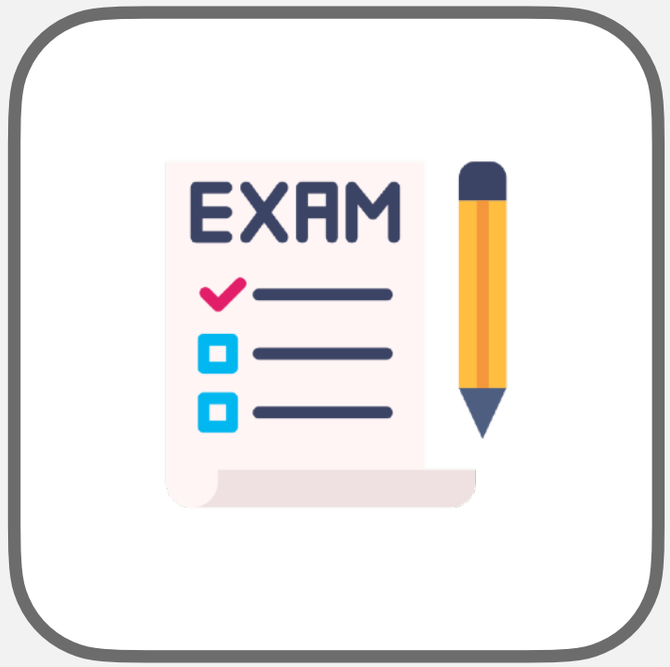
COP 3223H: Introduction to C Programming

Fall 2023

University of Central Florida

Dr. Kevin Moran

Week 15 - Class 11: Exam 3 Review



Administrivia

- Large *Programming Assignment 3* due today.
  - I *will* be holding office hours today.
- Course Experience Survey is due Today at 11:59 pm.
- Quiz 4 is due on Sunday by 11:59pm.
- *Exam 3* is Monday. December 4th!
  - We will review the format and content extensively today.

Today's Agenda

1. One Dynamic Memory topic
2. Exam Review

Exam 3 Format

- *2 Parts, In-class exam, closed book, 100 points total*
  - *<u>Part 1:</u>* Short Answer Questions
    - 4-5 questions
    - Either provide program output or answer with a code snippet or a few short sentences.
  - *<u>Part 2:</u>* Programming Questions
    - 4-5 questions with multiple parts
    - Either provide the output of a more complex program, or write several lines of code
- Focused on material from Weeks 11-15, but this builds on concepts from Weeks 1-5.
- You will have the ***entire*** exam period to complete the exam
- Please bring your UCF ID to the exam

Week 11 - Class 1: Strings Part I

Character Arrays (Strings)

- Strings are an array of characters.
- Each element of the array stores a character.
- The last character of the string array is the null character `\0`.
- `\0` helps the compiler know when it reaches the end of a string for certain function operations.
- Everything after `\0` in the character array is known as garbage values.

Character Arrays Declaration

- Declaring the string has the exact same procedures as declaring an array of ints and doubles.
- All you have to place is the data type char.
- Example:

```
char word[5];
```

Stack Space	
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	word[4] = ???
AA3	word[3] = ???
AA2	word[2] = ???
AA1	word[1] = ???
AA0	word[0] = ???

Character Arrays Declaration

- Strings have some unique syntaxes that allow for a proper declaration and initialization statement.
- C allows strings to be fully typed when be declared as long as the assignment operator is used.
- Double quotes are used to incorporate multiple characters.

```
char word[10] = "Pikachu";
```

Stack Space	
AA9	word[9] = ???
AA8	word[8] = ???
AA7	word[7] = ‘\0’
AA6	word[6] = ‘u’
AA5	word[5] = ‘h’
AA4	word[4] = ‘c’
AA3	word[3] = ‘a’
AA2	word[2] = ‘k’
AA1	word[1] = ‘i’
AA0	word[0] = ‘P’

Character Arrays Declaration

- Strings also have the initializer list like the ones we saw when working integers and doubles.
- They really don't need to be used, but you should know they exist.
- It's just extra time-consuming typing.
- You also MUST include the null character!

```
char word3[10] = {'P','i','k','a','c','h','u','\0'};
```

Char Array Declaration and Initialization

- you may think we can separate the declaration and initialization statements for strings, however we can't. It can only
- `word` is it expects an addr

There is another way we can separate the declaration and initialization statement. It involves the use of a string library function called `strcpy`. We see this very soon!

```
word = "Pikachu";
```

Collecting a String with `scanf`

- Collecting input for a string follows very similar procedures as collecting other data types.
- Two Differences:
  - Placeholder %s
  - No address operator (&)

```c
char pokemon[10];

scanf('%s', pokemon);

printf("the pokemon is %s\n", pokemon);
```

Week 11 - Class II: Strings Part II

Limitations of using `scanf` for Strings

- Strings are just an array characters.
- We can technically form words and even phrases.
- Does a single `scanf` statement with ONE placeholder allow multiple words to be collected into some string?

```c
char phrase[30];

scanf('%s', phrase);

printf("the phrase is %s\n", phrase);
```

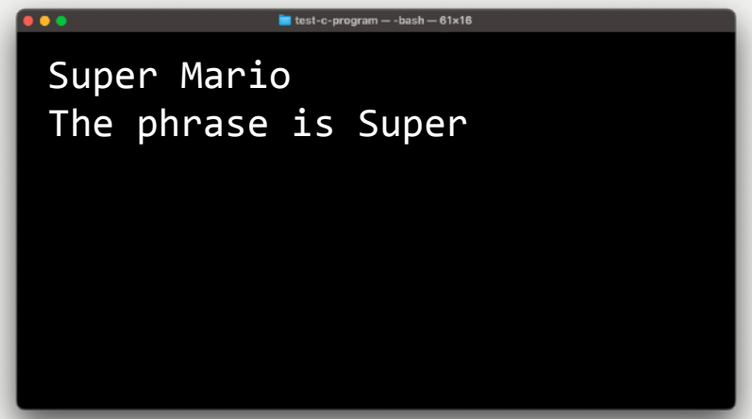
What if we type “Super Mario”?

Limitations of using `scanf` for Strings

- Strings are just an array characters.
- We can technically form words and even phrases.
- Does a single `scanf` statement with ONE placeholder allow multiple words to be collected into some string?

```c
char phrase[30];

scanf('%s', phrase);

printf("the phrase is %s\n", phrase);
```

What if we type “Super Mario”?



Limitations of using `scanf` for Strings

- Strings are just an array characters

Scanf stops reading values for a string when it encounters the whitespace character (‘ ‘)!

Clearing the Input Buffer

- Here is a simple user defined function that you can use to clear the input buffer.

- This function should be only when the buffer needs to be cleared. In other words, if your code is skipping an input collection statement, then that means the buffer had readable content.

```
void clearBuffer(){
    while(getchar() != '\n');
}
```

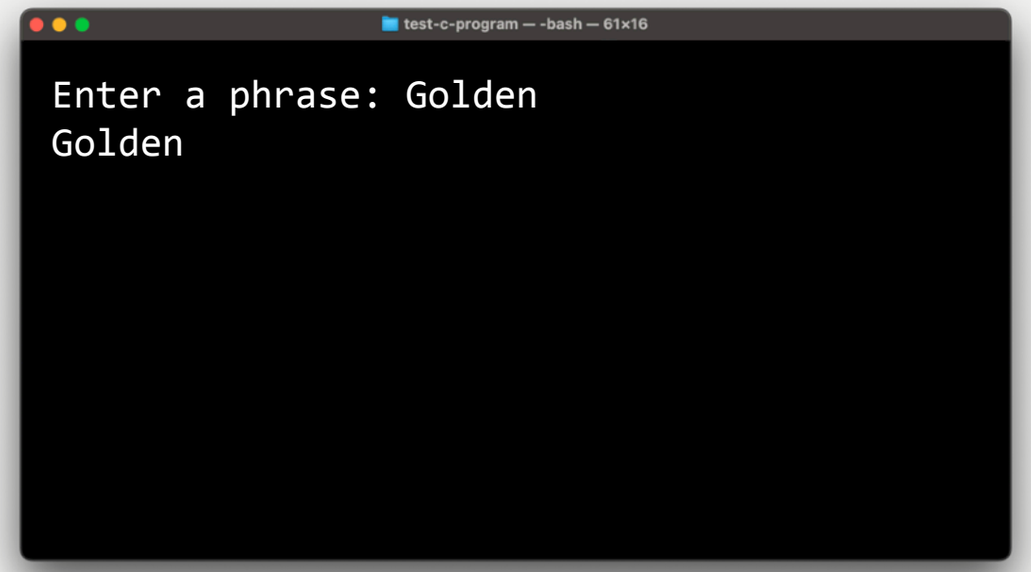
`gets()` and `puts()`

- Something we've noticed is that scanf has some limitations.
- scanf only allows one word to be read.
- How can programmers input a sentence as string?
- `gets()` is a simple function that allows user to input more than one word that can be stored in a character array (allowing whitespaces).
- `puts()` is another way to display a string onto the screen.

`gets()` and `puts()` Example

```c
char phrase[10];
printf("Enter a phrase: ");
gets(phrase);

puts(phrase);
```

```
'gets' has been explicitly marked
deprecated here
__deprecated_msg("This function is
provided for compatibility reasons only.
Due to security concerns inherent in the
design of gets(3), it is highly
recommended that you use fgets(3)
instead.")
```



fgets()

- `fgets()` is similar to gets(), but with extra syntax.
- `fgets()` meets the possible that gets() raises.
- `fgets()` takes three arguments
  - Array
  - String Length Limit
  - File to read from (stdin which is standard input)
- **`fputs()`** works like **`puts()`**, except that it doesn't automatically append a newline

Understanding how `fgets()` works

- `fgets` is a function that allows to process characters (including a whitespace characters) until newline character (‘\n’) is read

- If `fgets` receives a strings bigger than the provided limit, it will just append the null character in the last element and send the remaining characters into the buffer space

The String library

Function	Stack Space
`strcpy()`	Makes a copy of source, a string, in the character array accessed by dest:
`strncpy()`	Makes a copy of up to n characters from source in dest: strncpy(dest, source, 5) stores the first five characters of the source and does NOT add a null character.
`strcat()`	Appends source to the end of dest: strcat(dest, source)
`strncat()`	Appends up to n characters of source to end of dest, adding the null character if necessary.
`strcmp()`	Compares s1 and s2 alphabetically. Returns a negative value if s1 should precede s2, a zero if strings are equal, and a positive value if s2 should precede s1 in an alphabetized list. strcmp(s1,s2)
`strncmp()`	Compares the first n characters in s1 and s2 returning positive, zero, and negative values like strcmp.
`strlen()`	Returns the number of characters in s, not counting the terminating null. strlen(s)
`strtok()`	Breaks the parameter string into tokens finding groups of characters separated by any of the delimiter characters. Each group is separated with '\0'.
`strchr()`	Returns a pointer to the first location of a character located in the string. Null is returned if character is not found.
`strpbrk()`	Return a pointer to the first location in the strings that holds any character found in another string.
`strchr()`	Returns a pointer to the last occurrence of a character in the string. Null is returned if character not found.
`strstr()`	Returns a pointer to the first occurrence of string s2 in string s1. Null is returned if character not found.

Week 11 - Class III: Strings Part III

`strlen`

- The user defined function you just saw on the previous slide is already implemented in the **`string.h`** file.
  - Function takes one parameter which is the address of the string!
- It uses the same idea from our own custom function.
  - Start at the first address passed in the function.
  - Iterate through the string and count each character until the first null character is found.
  - Return the counter value.
  - IMPORTANT! The value return tells what index contains the null character!

`strlen` Example

```
char word[100] = "Mondays";
int len = strlen(word);
```

len = 7;

`strlen` Example 2

```
char word[100] = "Mondays";
word[3] = "\0";
int len = strlen(word);
```

len = 3;

strcpy

- The string library provides a function that allows you to completely copy the contents of a string into another including the null character (‘`\0`’).
- This is a very common task to do in many problems.
  - Hence why it even exists!
- The function takes two parameters.
  - The first parameter is the destination string (an address)
    - Where the contents copied need to be stored in memory
  - The second parameter is the source string (an address)
    - Where the contents that are needed to be copied are stored in memory
    - Important: You can also place a string literal has the source. This is the proper

`strcpy` Example

```
char string1[8];
char string2[8] = "Cakes";

strcpy(string1, string2);
strcpy(string1, "Cookies");
```

Something to Avoid

- Since we have learned that arrays are simply pointers, you might try to some sort statement like this…

```
char *string = "Hello!";
```

- This is a
- Now tha

This code will crash!!

```
strcpy(string, "hi");
```

Substrings and `strncpy`

- Substrings are a fragment of a longer string
- strncpy is the function to use to generate substrings of a string
  - The function takes 3 parameters
    - The first parameter is the destination (address)
    - The second parameter is the source (address)
    - The third parameter is the number of characters (integer)
- Examples
  - String called “Andrew”
  - Substring of this is “And”
  - Substring of this is “drew”
  - “Adw” is NOT a substring!!

`strcat`

- Concatenation is taking two strings and joining them together as one string. It basically appending one string to the end of the another one.
- Example: “Progr” concatenated with “amming” would be “Programming”
- `strcat` and strncat are the string functions that handle concatenation
  - strcat appends an entire string (simply copies the entire source string)
    - First argument is the destination string (address)
    - Second argument is the source string (address)
  - `strncat` appends the first n characters of a string (handles the null character properly)
    - First argument is the destination string (address)
    - Second argument is the source string (address)
    - Third argument is the number of characters (integer)

strcat

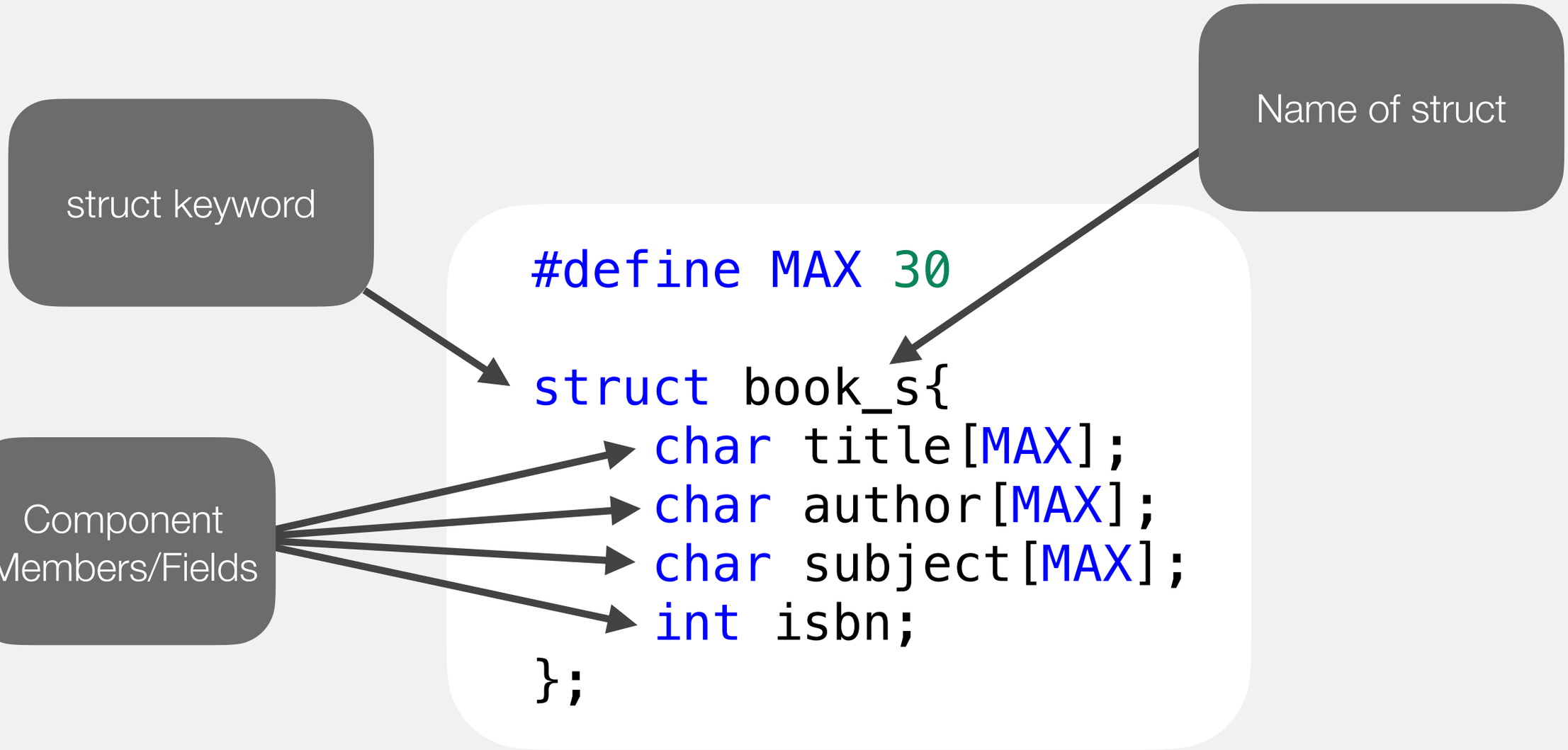
```c
char string5[8] = "Vanilla";
char string6[8] = "Cookie";

strcat(string5, string6);

printf("string5 = %s\n", string5);
printf("string6 = %s\n", string6);
```

Week 12 - Class 1: Structs

User Defined Structure Types

- A database is a collection of information stored in a computer's memory or in a disk file.
- A database is subdivided into records, which are a collection of information about one data object.
- The structure of the record is determined by the structure of the object's data type.
- C provides several ways to define structures.

User Defined Structure Syntax



Organizing User-defined `structs`

- The structure definition must be placed at the top of your C file. More specifically, it should be between your preprocessor statements and function prototypes.

```c
// preprocessor statements
#include<stdio.h>
#define MAX 30

struct book_s{
    char title[MAX];
    char author[MAX];
    char subject[MAX];
    int isbn;
};

// user defined function prototypes

int main(void){

return 0;

}

// user defined function definitions
```

Assigning Values to the Components of a `Struct`

- C has a simple operator called the direct selection operator (`.`).
- This allows us to properly access and assign values in the structure.

```
struct book_s mybook;

strcpy(mybook.title, "Julius Cesar");
strcpy(mybook.author, "William Shakespeare");
strcpy(mybook.title, "Play");
mybook.isbn = 1234;
```

.title	“Julius Caesar”
.author	“William Shakespeare”
.subject	“Play"
.isbn	1234

Stack Space Visualization with `Structs`

```
#define MAX 30

struct book_s{
    char title[MAX];
    char author[MAX];
    char subject[MAX];
    int isbn;
};

int main(void){

struct book_s mybook;

return 0;

}
```

Stack	Space
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	
AA3	my book.isbn
AA2	mybook.subject
AA1	mybook.author
AA0	mybook, my book.title

Typedef Structures

- As you may of saw, every time we must use struct (such as a declaration), we are required to type out the keyword struct.

- C provides a special keyword that will allow programmers to avoid using the struct keyword.

- **`typedef`** is a special keyword that allows C to assign a name to some type.

```c
#define MAX 30
typedef struct{
    char title[MAX];
    char author[MAX];
    char subject[MAX];
    int isbn;
}book_t;

int main(void){

book_t mybook;

strcpy(mybook.title, "Julius Cesar");
strcpy(mybook.author, "William Shakespeare");
strcpy(mybook.title, "Play");
mybook.isbn = 1234;

return 0;

}
```

Structs and Function Parameters

- Since structures are basically special variables, we can also use them as input/output parameters for user defined functions.
- We can perform both pass-by-value and pass-by-reference.
- With structures it is preferred that they are passed by reference since it easier (on the stack space) to pass the address (8 bytes always) of the struct rather than coping all the values of each component (number of bytes varies but could most likely be bigger than 8).

```
void displayBook(book_t mybook){
    printf("%s\n, mybook.title");
    printf("%s\n, mybook.author");
    printf("%s\n, mybook.subject");
    printf("%s\n, mybook.isbn");
}
```

Pass by Value

```
void displayBook(book_t *mybook){
    printf("%s\n, mybook.title");
    printf("%s\n, mybook.author");
    printf("%s\n, mybook.subject");
    printf("%s\n, mybook.isbn");
}
```

Pass by Reference

Indirect Component Selection Operator

- The indirect component selection operator is the character sequence -> placed between a pointer variable and a component name create a reference that follows the pointer to a structure and selects the component.

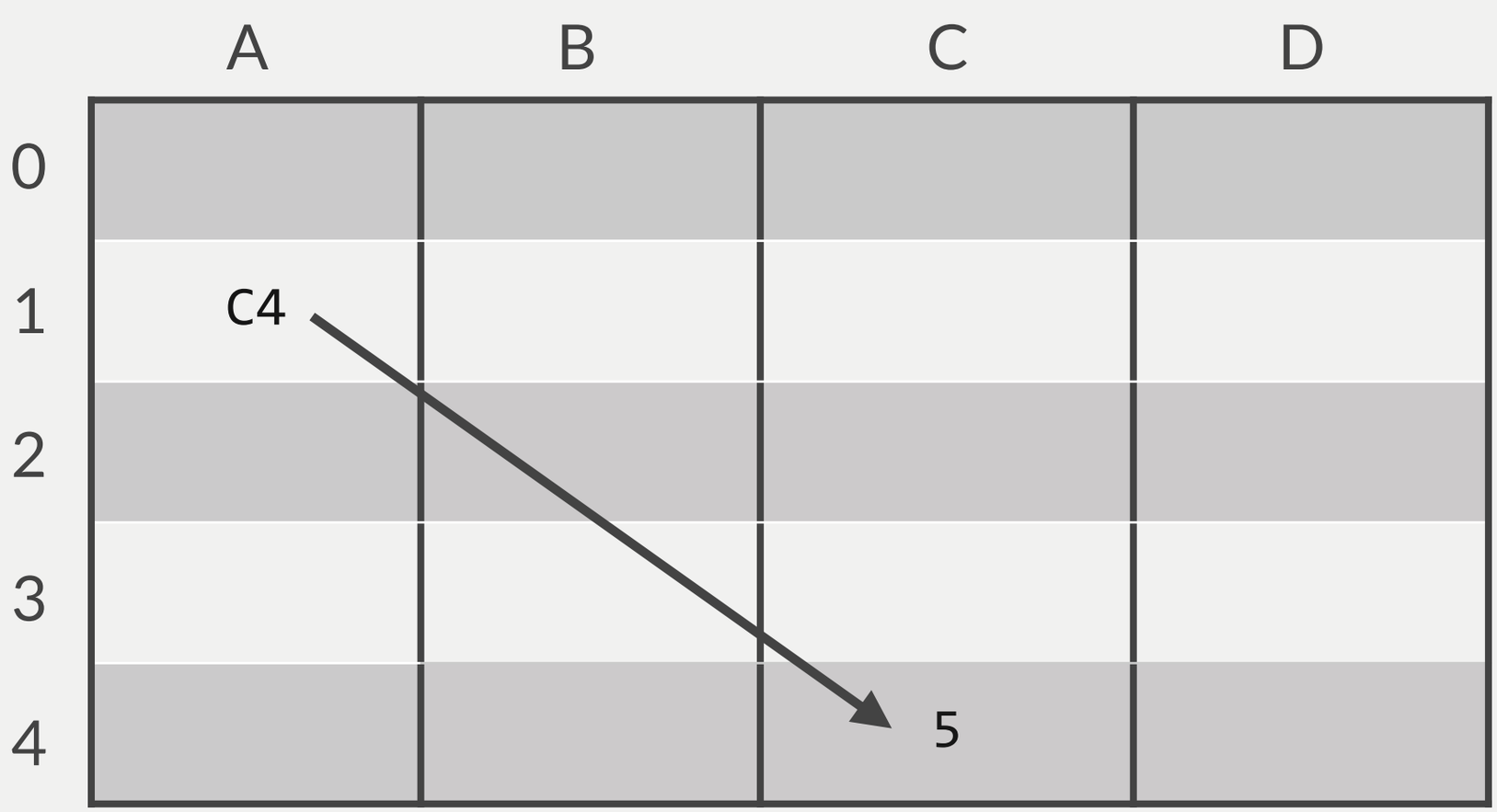
- While first one is valid to use, it can be a bit cumbersome to use, which is why C provides the indirect operator.

```
book_t *book_ptr = &mybook;

char title[MAX] = (*book_ptr).title;
char title2[MAX] = book_ptr->title;
```

Week 12 - Class II: Dynamic Memory Allocation

Pointer Refresher

- Special data type that holds an address a memory
- * is the deference operator
- & is the address operator




```c
int x = 5; //C4
int *ptrx = &x; // C4
printf("%d\n", *ptrx); // 5
```

Static Memory

- For this entire course, we have been provided by the OS memory to utilize for our program in the stack space.
  - Limitations:
    - Cannot change the size we are given
      - How can this be potentially bad?
- At compilation time (when code compiles) the memory allocation for the program is predetermined.
- “Get what you get and don’t get upset!”

Dynamic Memory

- Sometimes we may not know how much we really need for a program.
  - Example
    - Array Allocation – what if we allocated 5 elements and realized we need more elements?
- Memory that we can change in size during the program run (different then compilation time).
- Extra memory that we may need during a program is in the ***heap*** space.

`sizeof` Operator

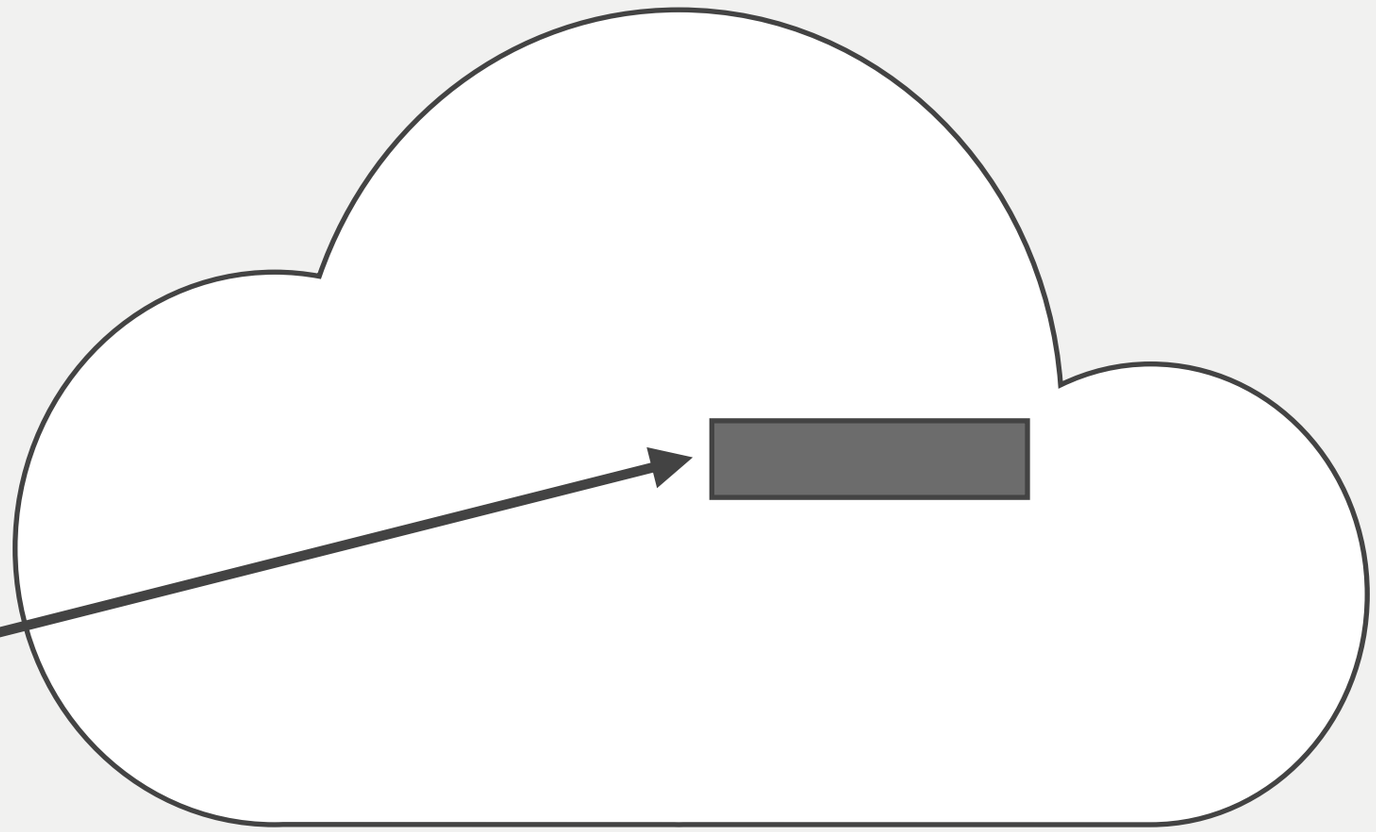
- Returns the size (in bytes) of a data type
  - `sizeof(int)` returns 4 bytes
  - `sizeof(double)` returns 8 bytes
  - `sizeof(char)` returns 1 byte

`malloc()`

- Part of the `stdlib.h` file
- Allocates a single memory block of any built in or user-defined type
- Function that returns memory based on the number of bytes needed
- Parameter of the function takes the number of bytes needed
- The function returns an address or NULL
  - What kind of variable will hold that address?
  - What happens if NULL is returned?
- Heap – region of memory in which the function `malloc` dynamically allocates blocks of storage

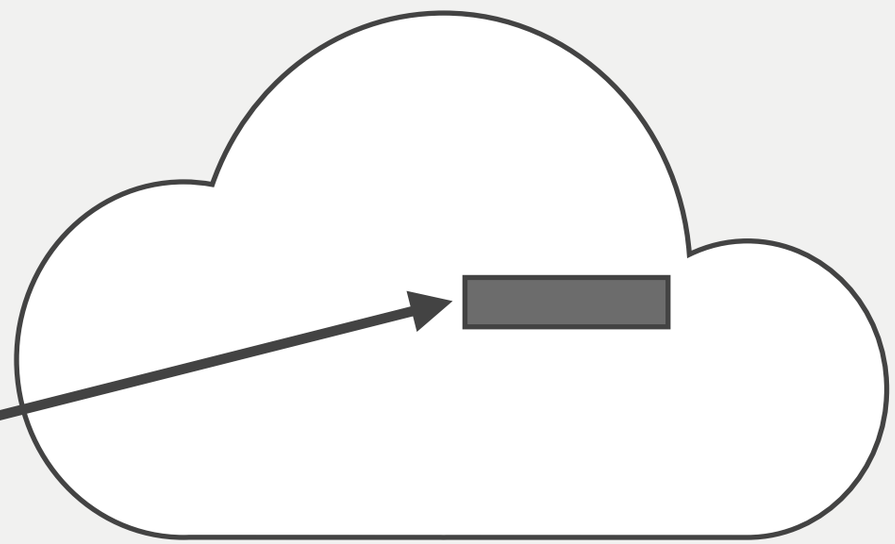
Stack and Heap Space

Stack	Space
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	
AA3	int *ptr
AA2	
AA1	
AA0	

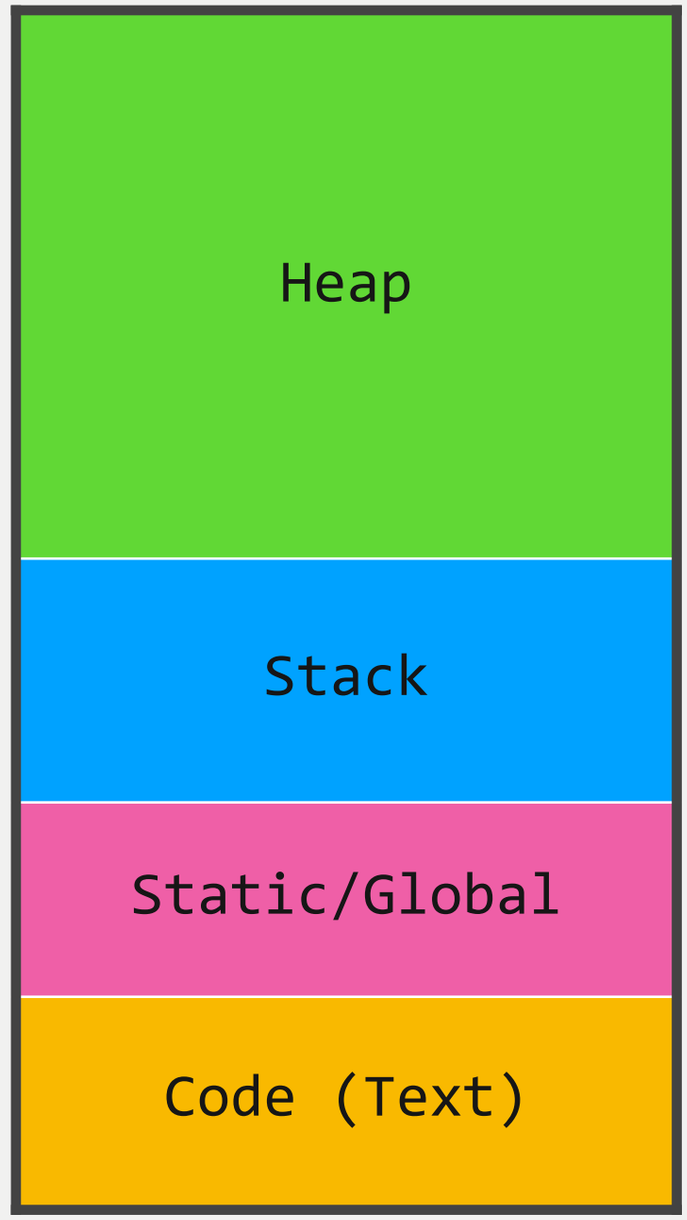


Stack and Heap Space

Stack	Space
AA9	
AA8	
AA7	
AA6	
AA5	
AA4	
AA3	int *ptr
AA2	
AA1	
AA0	



Program
Memory Allocation



malloc() + free() Example

```
int *pointer;
pointer = malloc(sizeof(int));
free(pointer);
```

Week 13 - Class 1: Dynamic Arrays

Variable Length Arrays ☹

- The arrays we are dealing with use static memory (stack space).
- Static means no flexibility in changing the size of memory required.
- Adding this flexibility results in dynamic memory
- We will study this at the end of the semester.
- Never use variables when declaring an array as you can have potential danger in what the value a variable can hold.
- VLAs pose danger if we accidentally change a value to a size that can’t be properly handled in memory.

Dynamic Array Example

```
int size;

printf ("How many elements would you like: "); scanf(“%d", &size);

int *array = (int *) malloc(size * sizeof(int));

for(int x = 0; x < size; ++x){

    printf("Enter a value: ");
    scanf(“%d", &array[x]) ;
}

for(int x = 0; x < size; ++x){

    printf(“array[%d] = %d\n”, x, array[x]) ;
    free (array) ;
    array = NULL;
}
```

About `sizeof()`

- You have learned that the sizeof operator returns the number of bytes.
- Since dynamic memory returns a heap for a pointer to point at, it will not return the number of elements but instead the size of the pointer.
- So how would you keep track of valid entries in a dynamic array?
  - ***Use a Regular Variable***

`free()`

- After **`free()`** is called, the value in the parameter doesn't change.
- Only significant is that the memory is labeled free from the OS perspective
- What do you think this means?
- What should we do with the pointer that is passed in the function call.
  - Set it to NULL!!!

Week 13 - Class II: Dynamic Arrays & Structs

Indirect Component Selection Operator

- The indirect component selection operator is the character sequence -> placed between a pointer variable and a component name create a reference that follows the pointer to a structure and selects the component.

- While first one is valid to use, it can be a bit cumbersome to use, which is why C provides the indirect operator.

```
book_t *book_ptr = &mybook;

char title[MAX] = (*book_ptr).title;
char title2[MAX] = book_ptr->title;
```

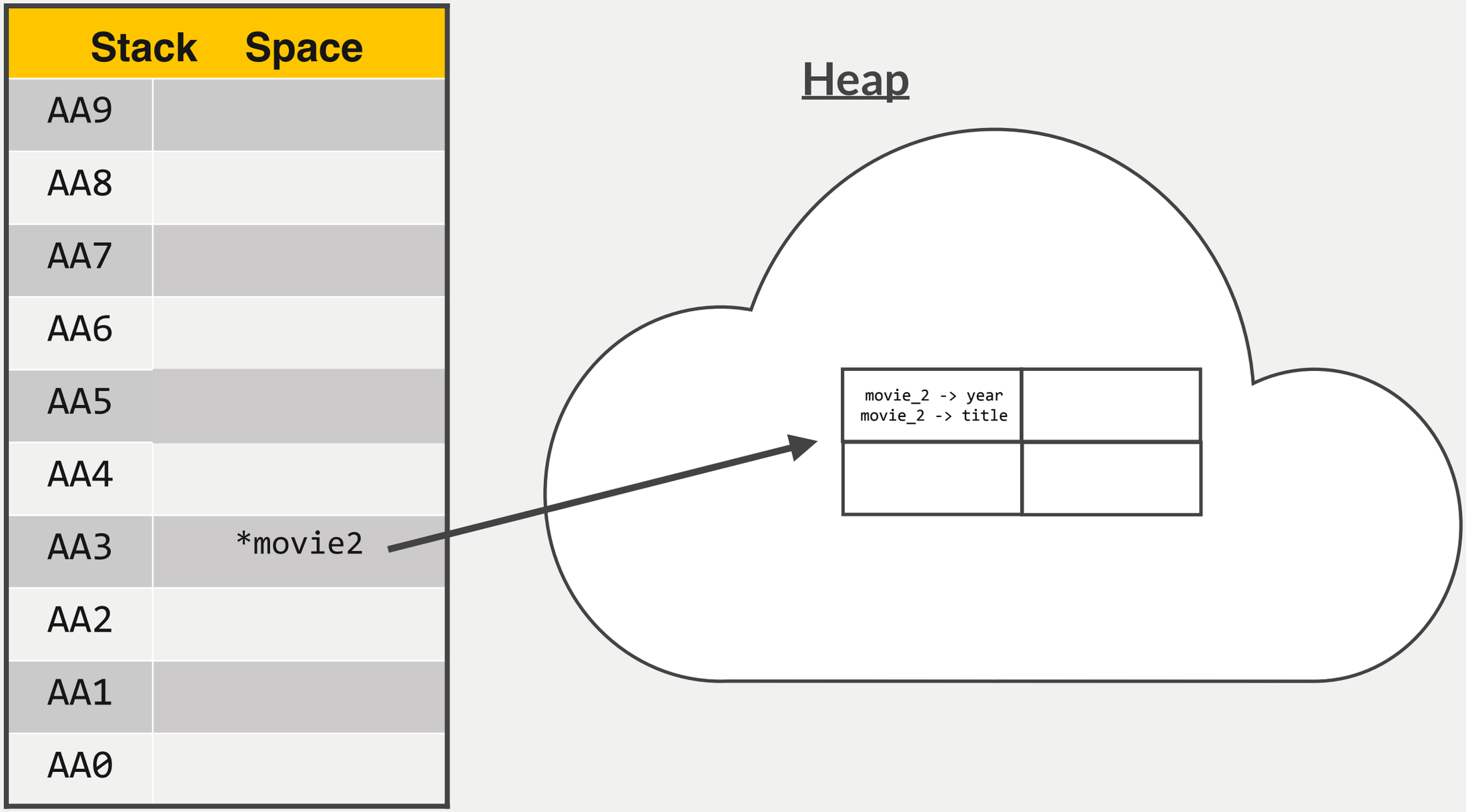
Dynamic Memory and Structs

- Similar syntax to dealing with primitive data types.

```
typedef struct{
    int year;
    char title[30];
}movie_t;
```

```
movie_t movie1; // declared in stack space
movie_t *movie2 = (movie_t *) malloc(sizeof(movie_t)); // declared in heap space

free(movie2);
```

Visualizing Dynamic Struct Allocation



Populating the Components of a Dynamic Struct

- Similar syntax to dealing with primitive data types.

```
typedef struct{
    int year;
    char title[30];
}movie_t;
```

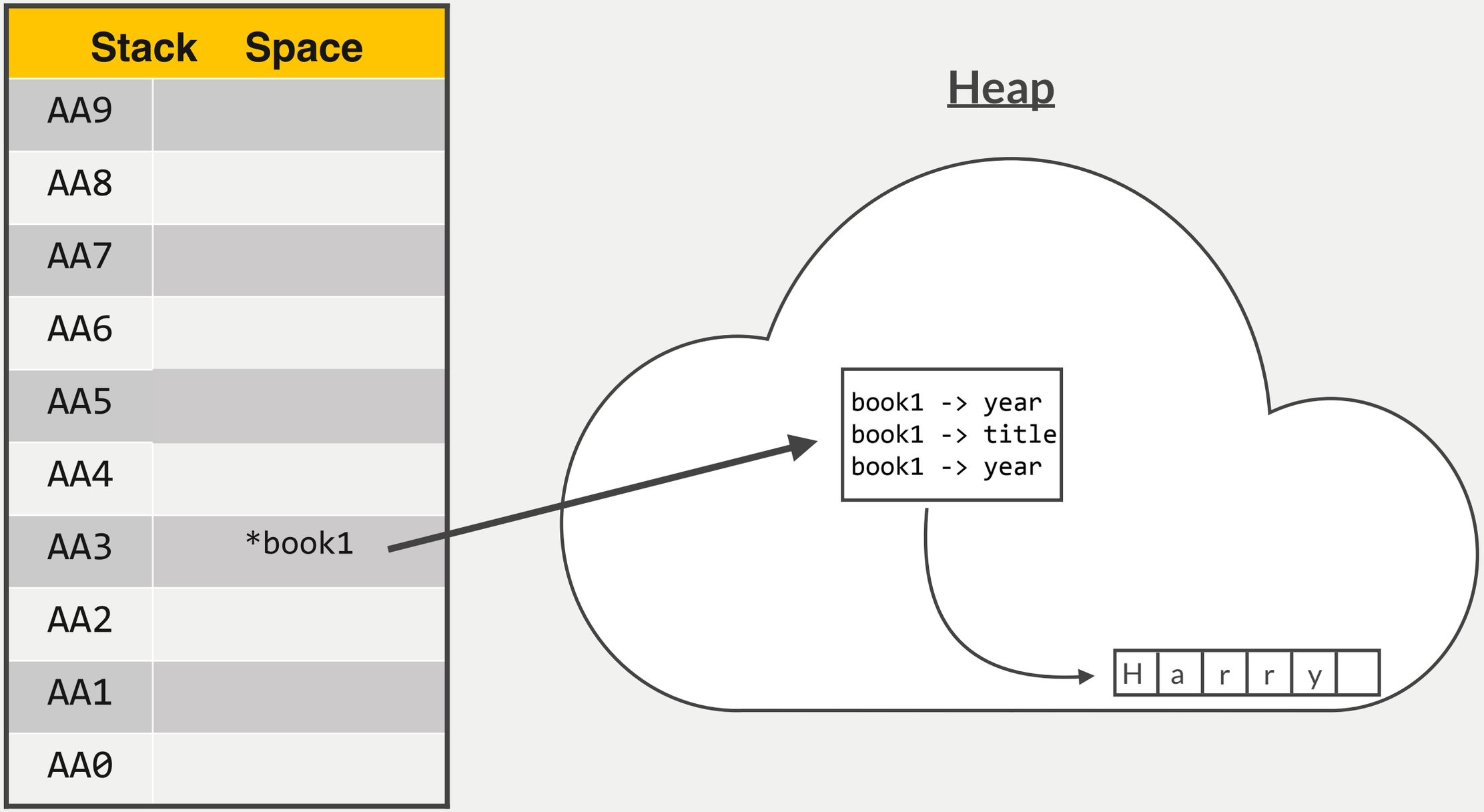
```
movie_t *movie2 = (movie_t *) malloc(sizeof(movie_t));
strcpy(movie2->title, "Avatar");
movie2->year = 2022;

printf("%s\n", movie2->title) ;
printf("%d\n", movie2->year);

free (movie2) ;
```

Dynamic Struct Components

```
typedef struct{
    int year;
    char * title;
    char * author;
}book_t;
```

Week 14 - Class 1: Dynamic Structs II

Visualizing Dynamic Struct Allocation



Deallocating Memory

```c
typedef struct{
    int year;
    char * title;
    char * author;
}book_t;
```

Anything
wrong with this?

```c
book_t * book1 = (book_t *) malloc(sizeof(book_t));

book1->title = (char *) malloc(sizeof(char) * 50);

strcpy(book1->title, "Harry Potter and the Goblet of Fire”);

book1 -> year = 1998;

free(book1);
```

Arrays and structs

```
typedef struct{
    int year;
    char * title;
    char * author;
}book_t;
```

- Guess What! We can also have a dynamic array of structs that contains dynamic components!
- The same rules apply that we have been learning with dynamic memory!

```
book_t * mylibrary = (book_t *) malloc(sizeof(book_t) * 10);
```

Week 15 - Class I & II: Linked Lists I & II

What is a Linked List?

- A linked list is a sequence of nodes in which each node but the last contains the address of the next node.
- *When to use a Linked List?*
  - You need constant-time insertions/deletions
  - You don’t know how many items will be in a list
  - You don’t need random access to elements
  - You want to be able to insert items into the middle of the list (Priority Queues)

Setting Up a Linked List in C

- We will use a typedef struct to set up the node that contains the data and pointer to the respective nodes in the linked list.

- Now you may notice **`node_s`** after struct. Why is that necessary? This allows C to know that the node will point to another type **`node_t`**. If you don't, your code won't compile.

```
typedef struct node_s{
    struct node_s * nextptr;
    int data;
}node_t;
```

Linked List Operations

- We can perform the basic operations with a linked lists
  - Insert
  - Remove
  - Display (traverse)
  - Search
  - Empty

Linked List Node Insertion

- *Insert a new node into the list*
- *Insert at the end of the list*
  - Traverse until nextptr is NULL
  - Make last nextptr new node
- *Insert in between two nodes*
  - Traverse to the position of the list
  - You will need reset the nodes pointers to properly maintain the linked list. It's good practice to draw the list to visually see how pointers work!

Linked List Node Removal

- See if the node even exists in the list
- Similar with inserting between two nodes, you will need to reset the previous adjacent node's next pointer to the old removed node's next pointer.
- Draw the picture to visualize!

Linked List Node Display

- Traverse each node to display the information
- Simple loop traversal

Linked List Node Search

- Traverse the list until the data you are seeking is found.
- Simple loop traversal

Acknowledgements

Slides adapted from Dr. Andrew Steinberg's COP 3223H course