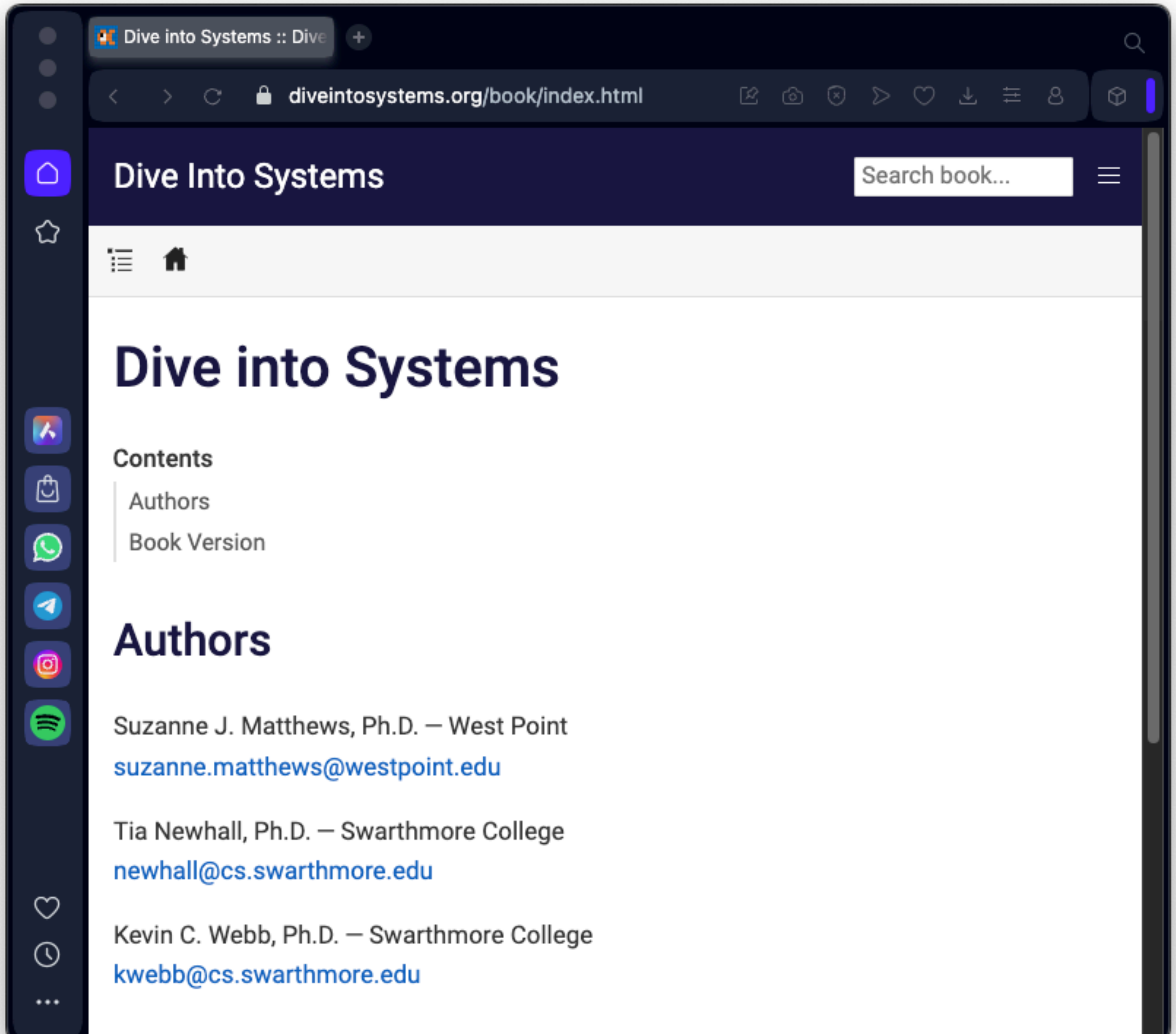
# 2. A Deeper Dive into C

## For COMSC 142

Sam Bowne

Updated Sep 16, 2025

# Free online textbook




- https://diveintosystems.org/book/index.html

# Topics

2.1. Parts of Program Memory and Scope

2.2. C Pointer Variables

2.3. Pointers and Functions

2.4. Dynamic Memory Allocation

2.5. Arrays in C

2.6. Strings and the String Library

2.7. Structs

2.8. Input / Output in C

2.9. Advanced C Features

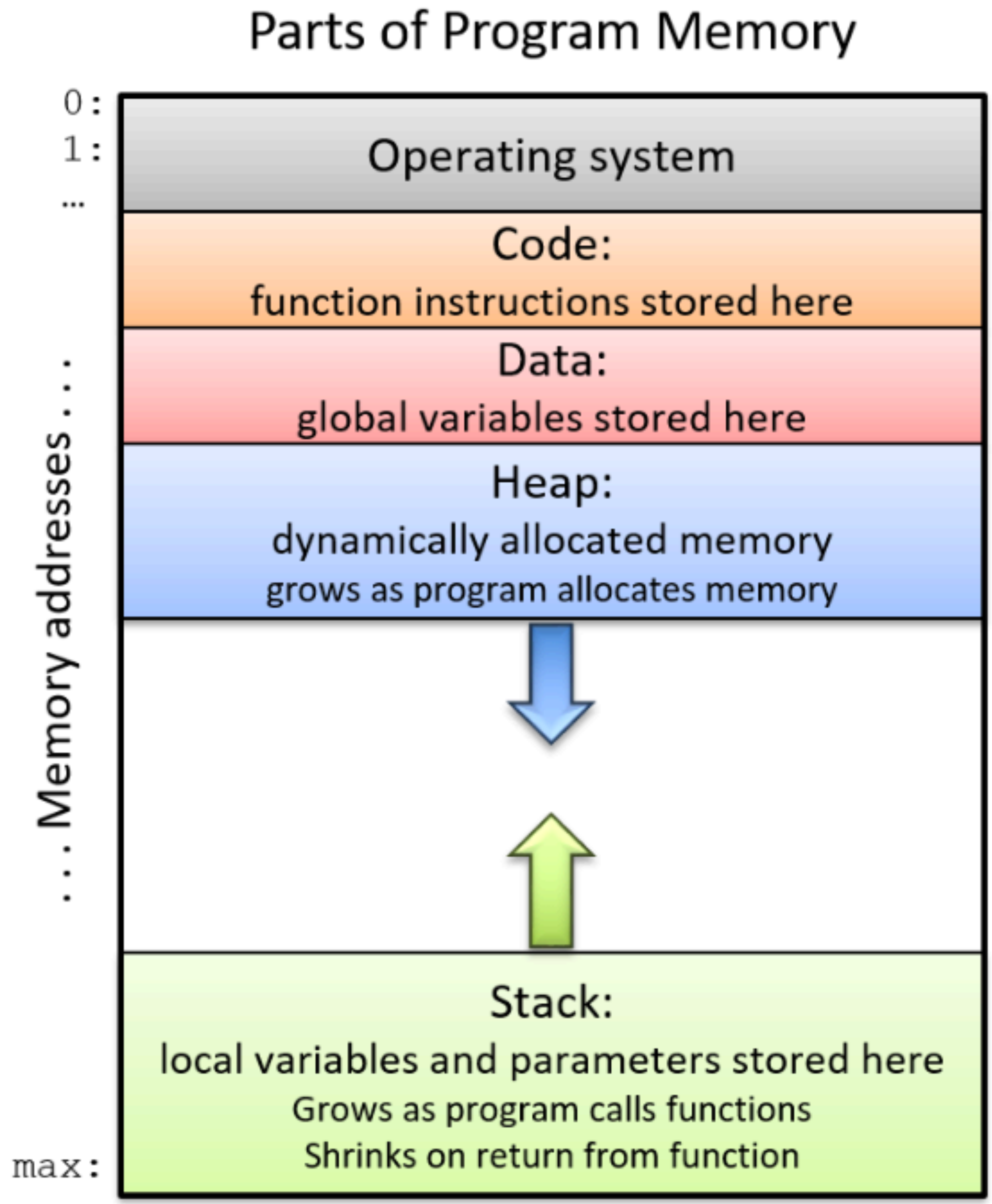
# 2.1. Parts of Program Memory and Scope

```c
/* An example C program with local and global variables */
#include <stdio.h>

int max(int n1, int n2); /* function prototypes */
int change(int amt);

int g_x;  /* global variable: declared outside function bodies */

int main(void) {
    int x, result;   /* local variables: declared inside function bodies */

    printf("Enter a value: ");
    scanf("%d", &x);
    g_x = 10;       /* global variables can be accessed in any function */

    result = max(g_x, x);
    printf("%d is the largest of %d and %d\n", result, g_x, x);

    result = change(10);
    printf("g_x's value was %d and now is %d\n", result, g_x);

    return 0;
}
```

```c
int max(int n1, int n2) {  /* function with two parameters */
    int val;    /* local variable */

    val = n1;
    if ( n2 > n1 ) {
        val = n2;
    }
    return val;
}

int change(int amt) {
    int val;

    val = g_x;  /* global variables can be accessed in any function */
    g_x += amt;
    return val;
}
```

Notice the same local variable name **val** used in both functions

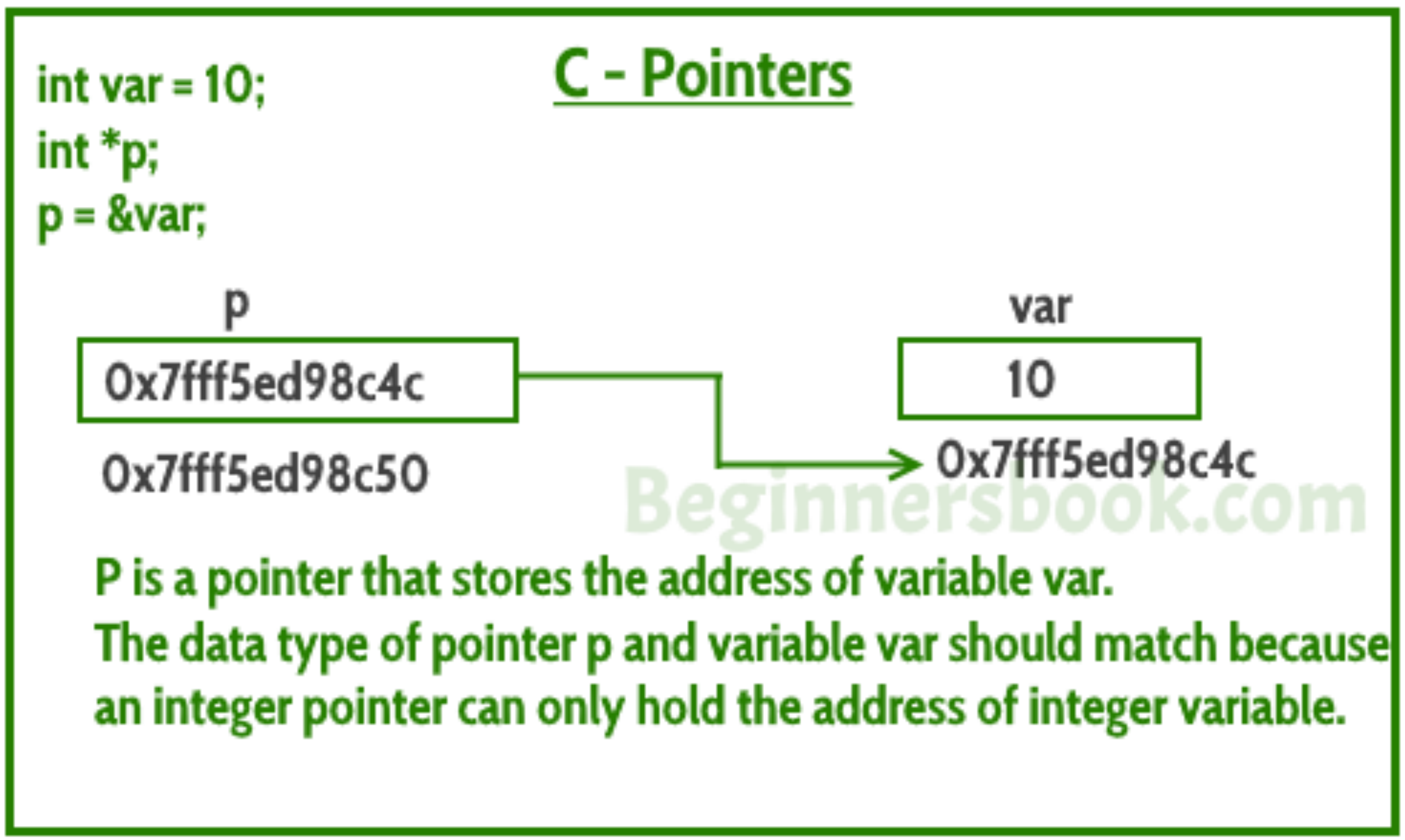
- Local variables and parameters reside on the **stack**



# 2.2. C Pointer Variables

# Pointers

- Pointer variable contains an address
- Data is stored at that address
- This is called **indirection**



# Declaring and Initializing a Pointer Variable

## Declaring

1. First, **declare a pointer variable** using `type_name *var_name`:

```
int *ptr;   // stores the memory address of an int (ptr "points to" an int)
char *cptr; // stores the memory address of a char (cptr "points to" a char)
```

## Initializing

```
int x;
char ch;

ptr = &x;     // ptr gets the address of x, pointer "points to" x
cptr = &ch;   // cptr gets the address of ch, pointer "points to" ch
```

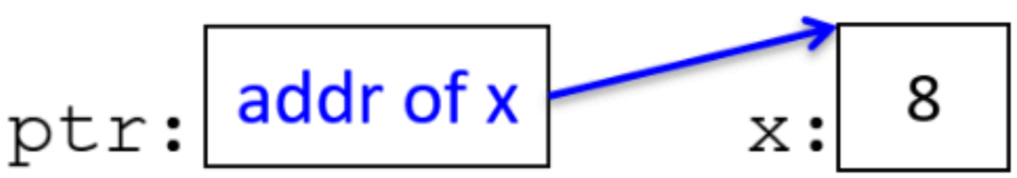
# Using NULL

- NULL represents an invalid address
- Null pointers should never be dereferenced

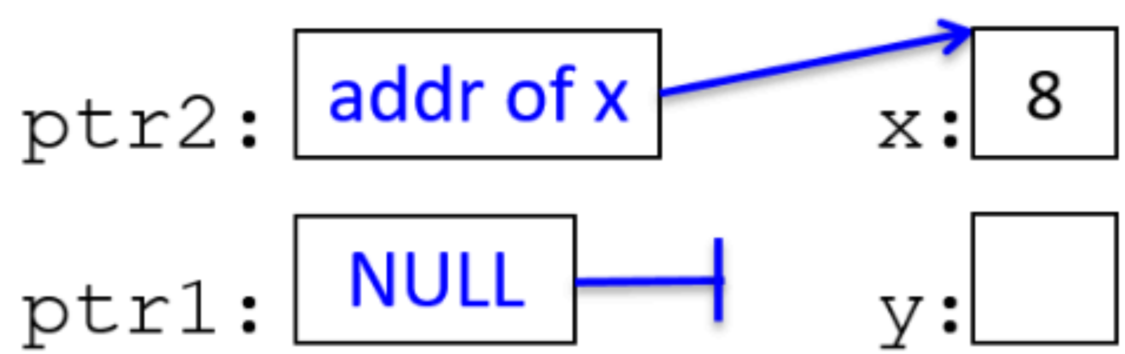
```
ptr = NULL;
cptr = NULL;
```

# Dereferencing a Pointer Variable

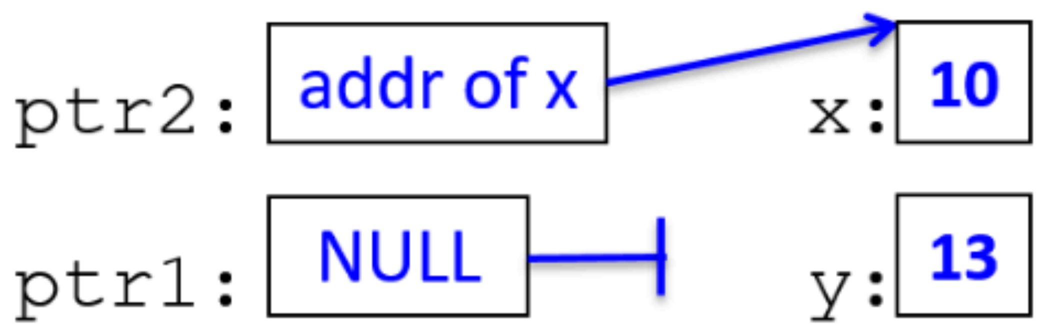
```
/* Assuming an integer named x has already been declared, this code sets the
   value of x to 8. */

ptr = &x;    /* initialize ptr to the address of x (ptr points to variable x) */
*ptr = 8;    /* the memory location ptr points to is assigned 8 */
```



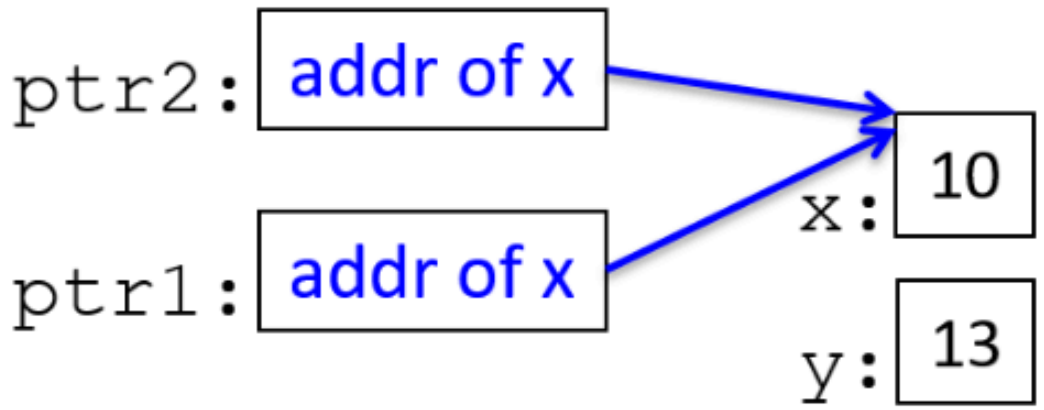
```
int *ptr1, *ptr2, x, y;

x = 8;
ptr2 = &x;     // ptr2 is assigned the address of x
ptr1 = NULL;
```



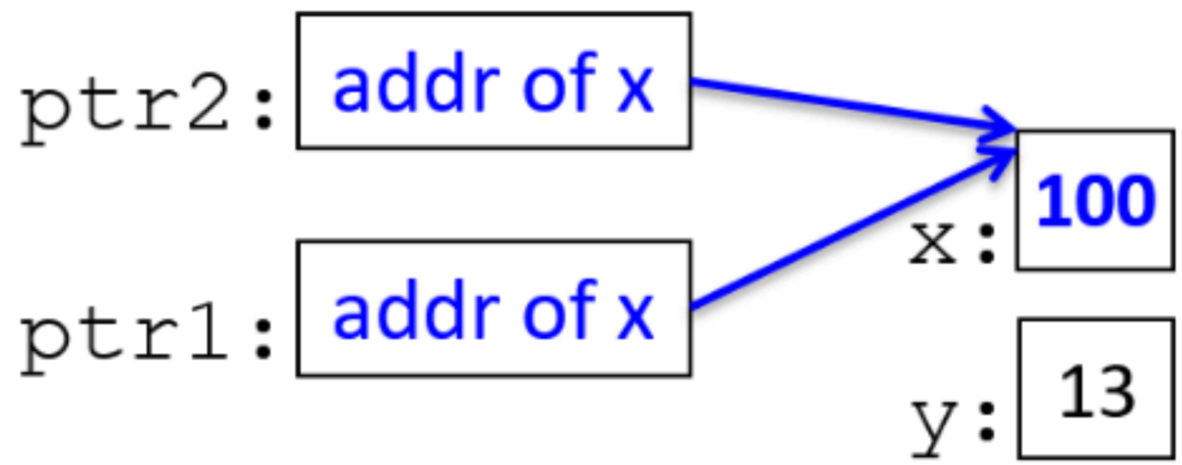
```
*ptr2 = 10;      // the memory location ptr2 points to is assigned 10
y = *ptr2 + 3;  // y is assigned what ptr2 points to plus 3
```



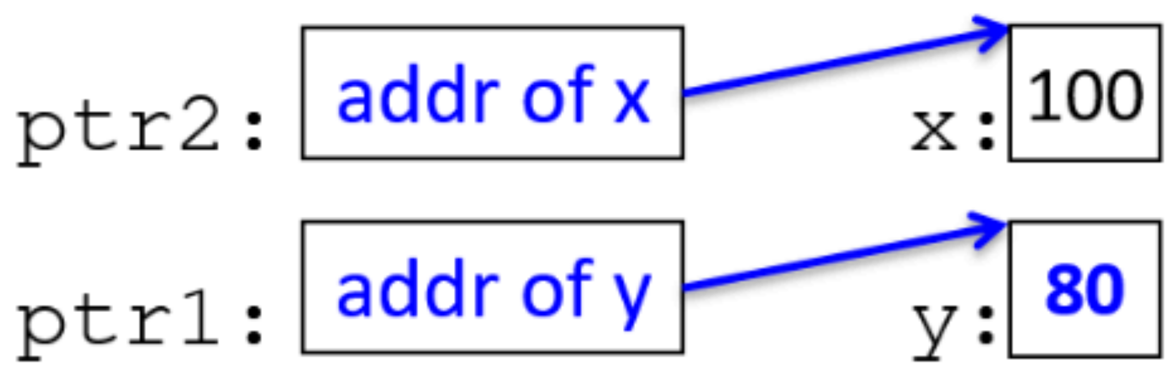

```
ptr1 = ptr2;   // ptr1 gets the address value stored in ptr2 (both point to x)
```



```
*ptr1 = 100;
```




```
ptr1 = &y;      // change ptr1's value (change what it points to)
*ptr1 = 80;
```



# Errors

```
ptr = 20;        // ERROR?:  this assigns ptr to point to address 20
ptr = &x;
*ptr = 20;       // CORRECT: this assigns 20 to the memory pointed to by ptr
```

If your program dereferences a pointer variable that does not contain a valid address, the program crashes:

```
ptr = NULL;
*ptr = 6;    // CRASH! program crashes with a segfault (a memory fault)

ptr = 20;
*ptr = 6;    // CRASH! segfault (20 is not a valid address)

ptr = x;
*ptr = 6;   // likely CRASH or may set some memory location with 6
            // (depends on the value of x which is used as an address value)

ptr = &x;   // This is probably what the programmer intended
*ptr = 6;
```

# Testing for NULL Pointers

```
if (ptr != NULL) {
    *ptr = 6;
}
```

Kahoot!

Ch 2a

# 2.3. Pointers and Functions

# Function to Double a Number

sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 61×22

```
debian@debian:~/COMSC-142$ cat double_number2.c
#include <stdio.h>

int double_number(int * val);  /* function prototype */

int main(void) {
  int number = 5;

  printf("Before double_number, number is %d\n", number);
  double_number(&number);
  printf("After double_number, number is %d\n", number);
}

int double_number(int * val) {
  *val = 2 * *val;
  return 0;
}

debian@debian:~/COMSC-142$ ./double_number2
Before double_number, number is 5
After double_number, number is 10
debian@debian:~/COMSC-142$
```

# Arguments Pass by Value

- C functions get a copy of an argument's value to work with
  - Modifying parameters in a function does not change its argument's value

# Function to Double a Number

sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 61×22

```
debian@debian:~/COMSC-142$ cat double_number.c
#include <stdio.h>

int double_number(int val);  /* function prototype */

int main(void) {
  int number = 5;

  printf("Before double_number, number is %d\n", number);
  double_number(number);
  printf("After double_number, number is %d\n", number);
}

int double_number(int val) {
  val = 2 * val;
  return 0;
}

debian@debian:~/COMSC-142$ ./double_number
Before double_number, number is 5
After double_number, number is 5
debian@debian:~/COMSC-142$
```

# Pointer Parameters

- Passing a pointer variable to a function
  - Allows the function to modify an argument value

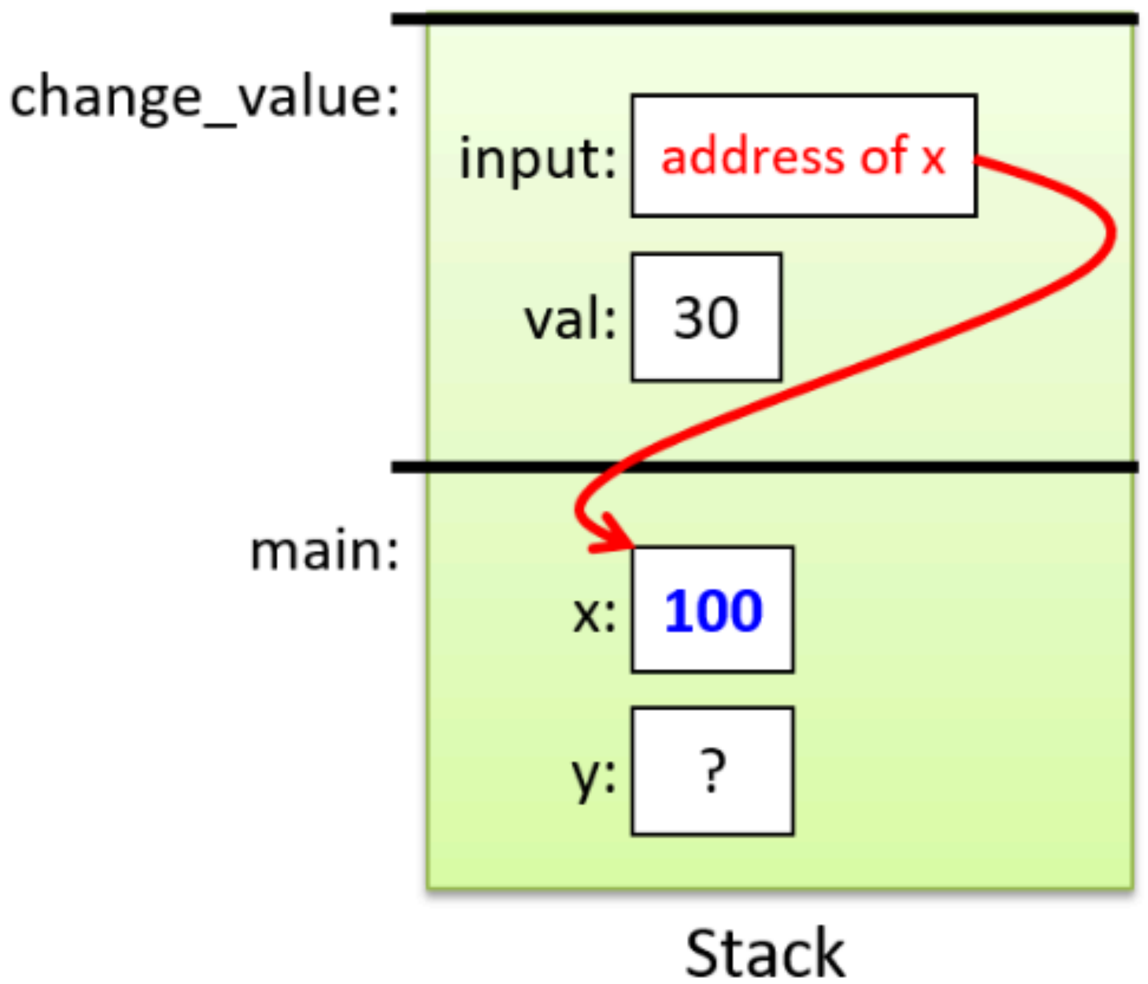
# Passing a pointer allows the function to change a value in the calling function

```c
#include <stdio.h>

int change_value(int *input);

int main(void) {
    int x;
    int y;

    x = 30;
    y = change_value(&x);
    printf("x: %d y: %d\n", x, y);  // prints x: 100 y: 30

    return 0;
}

/*
 * changes the value of the argument
 *     input: a pointer to the value to change
 *     returns: the original value of the argument
 */
int change_value(int *input) {
    int val;

    val = *input; /* val gets the value input points to */

    if (val < 100) {
        *input = 100;  /* the value input points to gets 100 */
    } else {
        *input =  val * 2;
    }
```

When run, the output is:

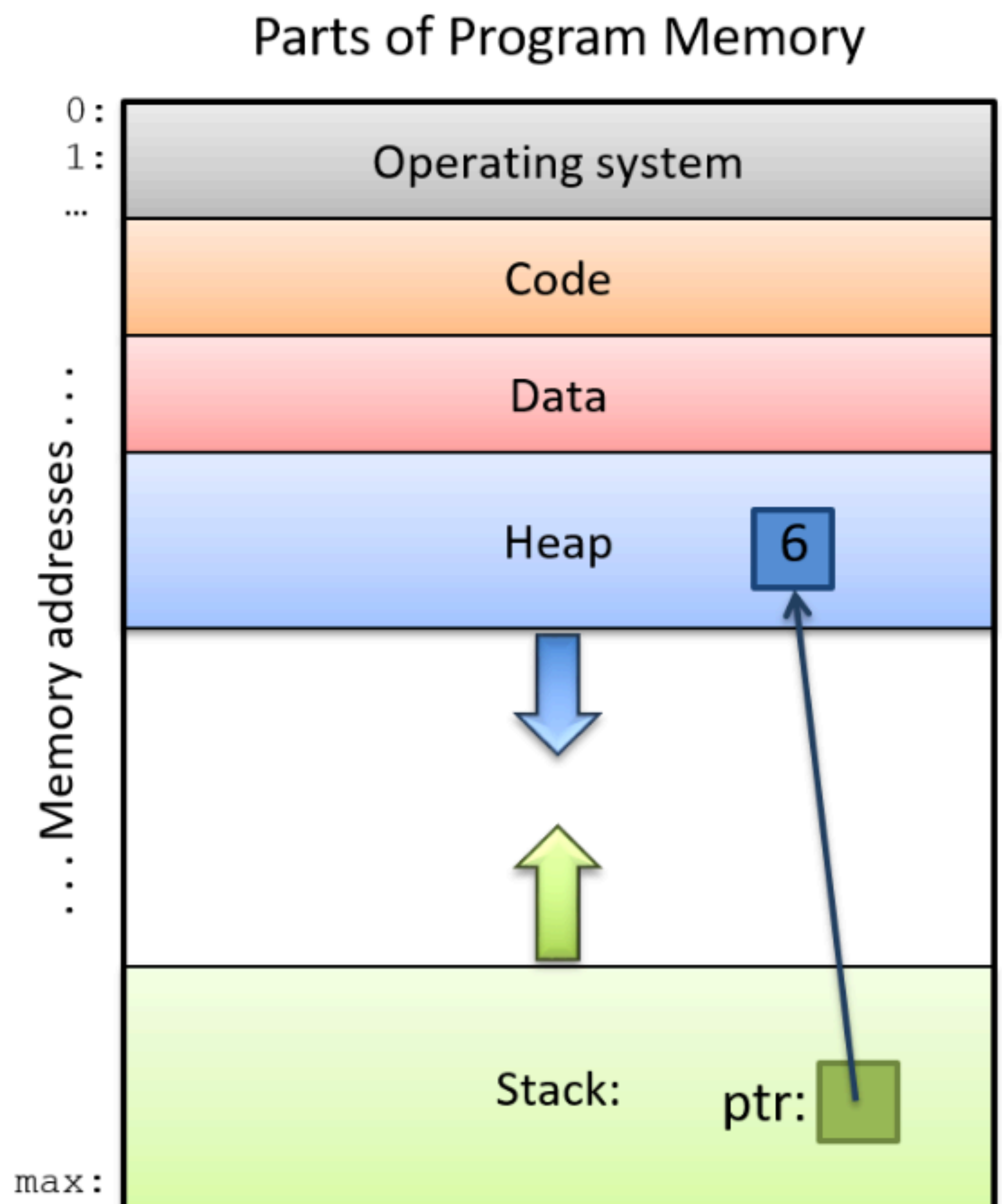
```
x: 100 y: 30
```

Figure 1 shows what the call stack looks like before executing the return in `change_value`.



# 2.4. Dynamic Memory Allocation

A **pointer** on the **stack** points to a block of memory allocated on the **heap**



# malloc and free

- malloc returns a **void *** type
  - Can point to any type of data

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p;

    p = malloc(sizeof(int));  // allocate heap memory for storing an int

    if (p != NULL) {
        *p = 6;   // the heap memory p points to gets the value 6
    }
}
```

# When malloc Fails

- If there's not enough free heap memory
  - malloc returns a NULL pointer

```
int *p;

p = malloc(sizeof(int));
if (p == NULL) {
    printf("Bad malloc error\n");
    exit(1);    // exit the program and indicate error
}
*p = 6;
```

# Freeing Memory

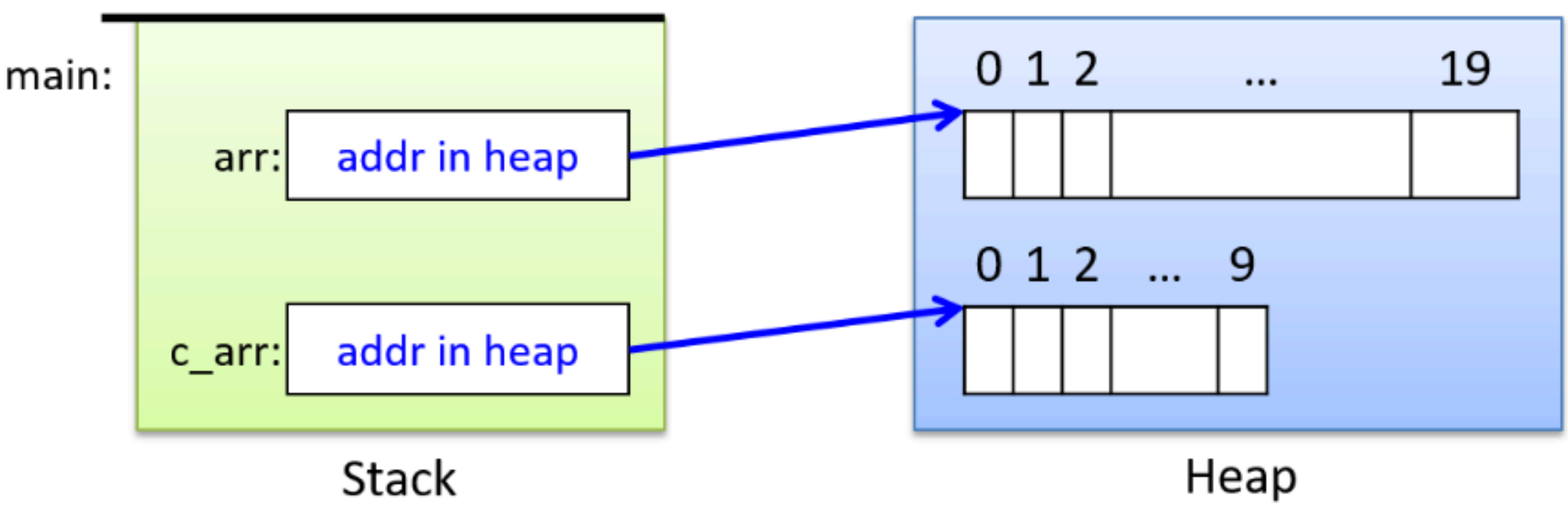
- When a program no longer needs the allocated memory, it should:
  - call **free()**
  - Set the pointer to **NULL**

```
free(p);
p = NULL;
```

- Failing to do this leads to many security problems, including
  - Dangling pointer (aka use-after-free)
  - Double-free

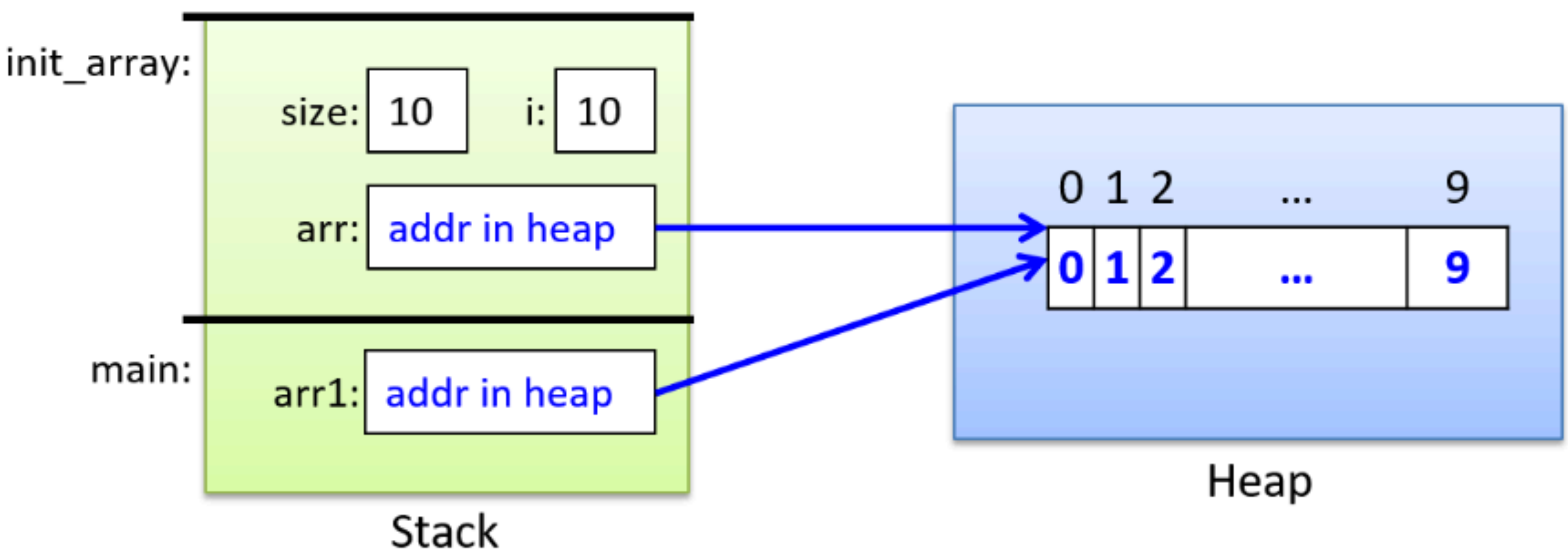
# Dynamically Allocated Arrays and Strings

```c
int *arr;
char *c_arr;

// allocate an array of 20 ints on the heap:
arr = malloc(sizeof(int) * 20);

// allocate an array of 10 chars on the heap:
c_arr = malloc(sizeof(char) * 10);
```



# Pointers to Heap Memory and Functions

- If a pointer is passed to a function, it can write to the data on the heap



Kahoot!

Ch 2b

# 2.5. Arrays in C

# Statically Allocated Arrays

```
// declare arrays specifying their type and total capacity
float averages[30];   // array of float, 30 elements
char  name[20];       // array of char, 20 elements
int i;

// access array elements
for (i = 0; i < 10; i++) {
    averages[i] = 0.0 + i;
    name[i] = 'a' + i;
}
name[10] = '\0';    // name is being used for storing a C-style string

// prints: 3 d abcdefghij
printf("%g %c %s\n", averages[3], name[3], name);

strcpy(name, "Hello");
printf("%s\n", name);  // prints: Hello
```

# Dynamically Allocated Arrays

```c
// declare a pointer variable to point to allocated heap space
int    *p_array;
double *d_array;

// call malloc to allocate the appropriate number of bytes for the array

p_array = malloc(sizeof(int) * 50);      // allocate 50 ints
d_array = malloc(sizeof(double) * 100);  // allocate 100 doubles

// always CHECK RETURN VALUE of functions and HANDLE ERROR return values
if ( (p_array == NULL) || (d_array == NULL) ) {
    printf("ERROR: malloc failed!\n");
    exit(1);
}

// use [] notation to access array elements
for (i = 0; i < 50; i++) {
    p_array[i] = 0;
    d_array[i] = 0.0;
}

// free heap space when done using it
free(p_array);
p_array = NULL;

free(d_array);
d_array = NULL;
```

# Array Memory Layout

```
int  iarray[6];  // an array of six ints, each of which is four bytes
char carray[4];  // an array of four chars, each of which is one byte
```

```
array [0]:  base address
array [1]:  next address
array [2]:  next address
  ...             ...
array [99]: last address
```

```
addr    element
----    -------
1230:   iarray[0]
1234:   iarray[1]
1238:   iarray[2]
1242:   iarray[3]
1246:   iarray[4]
1250:   iarray[5]
     ...
1280:   carray[0]
1281:   carray[1]
1282:   carray[2]
1283:   carray[3]
```

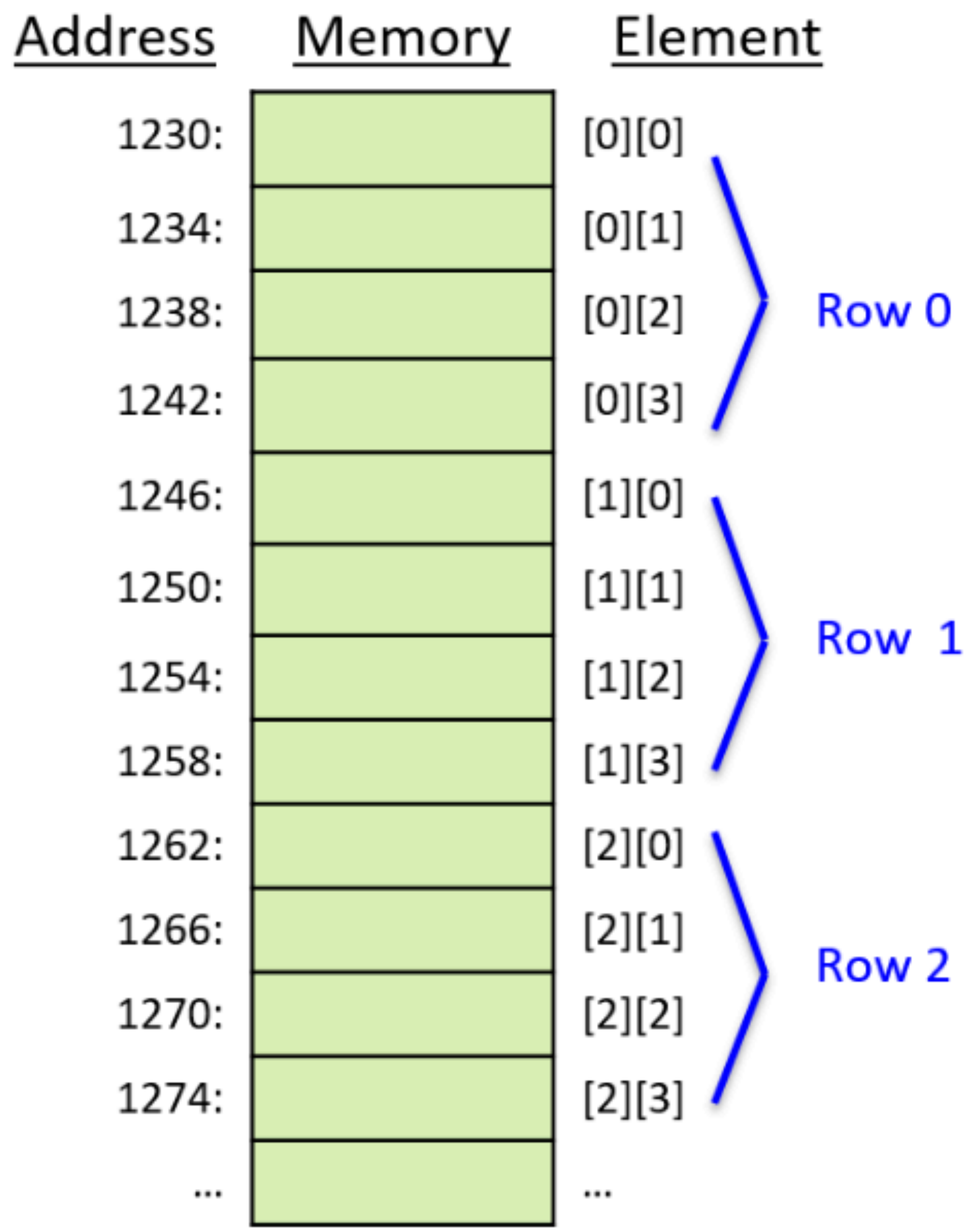
# Constants

- Easier to read and update than literal values buried deep in the code

```
#define N   20

int main(void) {
  int array[N];    // an array of 20 ints
  int *d_arr, i;

  // dynamically alloc array of 20 ints
  d_arr = malloc(sizeof(int)*N);
  if(d_arr == NULL) {
    exit(1);
  }

  for(i=0; i < N; i++) {
    array[i] = i;
    d_arr[i] = i*2;
  }
  ...
}
```

# Two-Dimensional Arrays

```
int arr[3][4];
```



# 2.6. Strings and the String Library

# Statically Allocated Strings (Arrays of char)

```
#include <stdio.h>
#include <string.h>   // include the C string library

int main(void) {
    char str1[10];
    char str2[10];

    str1[0] = 'h';
    str1[1] = 'i';
    str1[2] = '\0';   // explicitly add null terminating character to end

    // strcpy copies the bytes from the source parameter (str1) to the
    // destination parameter (str2) and null terminates the copy.
    strcpy(str2, str1);
    str2[1] = 'o';
    printf("%s %s\n", str1, str2);  // prints: hi ho

    return 0;
}
```

# C String Functions and Destination Memory

- **strcpy**, **strcat**, and many other string functions
  - Simply start writing at a string pointer
  - and write as many bytes as needed,
  - followed by a NULL byte
- They don't check to make sure enough room was reserved for the string
- That is the programmer's responsibility
- This leads to **buffer overflows**

# strlen, strcpy, strncpy

- **strncpy** is safer than **strcpy**

```
// returns the number of characters in the string (not including the null character)
int strlen(char *s);

// copies string src to string dst up until the first '\0' character in src
// (the caller needs to make sure src is initialized correctly and
// dst has enough space to store a copy of the src string)
// returns the address of the dst string
char *strcpy(char *dst, char *src);

// like strcpy but copies up to the first '\0' or size characters
// (this provides some safety to not copy beyond the bounds of the dst
// array if the src string is not well formed or is longer than the
// space available in the dst array); size_t is an unsigned integer type
char *strncpy(char *dst, char *src, size_t size);
```

```c
int len, i, ret;
char str[32];
char *d_str, *ptr;

strcpy(str, "Hello There");
len = strlen(str);  // len is 11

d_str = malloc(sizeof(char) * (len+1));
if (d_str == NULL) {
    printf("Error: malloc failed\n");
    exit(1);
}

strncpy(d_str, str, 5);
d_str[5] = '\0';   // explicitly add null terminating character to end

printf("%d:%s\n", strlen(str), str);      // prints 11:Hello There
printf("%d:%s\n", strlen(d_str), d_str);  // prints 5:Hello

free(d_str);
```

# strlcpy

- Only available in newer versions of Linux's GNU C library

```
// like strncpy but copies up to the first '\0' or size-1 characters
// and null terminates the dest string (if size > 0).
char *strlcpy(char *dest, char *src, size_t size);
```

# strcmp, strncmp

- Comparing string variables using the == operator does not compare the characters in the strings
  - it compares only the base addresses of the two strings

```
int strcmp(char *s1, char *s2);
// returns 0 if s1 and s2 are the same strings
// a value < 0 if s1 is less than s2
// a value > 0 if s1 is greater than s2

int strncmp(char *s1, char *s2, size_t n);
// compare s1 and s2 up to at most n characters
```

# Strcmp Example

```c
strcpy(str, "alligator");
strcpy(d_str, "Zebra");

ret =  strcmp(str,d_str);
if (ret == 0) {
    printf("%s is equal to %s\n", str, d_str);
} else if (ret < 0) {
    printf("%s is less than %s\n", str, d_str);
} else {
    printf("%s is greater than %s\n", str, d_str);  // true for these strings
}

ret = strncmp(str, "all", 3);  // returns 0: they are equal up to first 3 chars
```

# strcat and strncat

```
// append chars from src to end of dst
// returns ptr to dst and adds '\0' to end
char *strcat(char *dst, char *src)

// append the first chars from src to end of dst, up to a maximum of size
// returns ptr to dst and adds '\0' to end
char *strncat(char *dst, char *src, size_t size);
```

# strstr and strchr

```
// locate a substring inside a string
// (const means that the function doesn't modify string)
// returns a pointer to the beginning of substr in string
// returns NULL if substr not in string
char *strstr(const char *string, char *substr);

// locate a character (c) in the passed string (s)
// (const means that the function doesn't modify s)
// returns a pointer to the first occurrence of the char c in string
// or NULL if c is not in the string
char *strchr(const char *s, int c);
```

# sprintf

- Prints to a string

```c
char str[64];
float ave = 76.8;
int num = 2;

// initialize str to format string, filling in each placeholder with
// a char representation of its arguments' values
sprintf(str, "%s is %d years old and in grade %d", "Henry", 12, 7);
printf("%s\n", str);  // prints: Henry is 12 years old and in grade 7

sprintf(str, "The average grade on exam %d is %g", num, ave);
printf("%s\n", str);  // prints: The average grade on exam 2 is 76.8
```

# Functions for Individual Character Values

```
#include <stdlib.h>    // include stdlib and ctypes to use these
#include <ctype.h>

int islower(ch);
int isupper(ch);          // these functions return a non-zero value if the
int isalpha(ch);          // test is TRUE, otherwise they return 0 (FALSE)
int isdigit(ch);
int isalnum(ch);
int ispunct(ch);
int isspace(ch);
char tolower(ch);        // returns ASCII value of lower-case of argument
char toupper(ch);
```

# Functions to Convert Strings to Other Types

```c
#include <stdlib.h>

int atoi(const char *nptr);      // convert a string to an integer
double atof(const char *nptr);   // convert a string to a float
```

```c
printf("%d %g\n", atoi("1234"), atof("4.56"));
```

# Kahoot!

Ch 2c

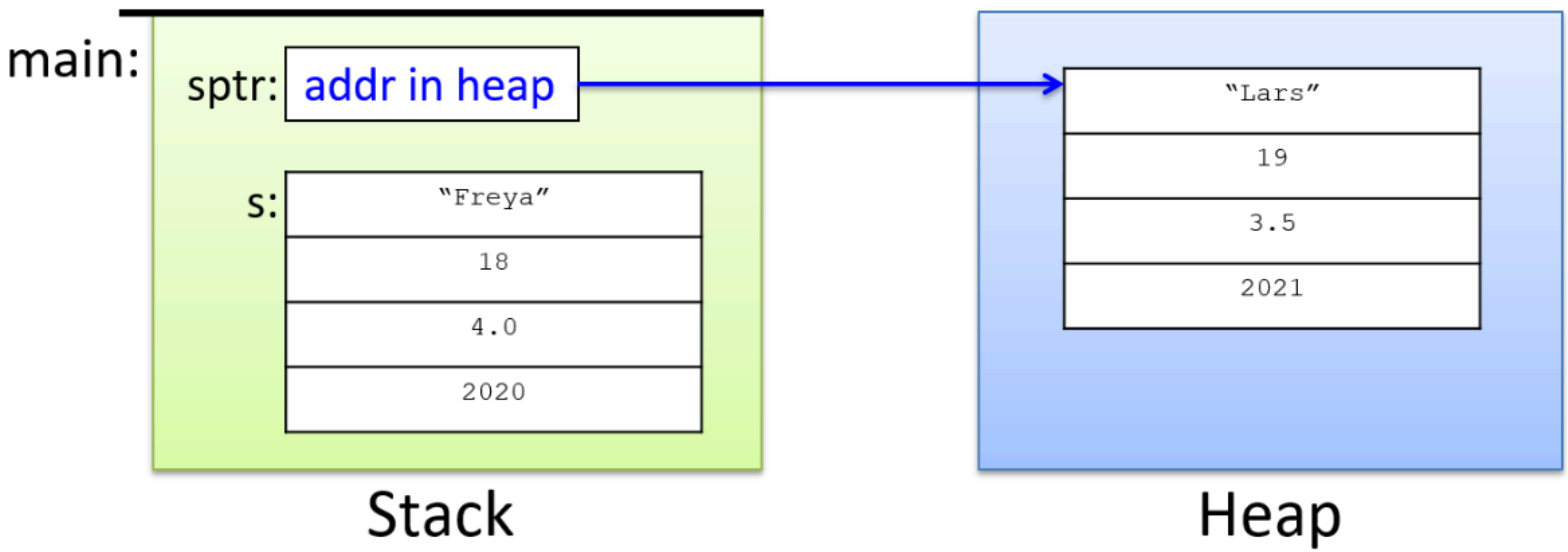
# 2.7. Structs

# The C struct Type

```
/* define a new struct type (outside function bodies) */
struct studentT {
    char  name[64];
    int   age;
    float gpa;
    int   grad_yr;
};
```

```
// access field values using .
strcpy(student1.name, "Ruth");
student1.age = 17;
student1.gpa = 3.5;
student1.grad_yr = 2021;
```

# structs on the Stack and the Heap



# 2.8. Input / Output in C

# Standard Input/Output

- **stdin**
  - Standard input
  - **scanf()** reads from **stdin**
    - The user on the keyboard
- **stdout**
  - Standard output
  - **printf()** writes to **stdout**
    - The monitor
- **stderr**
  - Standard error

# Input and Output Redirection

- From a shell (not C code)

```
#  redirect a.out's stdin to read from file infile.txt:
$ ./a.out < infile.txt

#  redirect a.out's stdout to print to file outfile.txt:
$ ./a.out > outfile.txt

# redirect a.out's stdout and stderr to a file out.txt
$ ./a.out &> outfile.txt

# redirect all three to different files:
#   (< redirects stdin, 1> stdout, and 2> stderr):
$ ./a.out < infile.txt 1> outfile.txt 2> errorfile.txt
```

# Another Way

1. Redirect stdout to one file and stderr to another file:

```
command > out 2>error
```

2. Redirect stdout to a file (`>out`), and then redirect stderr to stdout (`2>&1`):

```
command >out 2>&1
```

3. Redirect both to a file (this isn't supported by all shells, `bash` and `zsh` support it, for example, but `sh` and `ksh` do not):

```
command &> out
```

# printf

```
int x = 5, y = 10;
float pi = 3.14;

printf("x is %d and y is %d\n", x, y);

printf("%g \t %s \t %d\n", pi, "hello", y);
```

When run, these `printf` statements output:

```
x is 5 and y is 10
3.14     hello   10
```

# Formatting Placeholders

```
%f, %g: placeholders for a float or double value
%d:     placeholder for a decimal value (char, short, int)
%u:     placeholder for an unsigned decimal
%c:     placeholder for a single character
%s:     placeholder for a string value
%p:     placeholder to print an address value

%ld:    placeholder for a long value
%lu:    placeholder for an unsigned long value
%lld:   placeholder for a long long value
%llu:   placeholder for an unsigned long long  value
```

# Number Representations

```
%x:     print value in hexadecimal (base 16)
%o:     print value in octal (base 8)
%d:     print value in signed decimal  (base 10)
%u:     print value in unsigned decimal (unsigned base 10)
%e:     print float or double in scientific notation
(there is no formatting option to display a value in binary)
```

# scanf

- Reads data from **stdin**
- Input values must be separated by whitespace

```
int x;
float pi;

// read in an int value followed by a float value ("%d%g")
// store the int value at the memory location of x (&x)
// store the float value at the memory location of pi (&pi)
scanf("%d%g", &x, &pi);
```

# getchar and putchar

- Read or write a single character

```
ch = getchar();  // read in the next char value from stdin
putchar(ch);     // write the value of ch to stdout
```

# File Input/Output

1. *Declare* a `FILE *` variable:

```
FILE *infile;
FILE *outfile;
```

# File Input/Output

2. *Open* the file: associate the variable with an actual file stream by calling `fopen`. When opening a file, the *mode* parameter determines whether the program opens it for reading (`"r"`), writing (`"w"`), or appending (`"a"`):

```
infile = fopen("input.txt", "r");  // relative path name of file, read mode
if (infile == NULL) {
    printf("Error: unable to open file %s\n", "input.txt");
    exit(1);
}

// fopen with absolute path name of file, write mode
outfile = fopen("/home/me/output.txt", "w");
if (outfile == NULL) {
    printf("Error: unable to open outfile\n");
    exit(1);
}
```

# File Input/Output

3. *Use* I/O operations to read, write, or move the current position in the file:

```
int ch;  // EOF is not a char value, but is an int.
         // since all char values can be stored in int, use int for ch

ch = getc(infile);      // read next char from the infile stream
if (ch != EOF) {
    putc(ch, outfile);  // write char value to the outfile stream
}
```

# File Input/Output

4. *Close* the file:

```
fclose(infile);
fclose(outfile);
```

# File Input/Output

```
// ---------------
// Character Based
// ---------------

// returns the next character in the file stream (EOF is an int value)
int fgetc(FILE *f);

// writes the char value c to the file stream f
// returns the char value written
int fputc(int c, FILE *f);

// pushes the character c back onto the file stream
// at most one char (and not EOF) can be pushed back
int ungetc(int c, FILE *f);

// like fgetc and fputc but for stdin and stdout
int getchar();
int putchar(int c);
```

# File Input/Output

```
// -------------
// String  Based
// -------------

// reads at most n-1 characters into the array s stopping if a newline is
// encountered, newline is included in the array which is '\0' terminated
char *fgets(char *s, int n, FILE *f);

// writes the string s (make sure '\0' terminated) to the file stream f
int fputs(char *s, FILE *f);
```

# File Input/Output

```
// ---------
// Formatted
// ---------

// writes the contents of the format string to file stream f
//   (with placeholders filled in with subsequent argument values)
// returns the number of characters printed
int fprintf(FILE *f, char *format, ...);

// like fprintf but to stdout
int printf(char *format, ...);

// use fprintf to print stderr:
fprintf(stderr, "Error return value: %d\n", ret);
```

# File Input/Output

```
// read values specified in the format string from file stream f
//   store the read-in values to program storage locations of types
//   matching the format string
// returns number of input items converted and assigned
//   or EOF on error or if EOF was reached
int fscanf(FILE *f, char *format, ...);

// like fscanf but reads from stdin
int scanf(char *format, ...);
```

# Format String for fscanf

```
%d integer
%f float
%lf double
%c character
%s string, up to first white space

%[...] string, up to first character not in brackets
%[0123456789] would read in digits
%[^...] string, up to first character in brackets
%[^\n] would read everything up to a newline
```

# 2.9. Advanced C Features

# 2.9.2. Command Line Arguments

- Use **argc** and **argv** to refer to command-line arguments

```
int main(int argc, char *argv[]) { ...
```

- **argc** counts all items on the command line
- If a user enters:

```
./a.out 10 11 200
```

- **argc** will be 4

# 2.9.2. Command Line Arguments

```
int main(int argc, char *argv[]) { ...
```

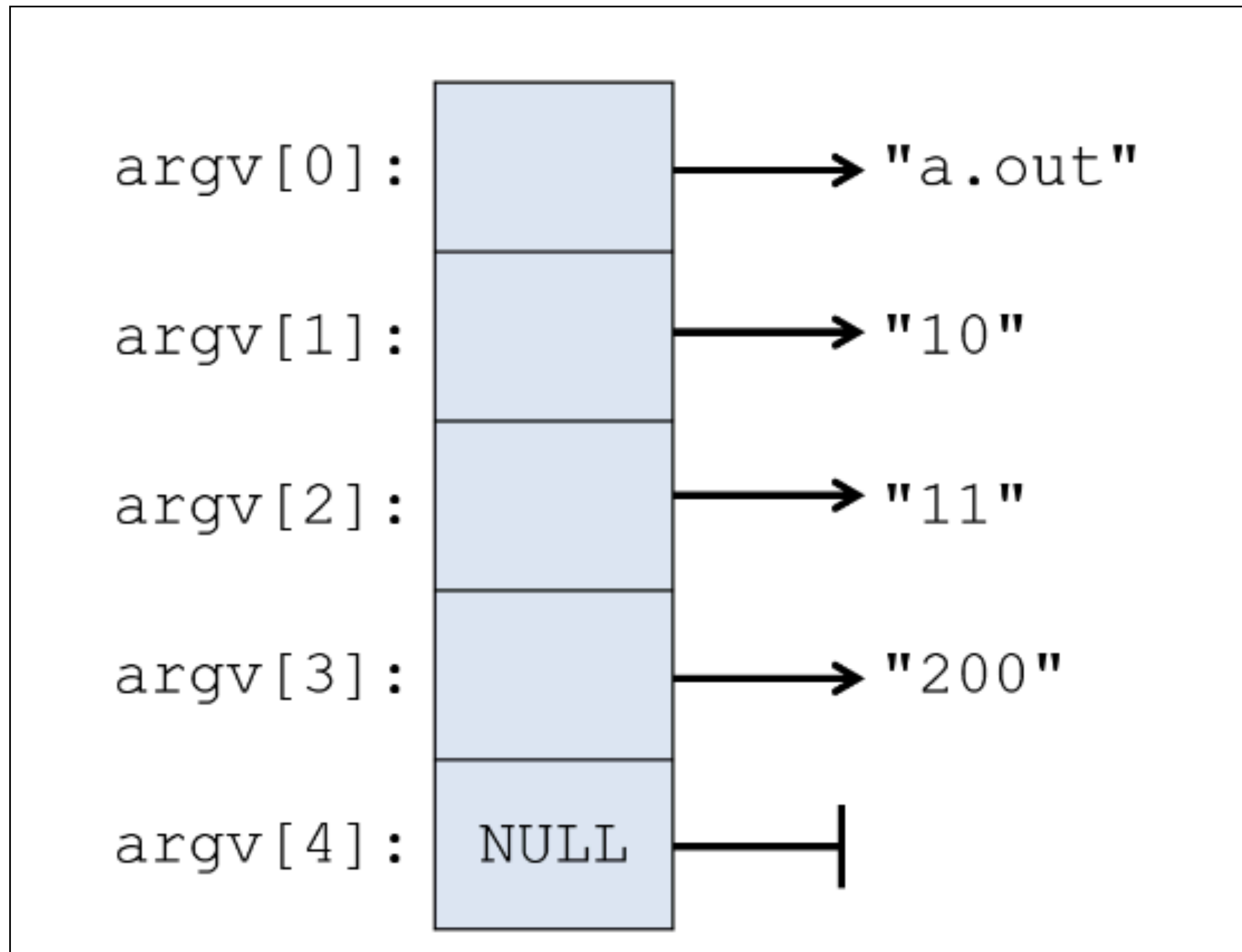
- **argv** is the argument vector
  - Contains the value of every argument
  - Followed by a NULL
  - A total of **argc** + 1 elements

# 2.9.2. Command Line Arguments

- If a user enters:

```
./a.out 10 11 200
```

- **argv** will have these elements
- They are **string** type



# Converting Data Types

```
#include <stdlib.h>

int atoi(const char *nptr);      // convert a string to an integer
double atof(const char *nptr);   // convert a string to a float
```

```
int x;
x = atoi(argv[1]);  // x gets the int value 10
```

# 2.9.4. Pointer Arithmetic

```c
#define N 10
#define M 20

int main(void) {
    // array declarations:
    char letters[N];
    int numbers[N], i, j;
    int matrix[N][M];

    // declare pointer variables that will access int or char array elements
    // using pointer arithmetic (the pointer type must match array element type)
    char *cptr = NULL;
    int *iptr = NULL;
    ...
```

```c
// make the pointer point to the first element in the array
cptr = &(letters[0]); //  &(letters[0])  is the address of element 0
iptr = numbers;       // the address of element 0 (numbers is &(numbers[0]))
```

# Pointer Arithmetic

```
// initialized letters and numbers arrays through pointer variables
for (i = 0; i < N; i++) {
    // dereference each pointer and update the element it currently points to
    *cptr = 'a' + i;
    *iptr = i * 3;

    // use pointer arithmetic to set each pointer to point to the next element
    cptr++;  // cptr points to the next char address (next element of letters)
    iptr++;  // iptr points to the next int address  (next element of numbers)
}
```

# Pointer Arithmetic

```
printf("\n array values using indexing to access: \n");
// see what the code above did:
for (i = 0; i < N; i++) {
    printf("letters[%d] = %c, numbers[%d] = %d\n",
           i, letters[i], i, numbers[i]);
}

// we could also use pointer arith to print these out:
printf("\n array values using pointer arith to access: \n");
// first: initialize pointers to base address of arrays:
cptr = letters;  // letters == &letters[0]
iptr = numbers;
for (i = 0; i < N; i++) {
    // dereference pointers to access array element values
    printf("letters[%d] = %c, numbers[%d] = %d\n",
            i, *cptr, i, *iptr);

    // increment pointers to point to the next element
    cptr++;
    iptr++;
}
```

# Pointer Arithmetic

```
 array values using indexing to access:
letters[0] = a, numbers[0] = 0
letters[1] = b, numbers[1] = 3
letters[2] = c, numbers[2] = 6
letters[3] = d, numbers[3] = 9
letters[4] = e, numbers[4] = 12
letters[5] = f, numbers[5] = 15
letters[6] = g, numbers[6] = 18
letters[7] = h, numbers[7] = 21
letters[8] = i, numbers[8] = 24
letters[9] = j, numbers[9] = 27

 array values using pointer arith to access:
letters[0] = a, numbers[0] = 0
letters[1] = b, numbers[1] = 3
letters[2] = c, numbers[2] = 6
letters[3] = d, numbers[3] = 9
letters[4] = e, numbers[4] = 12
letters[5] = f, numbers[5] = 15
letters[6] = g, numbers[6] = 18
letters[7] = h, numbers[7] = 21
letters[8] = i, numbers[8] = 24
letters[9] = j, numbers[9] = 27
```

Kahoot!

Ch 2d