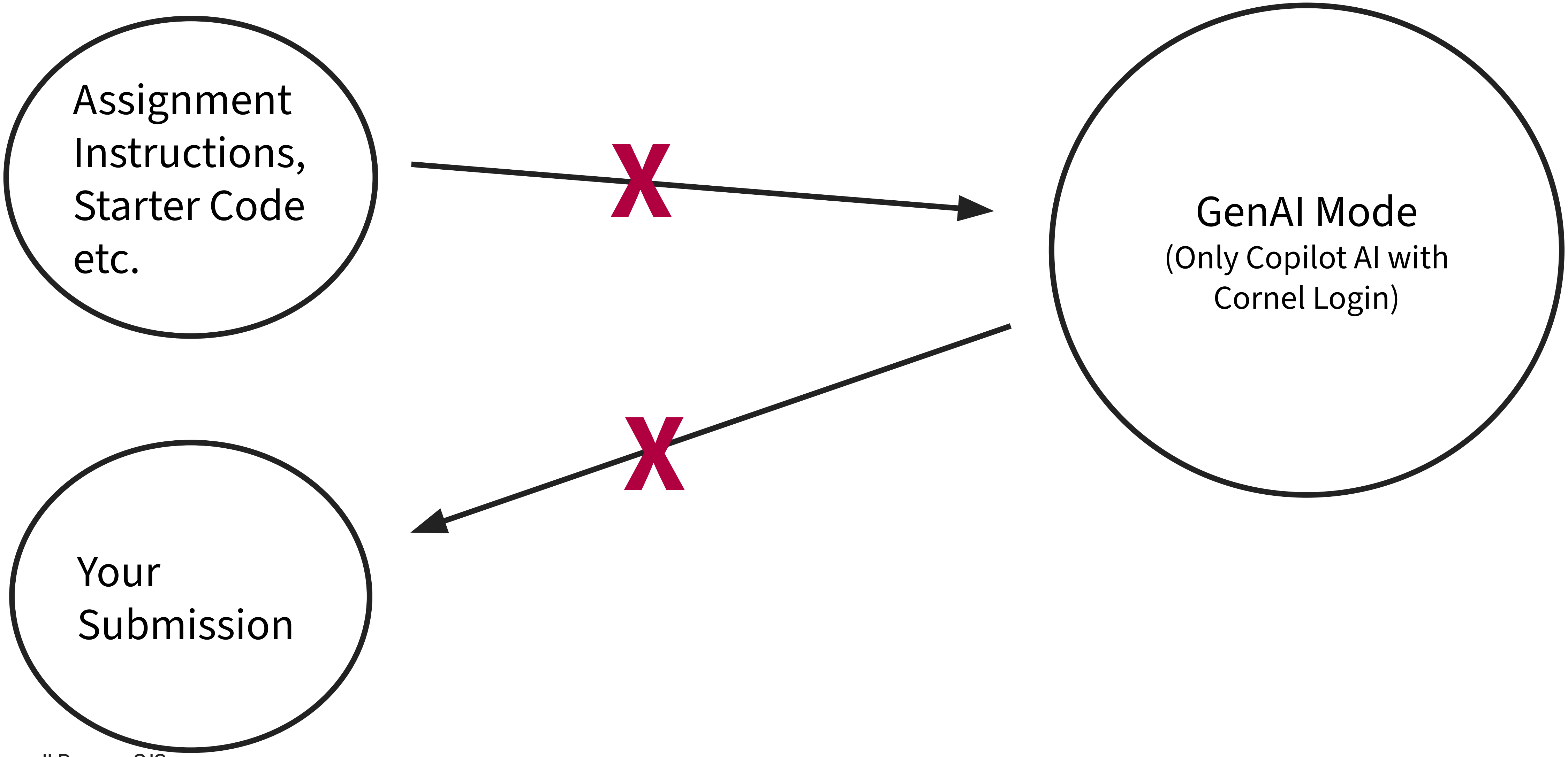
Review: GenAI Policy

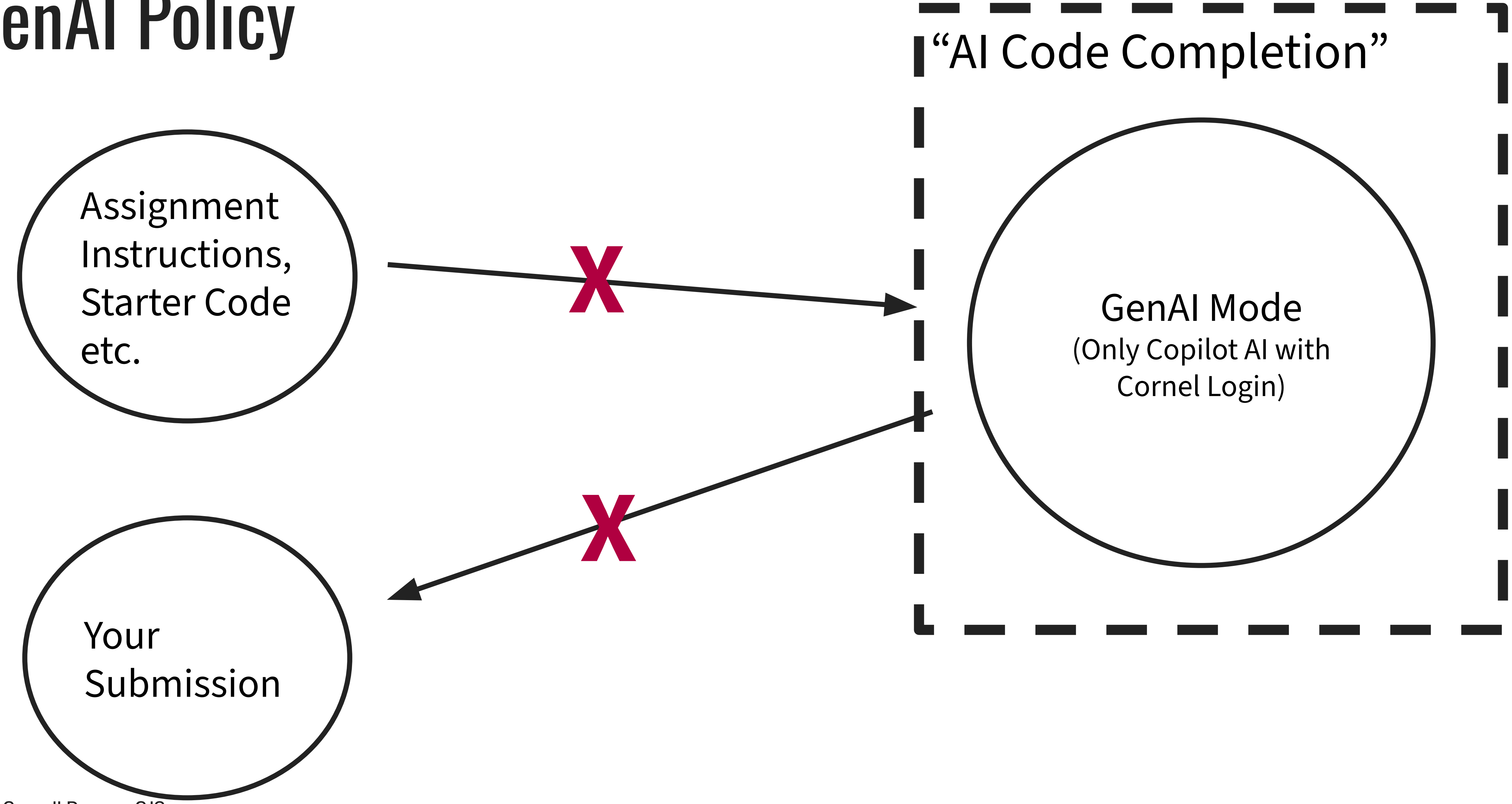
CS 3410: Computer System Organization and Programming

Fall 2025

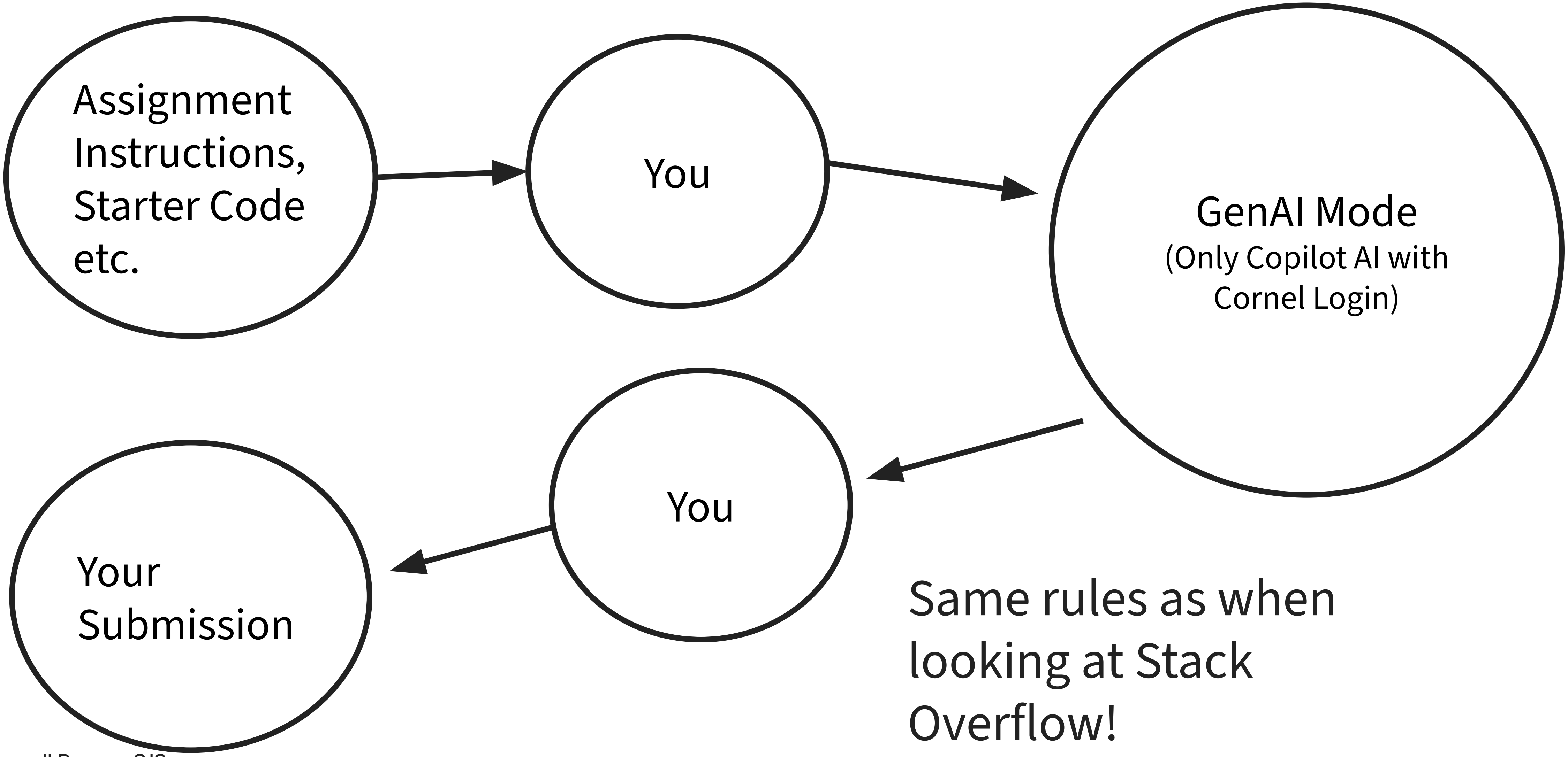
GenAI Policy



GenAI Policy

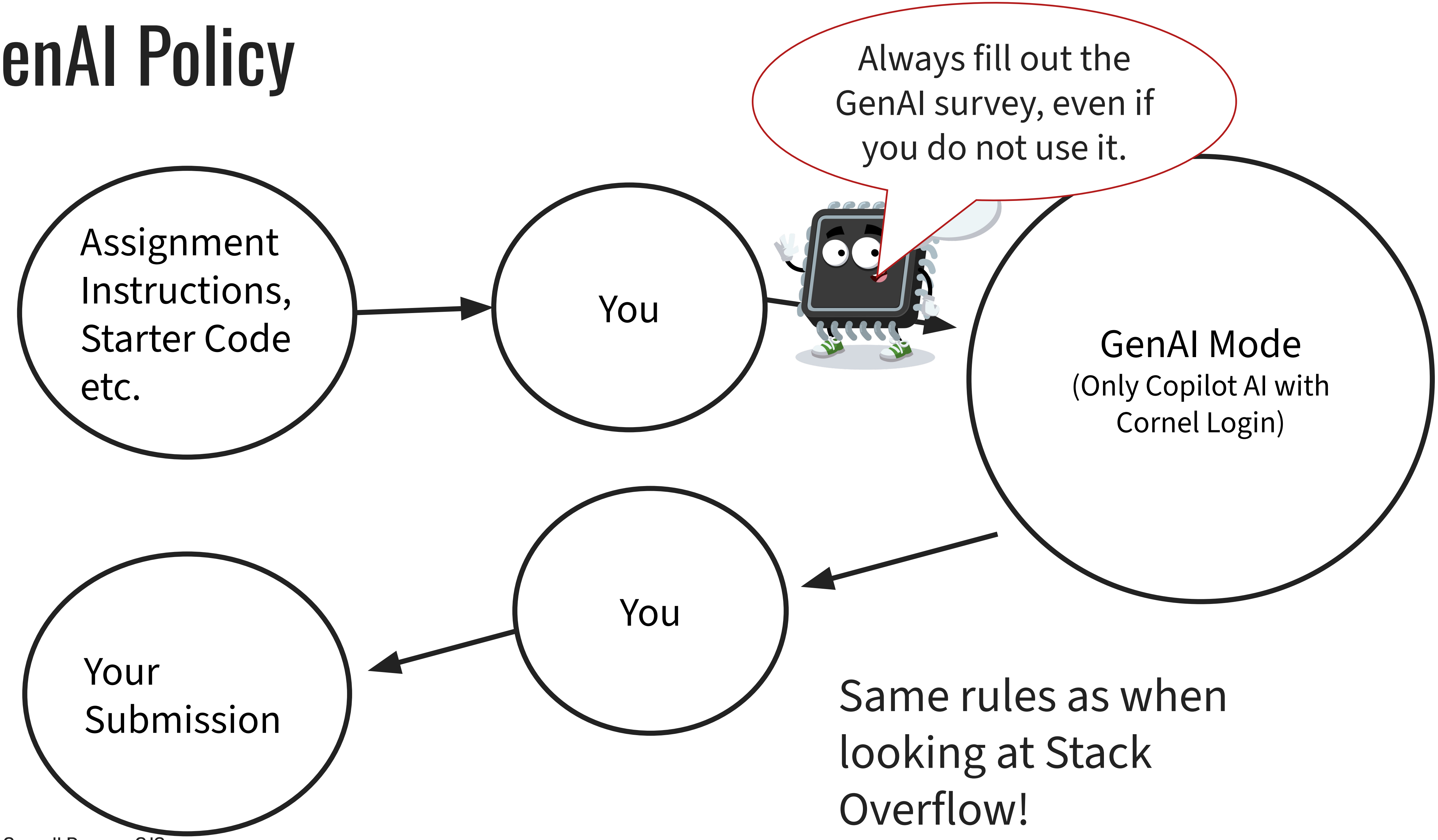


GenAI Policy



Same rules as when looking at Stack Overflow!

GenAI Policy



Same rules as when looking at Stack Overflow!

Memory Management in C

CS 3410: Computer System Organization and Programming

Fall 2025

[G. Guidi, A. Sampson, Z. Susag, H. Weatherspoon, and K. Laeufer]

Storage Duration and Lifetime

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Storage Duration and Lifetime

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Storage Duration and Lifetime

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Storage Duration and Lifetime

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

Storage Duration and Lifetime

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

Storage Duration and Lifetime

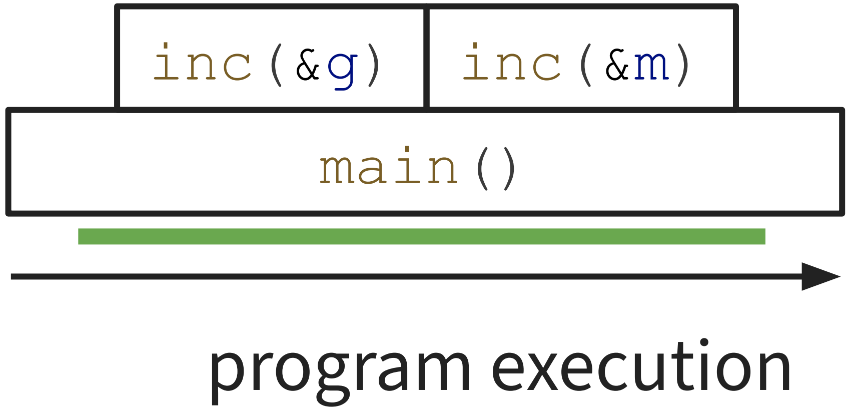
```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

Storage Duration and Lifetime

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

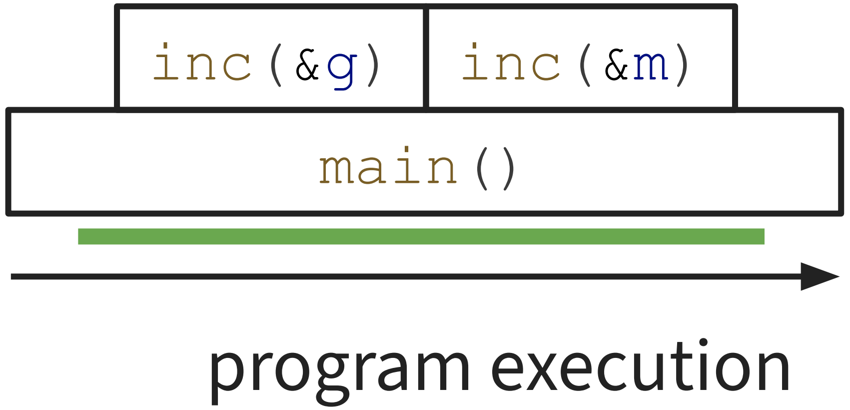
static



Storage Duration and Lifetime

```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static



Storage Duration and Lifetime

```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

automatic

Storage Duration and Lifetime

```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

automatic

Storage Duration and Lifetime

```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

automatic

automatic

Storage Duration and Lifetime

```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

automatic

automatic

Storage Duration and Lifetime

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

static

automatic

automatic

automatic

Storage Duration and Lifetime

```
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

Storage Duration and Lifetime

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

make sure type matches

Storage Duration and Lifetime

```
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

# of elements

make sure type matches

Cornell Bowers CIS
Computer Science

Storage Duration and Lifetime

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

# of elements

make sure type matches

do not forget to initialize

Storage Duration and Lifetime

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

# of elements

make sure type matches

do not forget to initialize

Storage Duration and Lifetime

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

???

PollEv.com/cs3410

Storage Duration and Lifetime

```
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

automatic

Storage Duration and Lifetime

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

automatic

Storage Duration and Lifetime

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic

automatic

automatic

Storage Duration and Lifetime

```
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

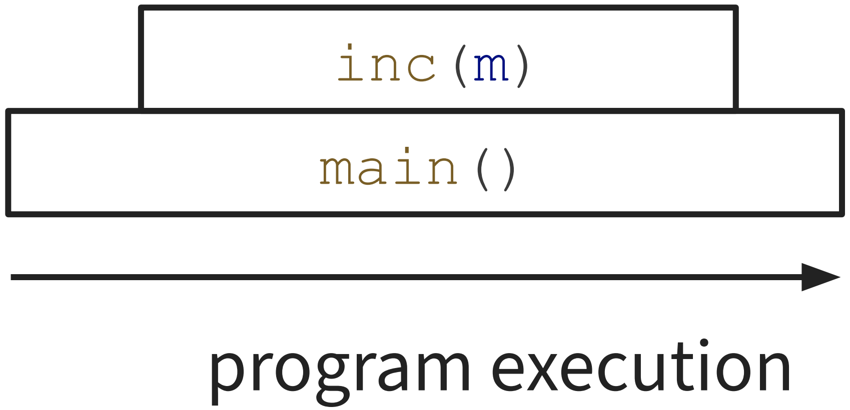
automatic

automatic

allocated

automatic

Storage Duration and Lifetime




```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

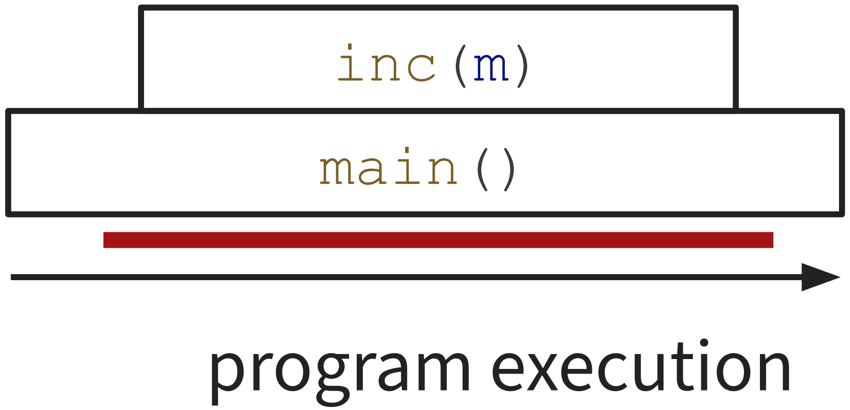
automatic

automatic

allocated

automatic

Storage Duration and Lifetime




```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

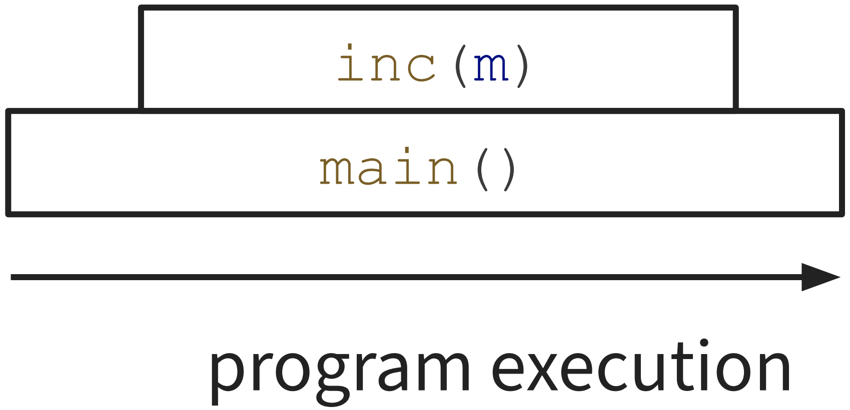
automatic

automatic

allocated

automatic

Storage Duration and Lifetime




```
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    free(m);
    inc(m);
    return 0;
}
```

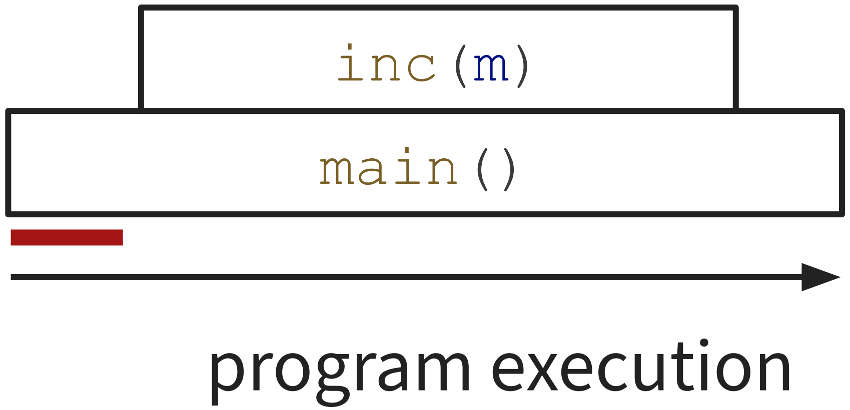
automatic

automatic

allocated

automatic

Storage Duration and Lifetime




```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    free(m);
    inc(m);
    return 0;
}
```

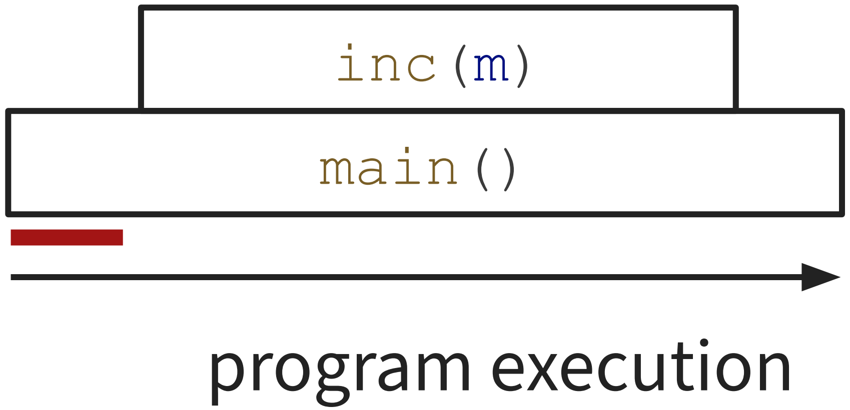
automatic

automatic

allocated

automatic

Storage Duration and Lifetime



allocated memory is not live!

```
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    free(m);
    inc(m);
    return 0;
}
```

automatic

automatic

allocated

automatic

Storage Duration and Lifetime

inc(m)

main()

ram execution

==17492==ERROR: AddressSanitizer: **heap-use-after-free** on address 0x7bdfa7a20010 at pc 0x00000040054d bp 0x7fff70b263e0 sp 0x7fff70b263d8

```
*m = 0;
```

allocated

```
}
```

Use after Free

Also called “dangling reference”

- Using a pointer after its been freed can cause memory corruption
    - Best case: your program immediately crashes

Use after Free

Also called “dangling reference”

- Using a pointer after its been freed can cause memory corruption
  - Best case: your program immediately crashes
  - Worse case: You end up on the NIST national security vulnerability database

CVE-2022-48771 Detail

RECEIVED

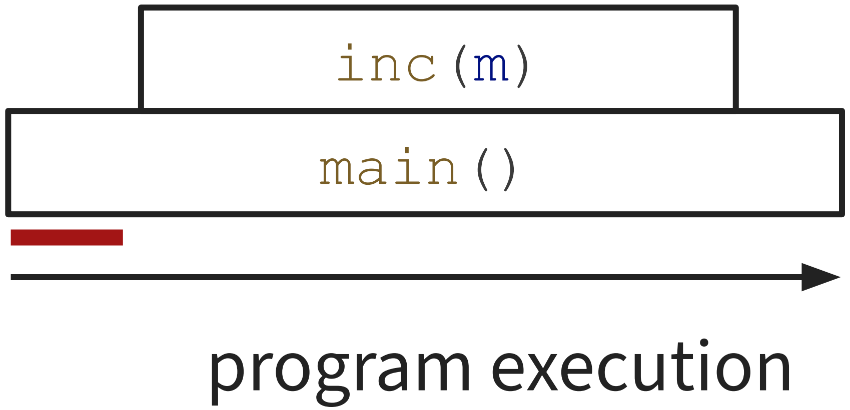
This vulnerability has been received by the NVD and has not been analyzed.

Description

In the Linux kernel, the following vulnerability has been resolved: drm/vmwgfx: Fix stale file descriptors on failed usercopy A failing usercopy of the fence_rep object will lead to a stale entry in the file descriptor table as put_unused_fd() won't release it. This enables userland to refer to a dangling 'file' object through that still valid file descriptor, leading to all kinds of use-after-free exploitation scenarios. Fix this by deferring the call to fd_install() until after the usercopy has succeeded.

src: UC Berkeley, CS61C slides

Storage Duration and Lifetime



allocated memory is not live!

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    free(m);
    inc(m);
    return 0;
}
```

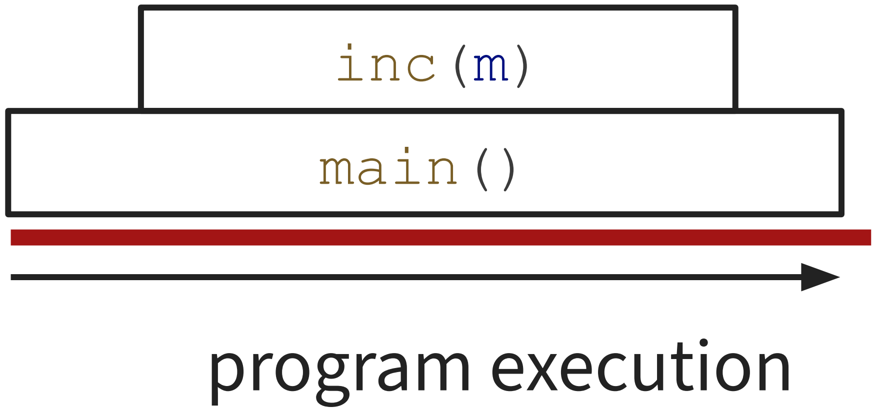
automatic

automatic

allocated

automatic

Storage Duration and Lifetime




```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    free(m);
    inc(m);
    return 0;
}
```

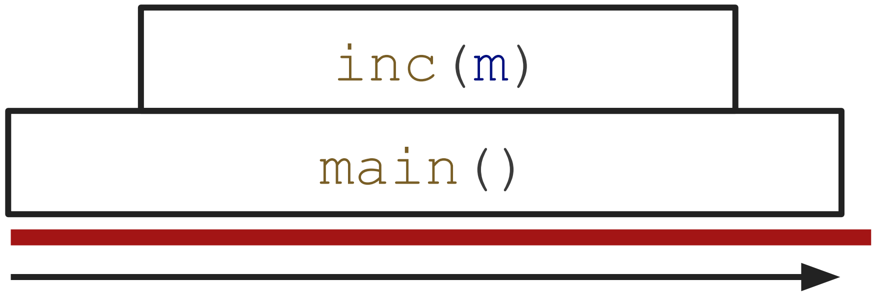
automatic

automatic

allocated

automatic

Storage Duration and Lifetime



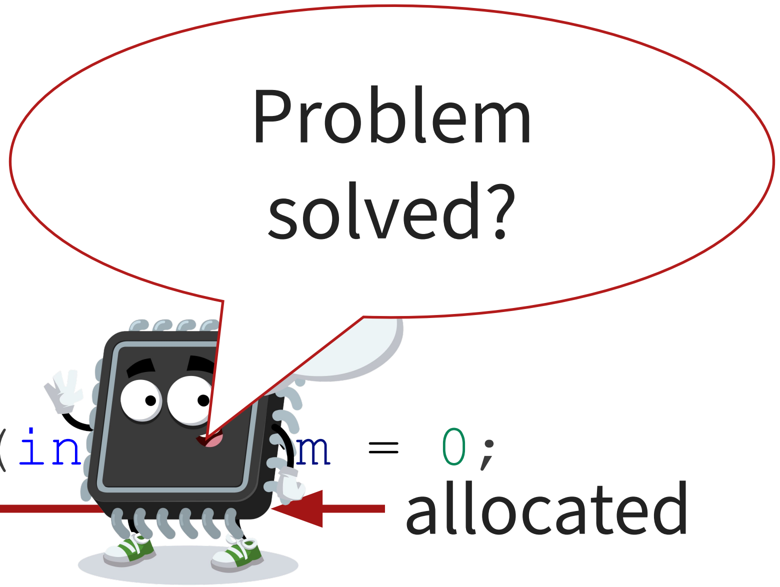

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(in        m = 0;
    free(m);
    inc(m);
    return 0;
}
```

automatic

automatic

allocated

automatic



Storage Duration and Lifetime

`inc(m)`

`main()`

ram execution

Problem solved?

==20833==ERROR: LeakSanitizer: detected memory leaks

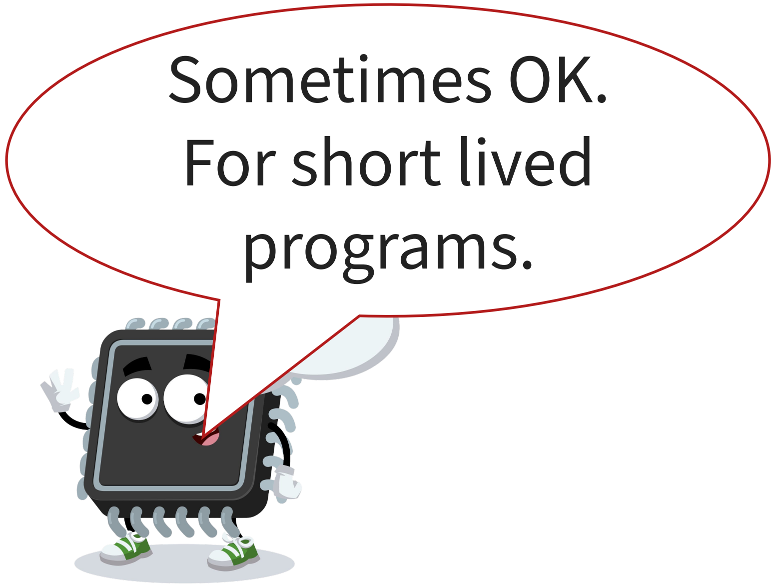
`m = 0;`

allocated

`}`

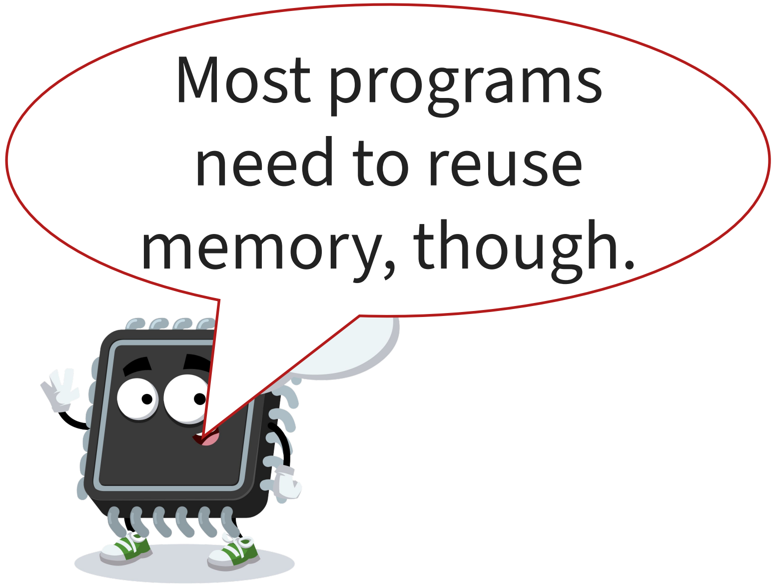
Memory Leak

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    free(m);
    inc(m);
    return 0;
}
```



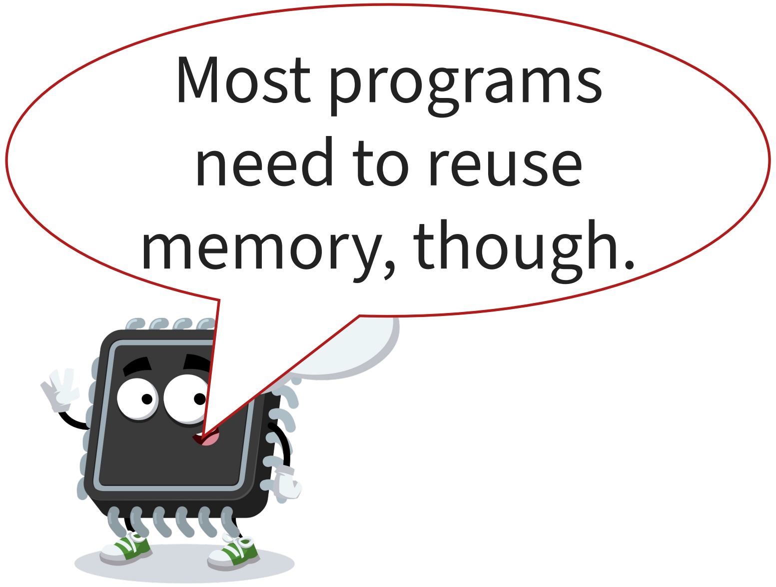
Memory Leak

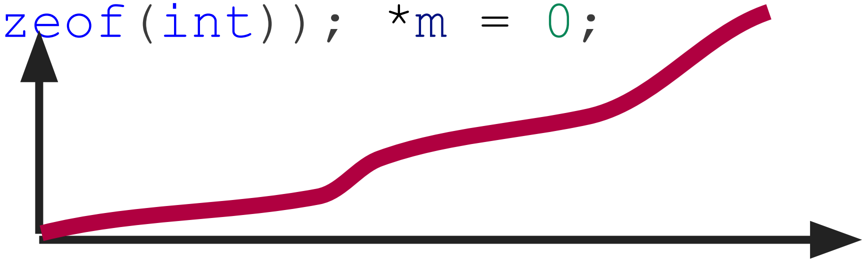
```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m;
    for(int i = 0; i < 100; i +=1) {
        m = (int*)malloc(1 * sizeof(int)); *m = 0;
        inc(m);
    }
    return 0;
}
```



Memory Leak

```c
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m;
    for(int i = 0; i < 100; i +=1) {
        m = (int*)malloc(1 * sizeof(int)); *m = 0;
        inc(m);
    }
    return 0;
```





Memory Leak

Most programs
d to reuse
ry, though.

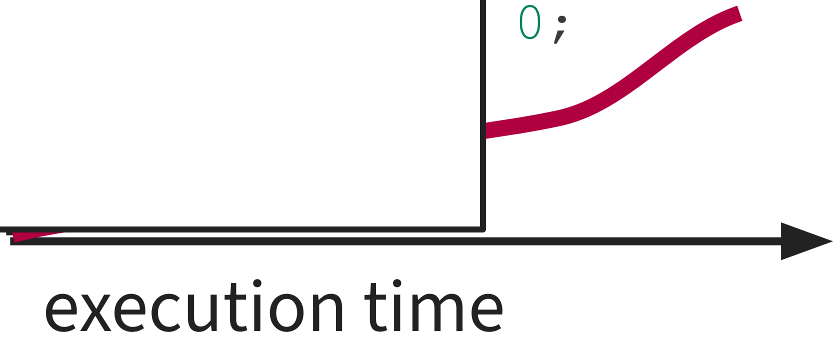
==22075==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 400 byte(s) in 100 object(s) allocated from:

```
    #0 0x7fd7d0ae6f2b in malloc
    #1 0x0000004005cb in main leak.c:9
```

SUMMARY: AddressSanitizer: 400 byte(s) leaked in 100 allocation(s).

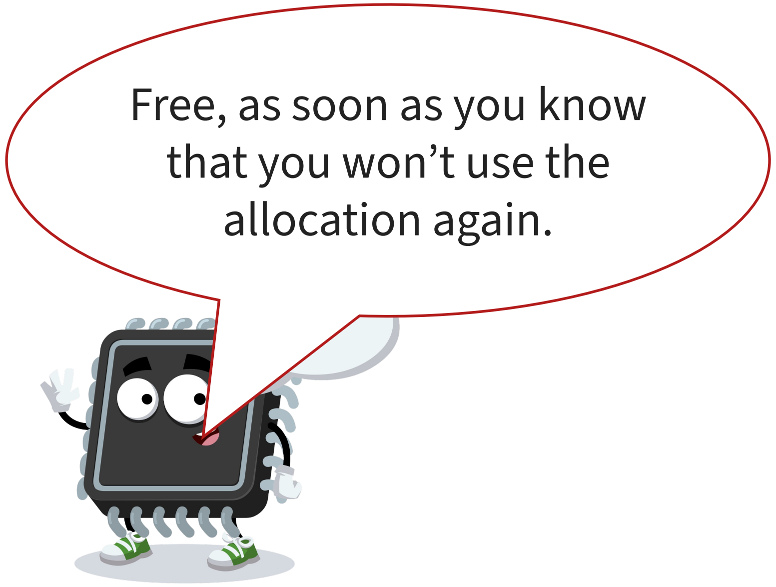
```
0;
```

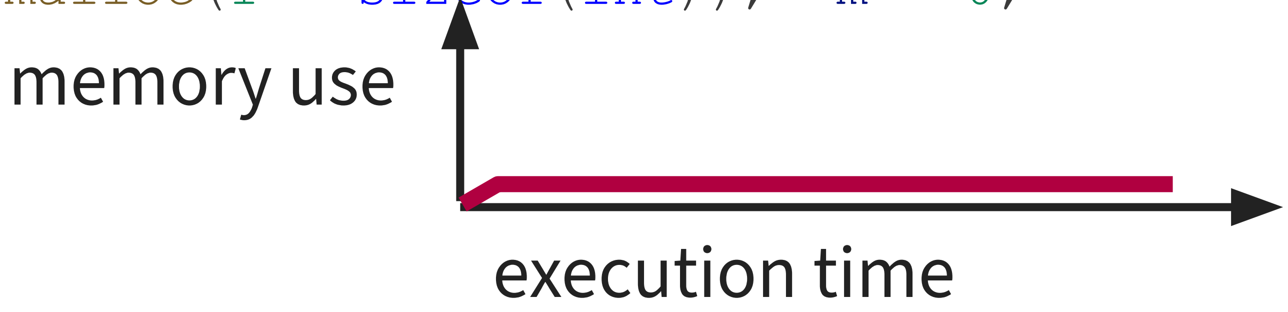



```
return 0;
```

Memory Leak

```
#include<stdlib.h>
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int* m;
    for(int i = 0; i < 100; i +=1) {
        m = (int*)malloc(1 * sizeof(int)); *m = 0;
        inc(m);
        free(m);
    }
    return 0;
```





Double Free

```c
#include<stdlib.h>
void inc(int* a) {
    // ...
}
int main() {
    int* m;
    for(int i = 0; i < 100; i +=1) {
        m = (int*)malloc(1 * sizeof(int)); *m = 0;
        inc(m);
        free(m);
    }
    free(m);
    return 0;
}
```

Double Free

Free as much as
ssible?!

==22429==ERROR: AddressSanitizer: attempting double-free on 0x7b3528a20c70 in thread T0:

    double_free.c:14

0x7b3528a20c70 is located 0 bytes inside of 4-byte region [0x7b3528a20c70,0x7b3528a20c74)

freed by thread T0 here:

    double_free.c:12

previously allocated by thread T0 here:

    double_free.c:10

0;

```
return 0;
```

execution time

Double Free


```
#include<stdlib.h>
void inc(int* a) {
    // ...
}
int main() {
    int* m;
    for(int i = 0; i < 100; i +=1) {
        m = (int*)malloc(1 * sizeof(int)); *m = 0;
        inc(m);
        free(m);
    }
    free(m);
    return 0;
}
```

previously allocated by thread T0 here: → `m = (int*)malloc(1 * sizeof(int)); *m = 0;`

freed by thread T0 here → `free(m);`

attempting double-free → `free(m);`

Double Free

Only free something once!

- Best case: you corrupt your heap and you crash
  - Because you didn’t free something returned by malloc

Double Free

Only free something once!

- Best case: you corrupt your heap and you crash
  - Because you didn't free something returned by malloc
- Worst case: you free something able to be freed on accident
  - … and end up on the NIST national vulnerability database.

CVE-2023-39975 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Description

kdc/do_tgs_req.c in MIT Kerberos 5 (aka krb5) 1.21 before 1.21.2 has a double free that is reachable if an authenticated user can trigger an authorization-data handling failure. Incorrect data is copied from one ticket to another.

src: UC Berkeley, CS61C slides

Common errors with manually allocation

1. **Use after free:** After you `free` memory, you can't use it.
2. **Double free:** You can only `free` memory once.
3. **Memory leak:** You must `free` all the memory you allocated with `malloc`

Storage Duration and Lifetime

- **static**:
- **automatic**:
- **allocated**:

Storage Duration and Lifetime

- **static**:
  - global variables, functions
  - live while the main function is executing
- **automatic**:
- **allocated**:

Storage Duration and Lifetime

- **static**:
  - global variables, functions
  - live while the main function is executing
- **automatic**:
  - function arguments, local variables
  - live from declaration until function exits
- **allocated**:

PollEv.com/cs3410

Storage Duration and Lifetime

- **static**:
  - global variables, functions
  - live while the main function is executing
- **automatic**:
  - function arguments, local variables
  - live from declaration until function exits
  - can have multiple copies live at the same time (recursive functions!)
- **allocated**:

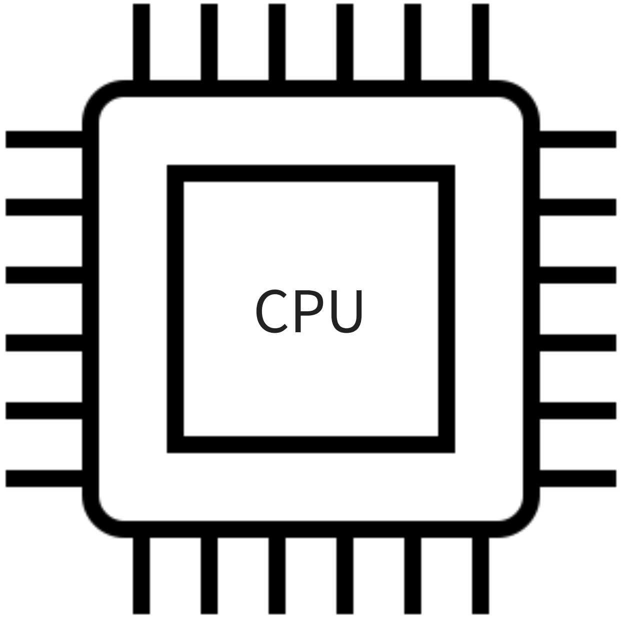
Storage Duration and Lifetime

- **static**:
  - global variables, functions
  - live while the main function is executing
- **automatic**:
  - function arguments, local variables
  - live from declaration until function exits
  - can have multiple copies live at the same time (recursive functions!)
- **allocated**:
  - manual allocation with `malloc`
  - live until `free` is called on pointer

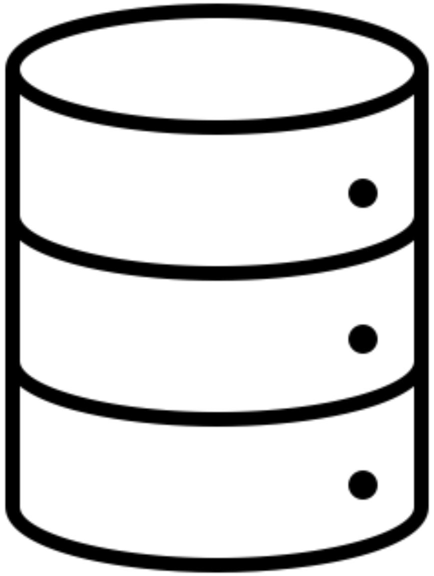
Storage Implementation

A Mental Model of Memory

Processor



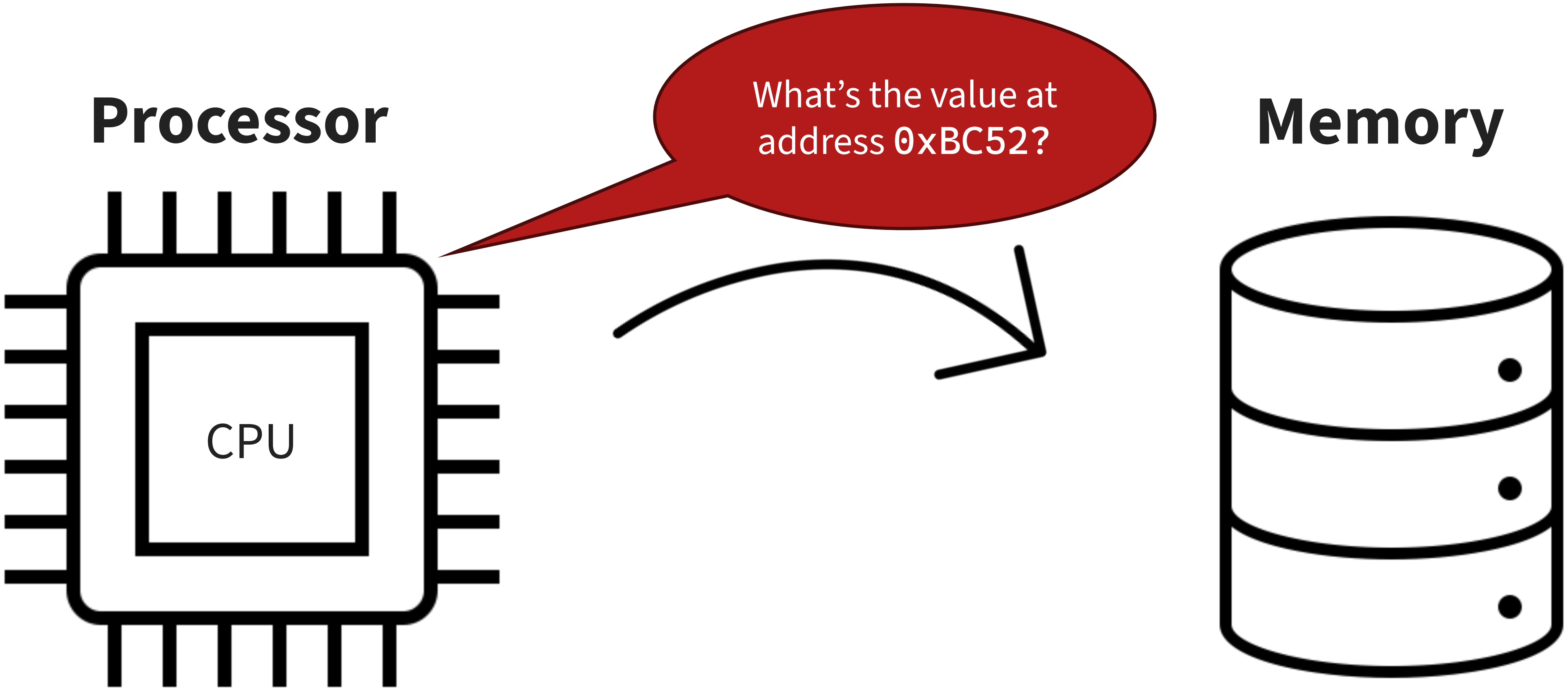
Memory



```c
uint8_t mem[SIZE];
```

$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$

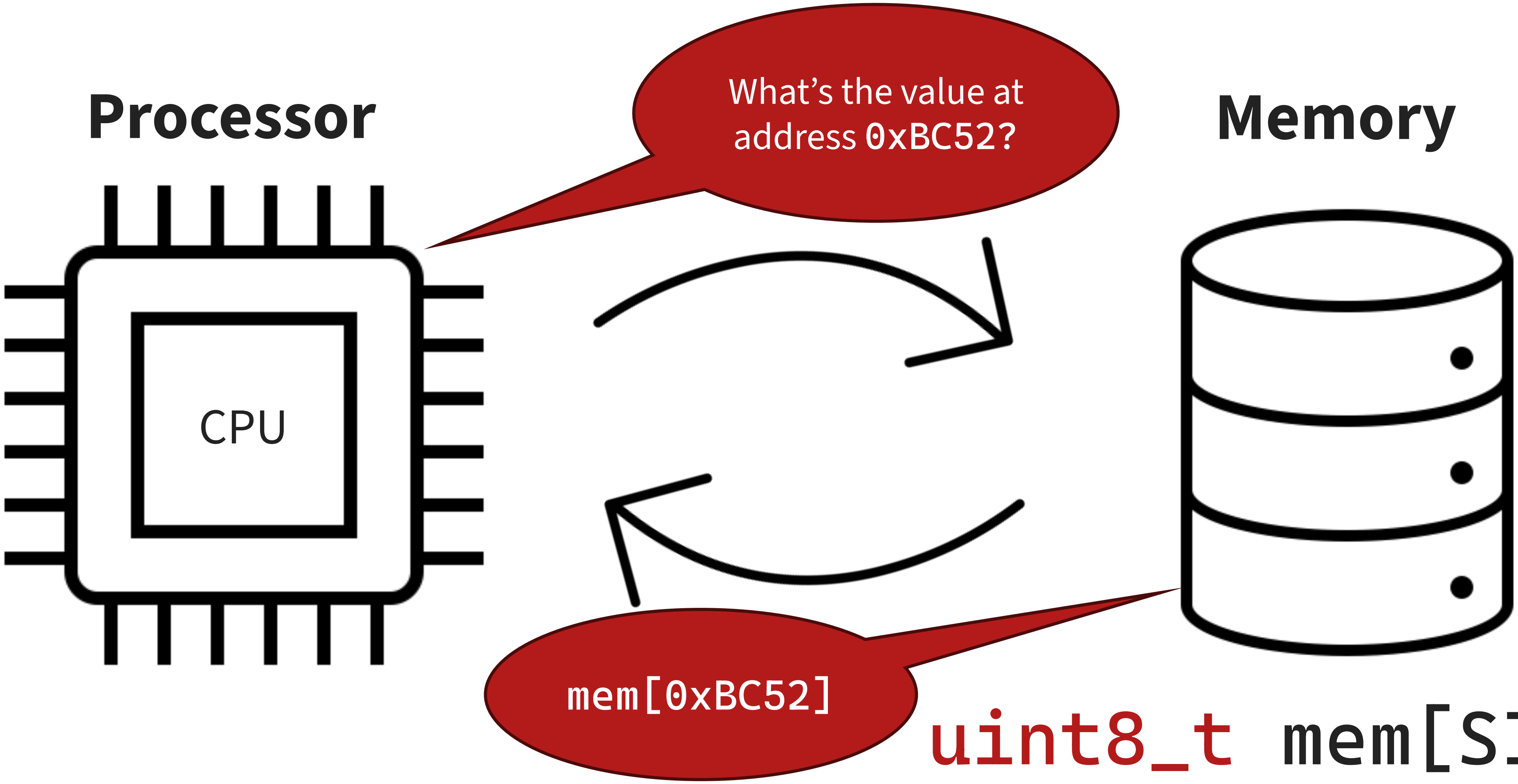
A Mental Model of Memory




```
uint8_t mem[SIZE];
```

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$$

A Mental Model of Memory

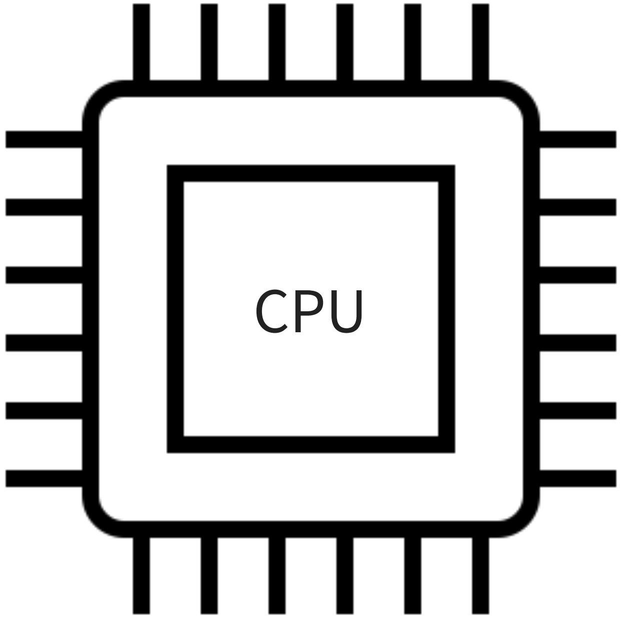



```
uint8_t mem[SIZE];
```

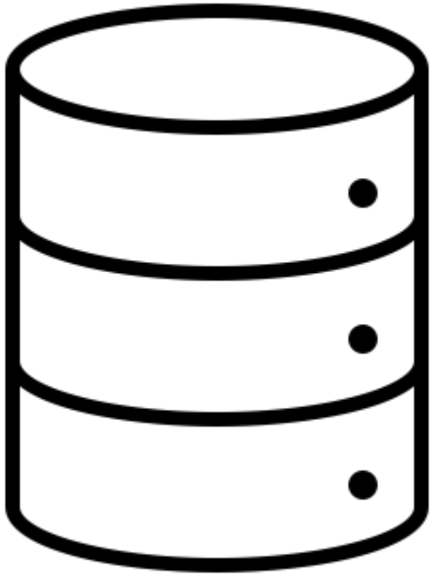
$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$$

A Mental Model of Memory

Processor



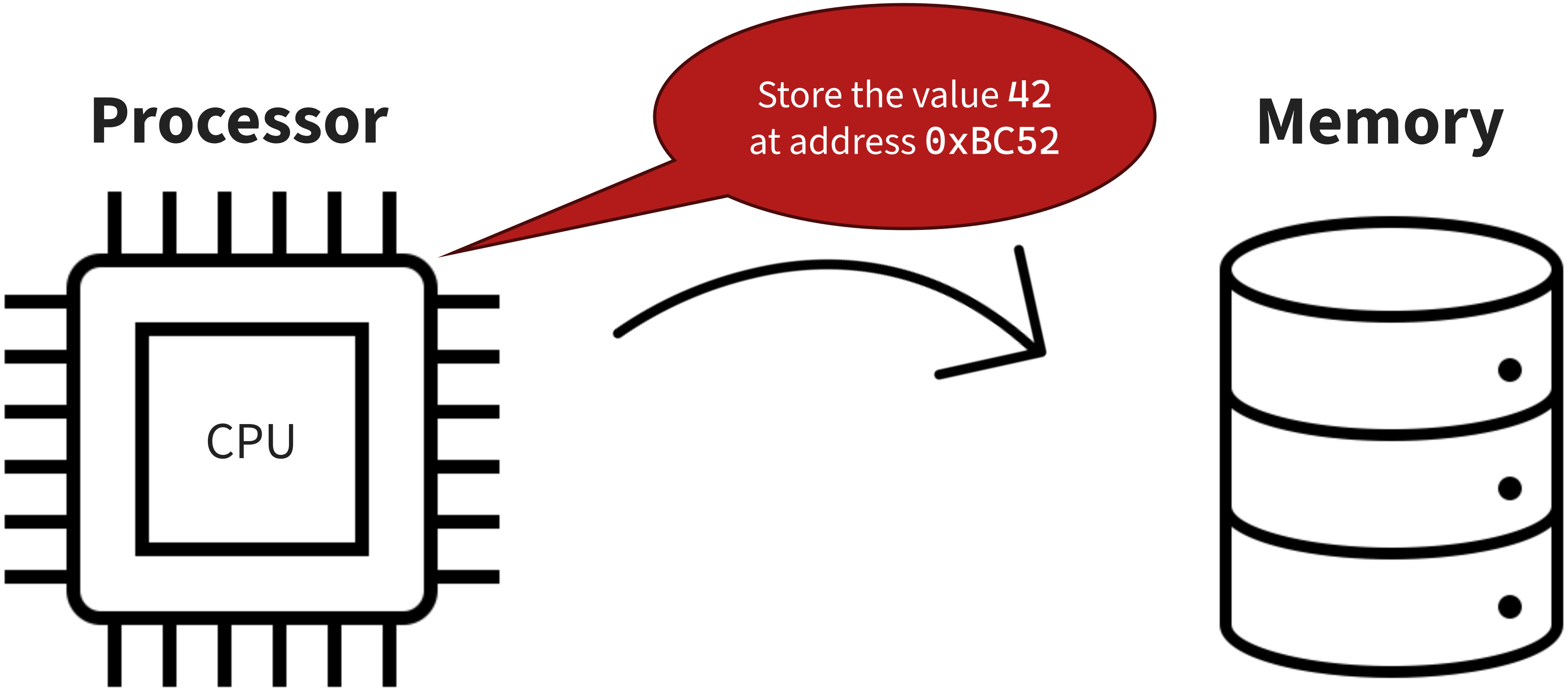
Memory



```
uint8_t mem[SIZE];
```

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$$

A Mental Model of Memory

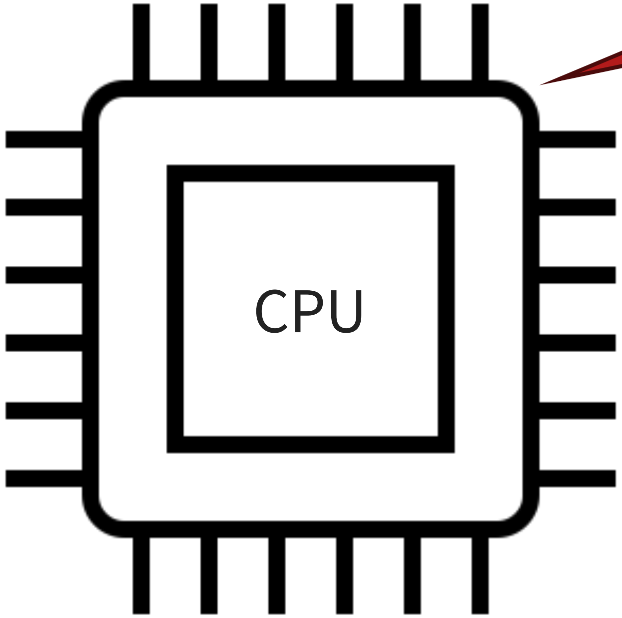



```
uint8_t mem[SIZE];
```

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$$

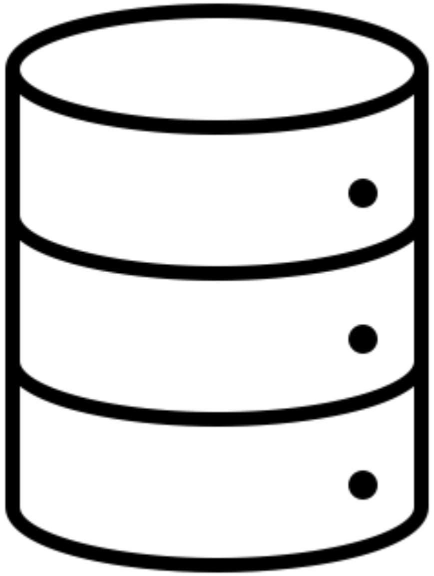
A Mental Model of Memory

Processor



Store the value 42
at address `0xBC52`

Memory



`mem[0xBC52] = 42;`

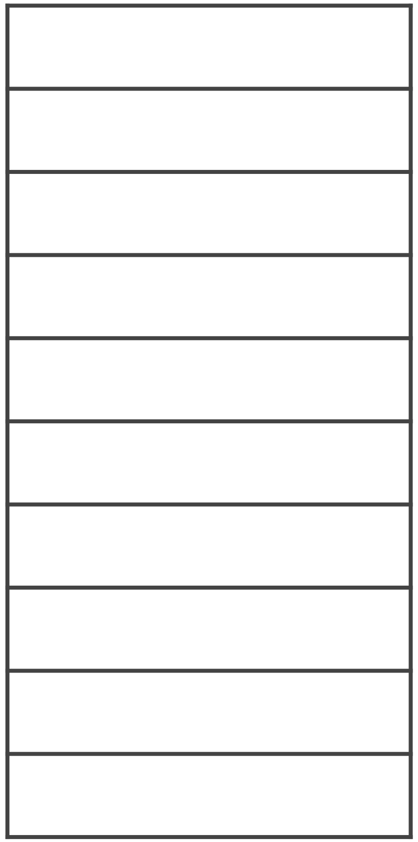
`8_t mem[SIZE];`

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34} = 17{,}179{,}869{,}184\text{B}$$

Implementation: static storage

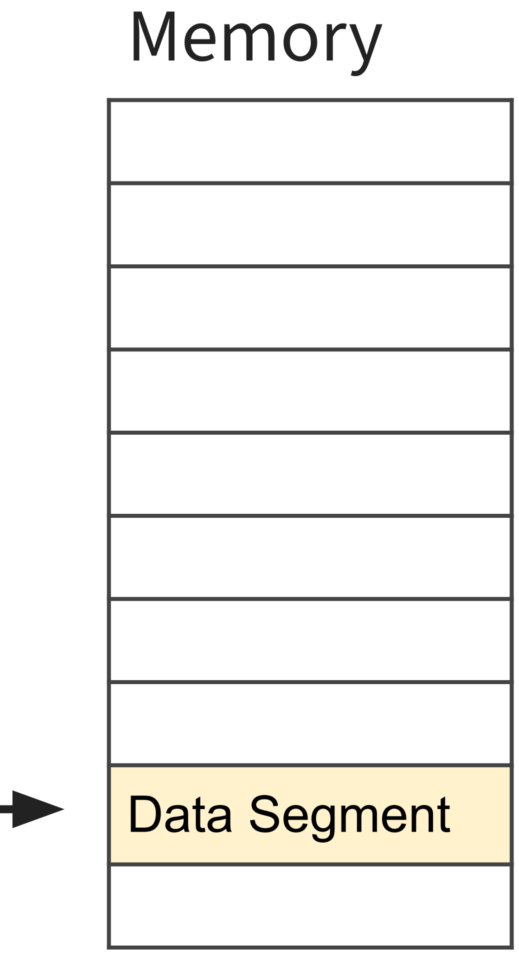
```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Memory



Implementation: static storage

```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```



Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Data Segment:

Address	Value
&g	2

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	2

Cornell Bowers CIS
Computer Science

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&g
&b	???

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	2

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&g
&b	2

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	2

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&g
&b	3

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	2

Cornell Bowers CIS
Computer Science

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&g
&b	3

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&m
&b	???

main frame

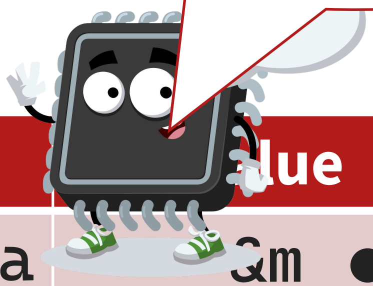
Address	Value
&m	0

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```



inc frame

Address	Value
&a	&m
&b	???

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&m
&b	0

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&m
&b	1

main frame

Address	Value
&m	0

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

inc frame

Address	Value
&a	&m
&b	1

main frame

Address	Value
&m	1

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

main frame

Address	Value
&m	1

Data Segment:

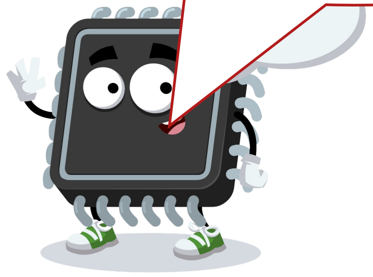
Address	Value
&g	3

Cornell Bowers CIS
Computer Science

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

What do we call this data structure?



PollEv.com/cs3410

main frame

Address	Value
&m	1

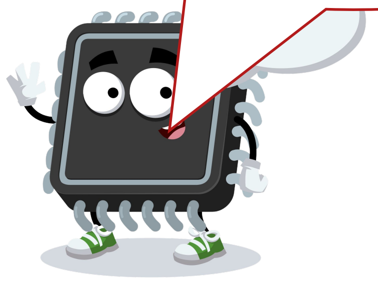
Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

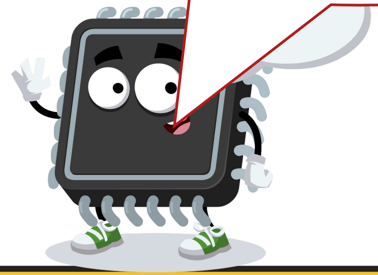
What do we call this data structure?



Data Segment:

Address	Value
&g	3

Implementation: automatic storage

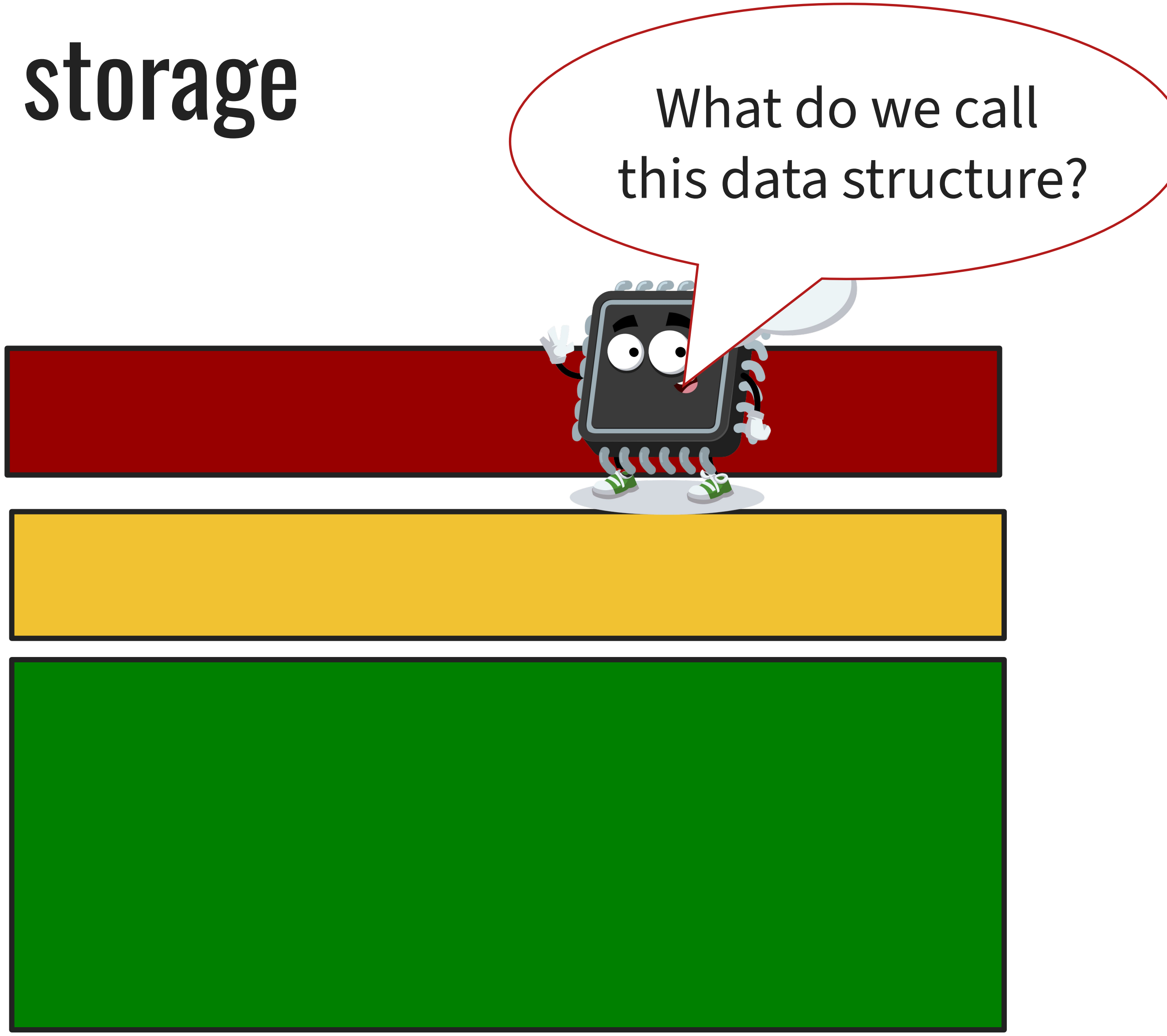



```c
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Data Segment:

Address	Value
&g	3

Implementation: automatic storage



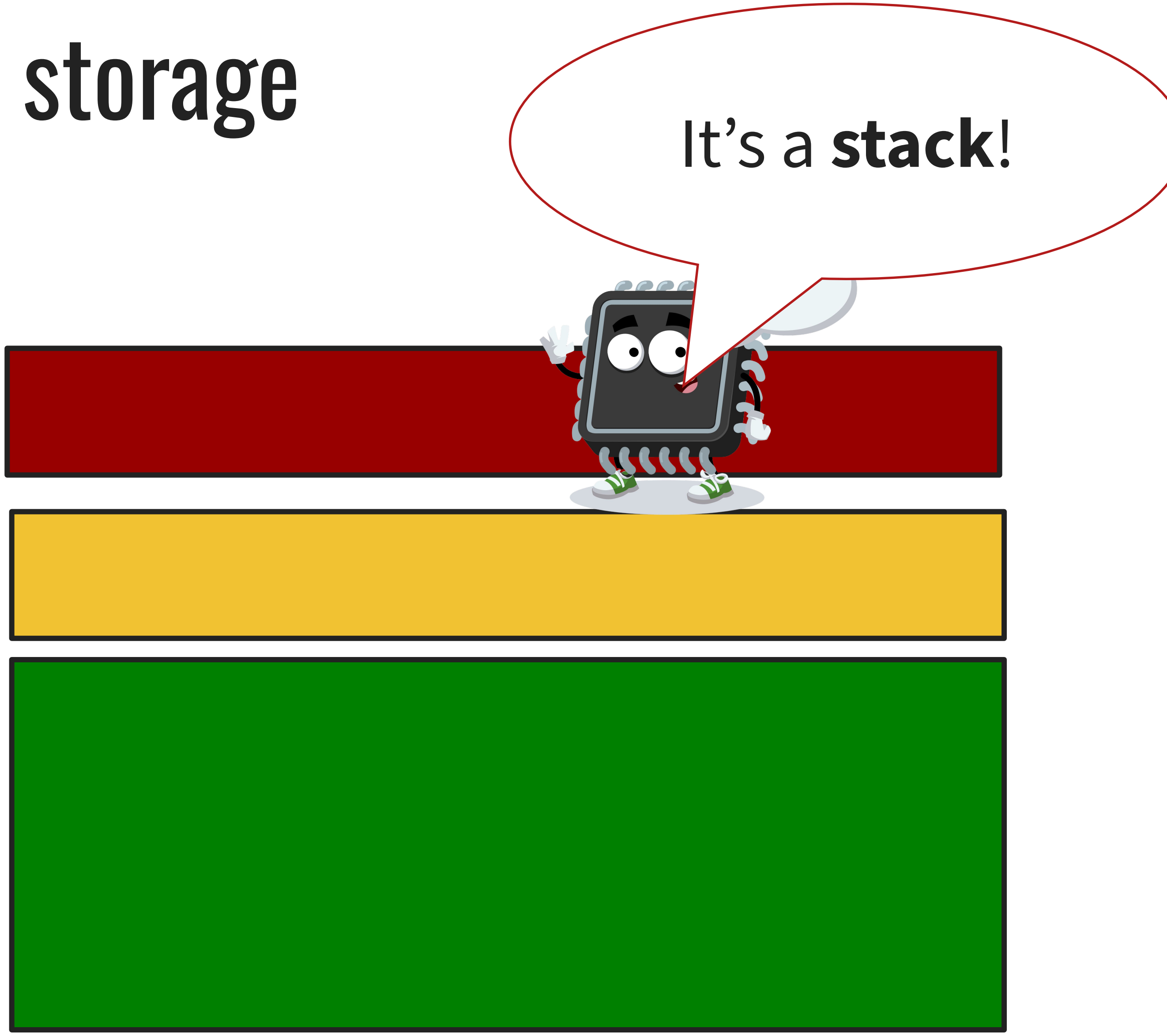

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```



Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

main frame

Address	Value
&m	1

Data Segment:

Address	Value
&g	3

Implementation: automatic storage

```
int g = 2;
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```

Data Segment:

Address	Value
&g	3

Caution: automatic storage

```
int* inc(int a) {
    a += 1;
    return &a;
}
int main() {
    int* res = inc(0);
    int m = *res;
    return 0;
}
```

`main` frame

Address	Value
&res	???
&m	???

Caution: automatic storage

```
int* inc(int a) {
    a += 1;
    return &a;
}
int main() {
    int* res = inc(0);
    int m = *res;
    return 0;
}
```

inc frame

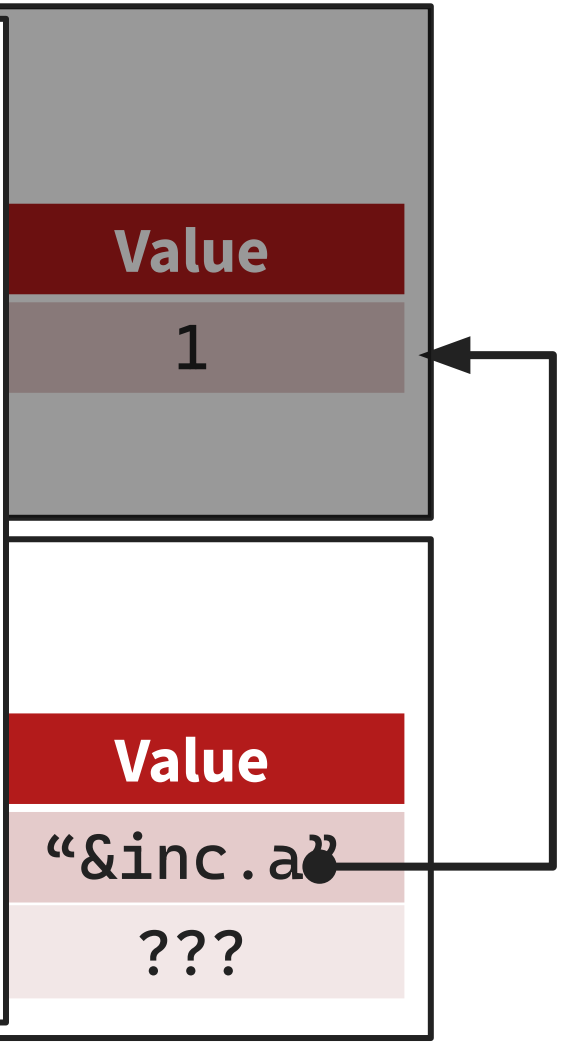
Address	Value
&a	1

main frame

Address	Value
&res	“&inc.a”
&m	???

Caution: automatic storage

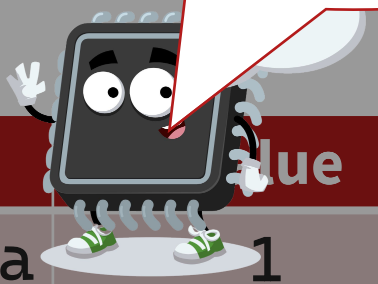
```
auto_err.c: In function 'inc':
auto_err.c:3:11: warning: function returns address
of local variable [-Wreturn-local-addr]
    3 |    return &a;
      |           ^~
Job 1, './a.out' terminated by signal SIGSEGV
(Address boundary error)
```



Caution: automatic storage

```
int* inc(int a) {
    a += 1;
    return &a;
}
int main() {
    int* res = inc(0);
    int m = *res;
    return 0;
}
```

Never return pointers to local variables!



inc frame

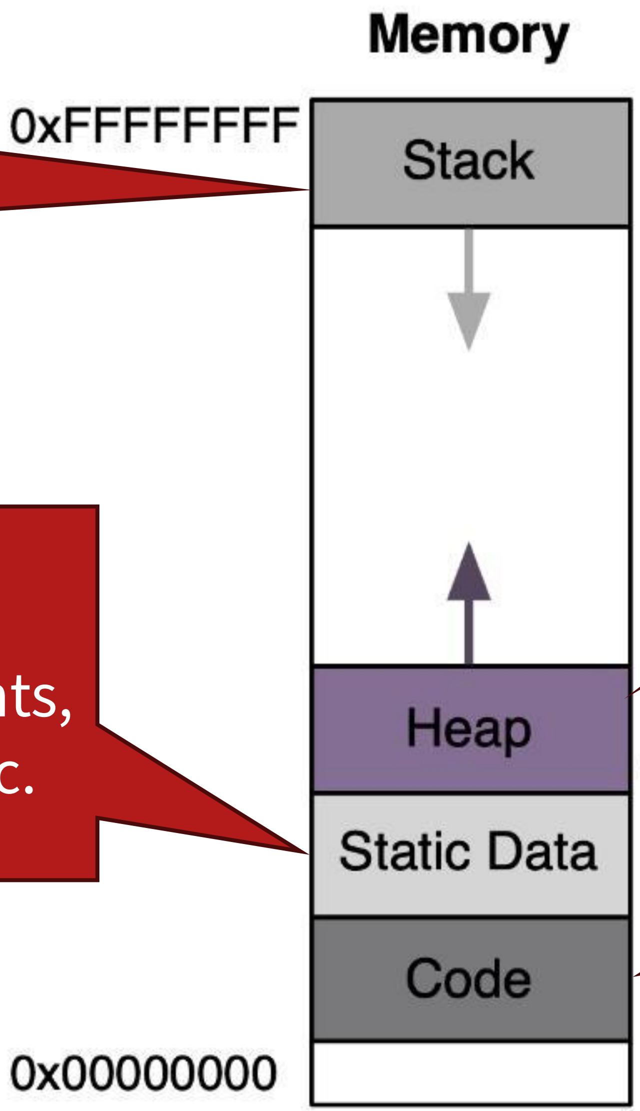
Address	Value
&a	1

main frame

Address	Value
&res	“&inc.a”
&m	???

Memory Layout

Local variables, function arguments



Dynamically allocated data

Stores global variables, constants, string literals, etc.

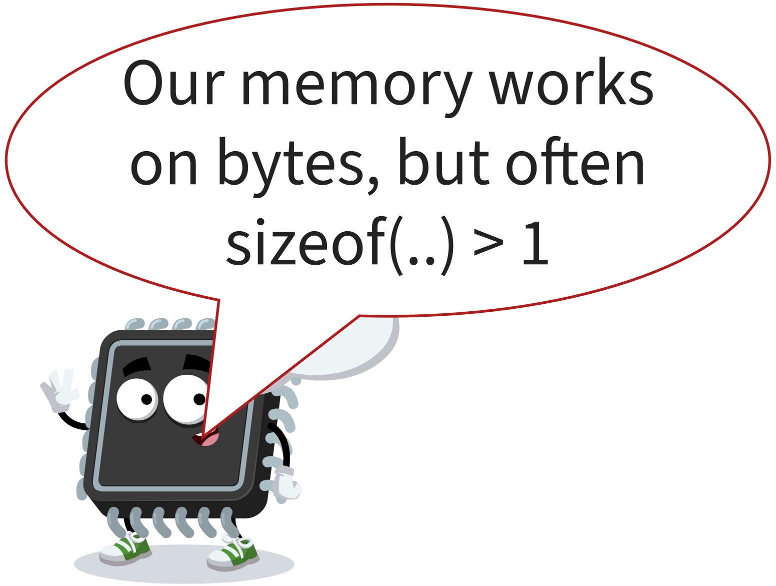
Machine code (i.e., the program!)

Different Types in Memory

```c
int main() {
  uint8_t a = 0;
  uint8_t b = 1;
  uint8_t *p = &a;
}
```

`main` frame

Address	Value
&a	0
&b	1
&p	&a



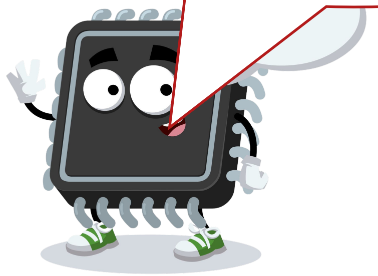
Different Types in Memory

```c
int main() {
  uint8_t a = 0;
  uint8_t b = 1;
  uint8_t *p = &a;
}
```

`main` frame

Address	Value
&a	0
&b	1
&p	&a

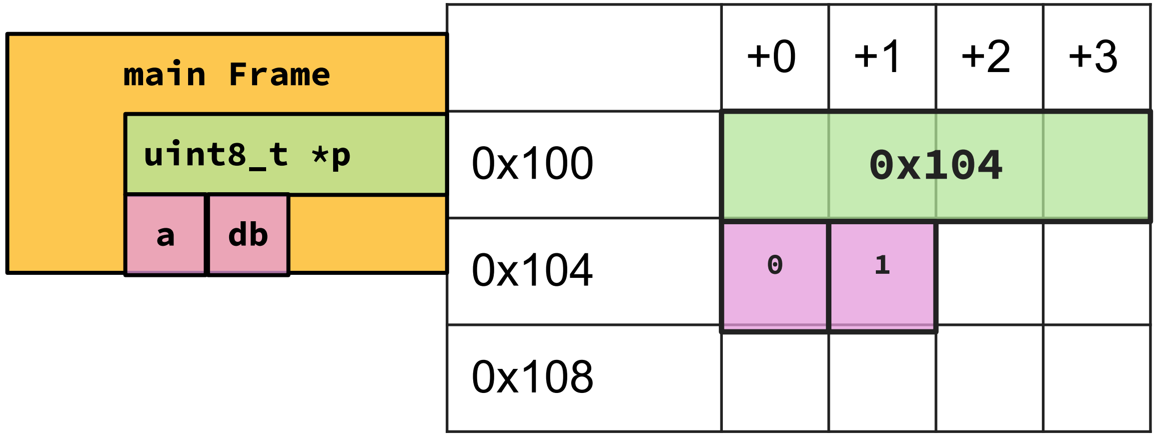
	+0	+1	+2	+3
0x100	u8	u8	u8*	
0x104	...			
0x108				



adapted from: UC Berkeley, CS61C slides

Alignment

```c
int main() {
  uint8_t a = 0;
  uint8_t b = 1;
  uint8_t *p = &a;
}
```

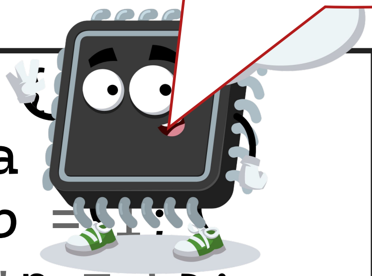


`main` frame

Address	Value
&a	0
&b	1
&p	&a

- In C, types also have alignment requirements
  - General rule: N-byte objects prefer to start at an address divisible by N
- Variables on the stack can get reordered.

Alignment




```c
int main() {
  uint8_t a
  uint8_t b = 1;
  uint8_t *p = &a;
}
```

in frame

Address	Value
&a	0
&b	1
&p	&a

main Frame

uint8_t *p

a | **db**

	+0	+1	+2	+3
0x100	**0x104**			
0x104	0	1		
0x108				

- In C, types also have alignment requirements
  - General rule: N-byte objects prefer to start at an address divisible by N
- Variables on the stack can get reordered.

Endianness

Memory: Bytes

`uint8_t mem[SIZE]`

Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07

$\texttt{load}_1\texttt{(0xBC52)}$

# of bytes

`mem[0xBC52]`

Memory: Bytes and Ints

$\mathtt{load}_4(\mathtt{0x0000})$

=

`uint8_t mem[SIZE]`

Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07

Cornell Bowers CIS
Computer Science

Memory: Bytes and Ints

Little-Endian

Least *significant byte at the* ***smallest*** *address*

$\texttt{load}_4(\texttt{0x0000})$

$=$

$\texttt{0x}\underbrace{\texttt{EA}}\underbrace{\texttt{51}}\underbrace{\texttt{B2}}\underbrace{\texttt{07}}$

`uint8_t mem[SIZE]`

Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07

Memory: Bytes and Ints

`uint8_t mem[SIZE]`

Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07

Big-Endian

Most *significant byte at the* ***smallest*** *address*

$$\mathtt{load}_4(\mathtt{0x0000})$$

$$=$$

$$\mathtt{0x}\underbrace{\mathtt{07}}\underbrace{\mathtt{B2}}\underbrace{\mathtt{51}}\underbrace{\mathtt{EA}}$$