

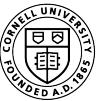
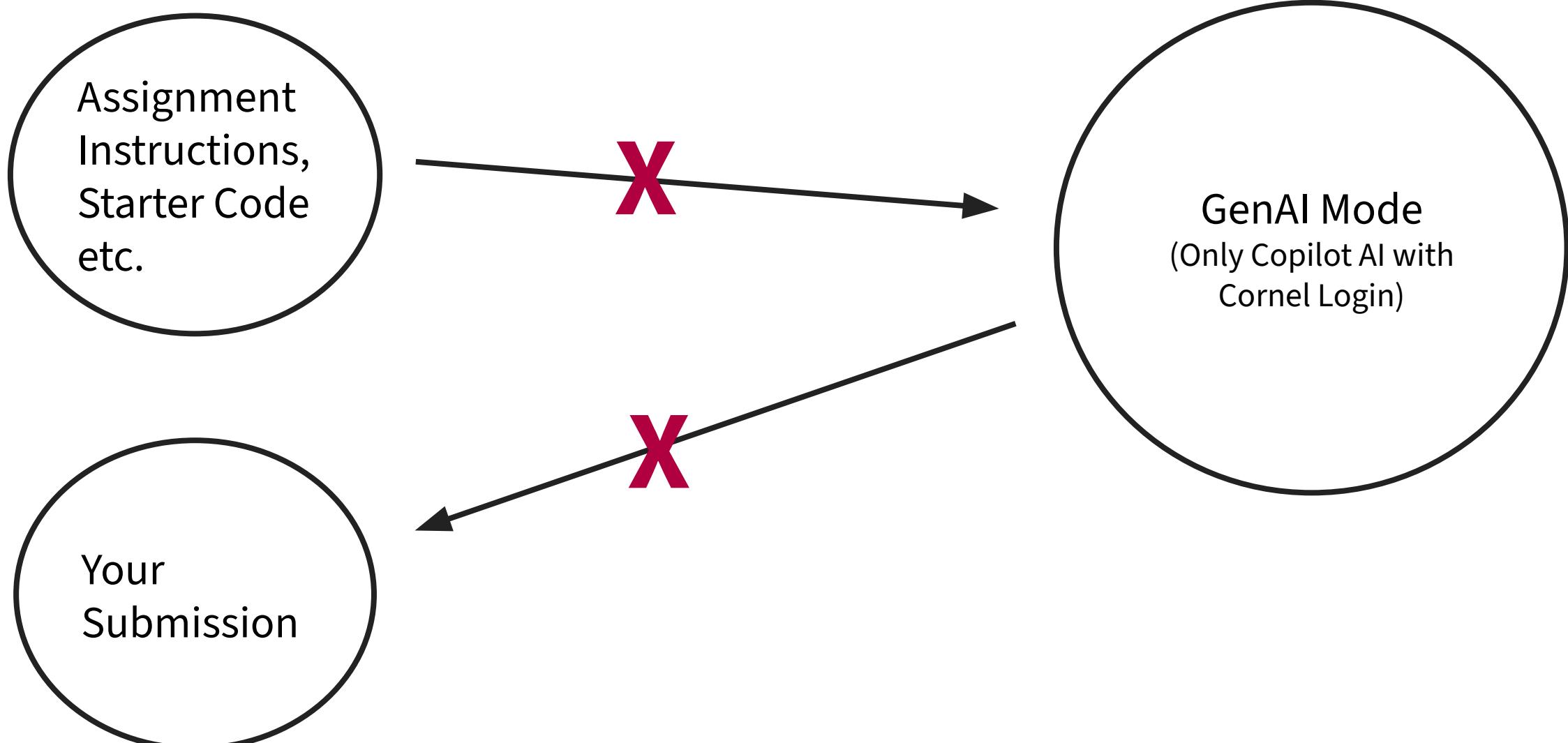
Review: GenAI Policy

CS 3410: Computer System Organization and Programming

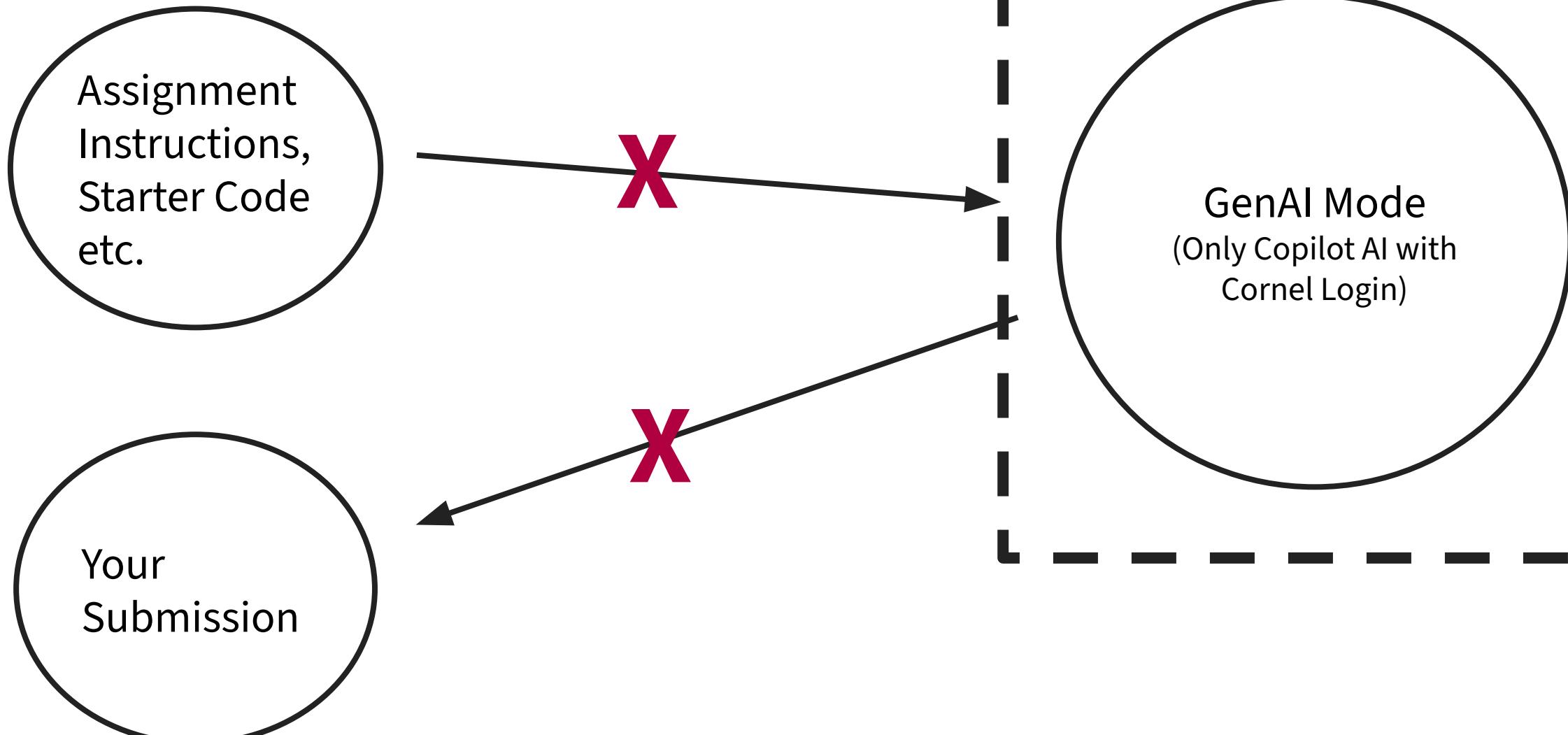
Fall 2025



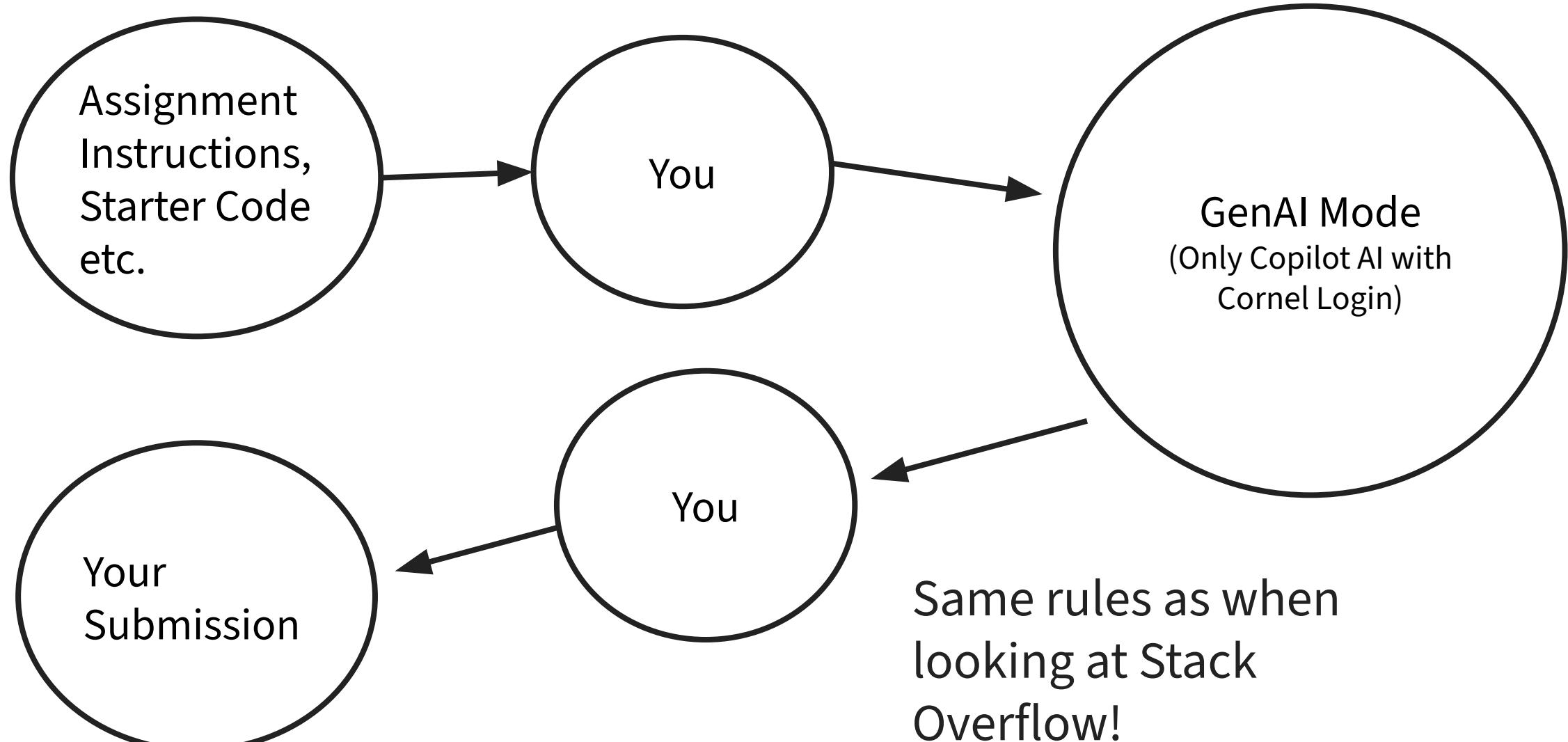
GenAI Policy



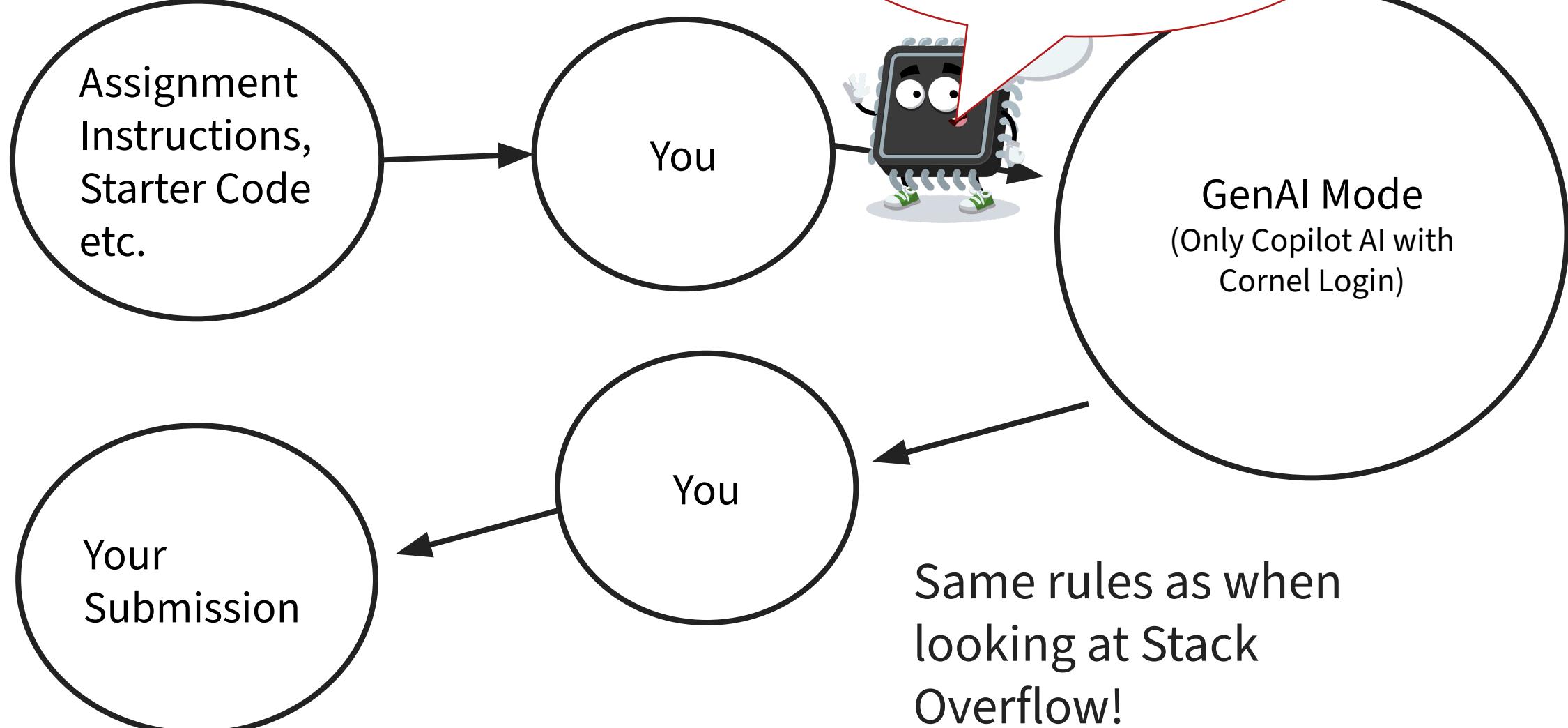
GenAI Policy



GenAI Policy



GenAI Policy



Memory Management in C

CS 3410: Computer System Organization and Programming

Fall 2025



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;           static
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;           static
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;           static
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}

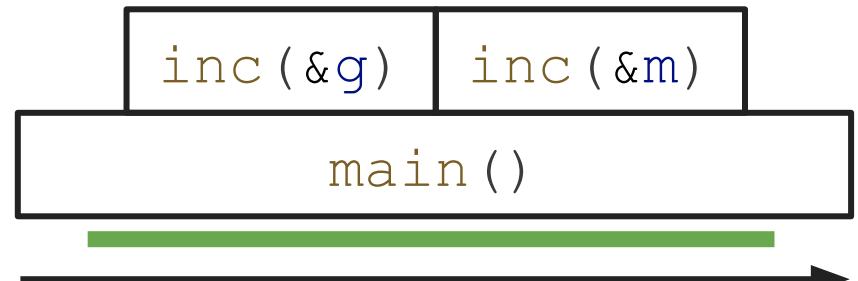
int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```



Storage Duration and Lifetime

```
int g = 2;           static
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}

int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```



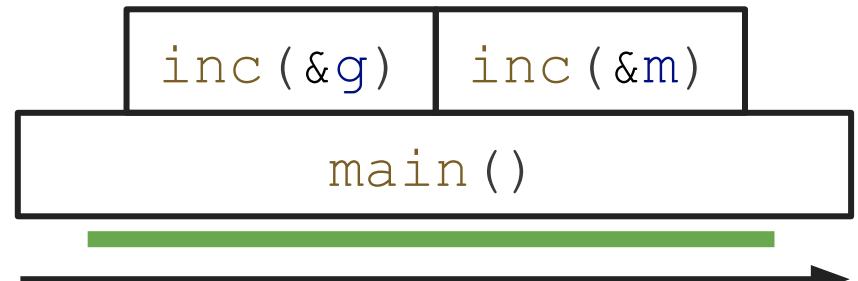
program execution



Storage Duration and Lifetime

```
int g = 2;           static
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}

int main() {
    int m = 0;
    inc(&g);
    inc(&m);
    return 0;
}
```



program execution



Storage Duration and Lifetime

```
int g = 2;                                static
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}

int main() {
    int m = 0;                            automatic
    inc(&g);
    inc(&m);
    return 0;
}
```



Storage Duration and Lifetime

```
int g = 2;                                static
void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}

int main() {
    int m = 0;                            automatic
    inc(&g);
    inc(&m);
    return 0;
}
```



Storage Duration and Lifetime

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



Storage Duration and Lifetime

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



Storage Duration and Lifetime

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

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;
}
```

The code illustrates storage duration and lifetime. The variable `a` is passed by pointer to the `inc` function. Inside `inc`, `a` is a **local automatic variable**. It is deallocated when `inc` exits. The variable `b` is a **local automatic variable** that also goes out of scope when `inc` exits. The variable `m` is a **global automatic variable** declared in `main`. It is deallocated when `main` exits.



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

make sure type matches

The diagram illustrates the storage duration and lifetime of variables in the provided C code. The variables 'a' and 'b' in the 'inc' function are both labeled as 'automatic', indicated by purple arrows pointing to the text. The variable 'm' in the 'main' function is dynamically allocated using 'malloc', so its type must match the pointer type passed to 'inc'. This is highlighted by a red arrow pointing from the malloc call to the text '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

The diagram illustrates the storage duration and lifetime of variables in the provided C code. It uses arrows to map variable declarations to their definitions:

- A purple arrow points from the declaration `int* a` to its definition `*a = b;`, labeled "automatic".
- A blue arrow points from the declaration `int b` to its definition `b = *a;`, labeled "automatic".
- A red arrow points from the declaration `int* m` to its definition `(int*)malloc(1 * sizeof(int))`, with the annotation "make sure type matches" below it.
- A black arrow points from the argument `1 * sizeof(int)` in the `malloc` call to the "# of elements" text above it.



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

do not forget to initialize

The diagram illustrates the storage duration and lifetime of variables. It shows a function inc that takes a pointer to an integer and increments its value. The variable 'a' is automatic within the function. The variable 'b' is also automatic. In the main function, a pointer 'm' is dynamically allocated using malloc. The size of the allocation is calculated as 1 * sizeof(int). The variable 'm' is also automatic. The function inc is called with 'm' as an argument. After the function call, free is used to deallocate the memory pointed to by 'm'. A note 'make sure type matches' is placed between the malloc and free calls. A note 'do not forget to initialize' is placed next to the initialization of 'm' in the main function.



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



do not forget to initialize



Storage Duration and Lifetime

```
#include<stdlib.h>

void inc(int* a) {
    int b = *a;
    b += 1;
    *a = b;
}

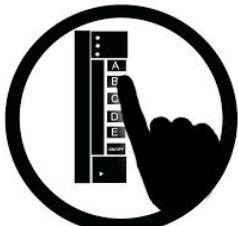
int main() {
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
    inc(m);
    free(m);
    return 0;
}
```

automatic
automatic

???



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;
}
```

The diagram illustrates the storage duration and lifetime of variables in the provided C code. It uses arrows to map variable names to their descriptions:

- A purple arrow points from the variable `a` in the `inc` function declaration to the word "automatic".
- A blue arrow points from the variable `b` in the `inc` function body to the word "automatic".
- A green arrow points from the variable `m` in the `main` function body to the word "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;
}
```

The diagram illustrates the storage duration and lifetime of variables in the provided C code. It uses arrows to map variable names to their descriptions:

- A purple arrow points from the variable `a` to the word "automatic".
- A blue arrow points from the variable `b` to the word "automatic".
- A green arrow points from the variable `m` to the word "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;
}
```

The diagram illustrates the storage duration and lifetime of variables in the provided C code. It uses arrows to map variable names to their descriptions:

- A purple arrow points from `a` to the word "automatic".
- A blue arrow points from `b` to the word "automatic".
- A green arrow points from `m` to the word "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

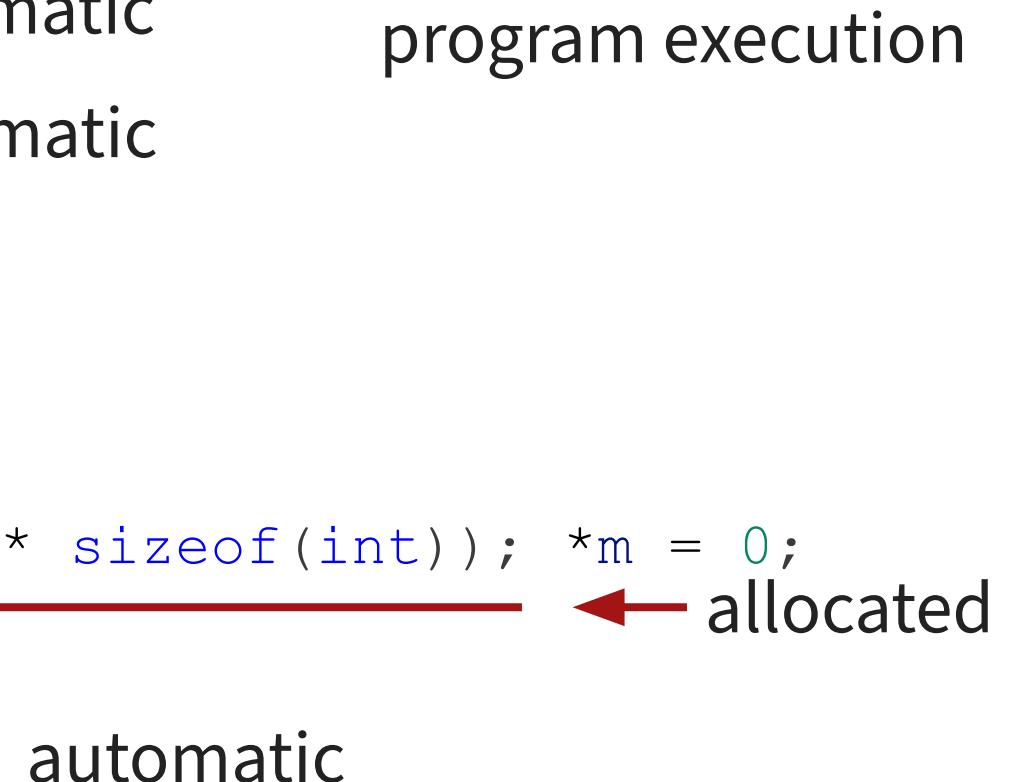
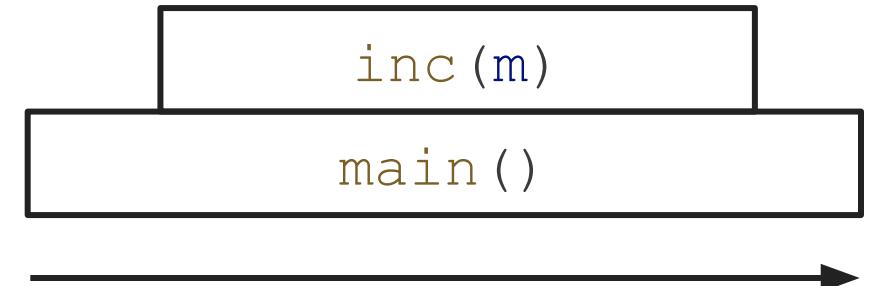


Storage Duration and Lifetime

```
#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;
}
```



Storage Duration and Lifetime

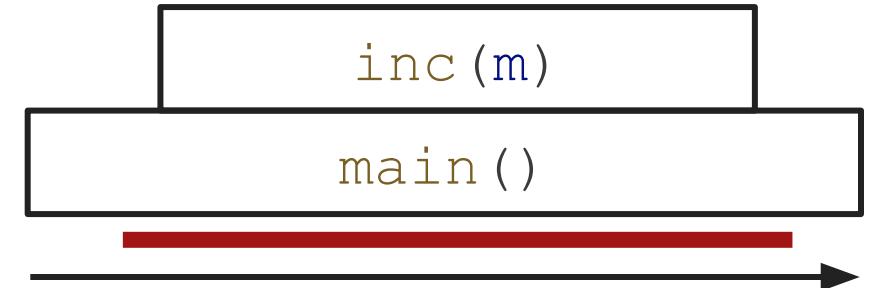
```
#include<stdlib.h>
```

```
void inc(int* a) {  
    int b = *a;  
    b += 1;  
    *a = b;  
}
```

automatic
automatic

```
int main() {  
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;  
    inc(m);  
    free(m);  
    return 0;  
}
```

allocated
automatic

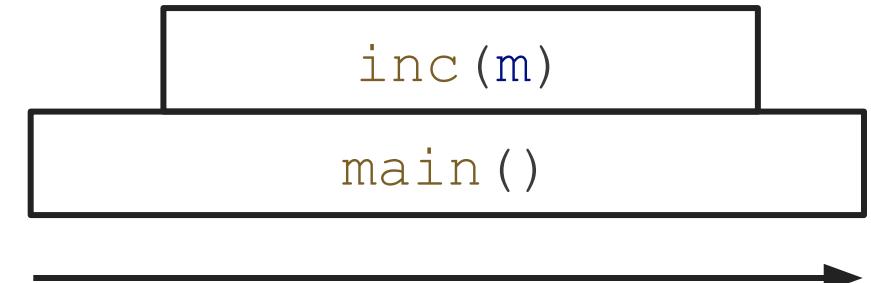


Storage Duration and Lifetime

```
#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

automatic

inc (m)

main ()

program execution

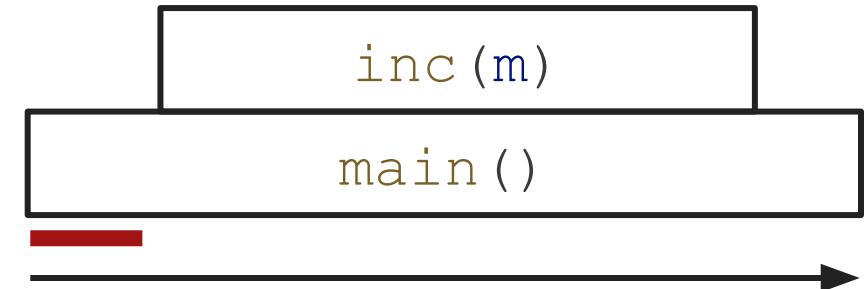
allocated

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

automatic

inc (m)

main ()

program execution

allocated

Storage Duration and Lifetime

allocated
memory
is not
live!

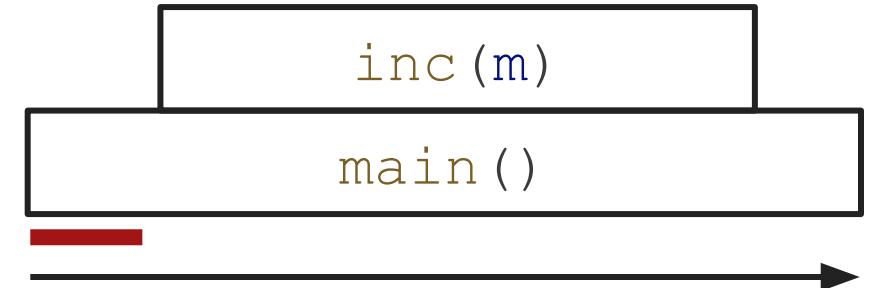
```
#include<stdlib.h>
```

```
void inc(int* a) {  
    int b = *a;  
    b += 1;  
    *a = b;  
}
```

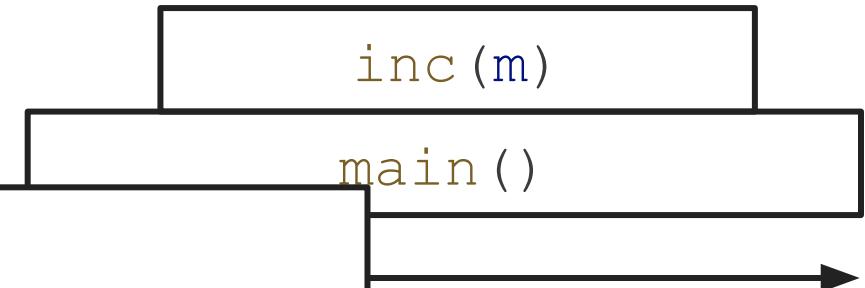
automatic
automatic

```
int main() {  
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;  
    free(m);  
    inc(m);  
    return 0;  
}
```

allocated
automatic



Storage Duration and Lifetime



==17492==ERROR: AddressSanitizer:

heap-use-after-free on address

0x7dfa7a20010 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.



Storage Duration and Lifetime

allocated
memory
is not
live!

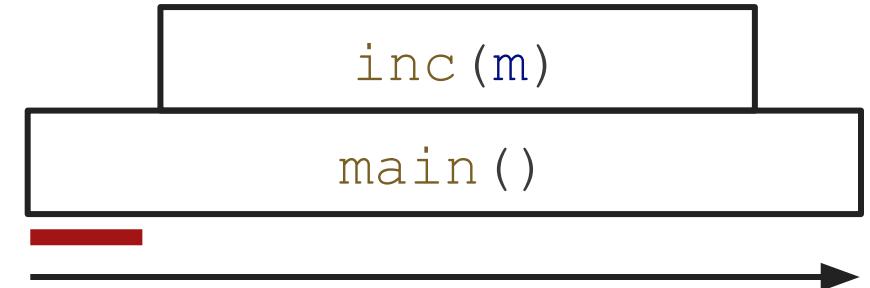
```
#include<stdlib.h>
```

```
void inc(int* a) {  
    int b = *a;  
    b += 1;  
    *a = b;  
}
```

automatic
automatic

```
int main() {  
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;  
    free(m);  
    inc(m);  
    return 0;  
}
```

allocated
automatic



Storage Duration and Lifetime

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

```
void inc(int* a) {  
    int b = *a;  
    b += 1;  
    *a = b;  
}
```

automatic
automatic

```
int main() {
```

```
    int* m = (int*)malloc(1 * sizeof(int)); *m = 0;
```

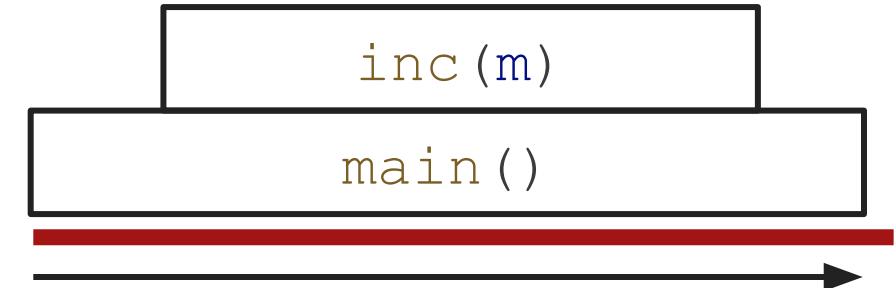
allocated

```
    free(m);
```

```
    inc(m);
```

```
    return 0;
```

automatic



Storage Duration and Lifetime

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

```
void inc(int* a) {  
    int b = *a;  
    b += 1;  
    *a = b;  
}
```

automatic
automatic

```
int main() {
```

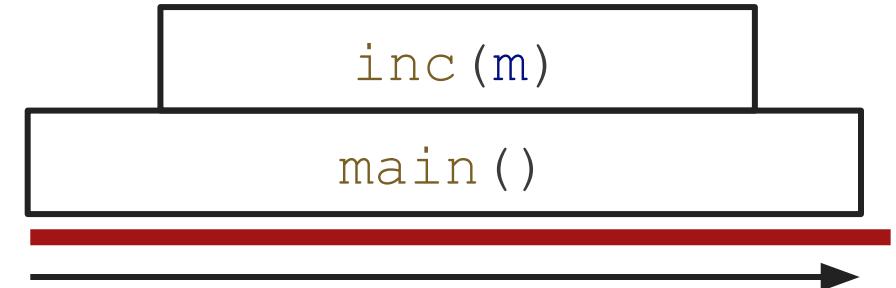
```
    int* m = (int*)malloc(1 * sizeof(int));
```

```
    free(m);
```

```
    inc(m);
```

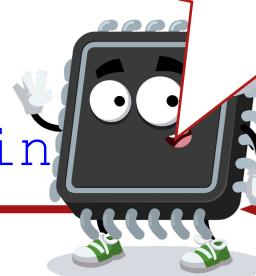
```
    return 0;
```

automatic



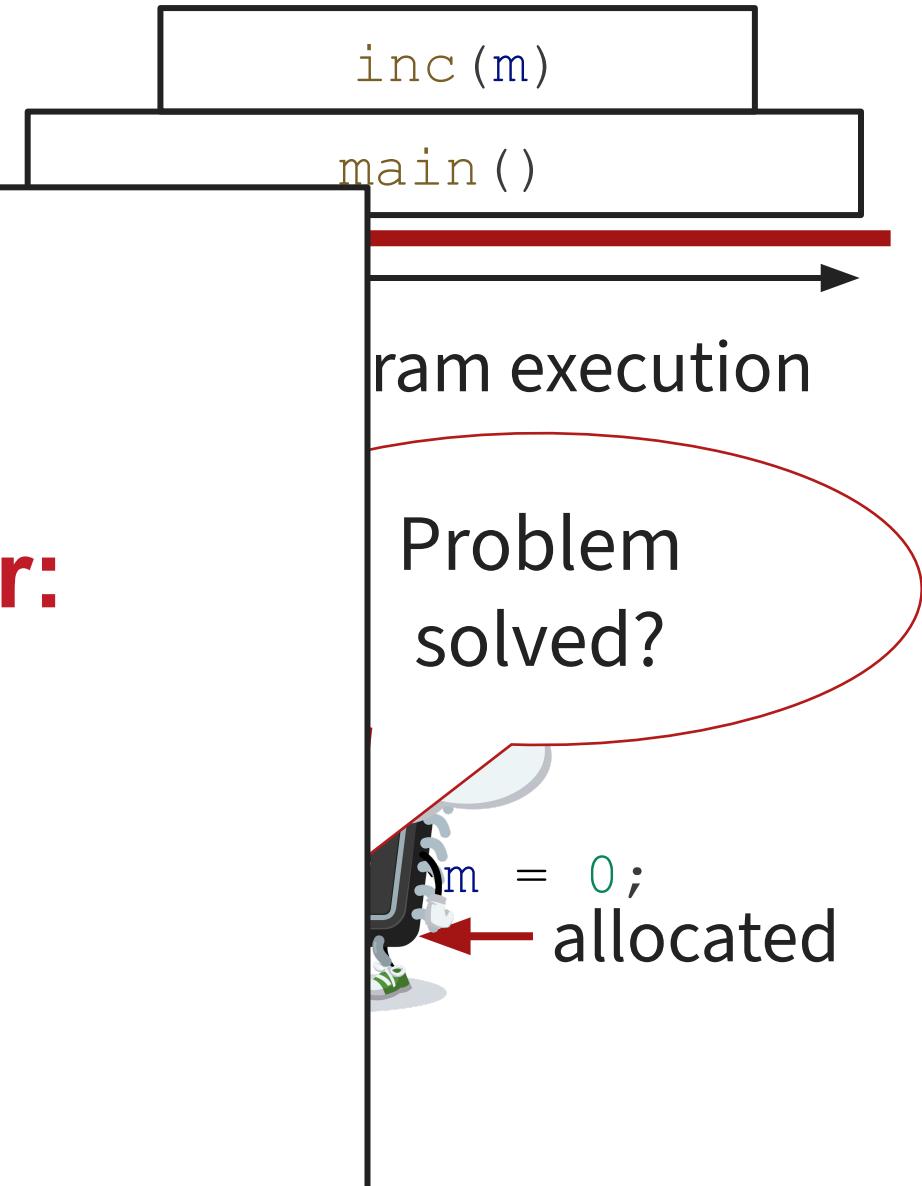
program execution

Problem solved?



m = 0;
allocated

Storage Duration and Lifetime



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

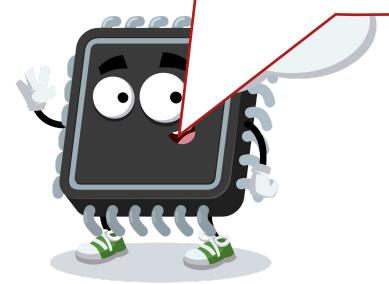
Memory Leak

```
#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;
}
```

Sometimes OK.
For short lived
programs.



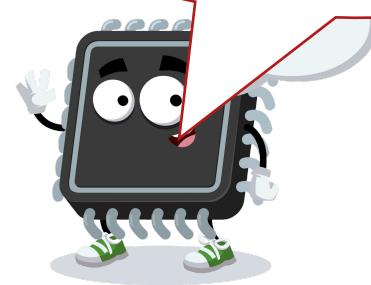
Memory Leak

```
#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);
    }
    return 0;
}
```

Most programs
need to reuse
memory, though.



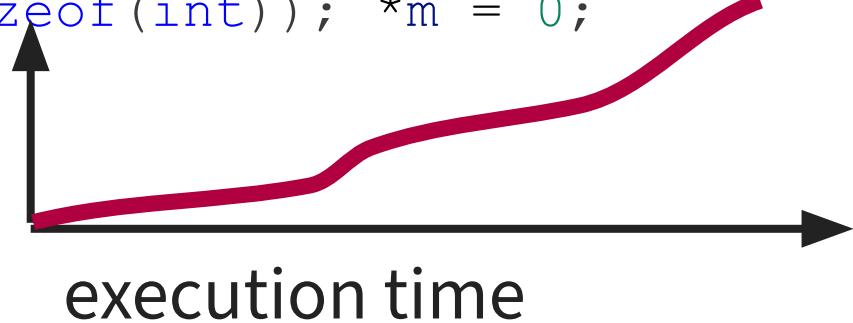
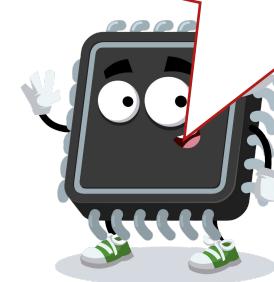
Memory Leak

```
#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);    memory use
    }
    return 0;
}
```

Most programs
need to reuse
memory, though.



Memory Leak

Most programs
try to reuse
memory, 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).

```
return 0;
```

execution time



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);      memory use
        free(m);
    }
    return 0;
}
```

Free, as soon as you know that you won't use the allocation again.

memory use

execution time

free(m);



Double Free

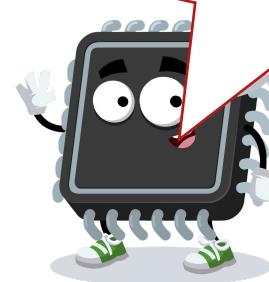
```
#include<stdlib.h>
void inc(int* a) {
    // ...
}

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;
```

memory use

execution time

Free as much as possible?!



Double Free

Free as much as
possible?!

```
==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
```

```
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);
    return 0;
}
```

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

freed by thread T0 here → free(m);

attempting double-free → free(m);



Free exactly
once!



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.

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:**

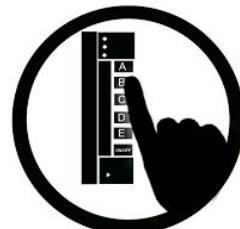


Storage Duration and Lifetime

- **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

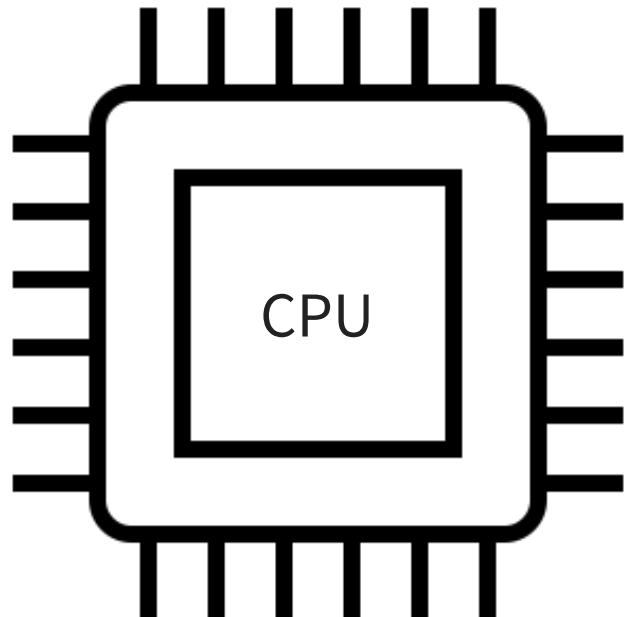


Storage Implementation

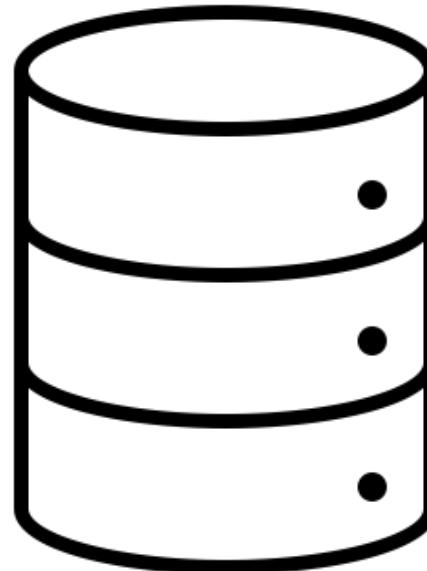


A Mental Model of Memory

Processor



Memory

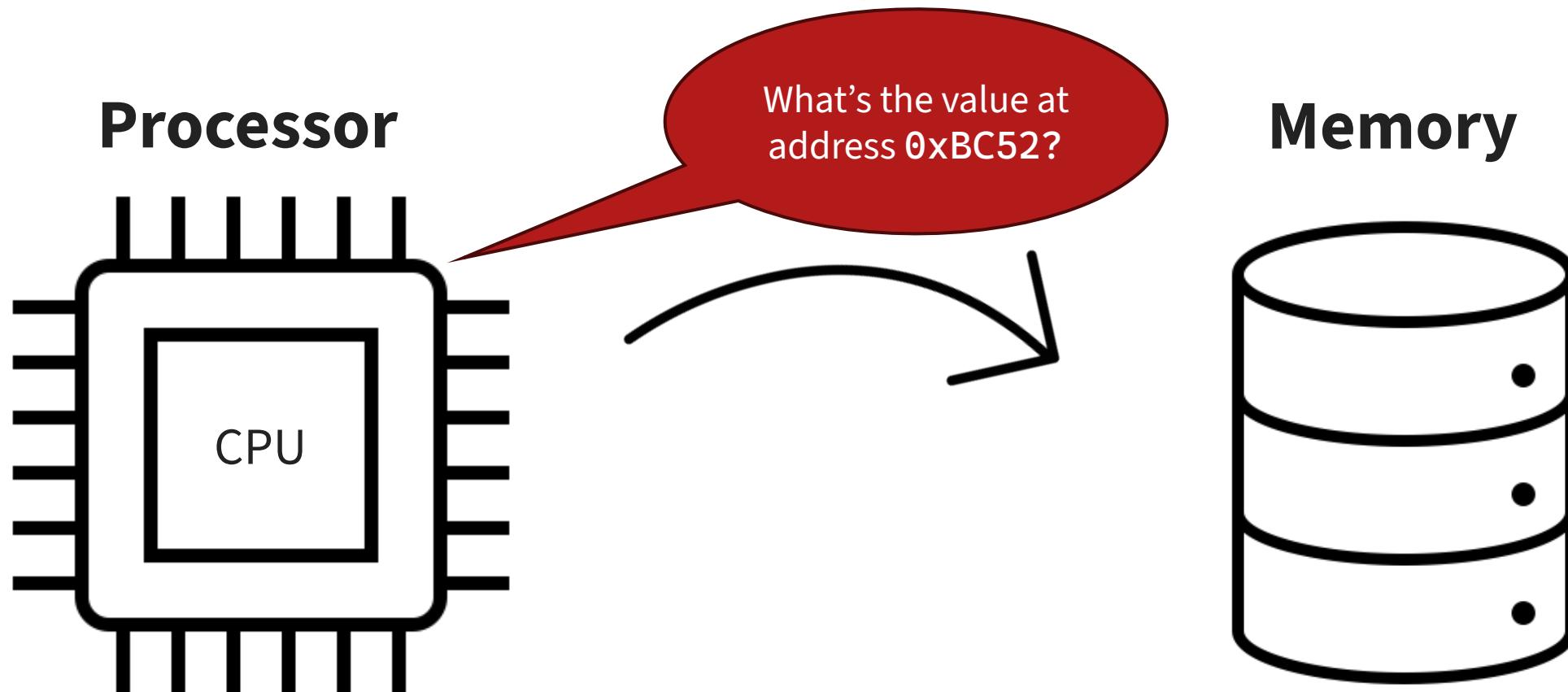


`uint8_t mem[SIZE];`

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



A Mental Model of Memory

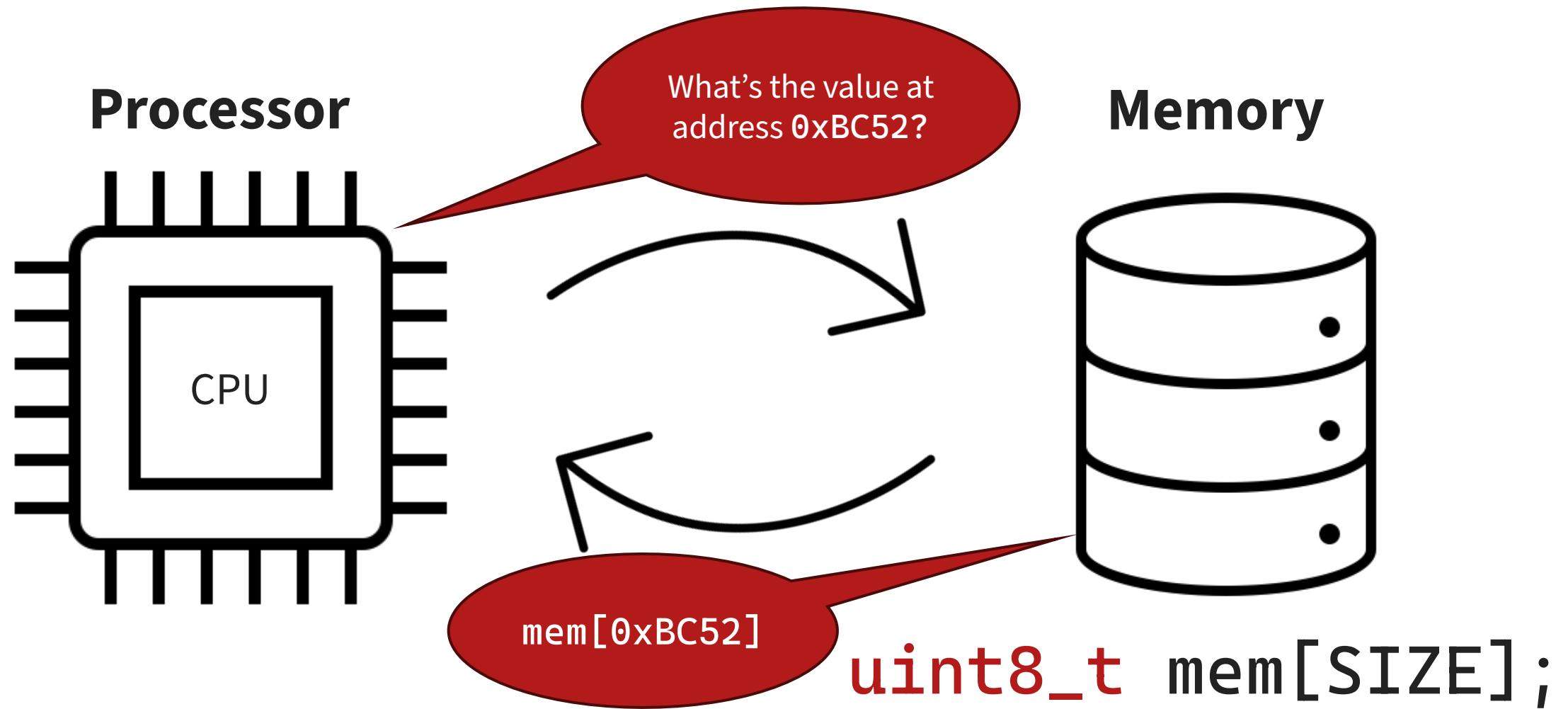


`uint8_t mem[SIZE];`

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

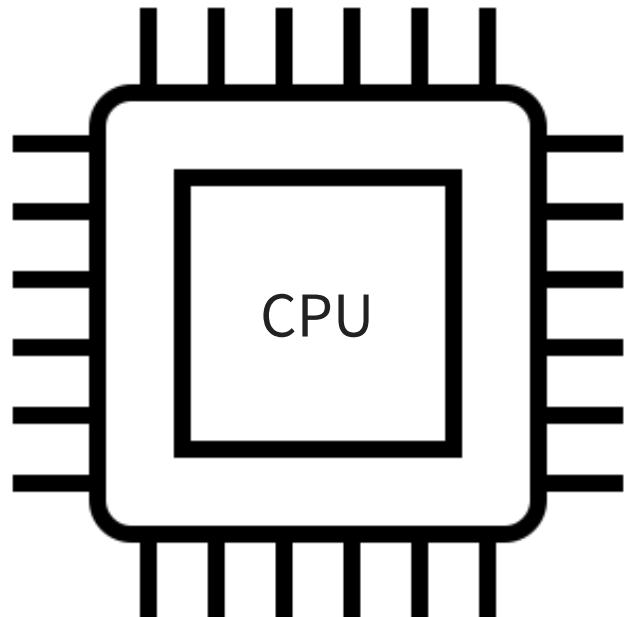


A Mental Model of Memory

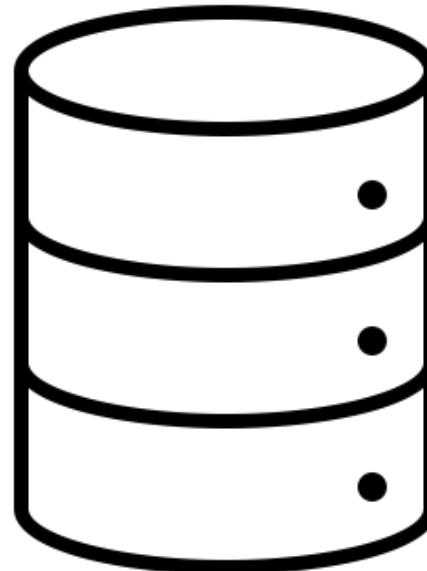


A Mental Model of Memory

Processor



Memory

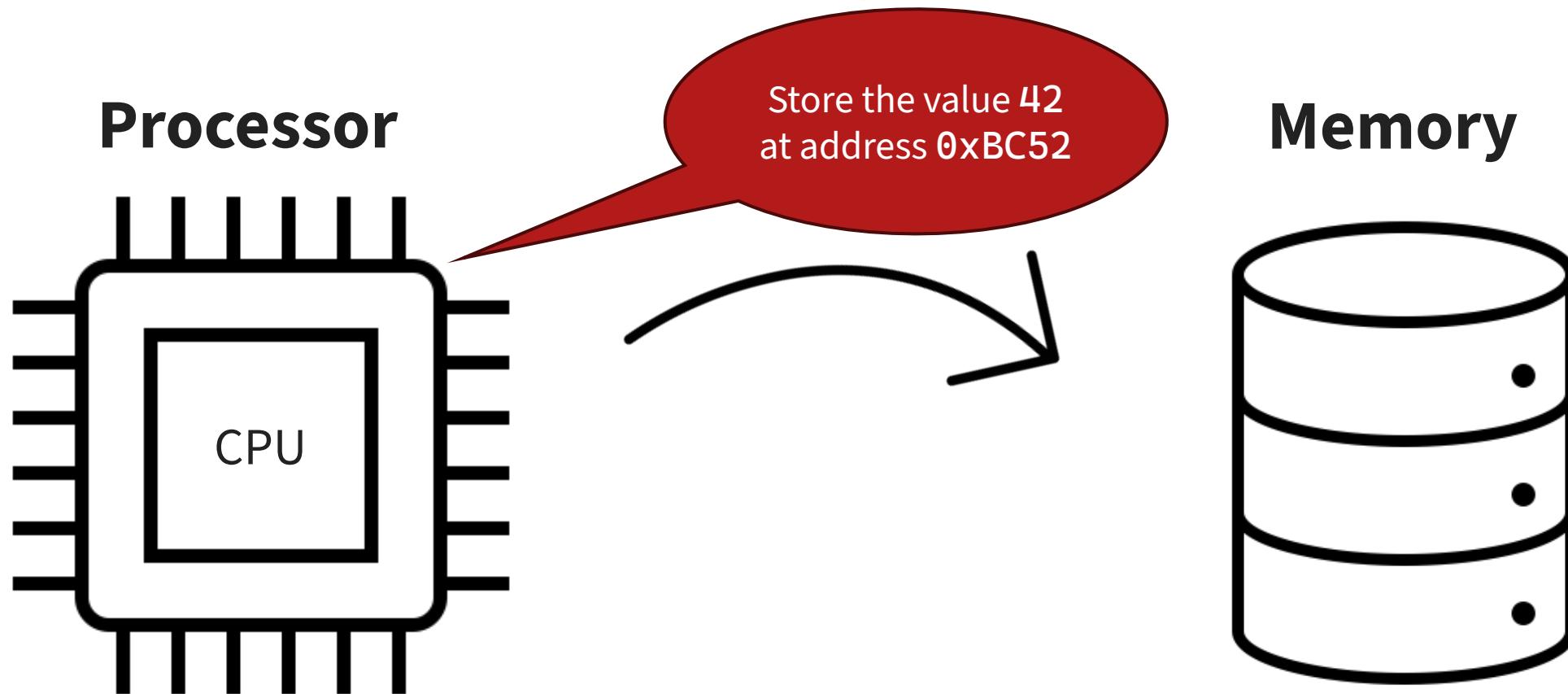


`uint8_t mem[SIZE];`

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



A Mental Model of Memory

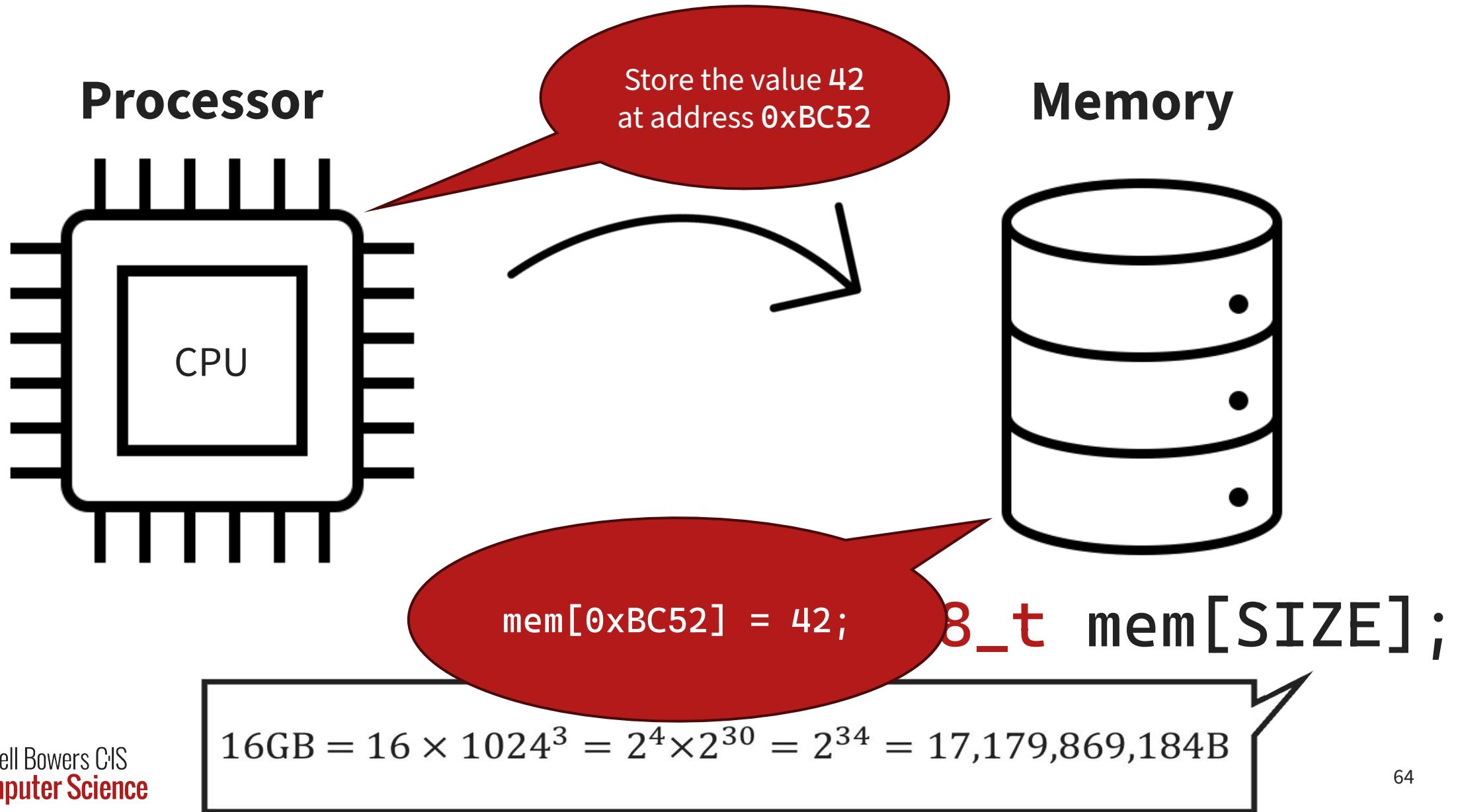


`uint8_t mem[SIZE];`

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



A Mental Model of Memory



Implementation: static storage

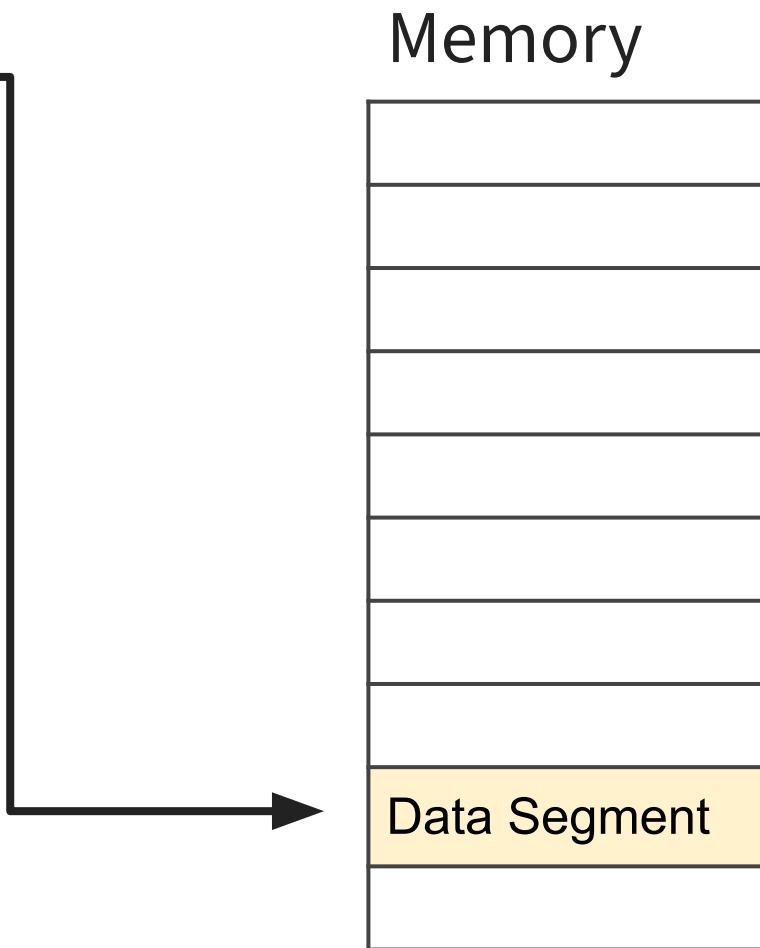
```
int g = 2;  
  
void inc(int* a) {  
  
    int b = *a;  
  
    b += 1;  
  
    *a = b;  
  
}  
  
int main() {  
  
    int m = 0;  
  
    inc(&g);  
  
    inc(&m);  
  
    return 0;  
}
```

Memory



Implementation: static 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;  
}
```



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



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

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

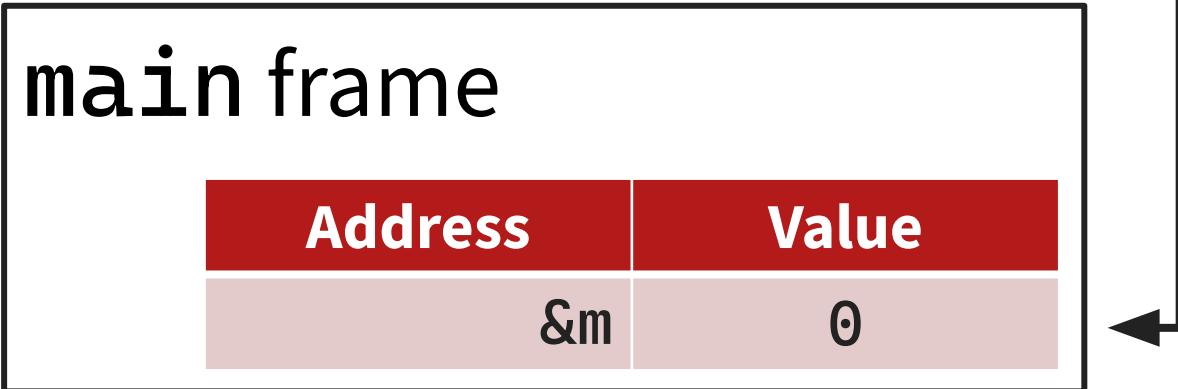
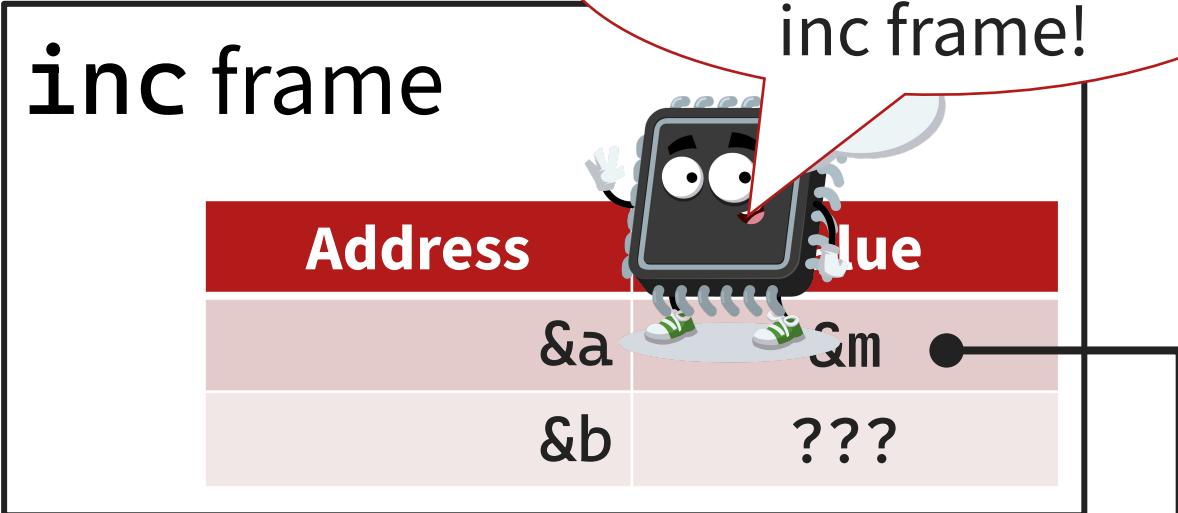
Data Segment:

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;  
}
```

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;  
}
```



Data Segment:

main frame	
Address	Value
&m	1

Data Segment:	
Address	Value
&g	3

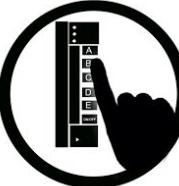
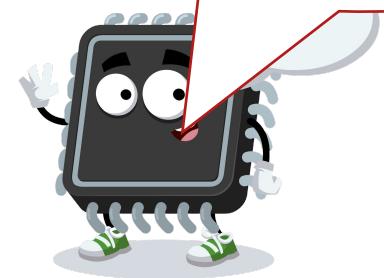


Implementation: automatic storage

```
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;  
}
```



What do we call
this data structure?



PollEv.com/cs3410

main frame

Address	Value
&m	1

Data Segment:

Address	Value
&g	3

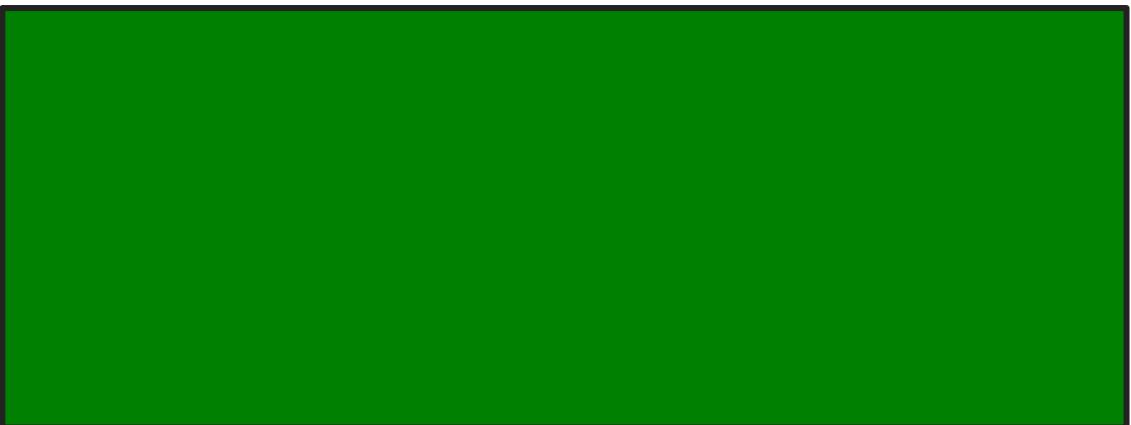
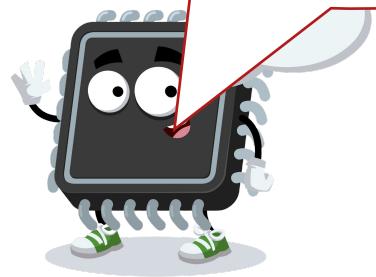
80



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?



Data Segment:

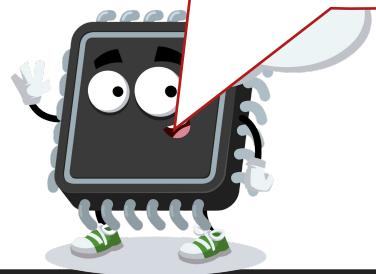
Address	Value
&g	3

81

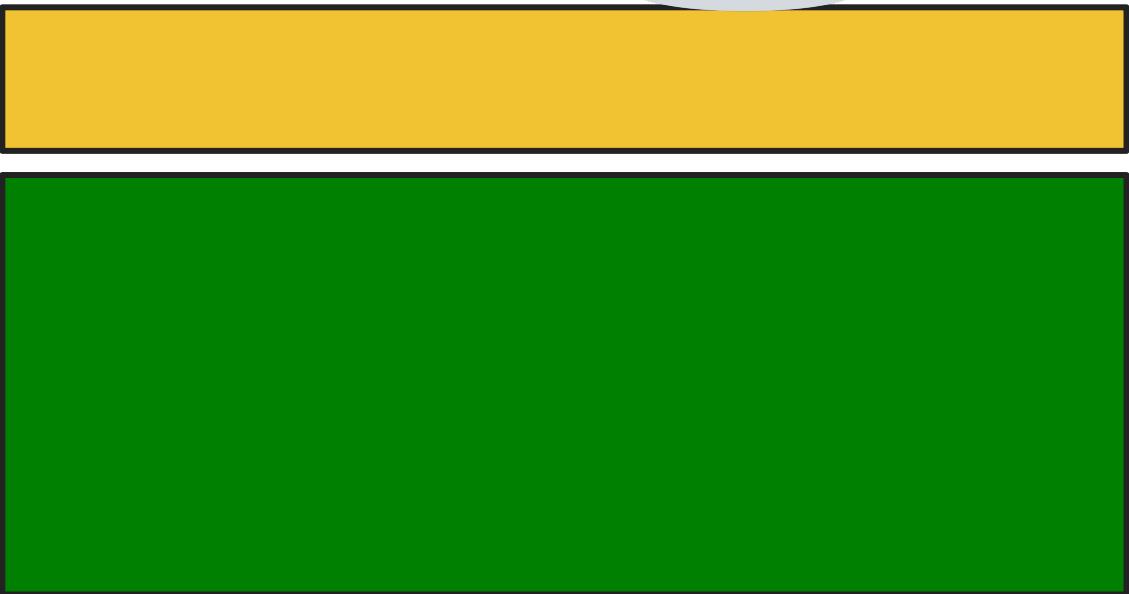


Implementation: automatic storage

What do we call
this data structure?



```
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

What do we call
this data structure?

```
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

83



Implementation: automatic storage

It's a **stack!**

```
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:

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	???



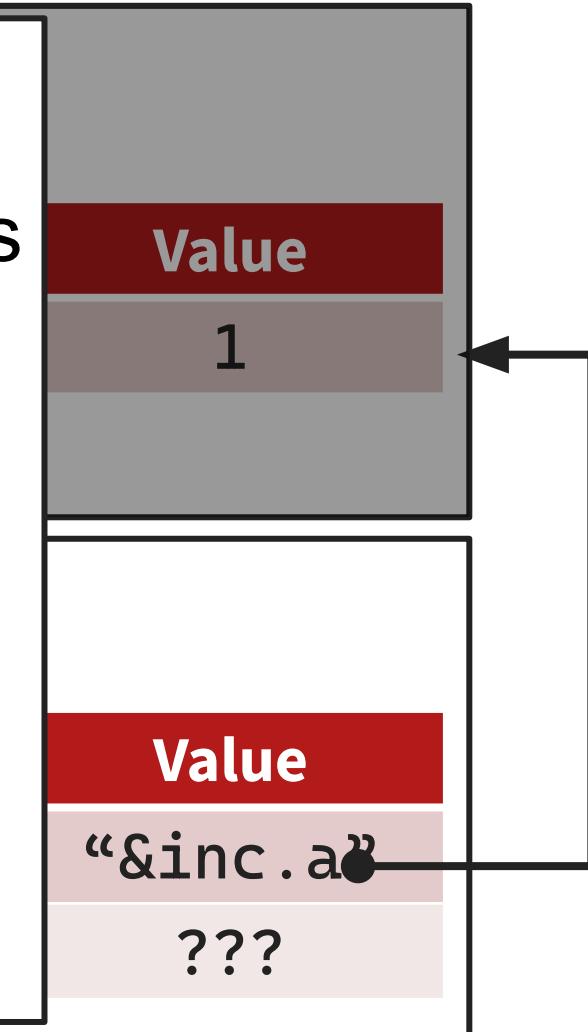
Caution: automatic storage

auto_err.c: In function ‘inc’:

auto_err.c:3:11: warning: function returns address
of local variable [-Wreturn-local-addr]

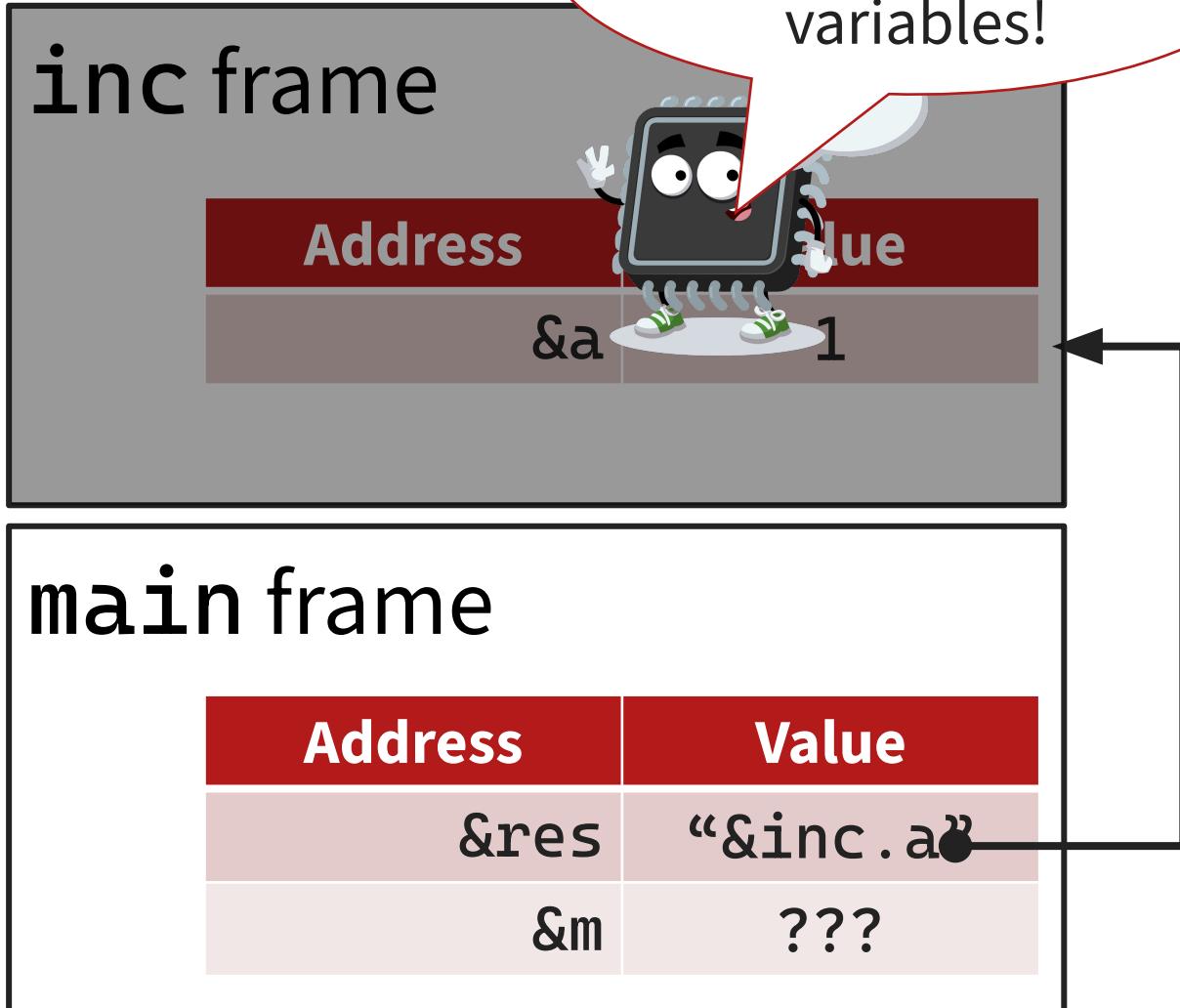
```
3 | return &a;  
|  
| ~
```

Job 1, './a.out' terminated by signal SIGSEGV
(Address boundary error)

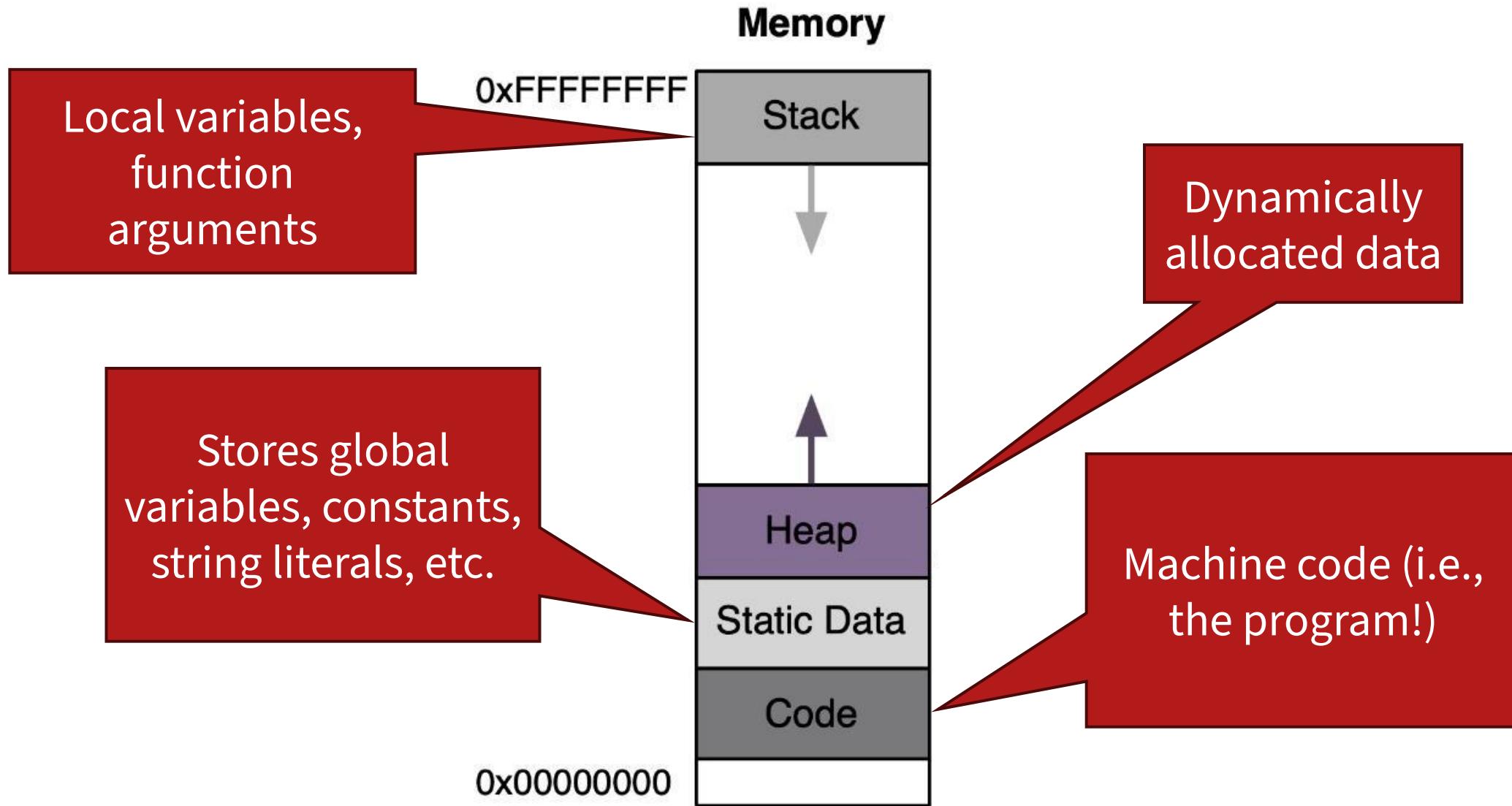


Caution: automatic storage

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



Memory Layout



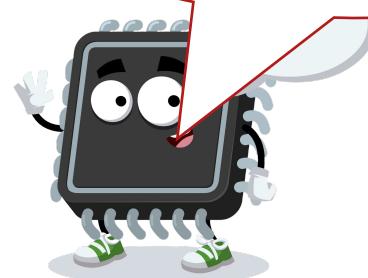
Different Types in Memory

```
1 int main() {  
2     uint8_t a = 0;  
3     uint8_t b = 1;  
4     uint8_t *p = &a;  
5 }
```

main frame

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

Our memory works
on bytes, but often
 $\text{sizeof}(\cdot) > 1$



Different Types in Memory

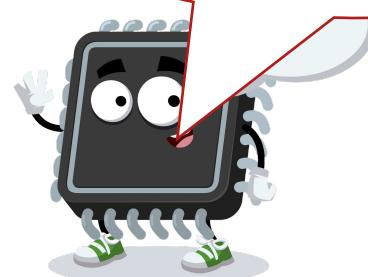
```
1 int main() {  
2     uint8_t a = 0;  
3     uint8_t b = 1;  
4     uint8_t *p = &a;  
5 }
```

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

main frame

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

Our memory works
on bytes, but often
`sizeof(..) > 1`

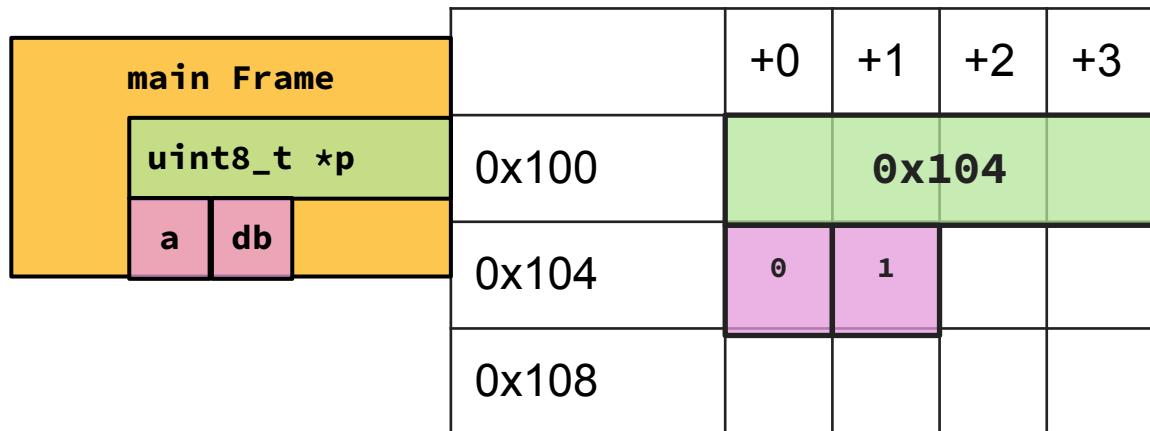


adapted from: UC Berkeley, CS61C slides



Alignment

```
1 int main() {  
2     uint8_t a = 0;  
3     uint8_t b = 1;  
4     uint8_t *p = &a;  
5 }
```



main frame

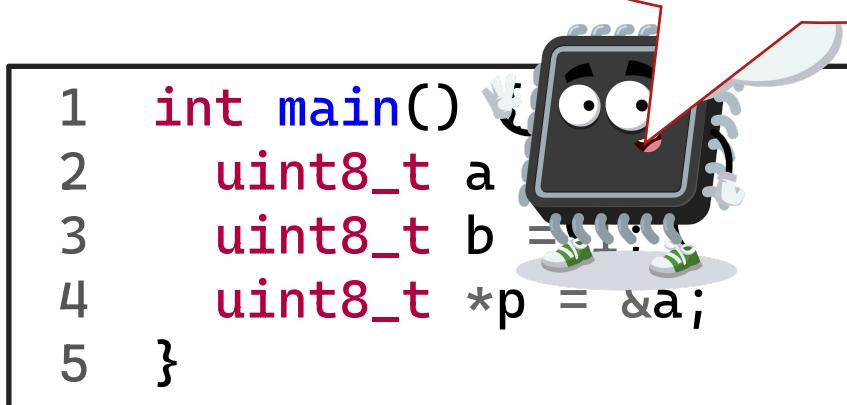
Address	Value
&a	0
&b	1
&p	&a

- 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.



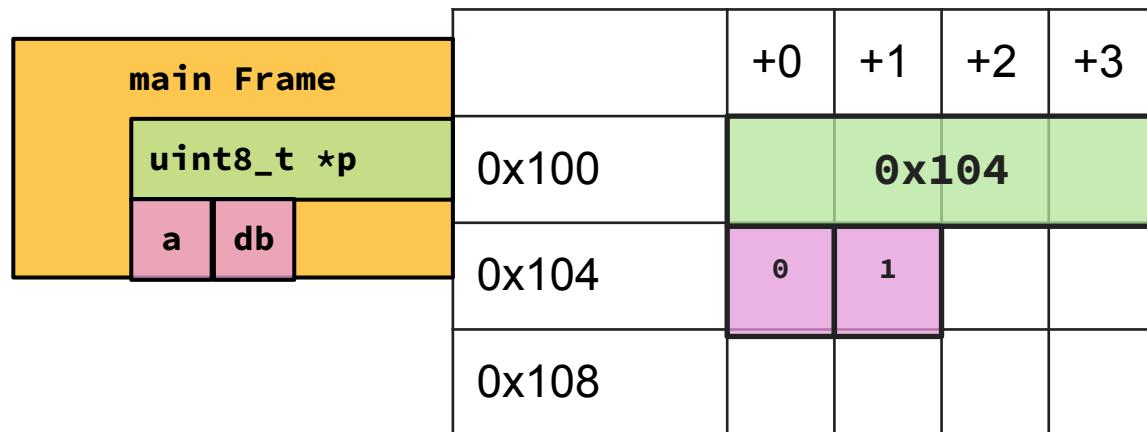
Alignment

This is why we aren't
allowed to write:
 $\&a > \&b$



in 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.



Endianness



Memory: Bytes

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

of bytes

$\text{mem}[0xBC52]$

`uint8_t mem[SIZE]`

Address	Value (<code>uint8_t</code>)
...	...
0xBC52	0xBF
...	...
0x000F	0x02
...	...
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07



Memory: Bytes and Ints

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

=

`uint8_t mem[SIZE]`

Address	Value (<code>uint8_t</code>)
...	...
0xBC52	0xBF
...	...
0x000F	0x02
...	...
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07



Memory: Bytes and Ints

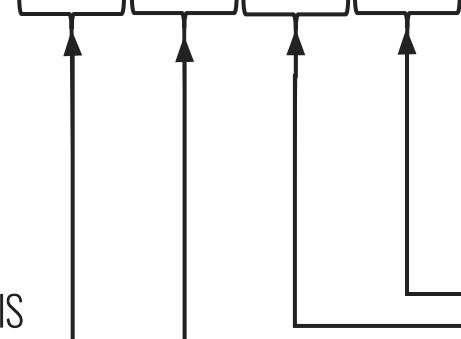
Little-Endian

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

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

=

0xEA51B207



uint8_t mem[SIZE]

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



Memory: Bytes and Ints

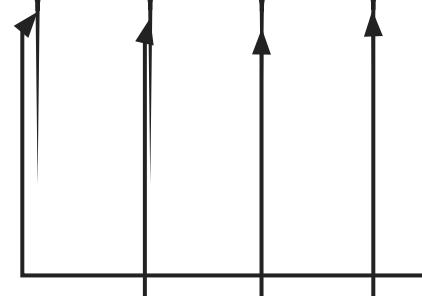
Big-Endian

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

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

=

0x07B251EA



uint8_t mem[SIZE]

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

