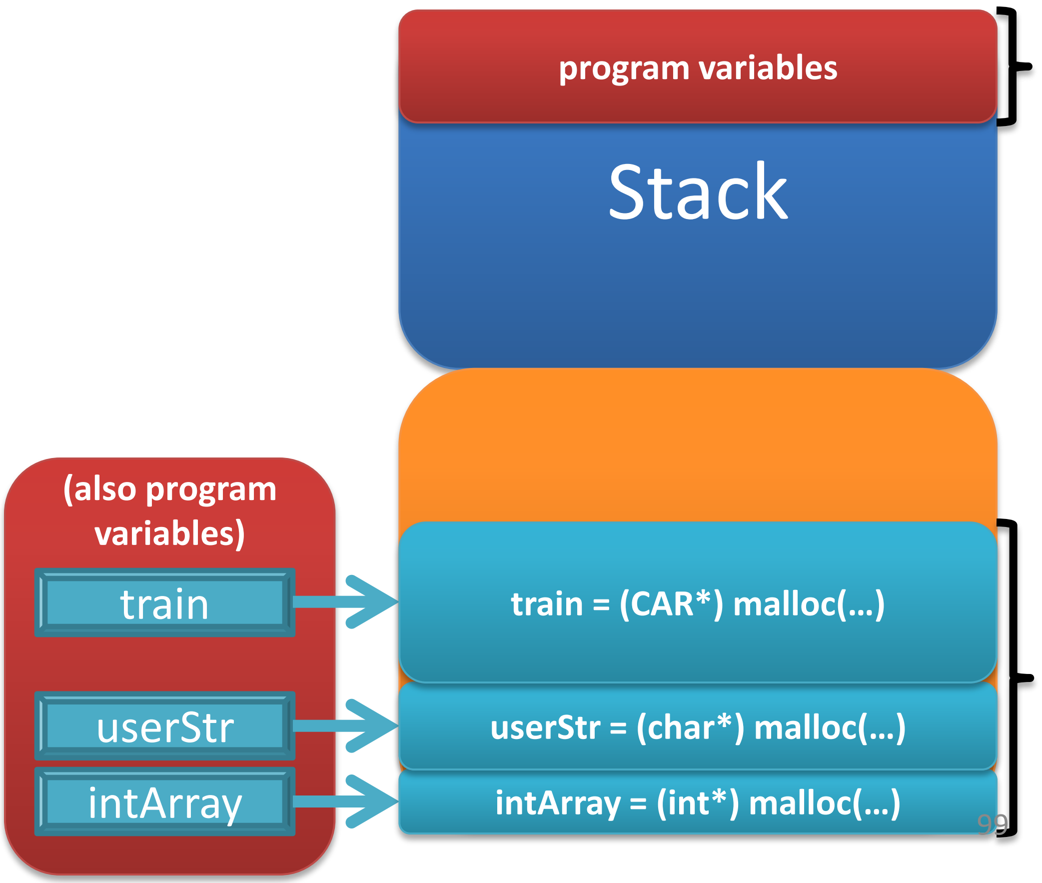
Memory Basics – “Returning” Memory

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



Memory Basics – "Returning" Memory

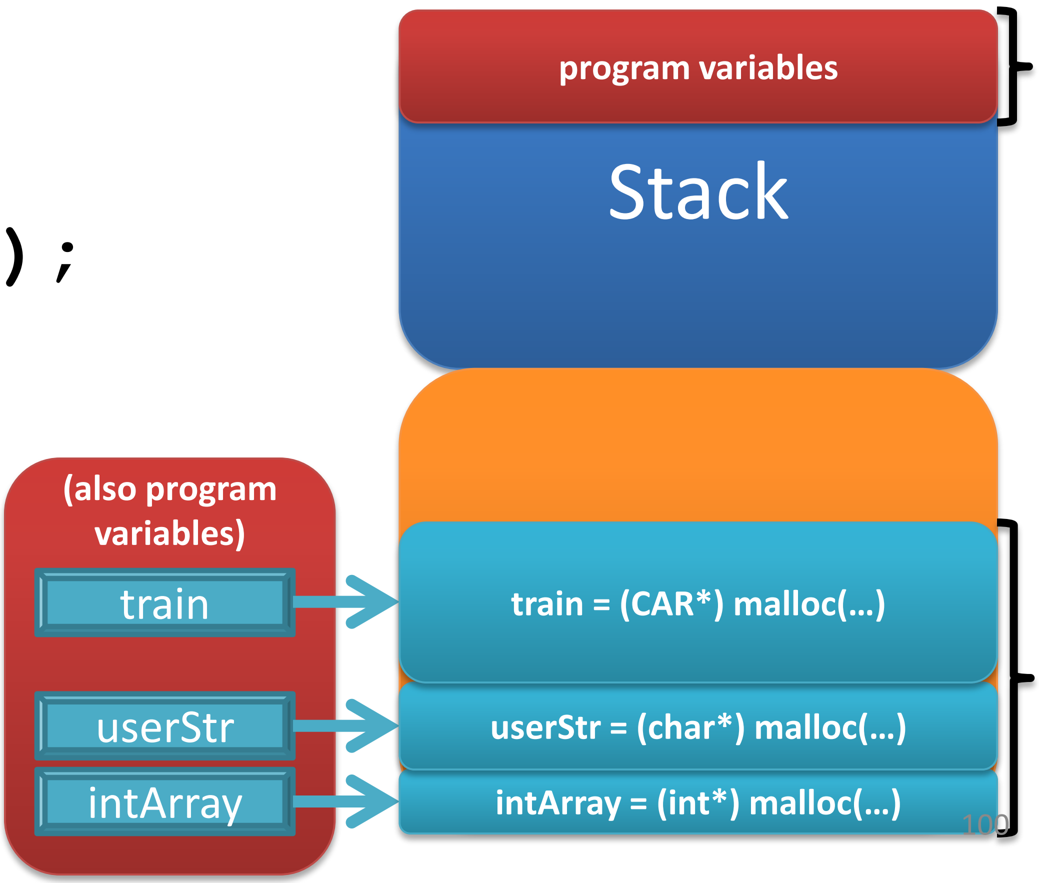
- and when you use free(), the memory you had on the <u>heap</u> is given **back** to *the process*



Memory Basics – "Returning" Memory

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

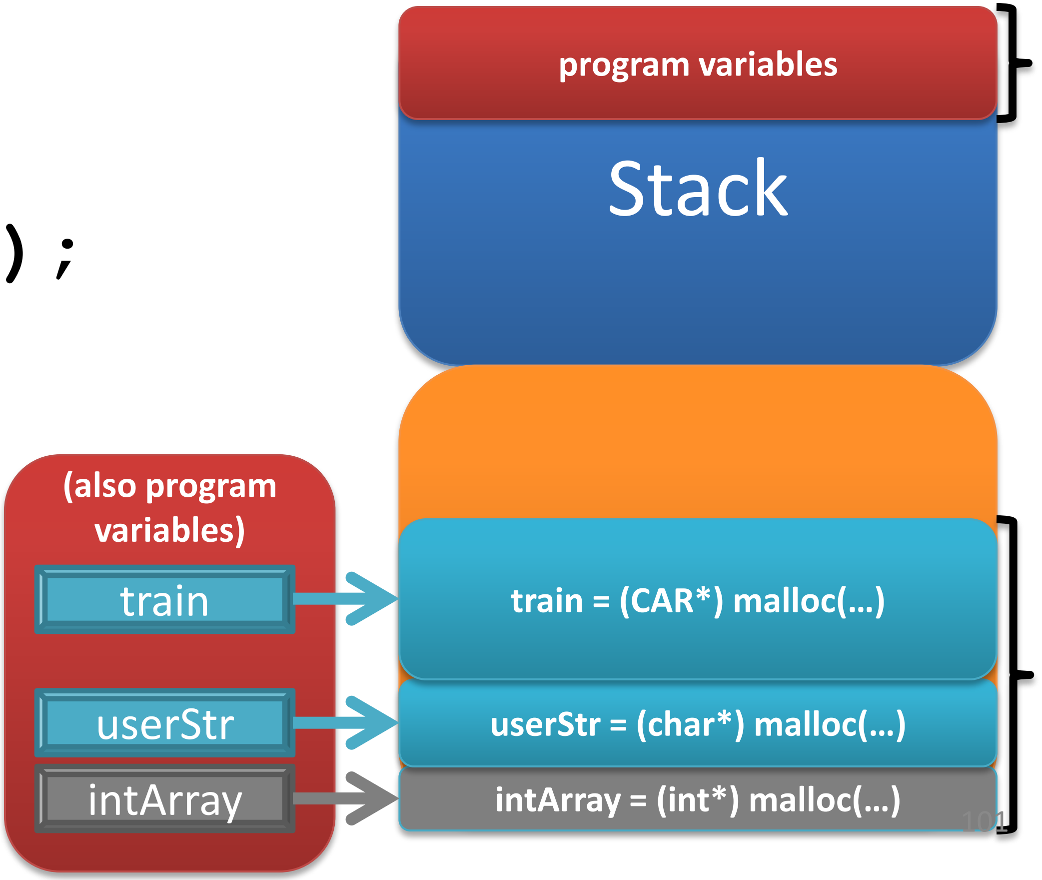
```
free(intArray);
```



Memory Basics – "Returning" Memory

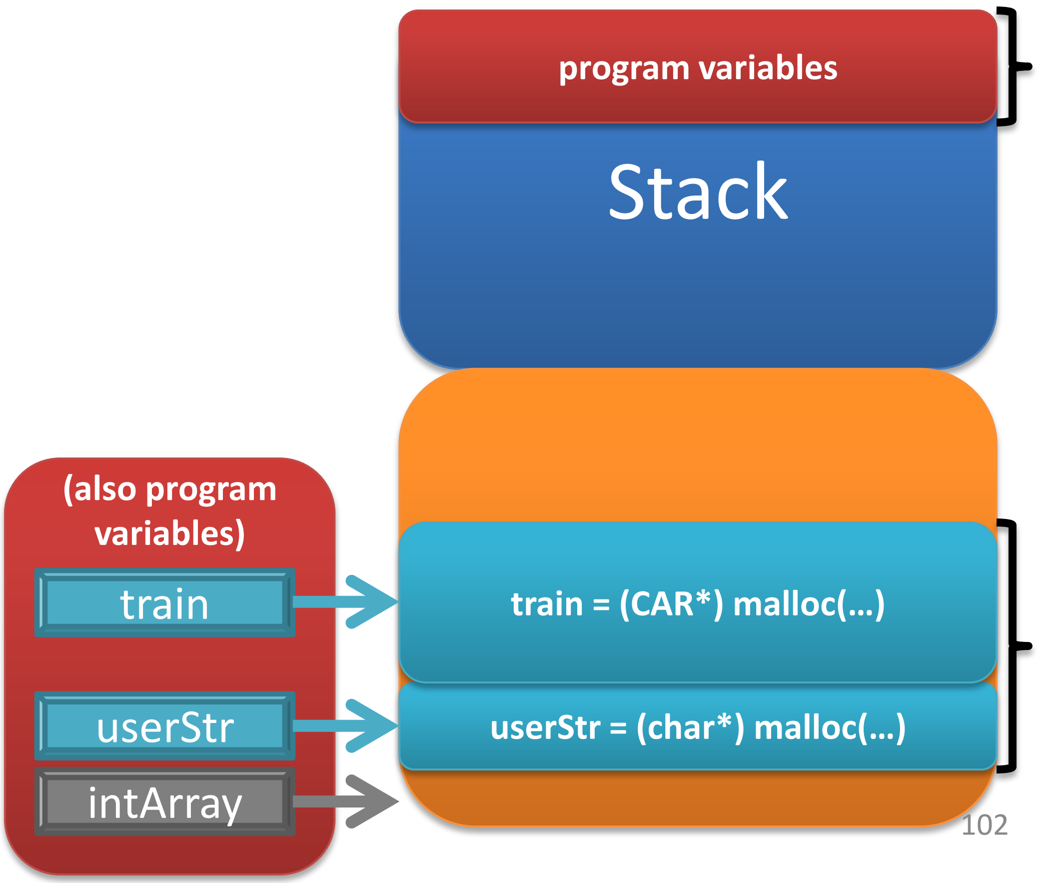
- and when you use free(), the memory you had on the <u>heap</u> is given **back** to *the process*

```
free(intArray);
```



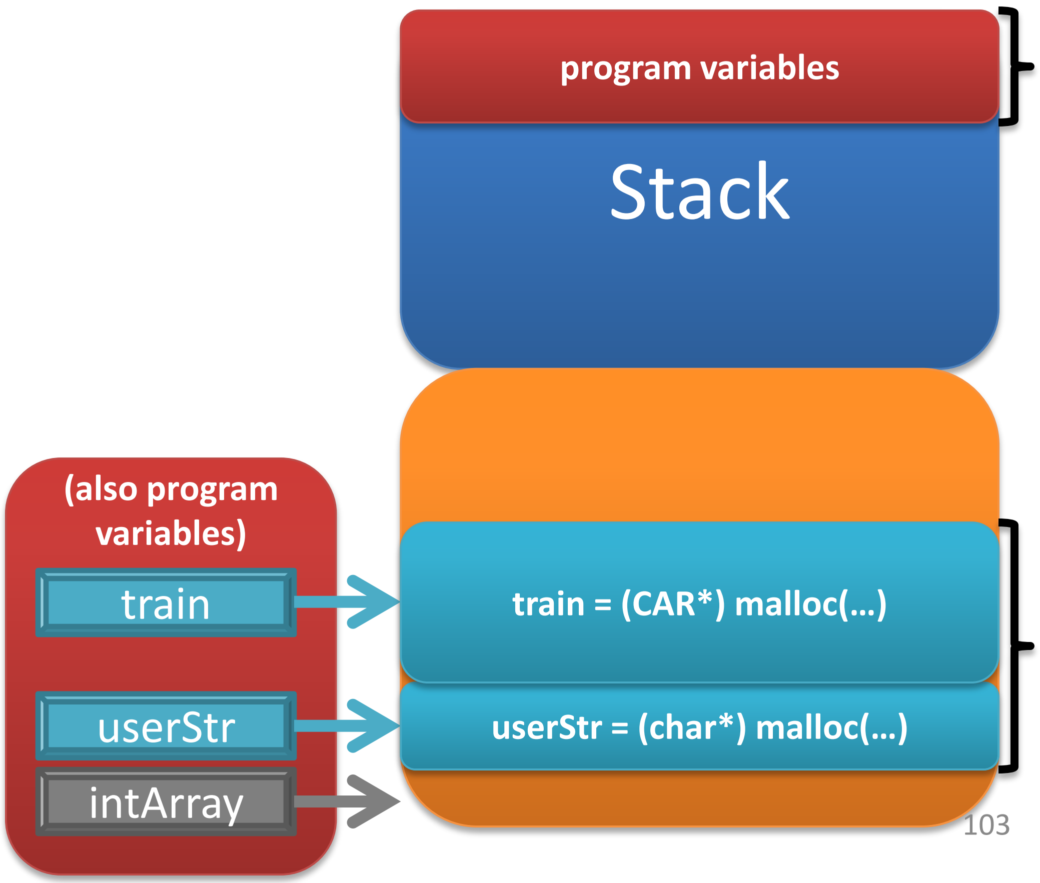
Memory Basics – “Returning” Memory

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



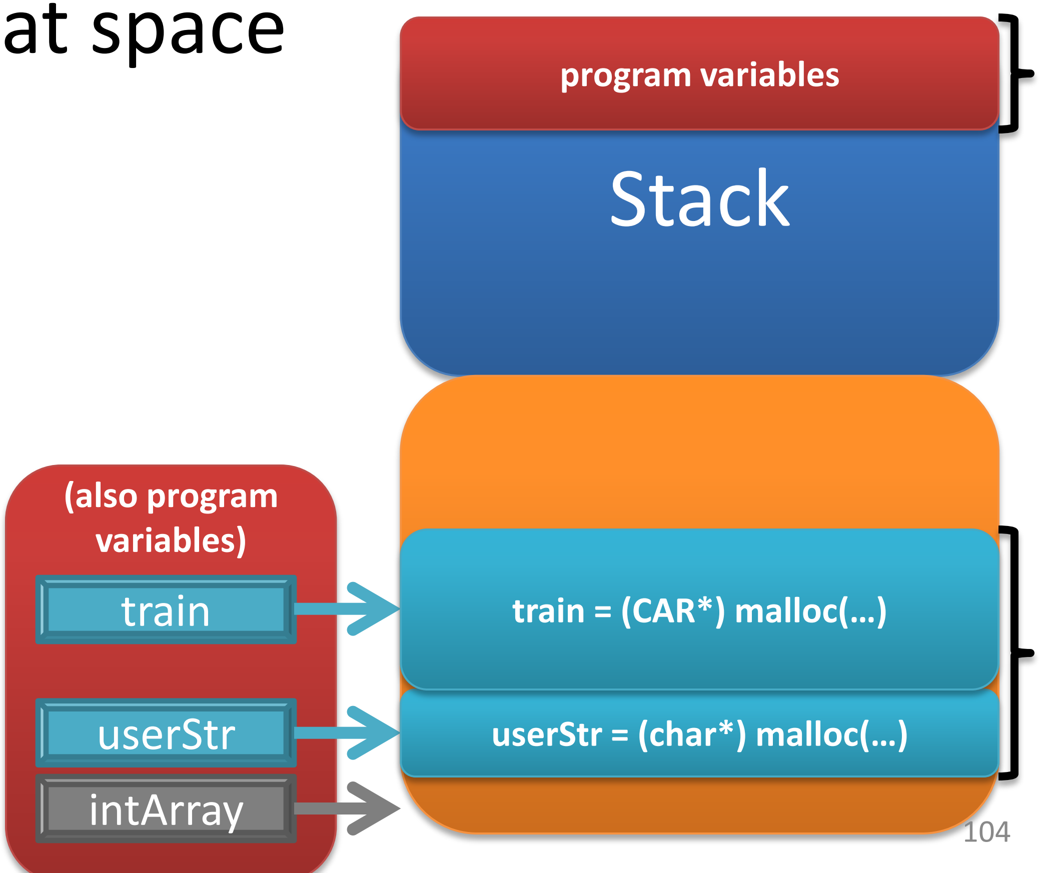
Memory Basics – Memory Errors

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



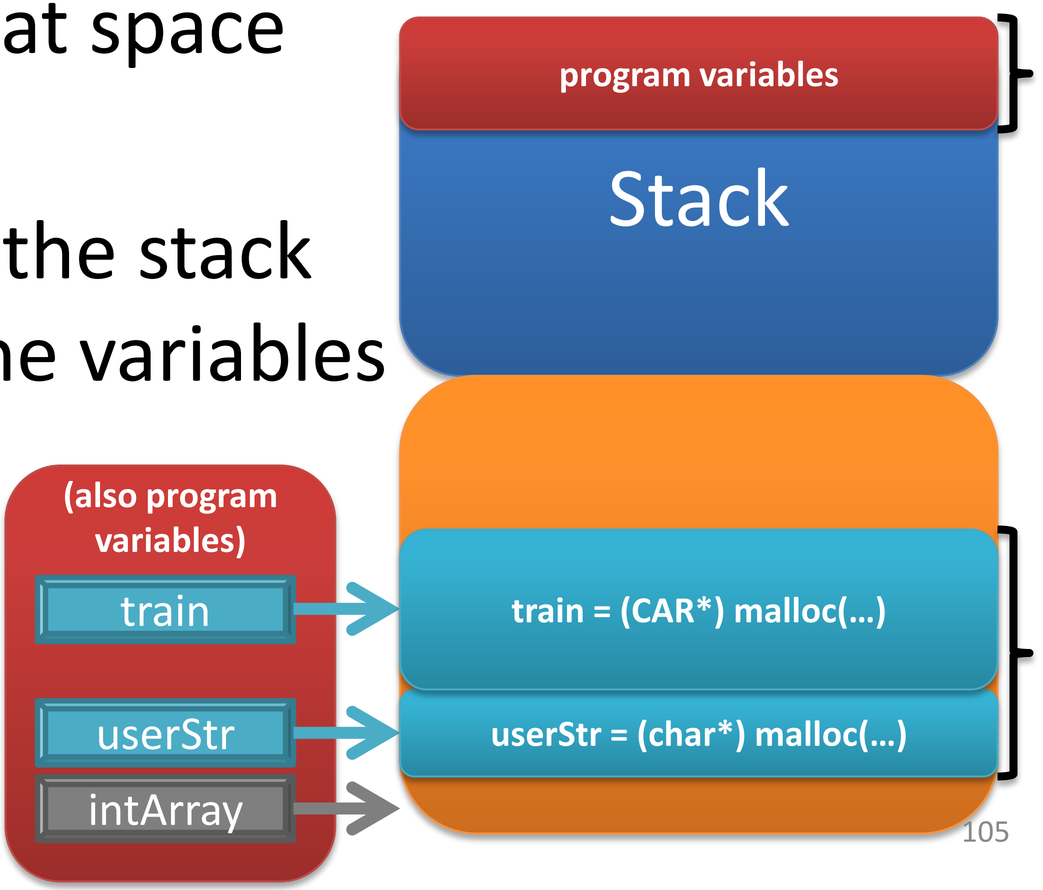
Memory Basics – Memory Errors

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



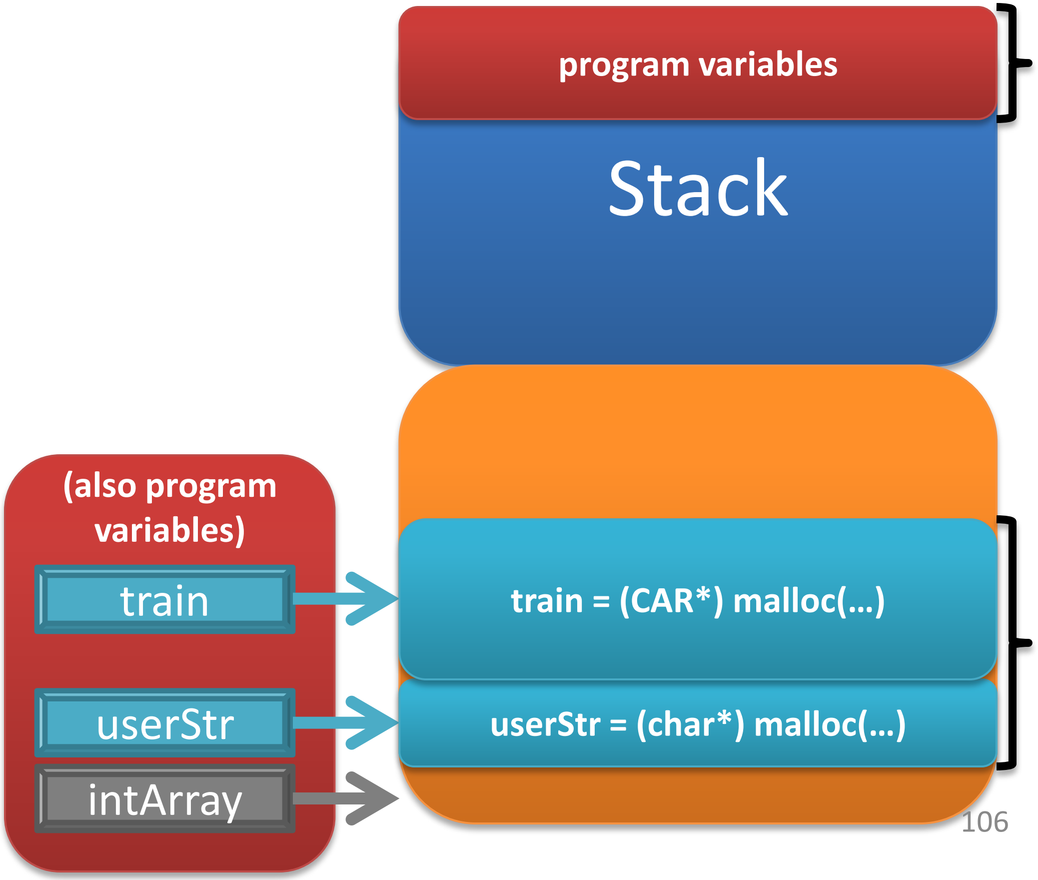
Memory Basics – Memory Errors

- 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



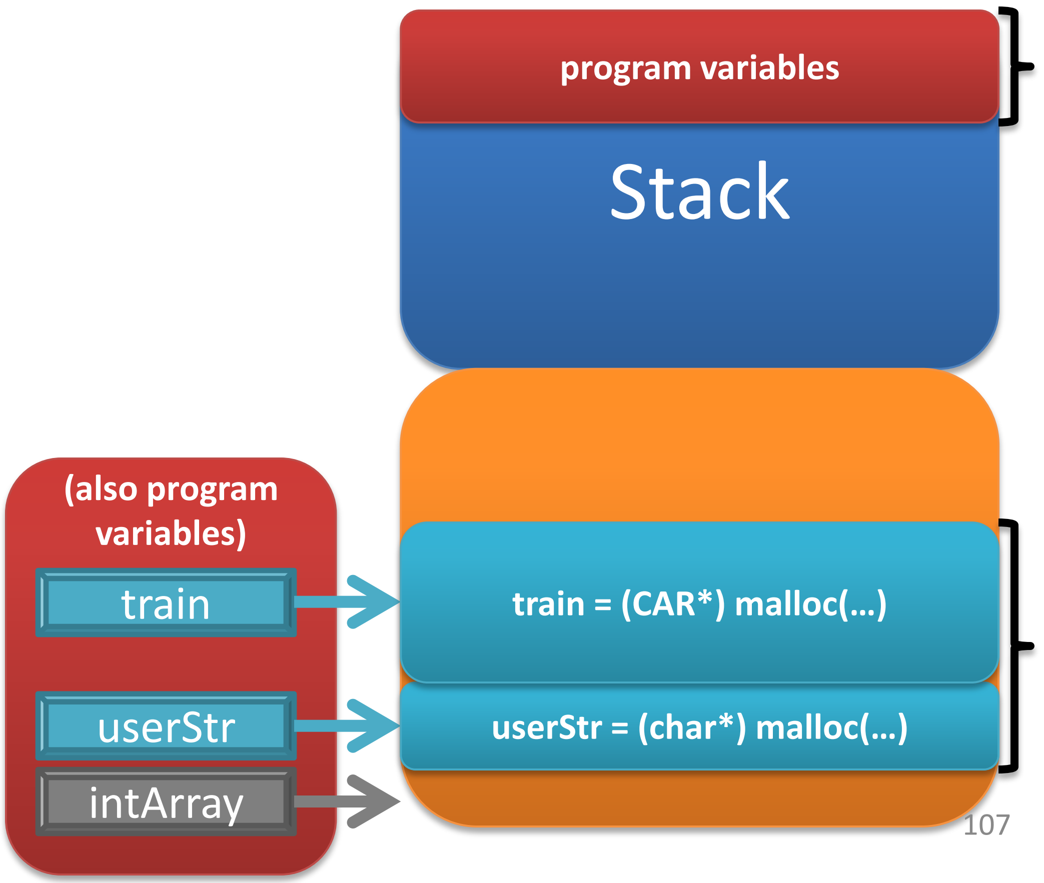
Memory Basics – Memory Errors

- intArray is now a



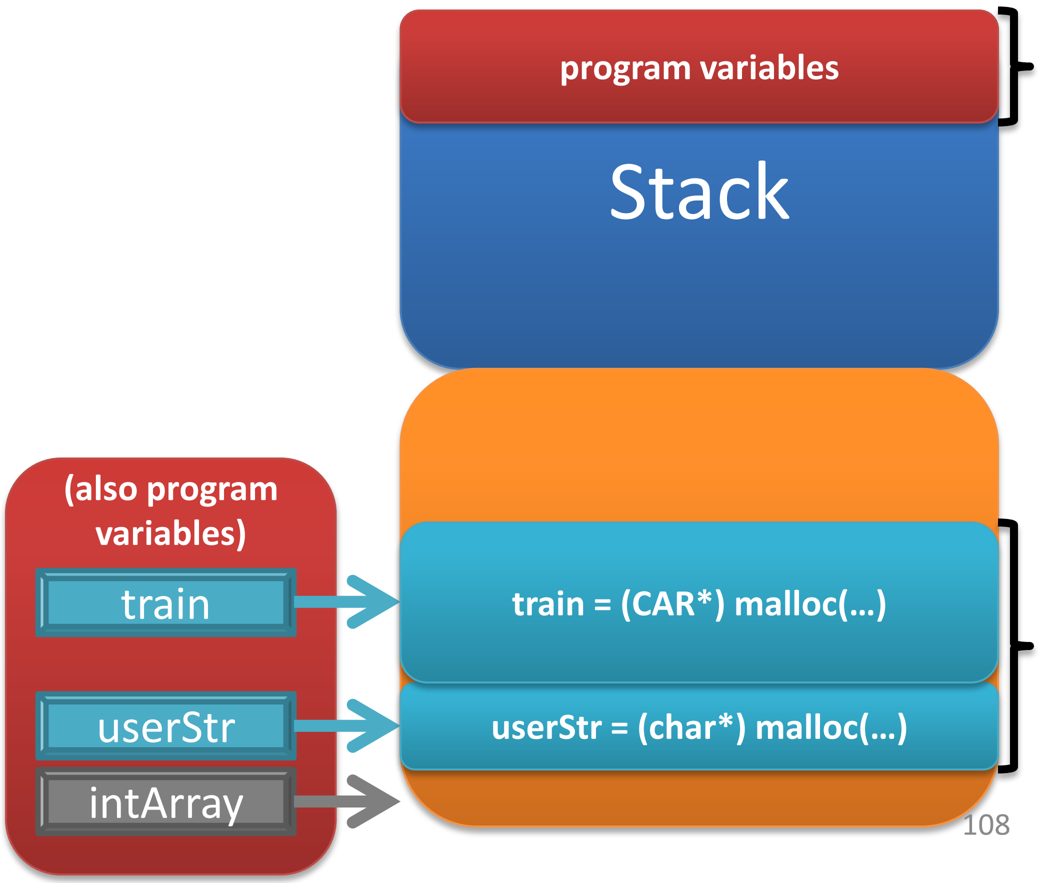
Memory Basics – Memory Errors

- intArray is now a **dangling pointer**



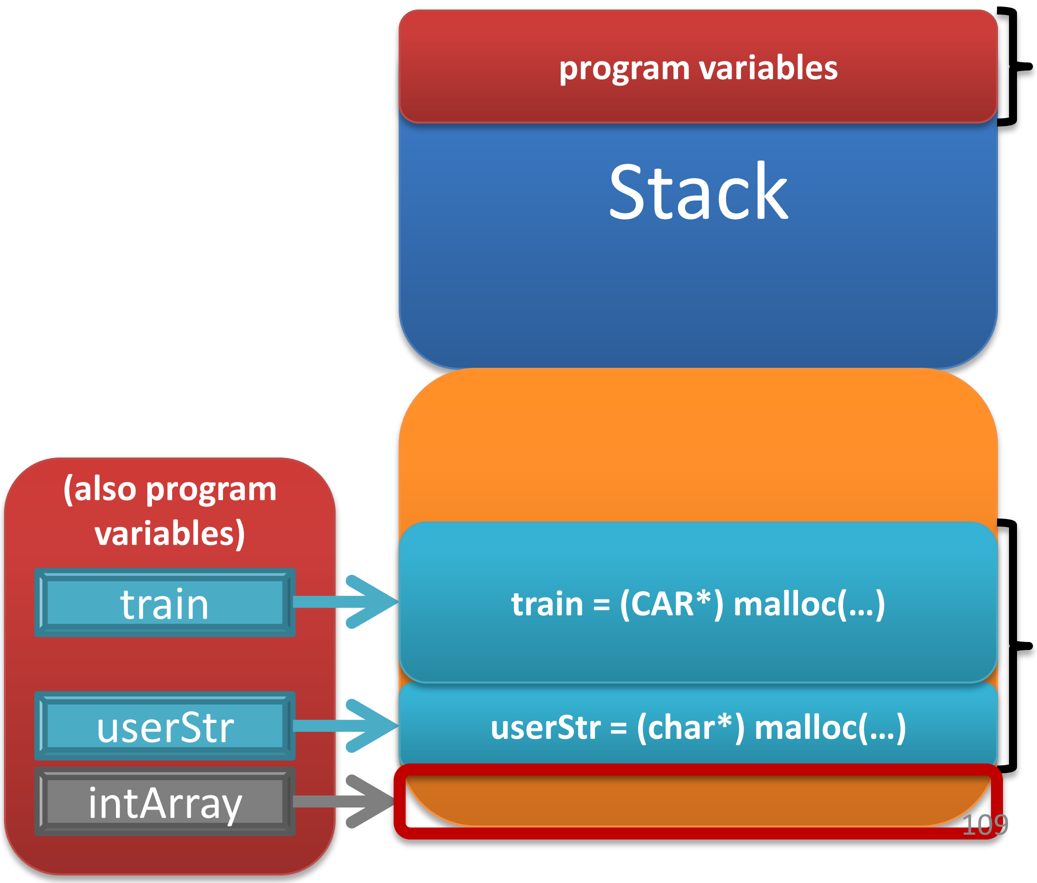
Memory Basics – Memory Errors

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



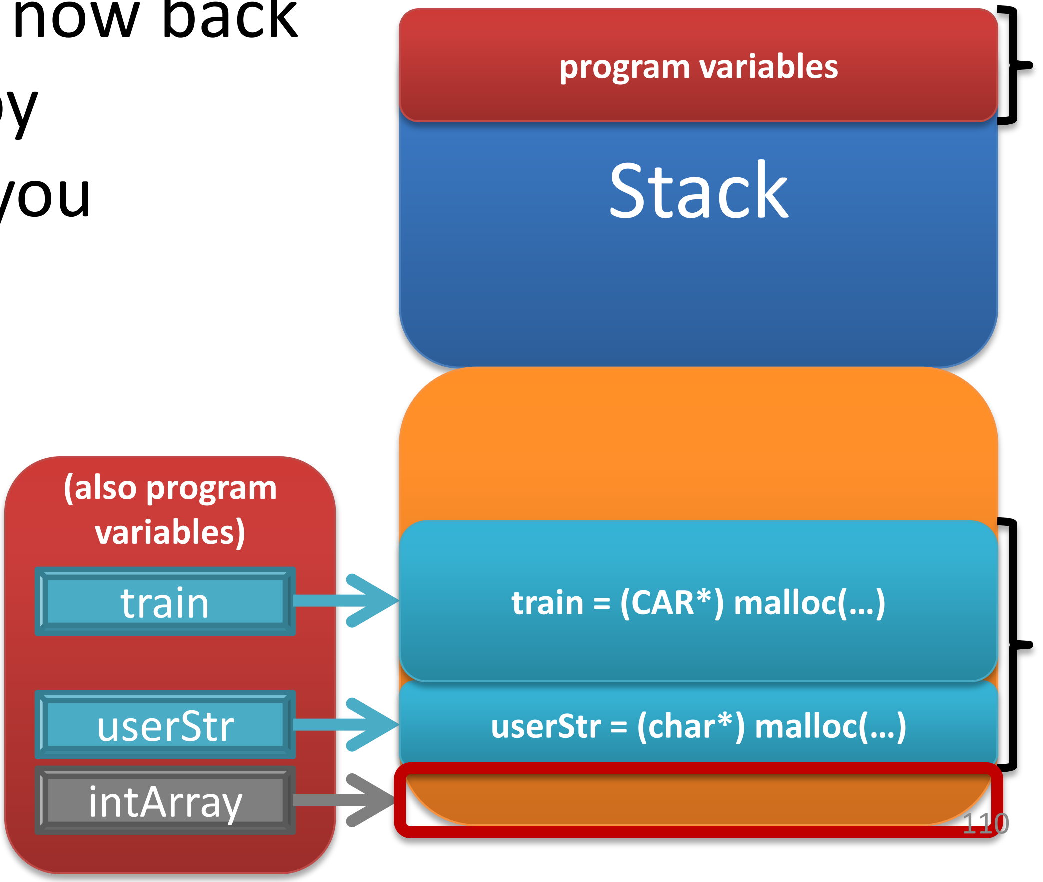
Memory Basics – Memory Errors

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



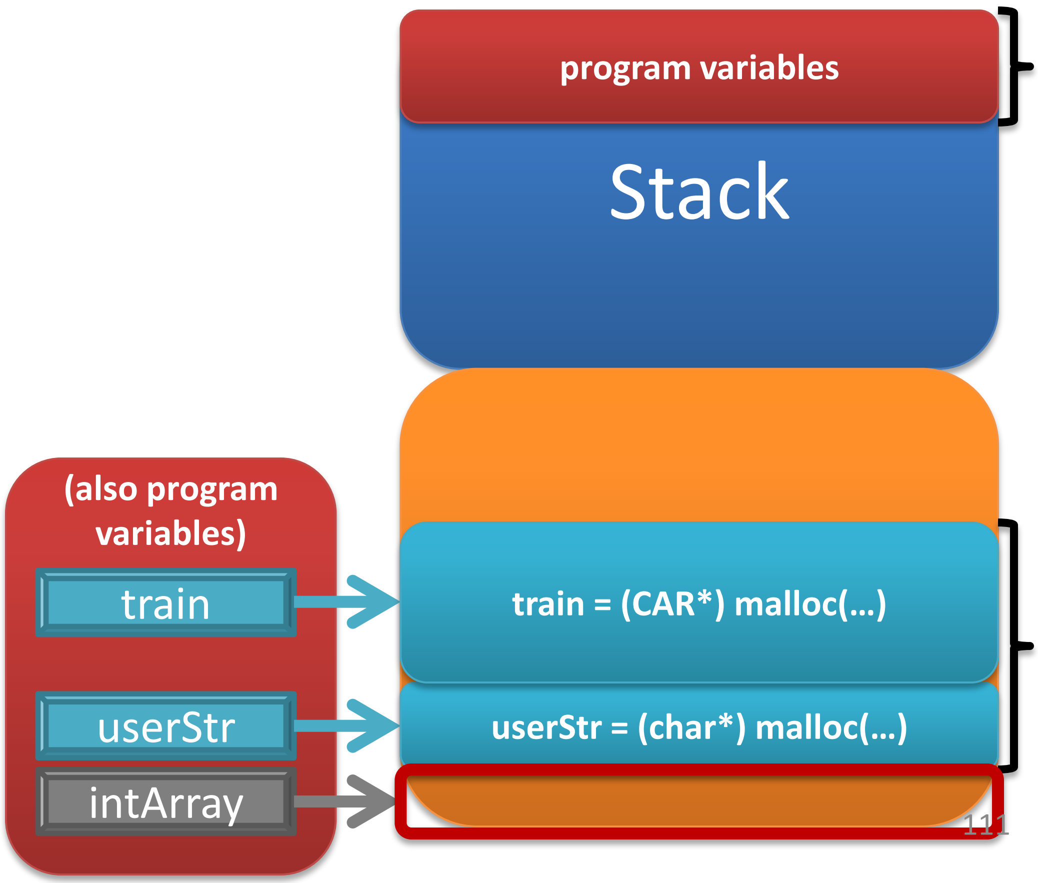
Memory Basics – Memory Errors

- 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



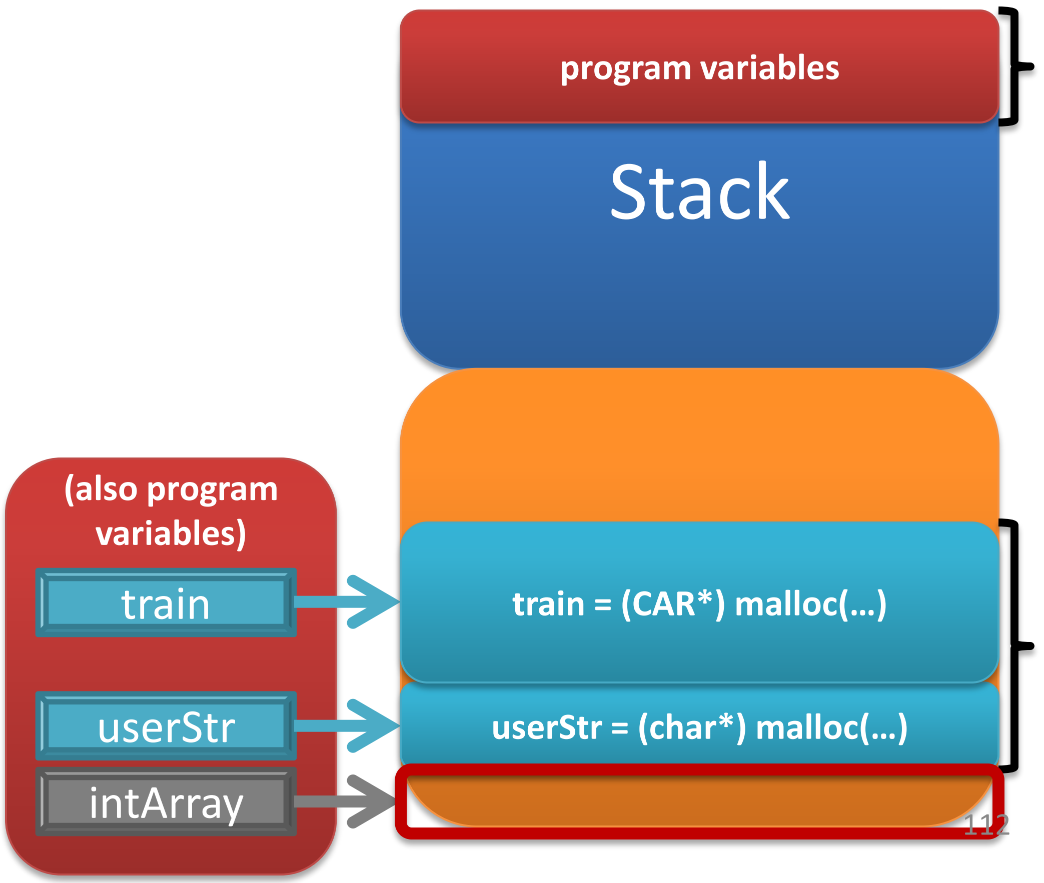
Memory Basics – Memory Errors

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



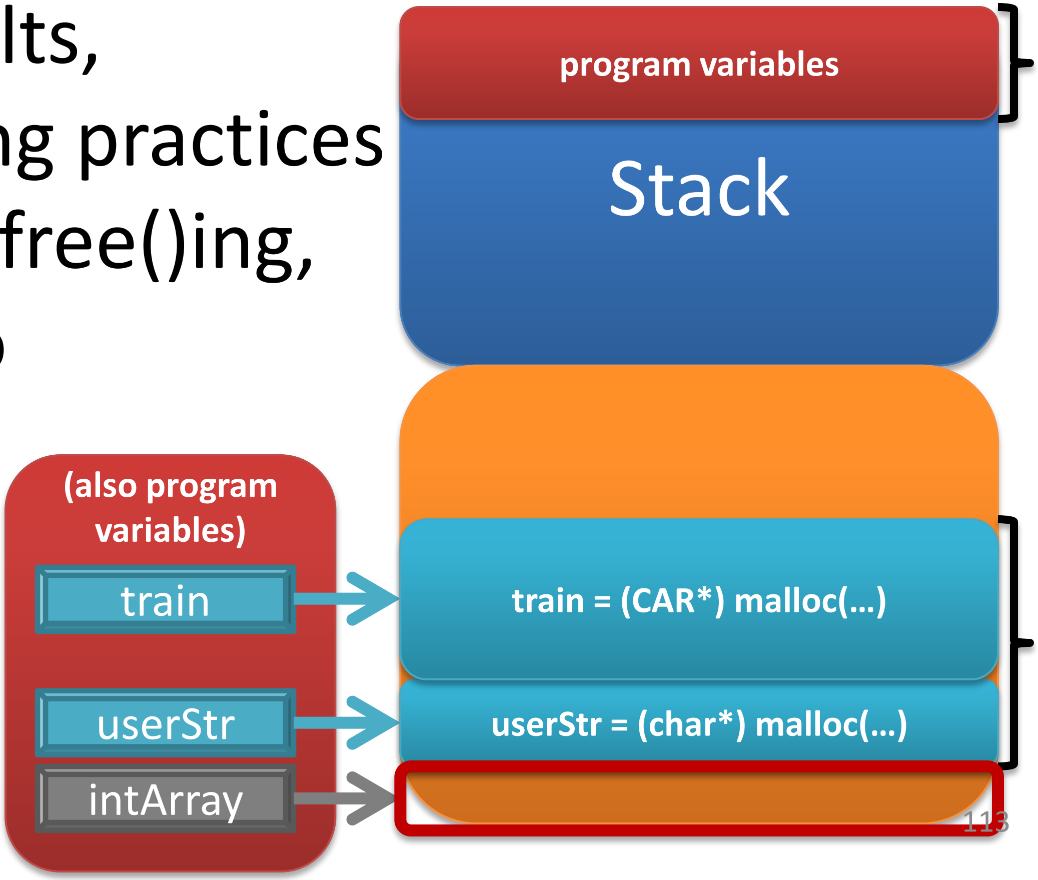
Memory Basics – Memory Errors

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



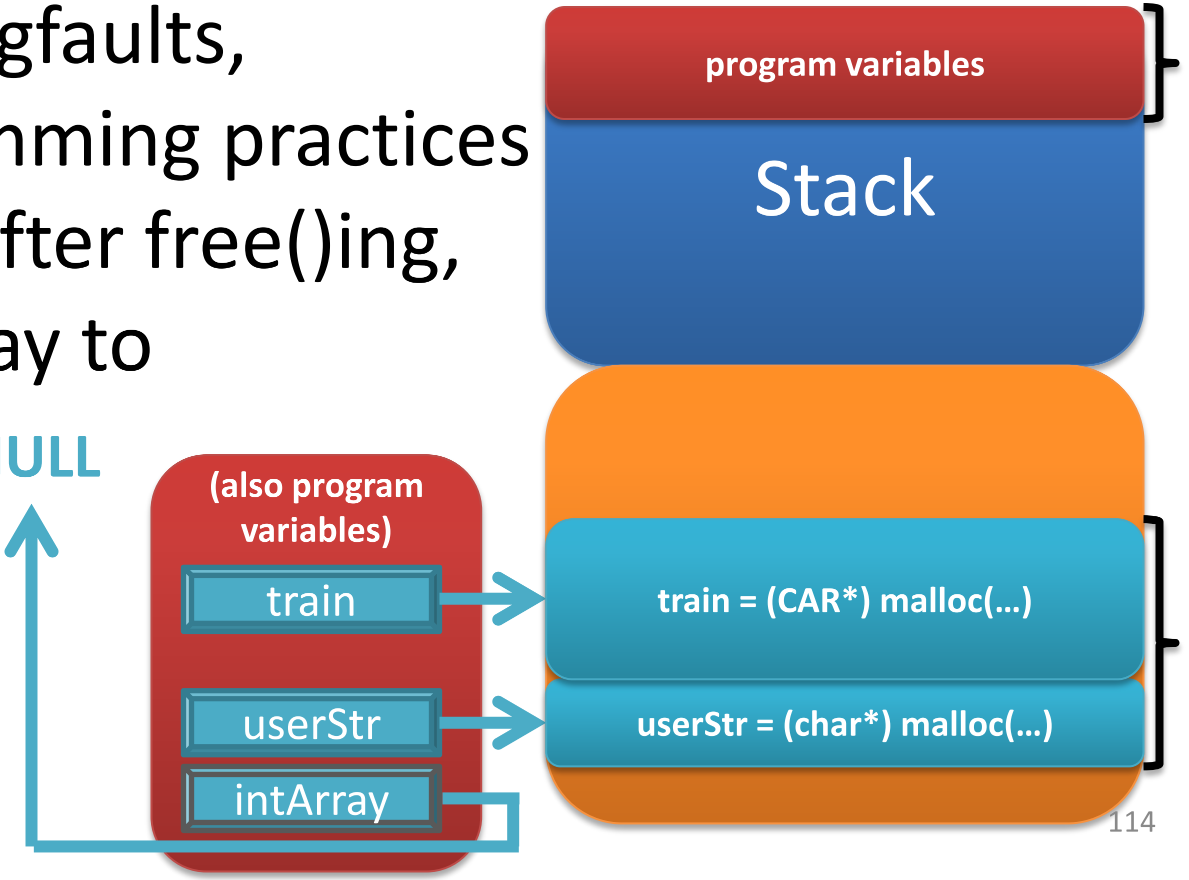
Memory Basics – Memory Errors

- 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

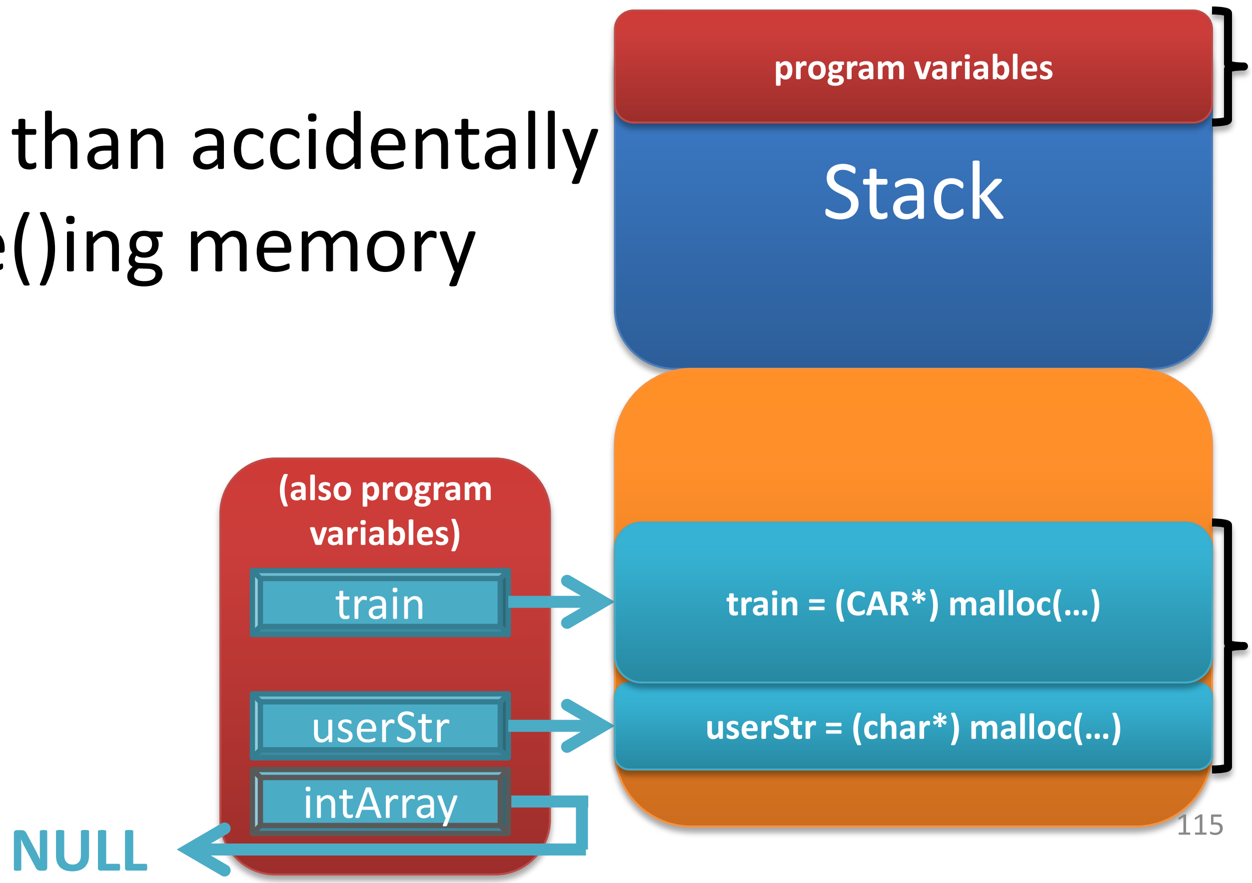
- if we tried to free() intArray's memory again
- we would get a **SEGFAULT**
- to prevent segfaults, good programming practices dictate that after free()ing, we set intArray to be equal to **NULL**



Memory Basics – Memory Errors

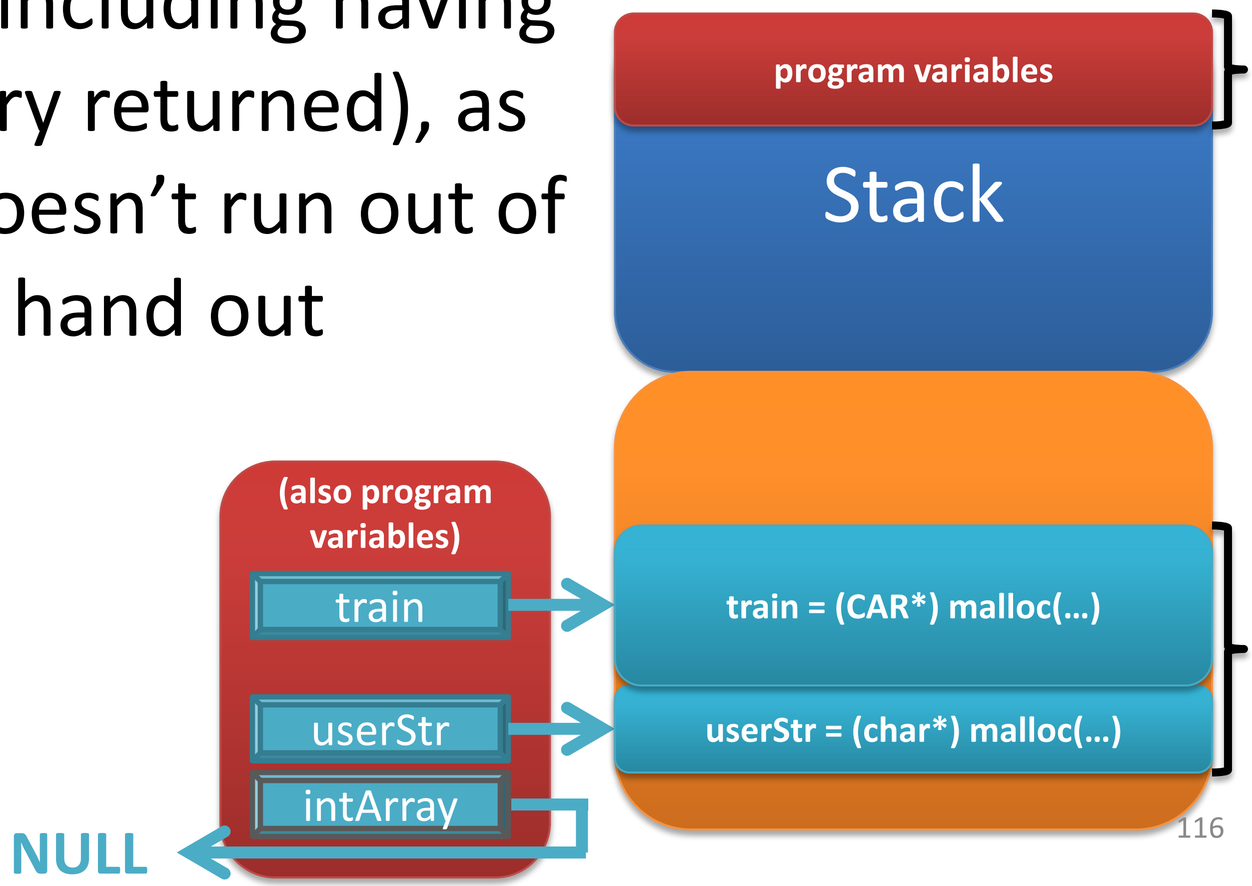
- NOTE: if you try to free a NULL pointer, no action occurs (and it doesn't segfault!)

- much safer than accidentally double free()ing memory



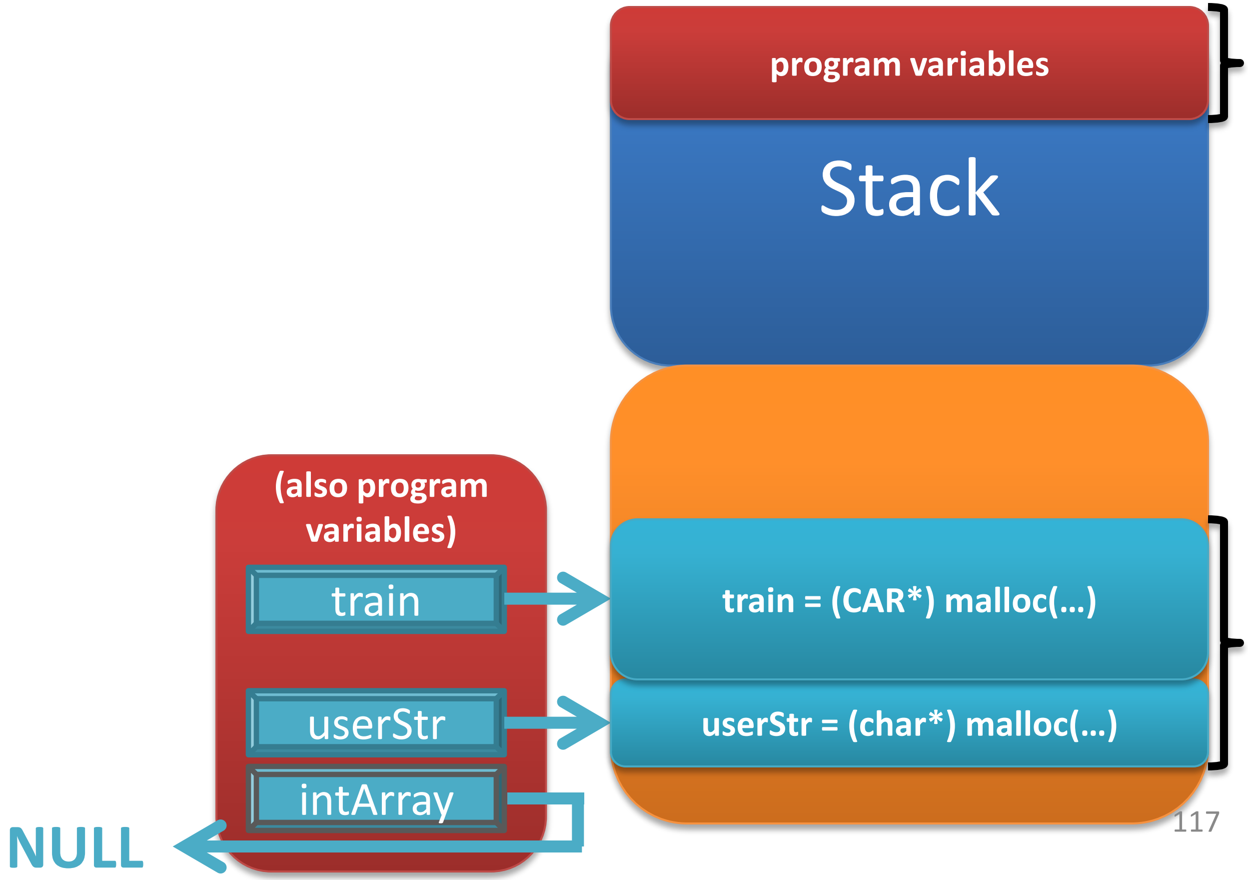
Memory Basics – Running Out

- 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

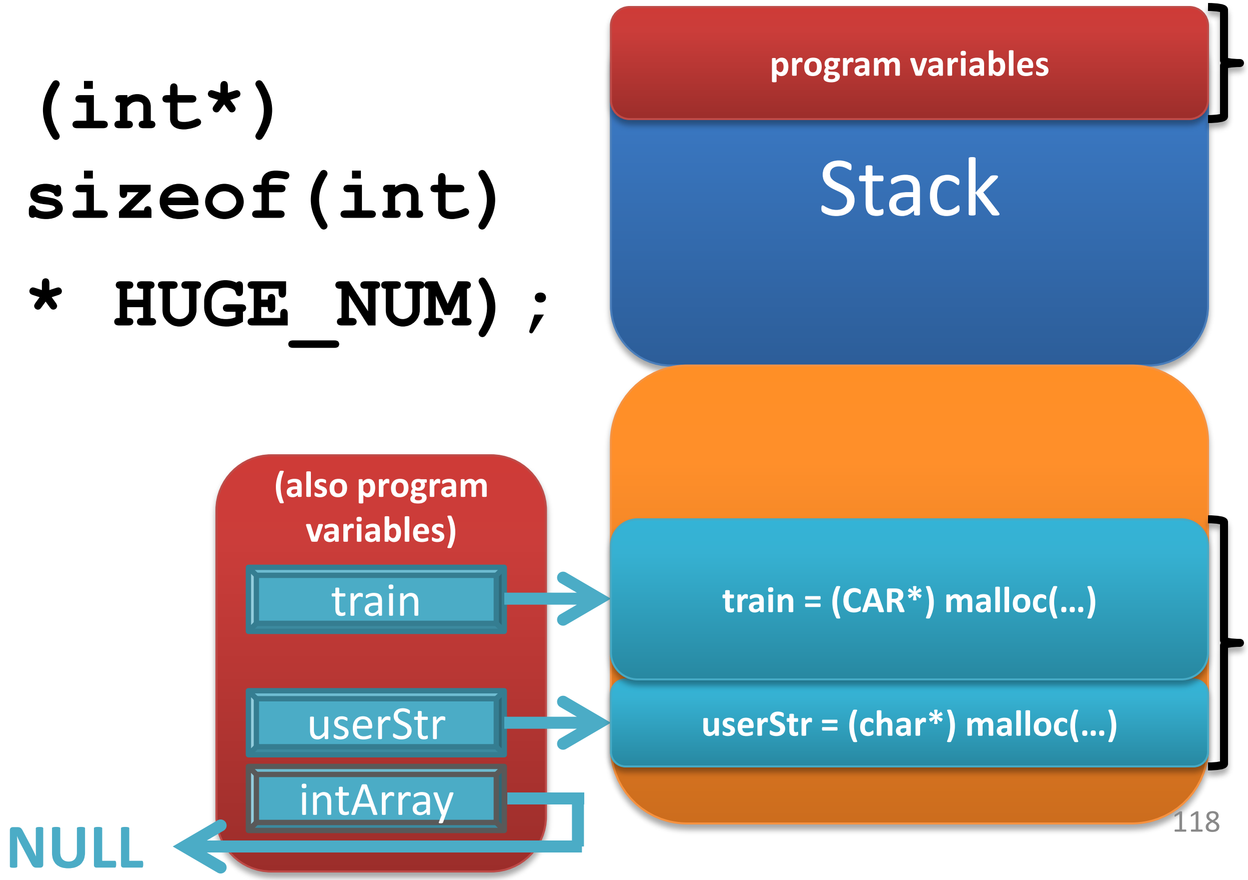
- if you try to allocate memory, but there's not enough contiguous space to handle your request



Memory Basics – Running Out

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

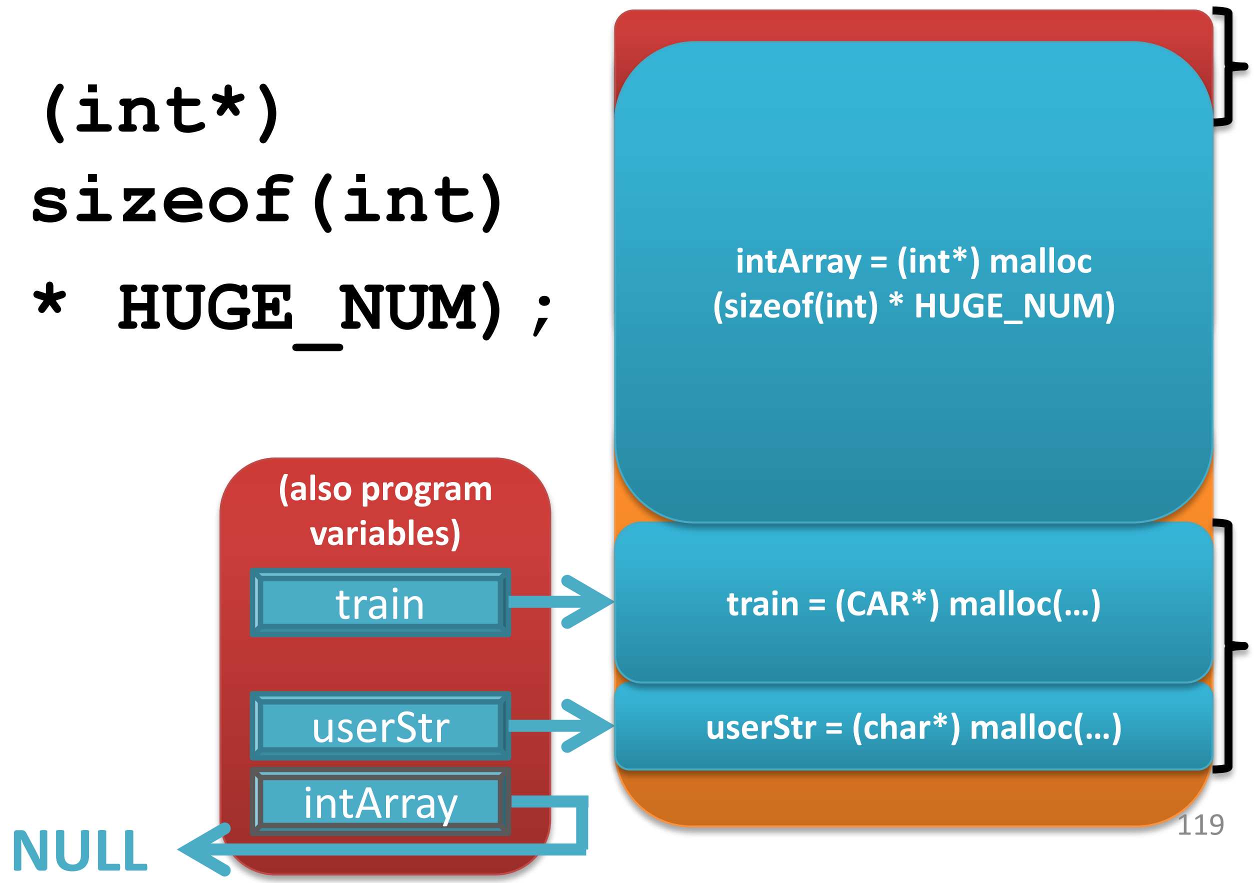
```
intArray = (int*)
  malloc ( sizeof(int)
          * HUGE_NUM);
```



Memory Basics – Running Out

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

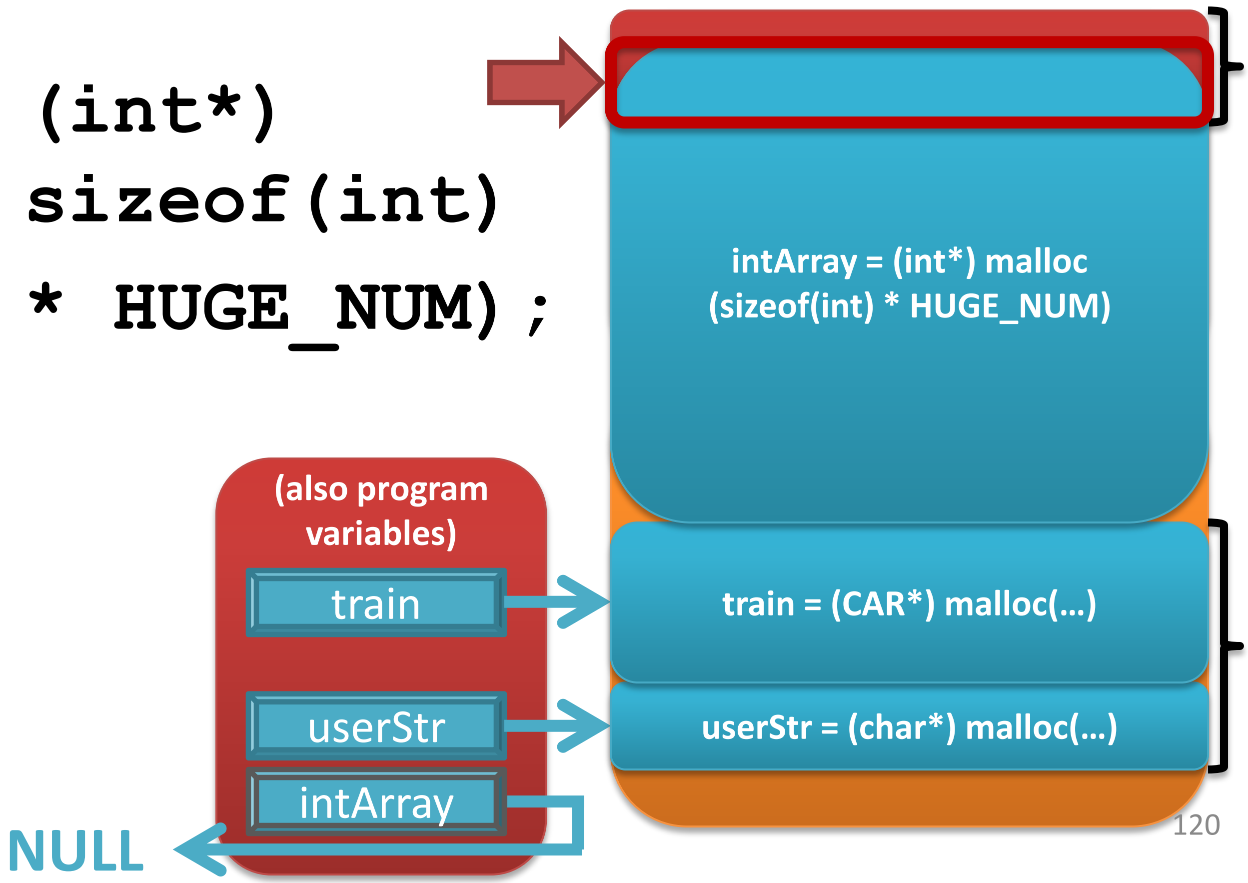
```
intArray = (int*)
  malloc ( sizeof(int)
           * HUGE_NUM);
```



Memory Basics – Running Out

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

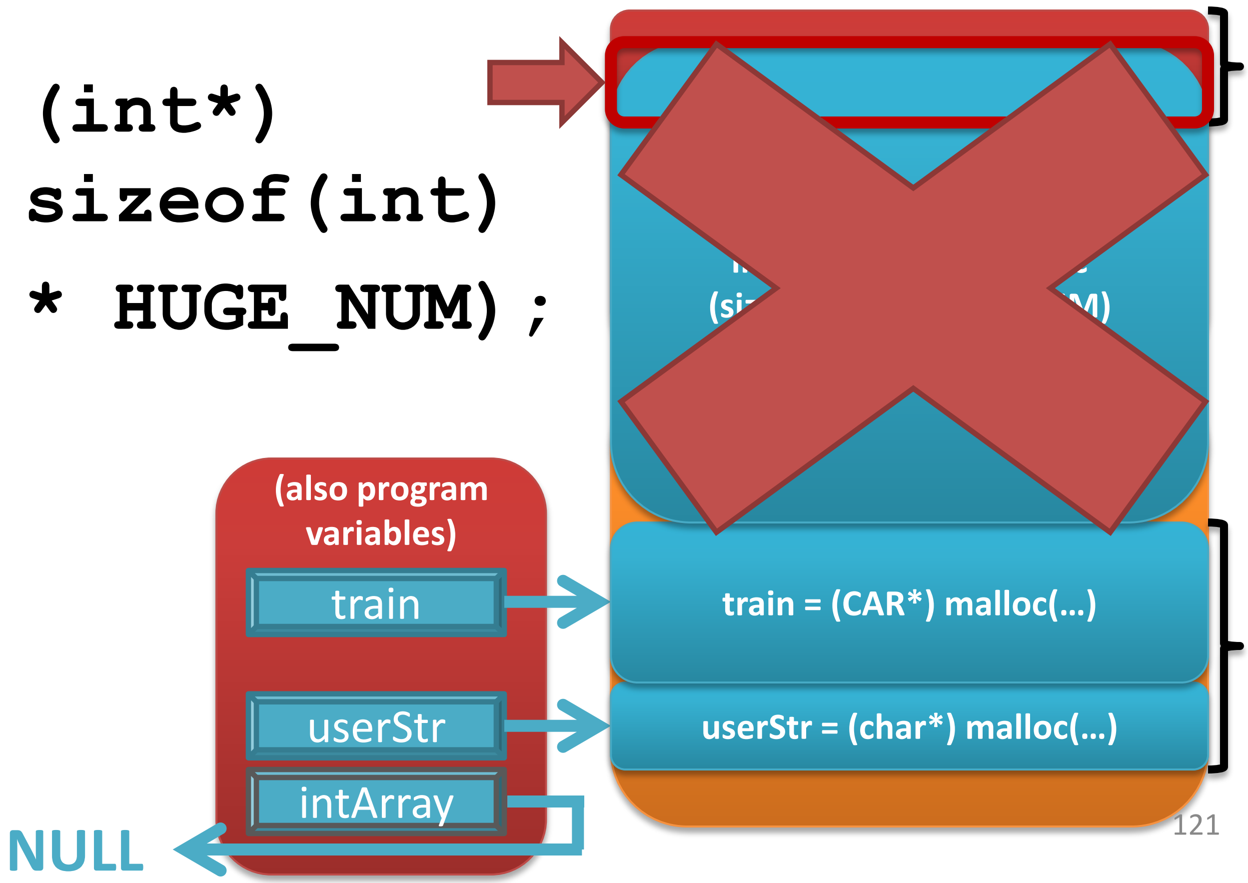
```
intArray = (int*)
   malloc ( sizeof(int)
          * HUGE_NUM);
```



Memory Basics – Running Out

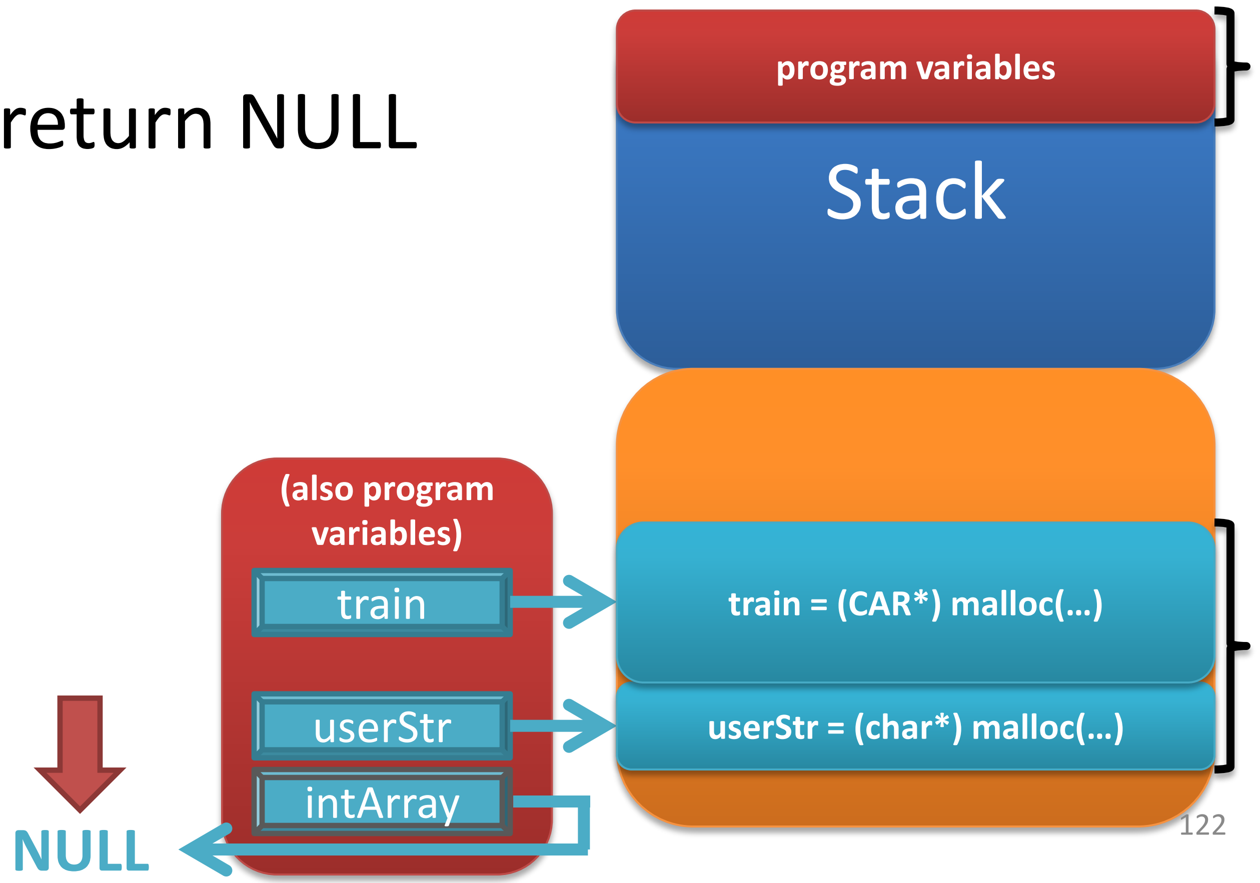
- if you try to allocate memory, but there's not enough contiguous space to handle your request

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



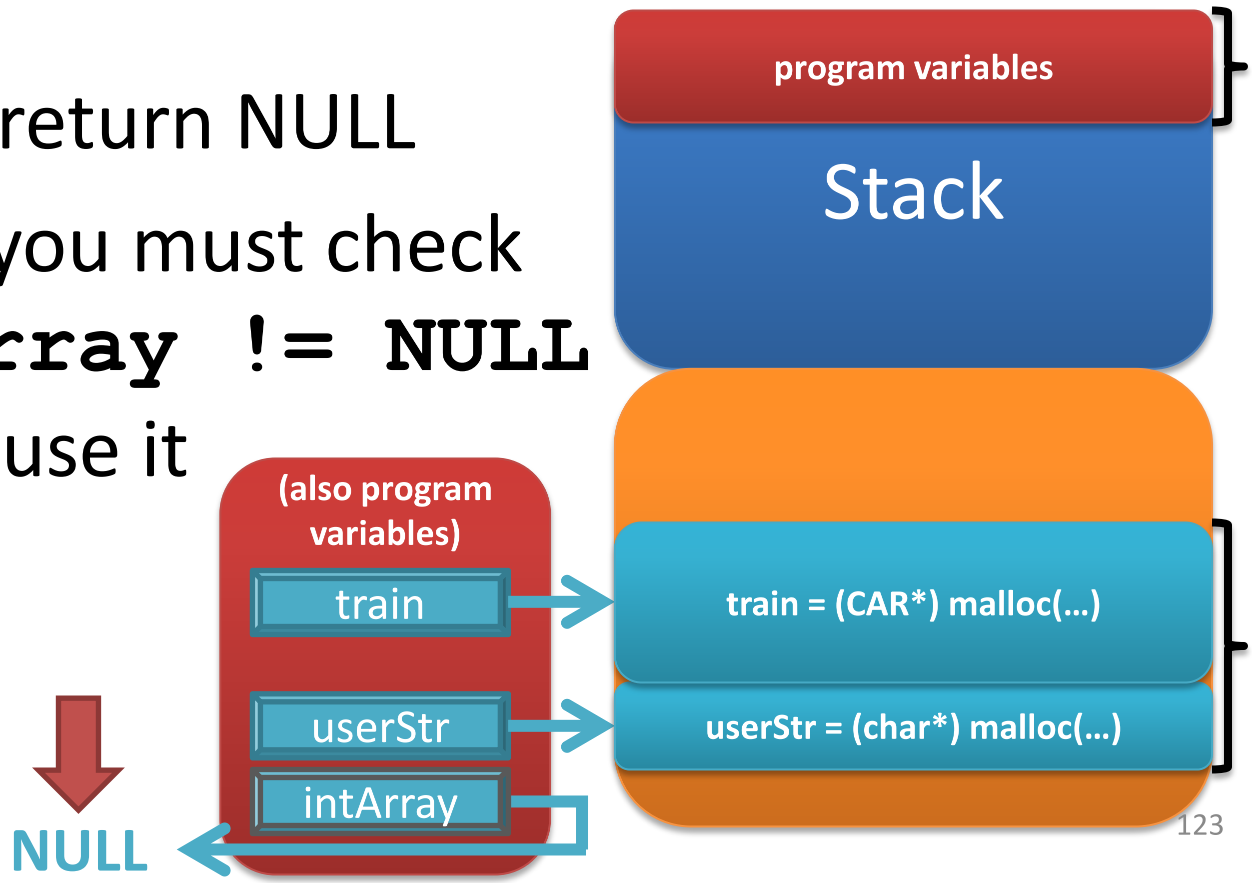
Memory Basics – Running Out

- 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!!!

Outline

- Makefiles
- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers
- Memory and Functions
- Homework

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 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[]);
void PrintName(char *name  );

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

Outline

- Makefiles
- File I/O
- Command Line Arguments
- Random Numbers
- Re-Covering Pointers
- Memory and Functions
- Homework

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

Quick Notes

- answered questions from HW2 on Piazza
- magic numbers
  - should use #defines as you code
  - not replace with #define after you're done
- elegant solution to printing the full train