Week 13: Dynamic Memory, Linked Lists

CIT-593, Spring 2022

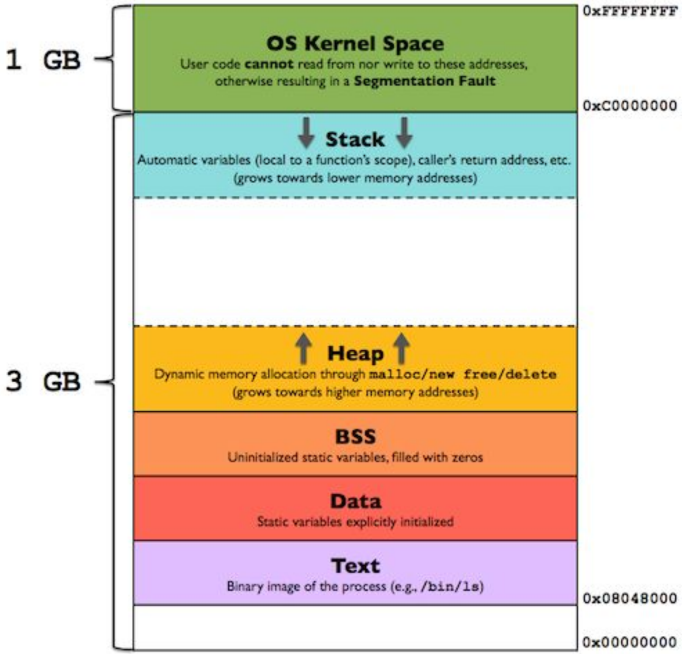
Sarah Santos

April 13, 2022

Penn Engineering

What is the heap?

A region of memory used for dynamic memory allocation



OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**

Stack
Automatic variables (local to a function's scope), caller's return address, etc.
(grows towards lower memory addresses)

Heap
Dynamic memory allocation through `malloc/new` `free/delete`
(grows towards higher memory addresses)

BSS
Uninitialized static variables, filled with zeros

Data
Static variables explicitly initialized

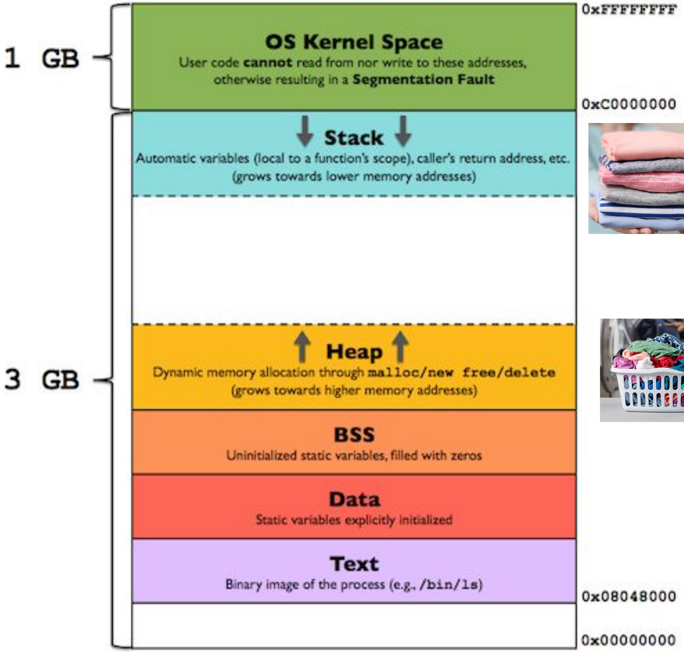
Text
Binary image of the process (e.g., `/bin/ls`)

image credit: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

What is the heap?

A region of memory used for dynamic memory allocation

Note: This image is flipped compared to how Dr. Farmer draws memory, but the stack still "grows towards 0"



1 GB

3 GB

0xFFFFFFFF

OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**

0xC0000000

Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)

Heap
Dynamic memory allocation through `malloc/new` `free/delete` (grows towards higher memory addresses)

BSS
Uninitialized static variables, filled with zeros

Data
Static variables explicitly initialized

Text
Binary image of the process (e.g., `/bin/ls`)

0x08048000

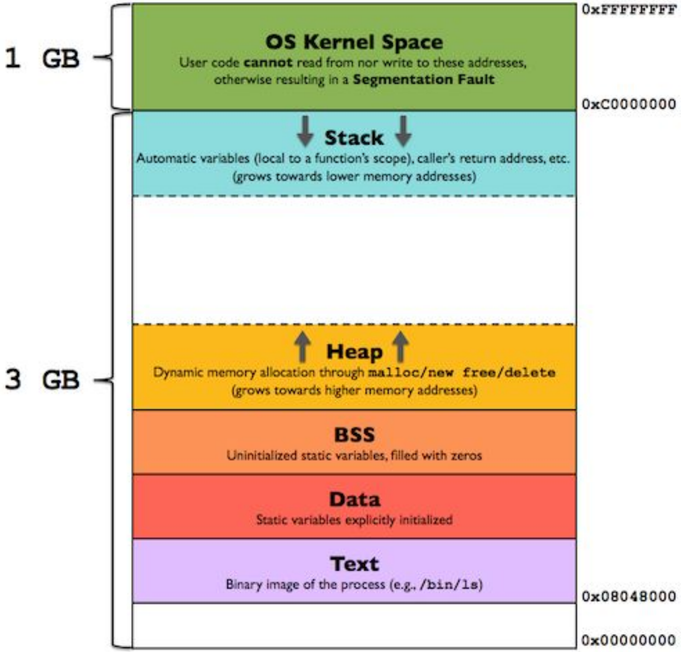
0x00000000

Higher Addresses

Lower Addresses

image credit: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

What is the heap?



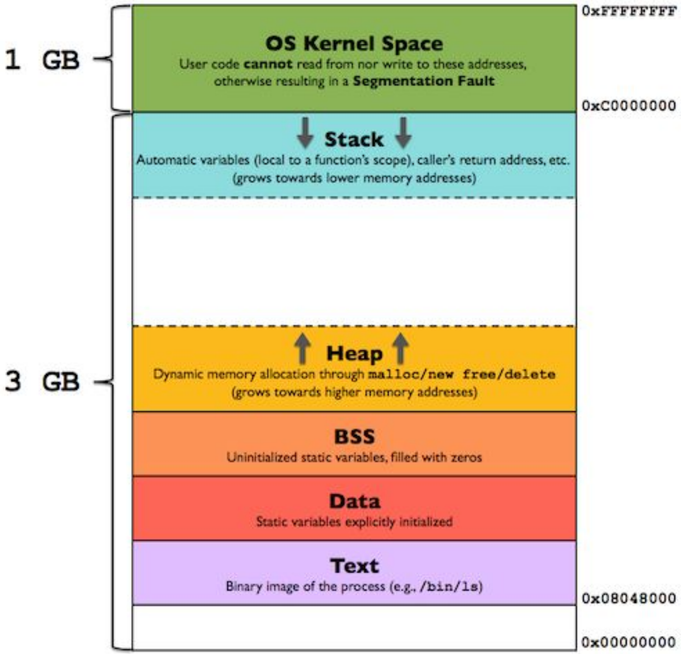
1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

image credit: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

- Large Block of unorganized Memory
  - As opposed to the stack, with its stack frames

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

What is the heap?

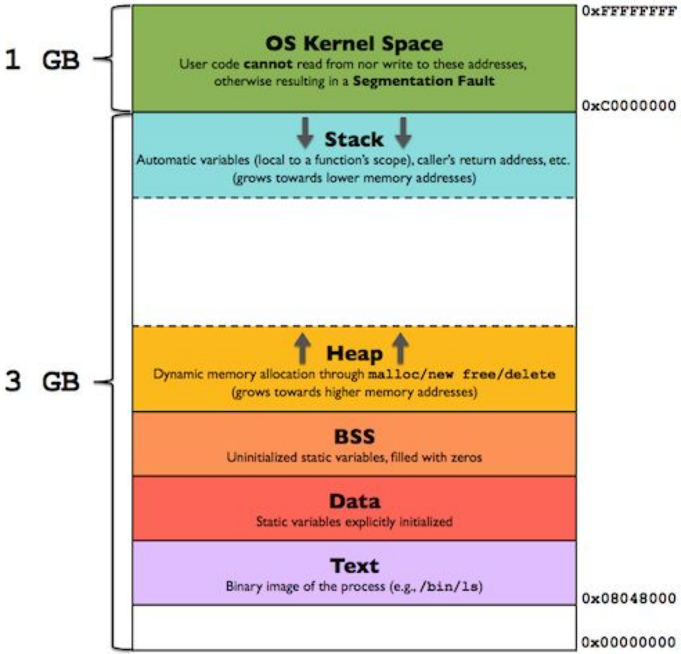

1 GB
3 GB
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000

- Large Block of unorganized Memory
  - As opposed to the stack, with its stack frames
- Memory management done by programmer

image credit: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

What is the heap?



0xFFFFFFFF
1 GB
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
3 GB
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

- Large Block of unorganized Memory
  - As opposed to the stack, with its stack frames
- Memory management done by programmer
- **Heap memory persists outside scope of function!**
  - **Stack frames destroyed when function returns**

image credit: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

Penn Engineering

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

How do we interact with the heap?

Two Main Functions

Helper Function

How do we interact with the heap?

Two Main Functions

memory allocation

memory deallocation

Helper Function

returns size of data type in bytes

How do we interact with the heap?

Two Main Functions

`void *malloc(size_t size);`	*memory allocation*
`void free(void *ptr);`	*memory deallocation*

Helper Function

`size_t sizeof(type);`	*returns size of data type in bytes*

`void *malloc(size_t size)`

- So far: size of array must be known at runtime
  - Memory allocated on stack

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

`void *malloc(size_t size)`

- So far: size of array must be known at runtime
  - Memory allocated on stack
- Don’t know array size at runtime? Use malloc()!

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

void *malloc(size_t size)

- So far: size of array must be known at runtime
  - Memory allocated on stack
- Don’t know array size at runtime? Use malloc()!
- Allocates contiguous block of memory in heap

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

`void *malloc(size_t size)`

- So far: size of array must be known at runtime
  - Memory allocated on stack
- Don’t know array size at runtime? Use malloc()!
- Allocates contiguous block of memory in heap
- Returns pointer to allocated memory

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

`void *malloc(size_t size)`

- Use `sizeof(type)` to get the size of a type

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

`void *malloc(size_t size)`

- Use `sizeof(type)` to get the size of a type
- Recall: void* is a generic pointer type
- May need to cast memory to our desired type

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

void *malloc(size_t size)

- Use `sizeof(type)` to get the size of a type
- Recall: void* is a generic pointer type
- May need to cast memory to our desired type
- ...but we can usually just do something like this:

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

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

Check what malloc() returns!

Always check the return value

- ○ Returns a pointer to the memory block if success
- ○ Returns null pointer if failed

Penn Engineering

Adapted from slides by Ludwig Zhao and Dr. Farmer

Check what malloc() returns!

Always check the return value

- Returns a pointer to the memory block if success
- Returns null pointer if failed

Example from lecture:

```
int length = 2;
int* int_array = NULL;

int_array = malloc (length * sizeof(int)) ;
if (int_array == NULL) return 1 ;
```

Adapted from slides by Ludwig Zhao and Dr. Farmer

Check what malloc() returns!

Always check the return value

- Returns a pointer to the memory block if success
- Returns null pointer if failed

Example from lecture:

```
int length = 2;
int* int_array = NULL;

int_array = malloc (length * sizeof(int)) ;
if (int_array == NULL) return 1 ;
```

Penn Engineering

Adapted from slides by Ludwig Zhao and Dr. Farmer

`void free(void *ptr)`

The bread to malloc's butter

- Frees memory allocated by malloc() call

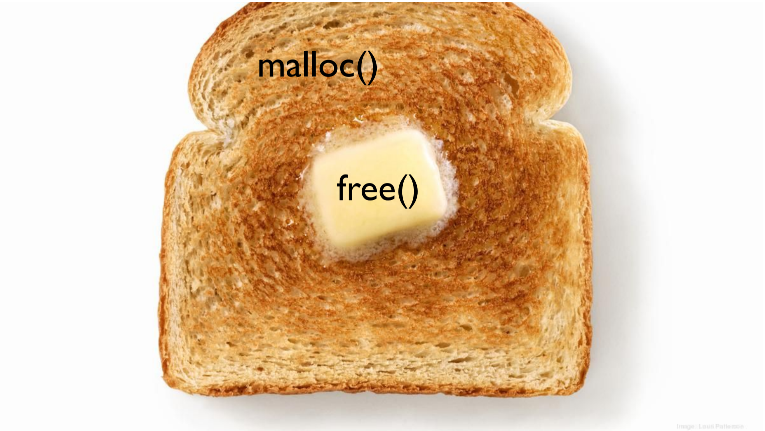


Image : Lauri Patterson

(not a metaphor for functionality, just to remind you to make sure every `malloc` comes with a `free`. bread without butter is not good imo)

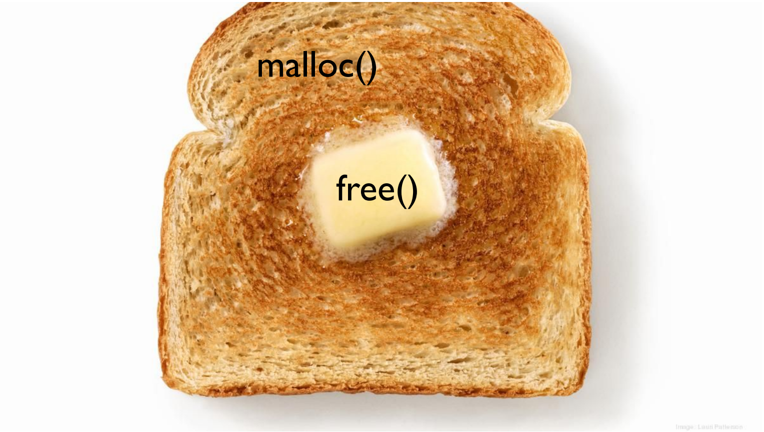
Penn Engineering

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

void free(void *ptr)

The bread to malloc's butter

- Frees memory allocated by malloc() call
- Every malloc() must be free()'d eventually
  - Else: memory leaks!



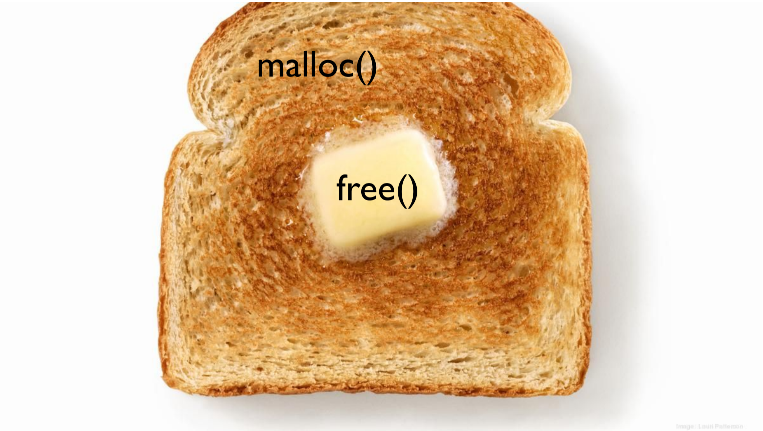
(not a metaphor for functionality, just to remind you to make sure every `malloc` *comes with a* `free`*. bread without butter is not good imo)*

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

void free(void *ptr)

The bread to malloc's butter

- Frees memory allocated by malloc() call
- Every malloc() must be free()'d eventually
  - Else: memory leaks!
- Memory is freed, but technically speaking..



(not a metaphor for functionality, just to remind you to make sure every `malloc` comes with a `free`. bread without butter is not good imo)

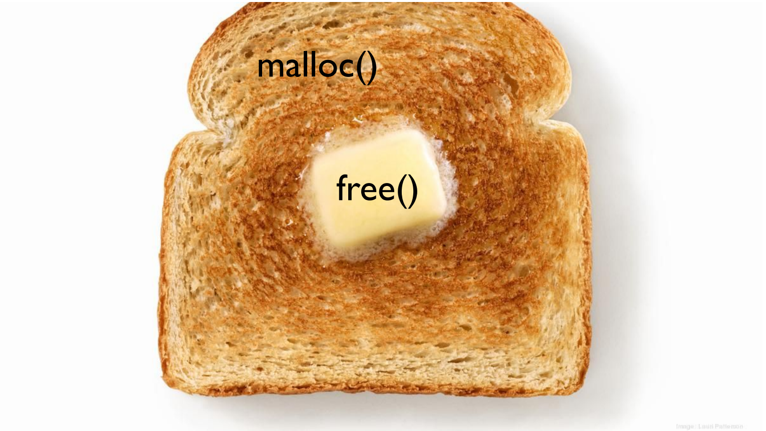
Penn Engineering

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

void free(void *ptr)

The bread to malloc's butter

- Frees memory allocated by malloc() call
- Every malloc() must be free()'d eventually
  - Else: memory leaks!
- Memory is freed, but technically speaking..
  - Pointer value on stack not modified (too much overhead)
    - *i.e. left your table at restaurant, but you still have the "text reminder about your reservation"*



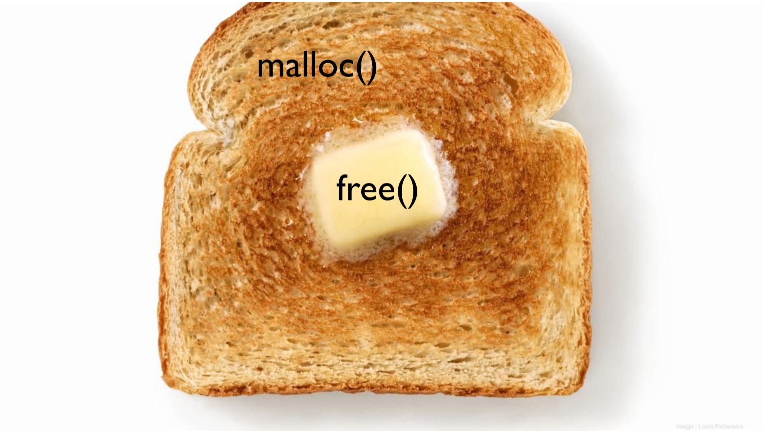
(not a metaphor for functionality, just to remind you to make sure every `malloc` comes with a `free`. bread without butter is not good imo)

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

`void free(void *ptr)`

The bread to malloc's butter

- Frees memory allocated by malloc() call
- Every malloc() must be free()'d eventually
  - Else: memory leaks!
- Memory is freed, but technically speaking..
  - Pointer value on stack not modified (too much overhead)
    - *i.e. left your table at restaurant, but you still have the "text reminder about your reservation"*
  - Allocation on heap no longer valid for use
    - *i.e. left your table at restaurant, but your dirty dishes and food scraps still there until cleaned*



(not a metaphor for functionality, just to remind you to make sure every `malloc` *comes with a* `free`*. bread without butter is not good imo)*

Penn Engineering

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

void free(void *ptr)

- Do not free() memory not returned by malloc()

- **Bad:**

```
int i[] = {1, 2, 3, 4, 5};
free(*i); // DO NOT DO THIS
```

- **Good:**

```
int* intArray = malloc(sizeof(int) * 5);
free(intArray);
```

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

Tips when calling free()

Always deallocate all the memory blocks before exit

- Do not deallocate the same memory blocks twice
- Set freed ptr to NULL ptr

```
free(linkedList);
linkedList = NULL;
```

Adapted from slides by Ludwig Zhao

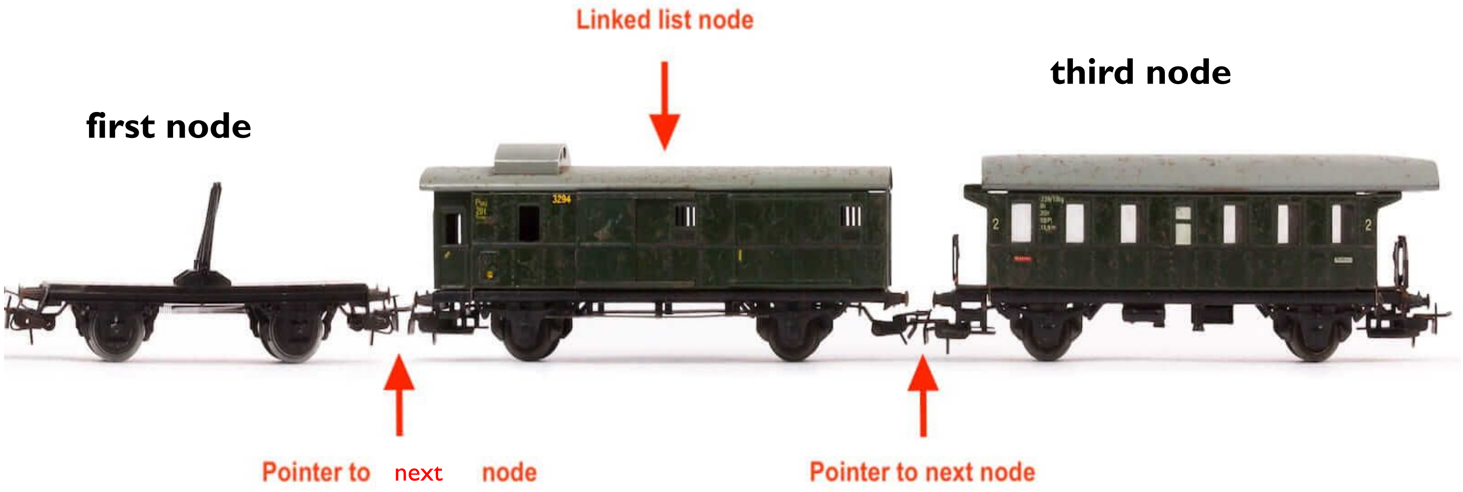
What is a linked list?

A data structure that uses dynamically allocated memory

What is a linked list?

A data structure that uses dynamically allocated memory

second node



third node

first node

What is a linked list?

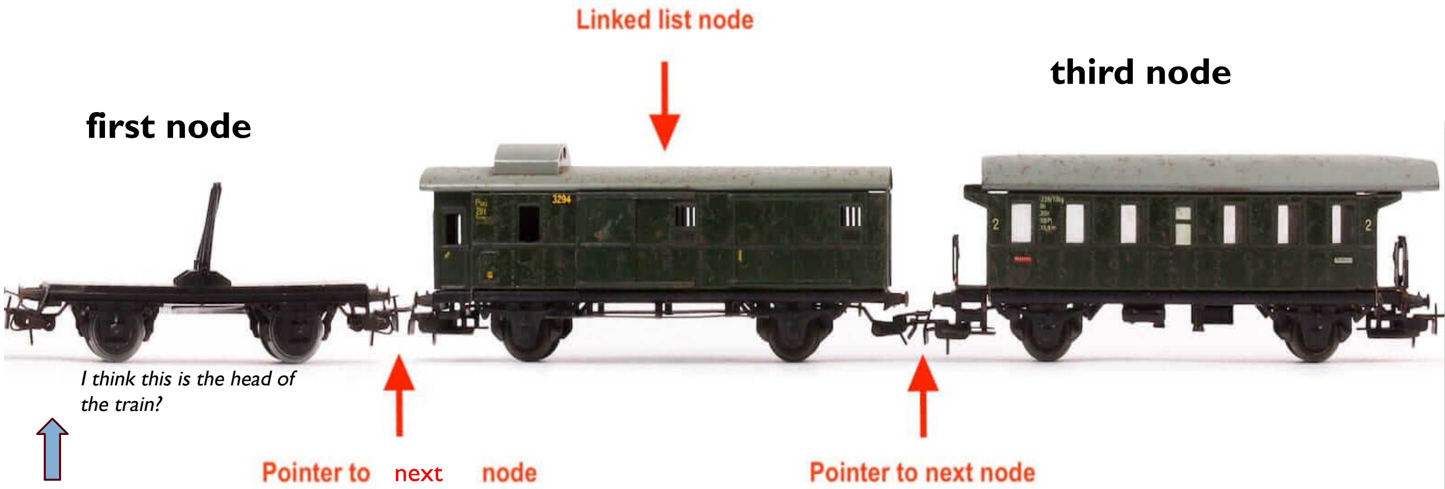
A data structure that uses dynamically allocated memory

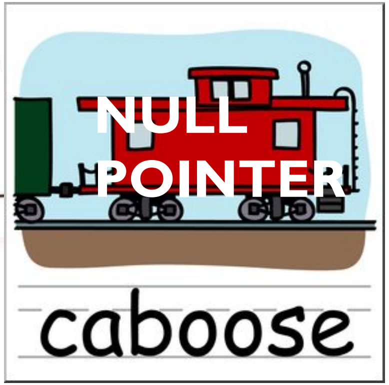
second node

third node

The caboose is the NULL pointer signalling the end of the train (linked list)

first node



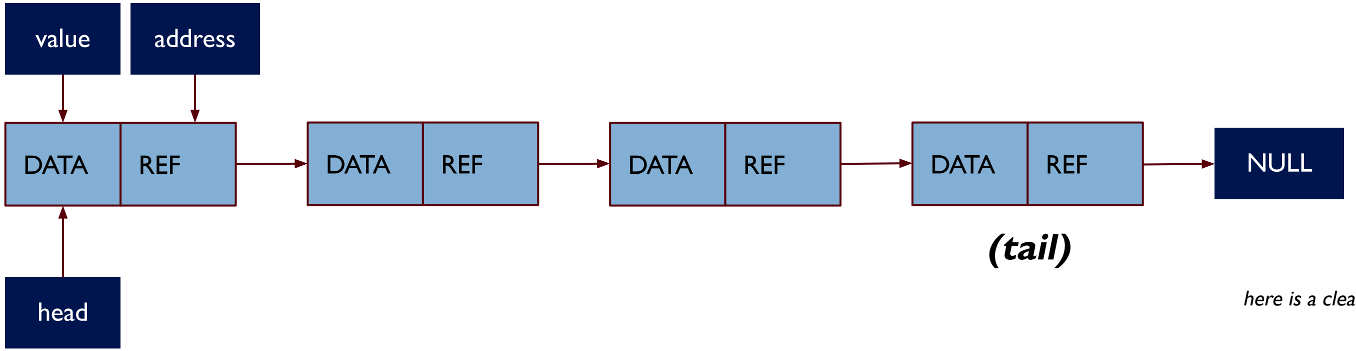


I think this is the head of the train?

↑

Pointer to head of linked list

What is a linked list?



value

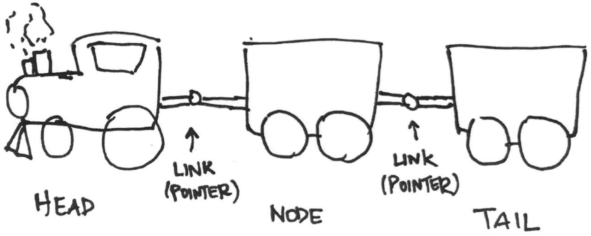
address

DATA REF → DATA REF → DATA REF → DATA REF → NULL

(tail)

head

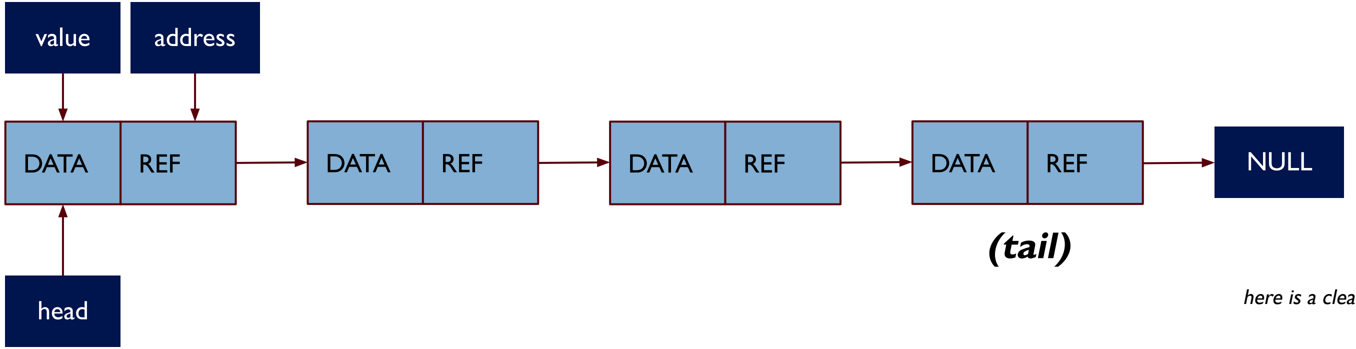
here is a cleaner train cartoon and diagram!



- Head pointer points to the first node (never lose your head pointer!)

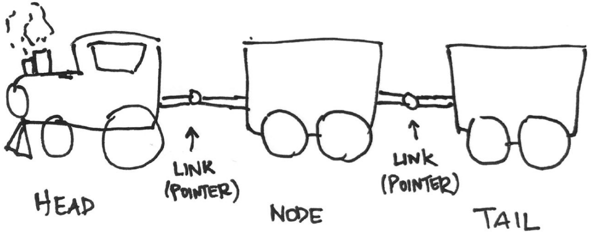
Adapted from slides by Ludwig Zhao

What is a linked list?



(tail)

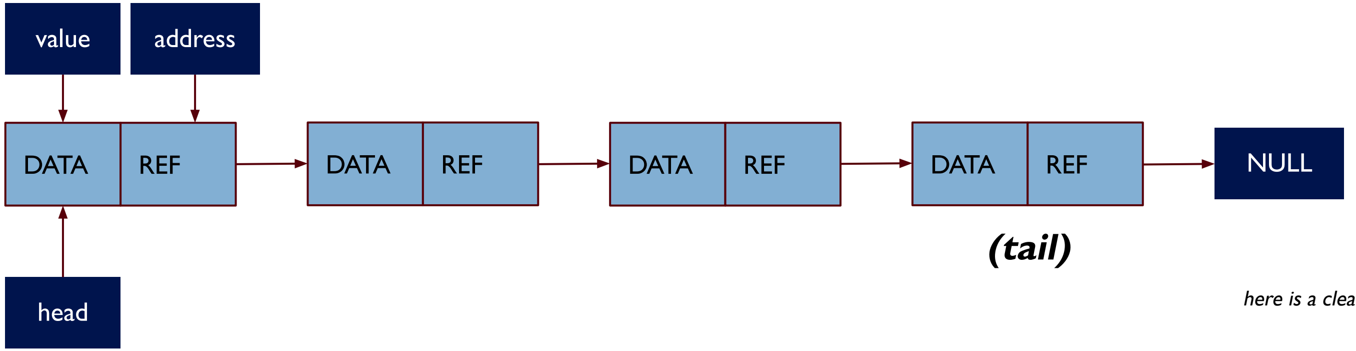
here is a cleaner train cartoon and diagram!



- Head pointer points to the first node (never lose your head pointer!)
- Each node has the data field(s) and a pointer points to the next node, next

Adapted from slides by Ludwig Zhao

What is a linked list?



value

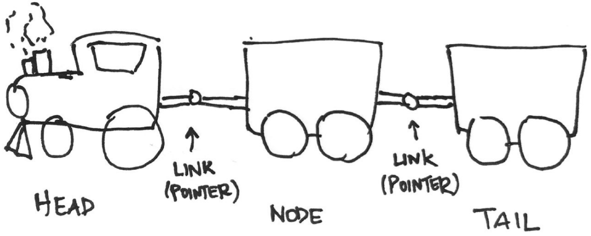
address

DATA REF → DATA REF → DATA REF → DATA REF → NULL

head

(tail)

here is a cleaner train cartoon and diagram!



- Head pointer points to the first node (never lose your head pointer!)
- Each node has the data field(s) and a pointer points to the next node, next
- The last node has the next pointer points to null ptr

Adapted from slides by Ludwig Zhao

Other notes on linked lists

- As program runs, memory is dynamically allocated

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

Other notes on linked lists

- As program runs, memory is dynamically allocated
- More efficient:
    - All memory allocated is used
    - Can grow/shrink data structure as needed

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

Other notes on linked lists

- As program runs, memory is dynamically allocated
- More efficient:
  - All memory allocated is used
  - Can grow/shrink data structure as needed
- Cons:
  - Annoying to traverse
  - Need to know how to work with pointers

"pointers put the link in linked lists"

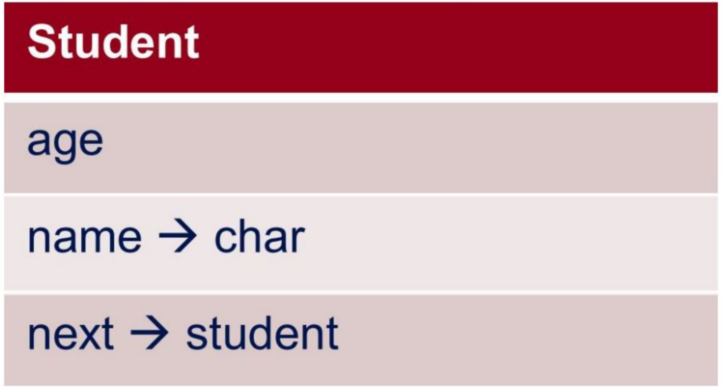
Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

addToList.c

Sample Code Discussion

We use a struct to create nodes

```
typedef struct studentStruct {
   int age;
   char *name;
   struct studentStruct *next;
} student;
```

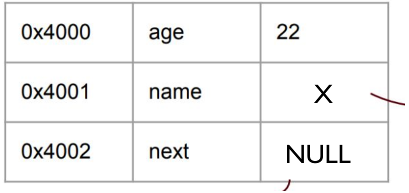


Student
age
name → char
next → student

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

Heap Allocation Example

- First student: Bob

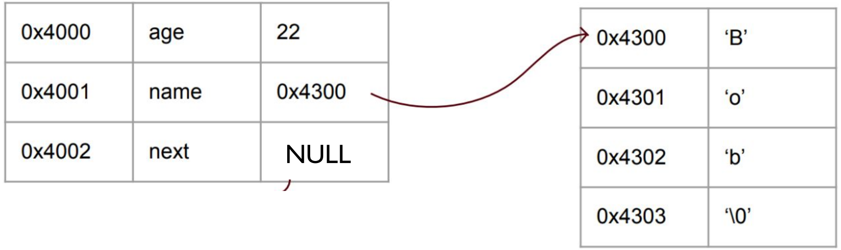


0x4000	age	22
0x4001	name	X
0x4002	next	NULL

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

Heap Allocation Example

- First student: Bob



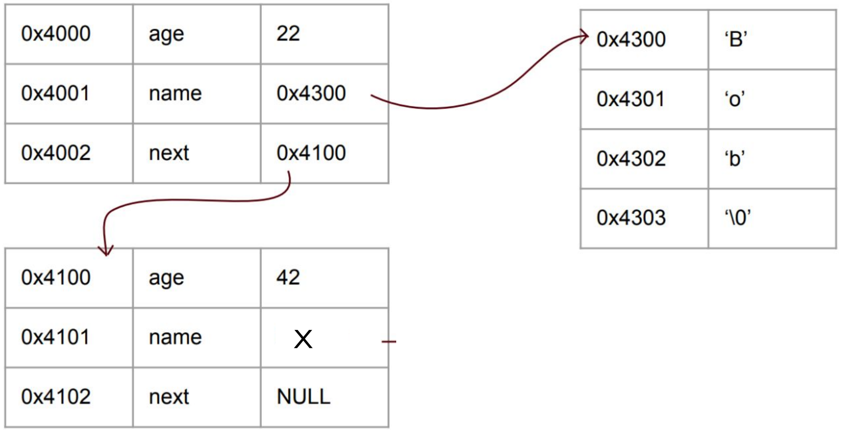
0x4000	age	22
0x4001	name	0x4300
0x4002	next	NULL

0x4300	‘B’
0x4301	‘o’
0x4302	‘b’
0x4303	‘\0’

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

Heap Allocation Example

- First student: Bob, Second student: Tables



0x4000	age	22
0x4001	name	0x4300
0x4002	next	0x4100

0x4300	‘B’
0x4301	‘o’
0x4302	‘b’
0x4303	‘\0’

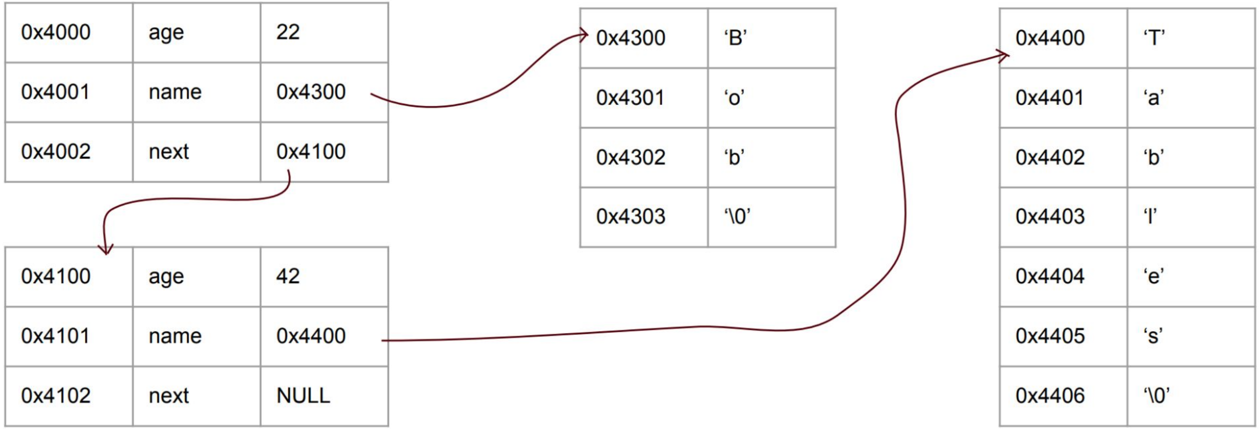
0x4100	age	42
0x4101	name	X
0x4102	next	NULL

Penn Engineering

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

Heap Allocation Example

- First student: Bob, Second student: Tables



0x4000	age	22
0x4001	name	0x4300
0x4002	next	0x4100

0x4100	age	42
0x4101	name	0x4400
0x4102	next	NULL

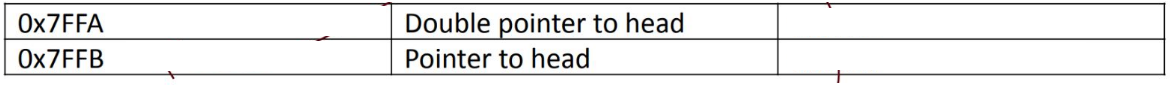
0x4300	‘B’
0x4301	‘o’
0x4302	‘b’
0x4303	‘\0’

0x4400	‘T’
0x4401	‘a’
0x4402	‘b’
0x4403	‘l’
0x4404	‘e’
0x4405	‘s’
0x4406	‘\0’

Adapted from slides by Makarios Chung, Alexandra Ulven, and Daniel Sullivan

Double Pointers

- Pointer to a Pointer



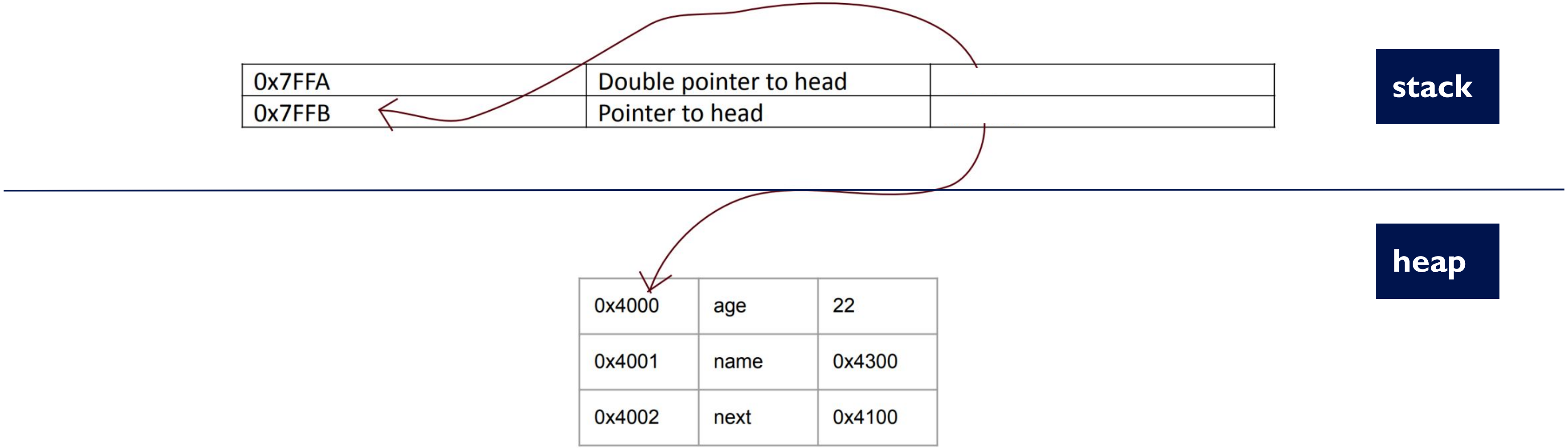
0x7FFA	Double pointer to head	
0x7FFB	Pointer to head	

stack

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

Double Pointers

- Pointer to a Pointer



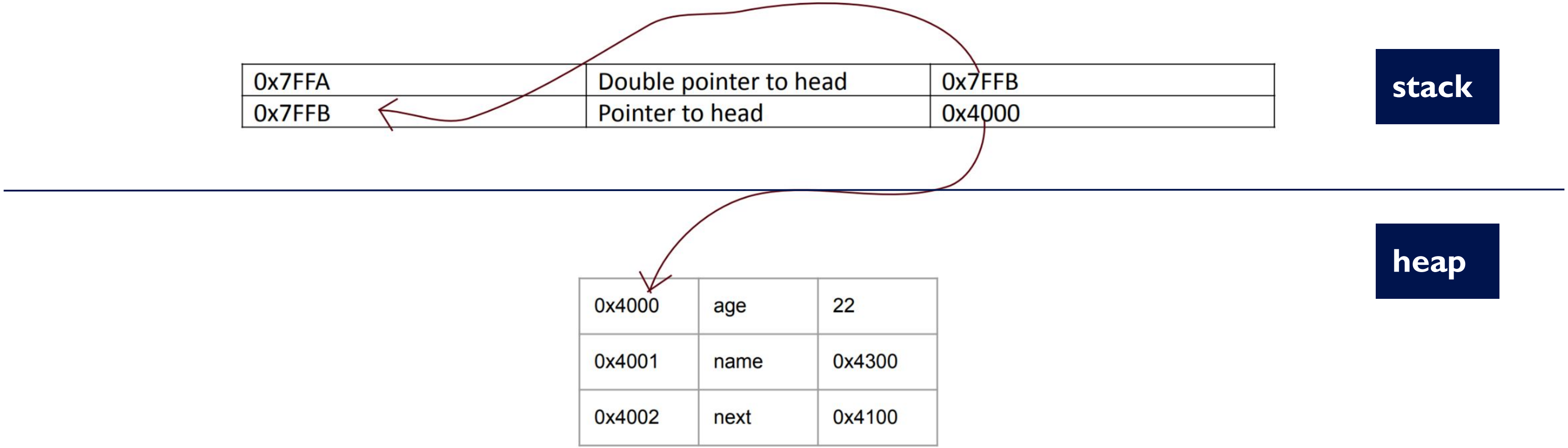
0x7FFA	Double pointer to head	
0x7FFB	Pointer to head	

0x4000	age	22
0x4001	name	0x4300
0x4002	next	0x4100

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

Double Pointers

- Pointer to a Pointer



0x7FFA	Double pointer to head	0x7FFB
0x7FFB	Pointer to head	0x4000

stack

heap

0x4000	age	22
0x4001	name	0x4300
0x4002	next	0x4100

Adapted from slides by Makarios Chung,
Alexandra Ulven, and Daniel Sullivan

Debugging practice