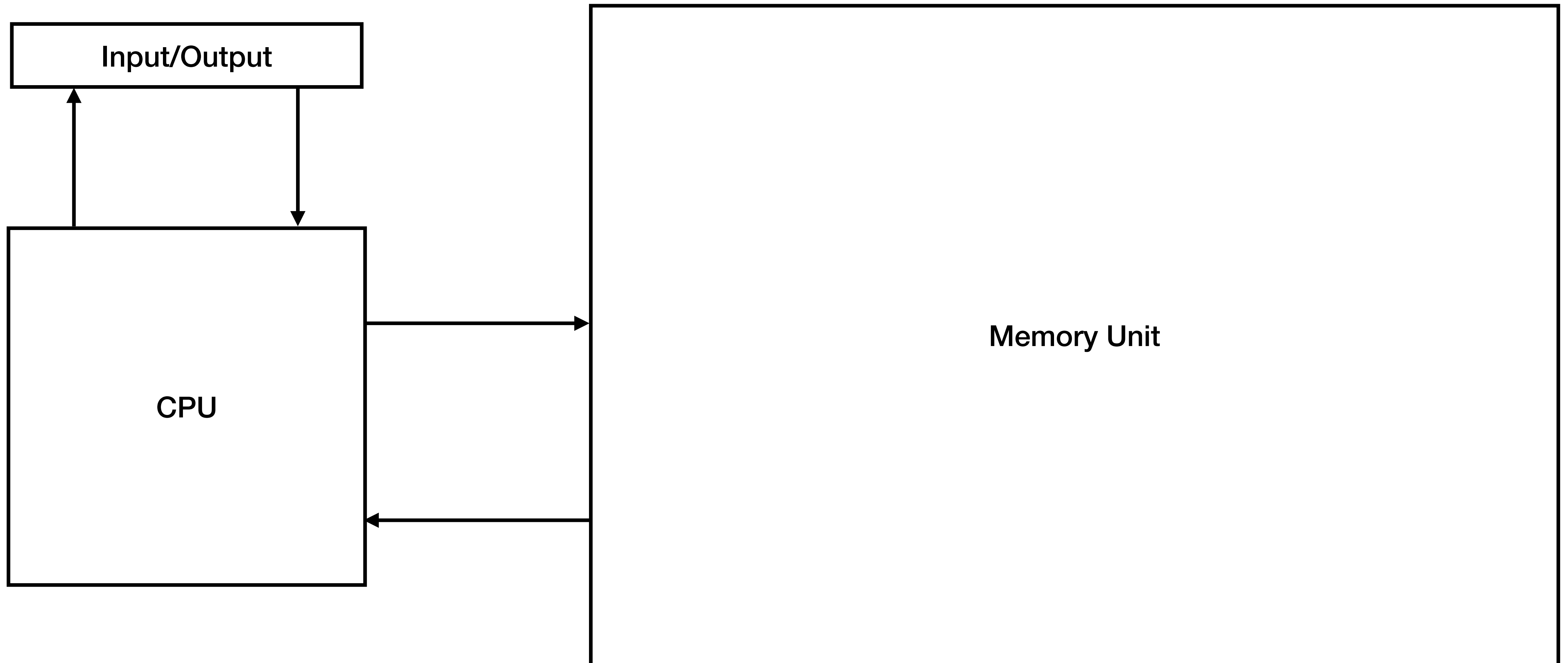


# Wrap Up

CS143: lecture 21

Byron Zhong, July 30

# A Von Neumann Machine



# A new perspective...

- Variables (data) and functions (code) live in memory
  - Memory is a contiguous storage of bytes
  - Each byte has an address -- variables and functions have addresses
- When executing a program, CPU fetches an instruction from memory and performs actions:
  - Read (n bytes) from an address to a register, write (n bytes) to an address to a register
  - Manipulate the bits in registers -- computation
  - Jump to another instruction, check for conditions, ...
- The compiler `clang` translates your C program into these instructions

# A new perspective...

- A process's memory is partitioned into
  - The stack: the compiler uses this to manage local variables. Stack frames come and go as functions are called and return
  - The heap: you use this to store data with complicated lifetime
    - `ptr = malloc(n);`
    - `free(ptr);`
    - One malloc, one free
  - Code, global variables, string literals ...
- Virtual memory: OS gives each process its own memory address space (0 -- FFFFFFFF...)

# A new perspective...

- Data and code are just bits
  - A bit answers a yes/no question -- we specify what the questions are by agreeing on an encoding
  - Unsigned integer encoding -- each bit indicates the presence of a power of 2
  - Signed integer encoding (2's complement) -- the highest bit is negative
  - We can come up with our own encodings (e.g. student record)
- Types are used to keep track of the encodings

# A new perspective...

- Statically, we can organize data...
  - ... of different types into a `struct`
    - to represent a real-world object
    - to group variables that are dependent (invariants)
  - ... of the same type into an array
    - to represent multiple instances of the same thing
    - to apply the same action repeatedly
- Compiler translates structs and arrays access into direct memory access

# A new perspective...

- Dynamically, we can organize data as:
  - `list`: an ordered sequence
    - If we use pointers to keep track of the order -- linked list
      - Easy to reorder, insert, delete, ...
    - If we use relative memory position to keep track of the order -- arraylist
      - Easy to access specific element
  - `map`: a collection of key-value pairs
    - BST -- if the keys are ordered
    - Hash Table -- if the keys can be converted to an integer -- need to handle collision

# Topics Covered

## Memory:

- Variables and types
- Array
- Types
- Pointers
- Pass by reference
- Function frames
- Stack and Heap

## Data structure:

- Array List
- Linked List
- Tree & BST
- Hash Table
- Max Heap
- Selection, insertion, bubble sort
- Tree sort, heap sort,
- Counting sort

## Bits:

- Bitwise operations
- Integer representation
- Bit-packing
- Masks
- Binary and hex
- Endianness

## Other:

- Threads
- Virtual memory
- Dynamic dispatch
- Terminal
- Git
- Compiler
- Makefile
- Valgrind
- Machine structure



# Review

## C

- All operators:
  - Unary: `!x` `~x` `-x` `x++` `++x` `x--` `--x`
  - Binary:
    - Arithmetic: `x + y`, `x - y`, `x * y`, `x / y` (two kinds), `x % y`,
    - Comparison: `x == y`, `x != y`, `x > y`, `x < y`, `x >= y`, `x <= y`
    - Bitwise: `x & y`, `x | y`, `x ^ y`, `x << y`, `x >> y`

# Review

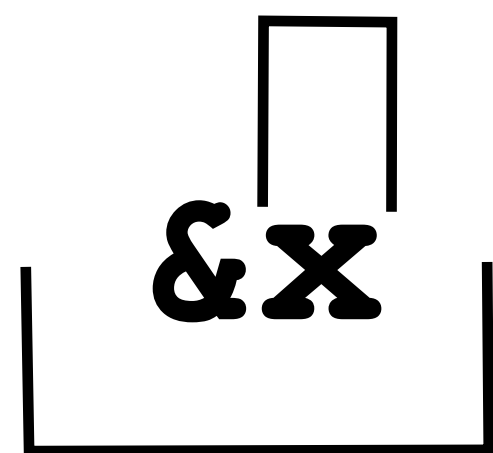
## C

- All operators: (Cont.)
  - Pointers:  $*a$ ,  $\&a$

# Pointers

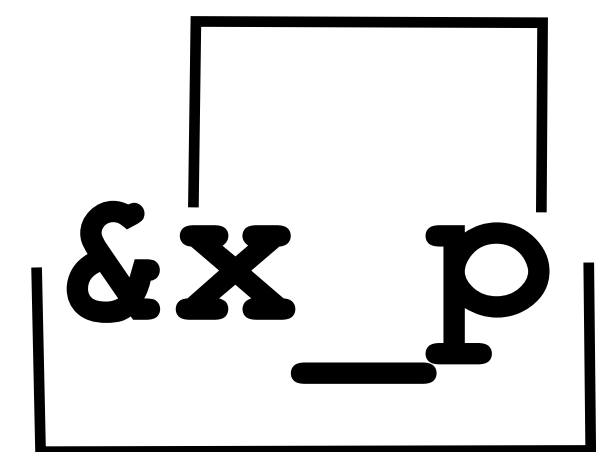
## Review

type : int  
value: 25



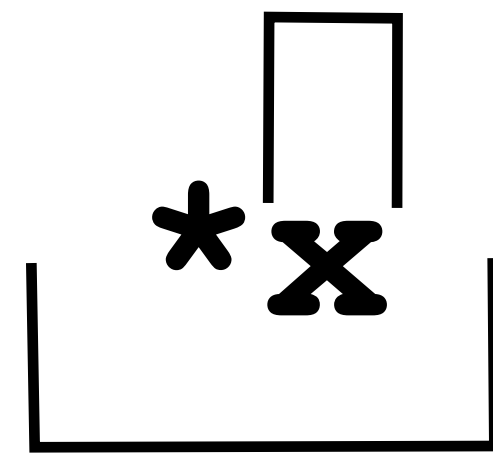
type : int \*  
value: 100

type : int \*  
value: 100



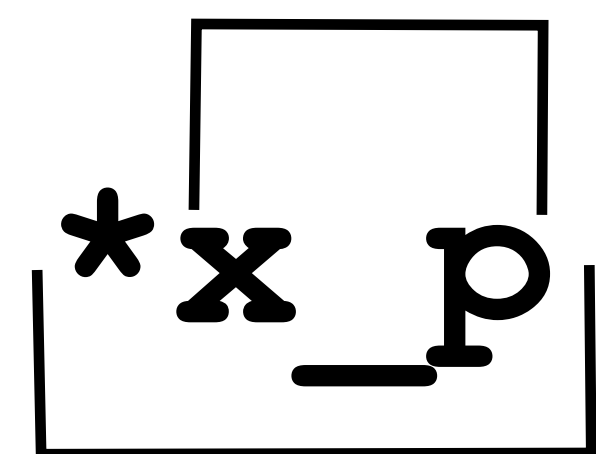
type : int \*\*  
value: 108

type : int  
value: 25

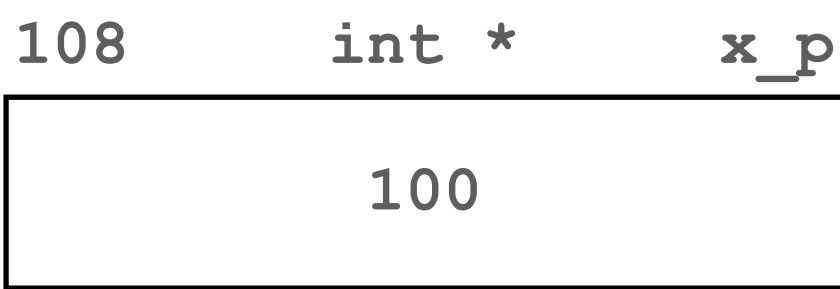
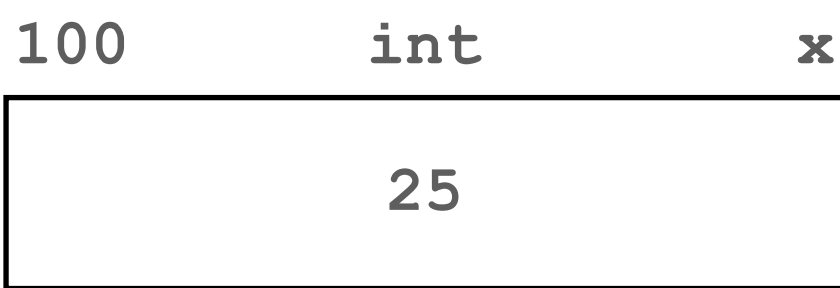


error

type : int \*  
value: 100



type : int  
value: 25



# Review

## C

- All operators: (Cont.)
  - Pointers: `*a`, `&a`
  - Subscript and member:
    - `a.field`
    - `a[b]` is a short hand of `*(a + b)`
    - `a->field` is a short hand of `(*a).field`

# Review

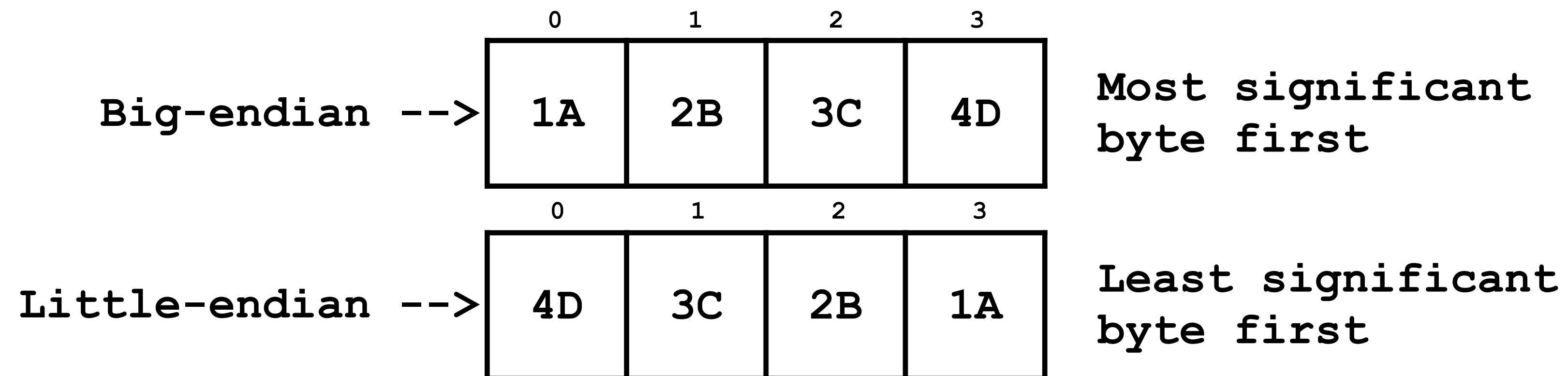
## C

- All operators: (Cont.)
  - Pointers: `*a`, `&a`
  - Subscript and member: `a.field`, `a[b]`, `a->field`
  - Ternary conditional: `a ? b : c` (In Python: `b if a else c`)
  - Type cast: `(type) x`

# Bits

## Endian

- We think of an integer as one atomic value:
  - `int x = 0x1A2B3C4D;`
- But if an integer has 4 bytes and each byte is addressable, which of the 4 bytes is stored first?



# Bits

## Endian

```
int main(void)
{
    int x = 0x1A2B3C4D;
    char *ptr = (char *) &x;

    for (int i = 0; i < 4; ++i) {
        printf("0x%hhx\n", ptr[i]);
    }

    return 0;
}
```

# Review

## Function Frames

- When a function returns, we can recycle the memory used by the variables declared inside the function.
  - Variables declared in { . . . } can only be accessed in { . . . } (Scope)
- Local variables and arguments live in a *frame*.



# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```

---

# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}
```

```
→ int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```

main

b: ??

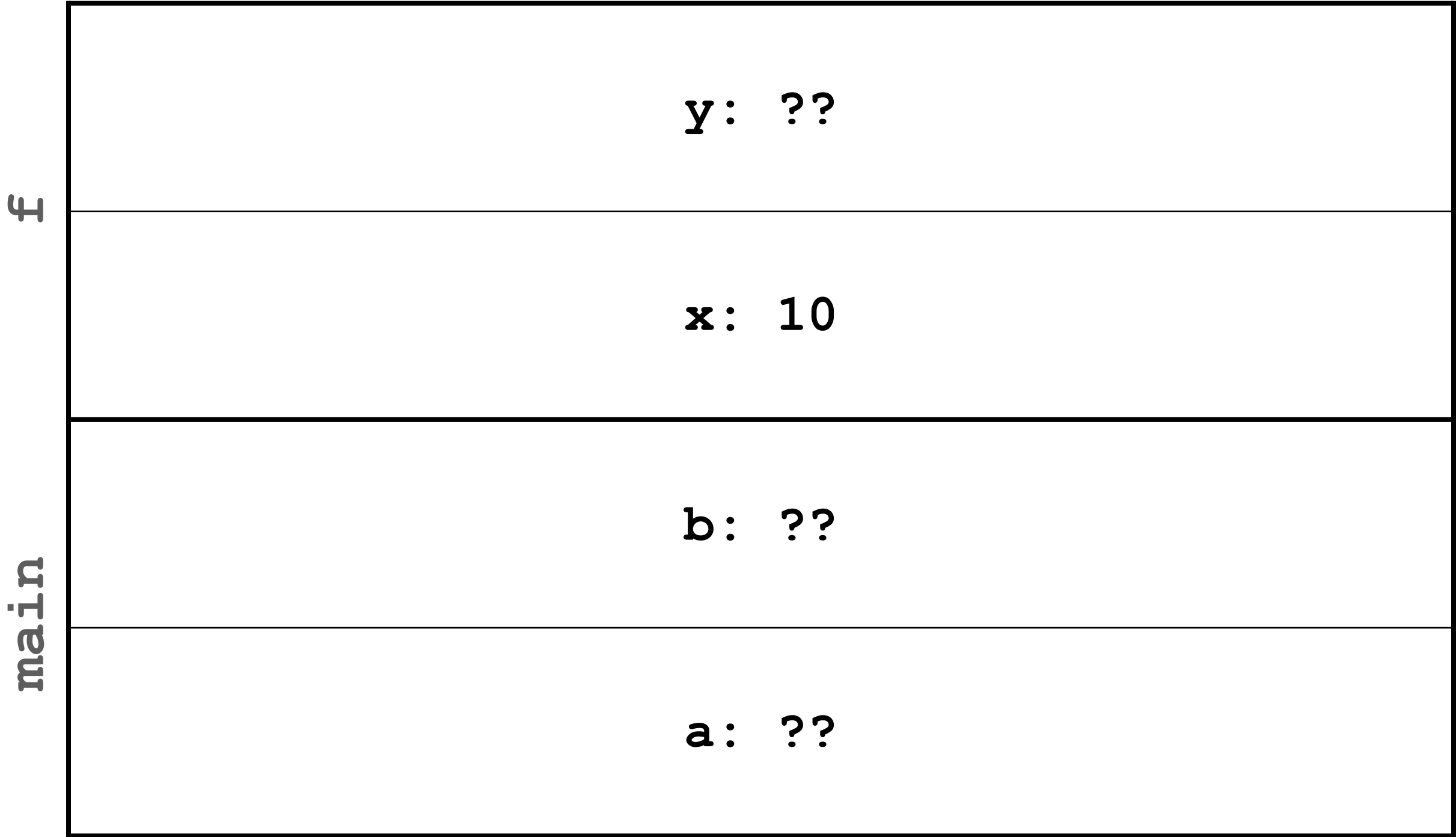
a: ??

# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```

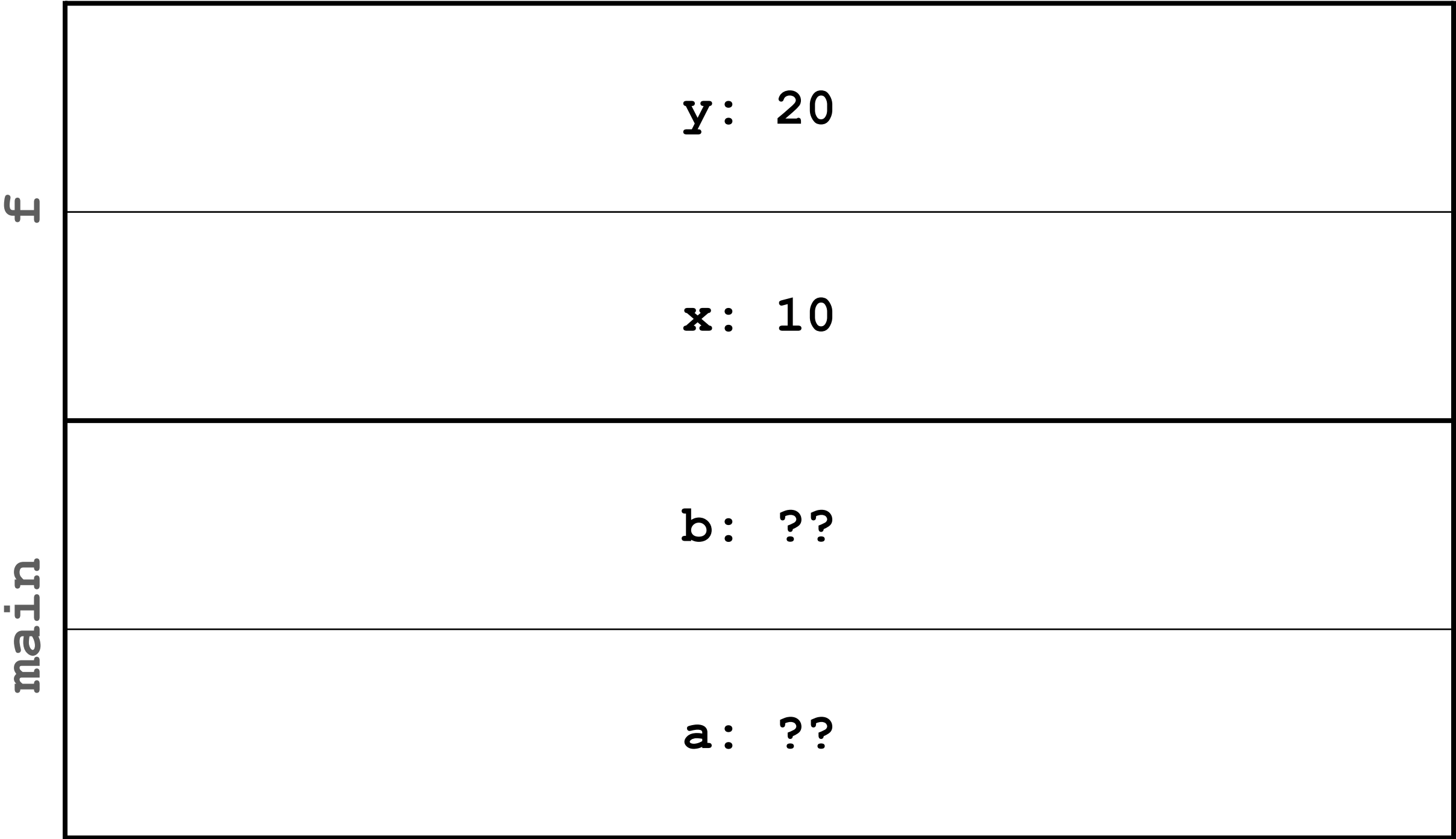


# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```

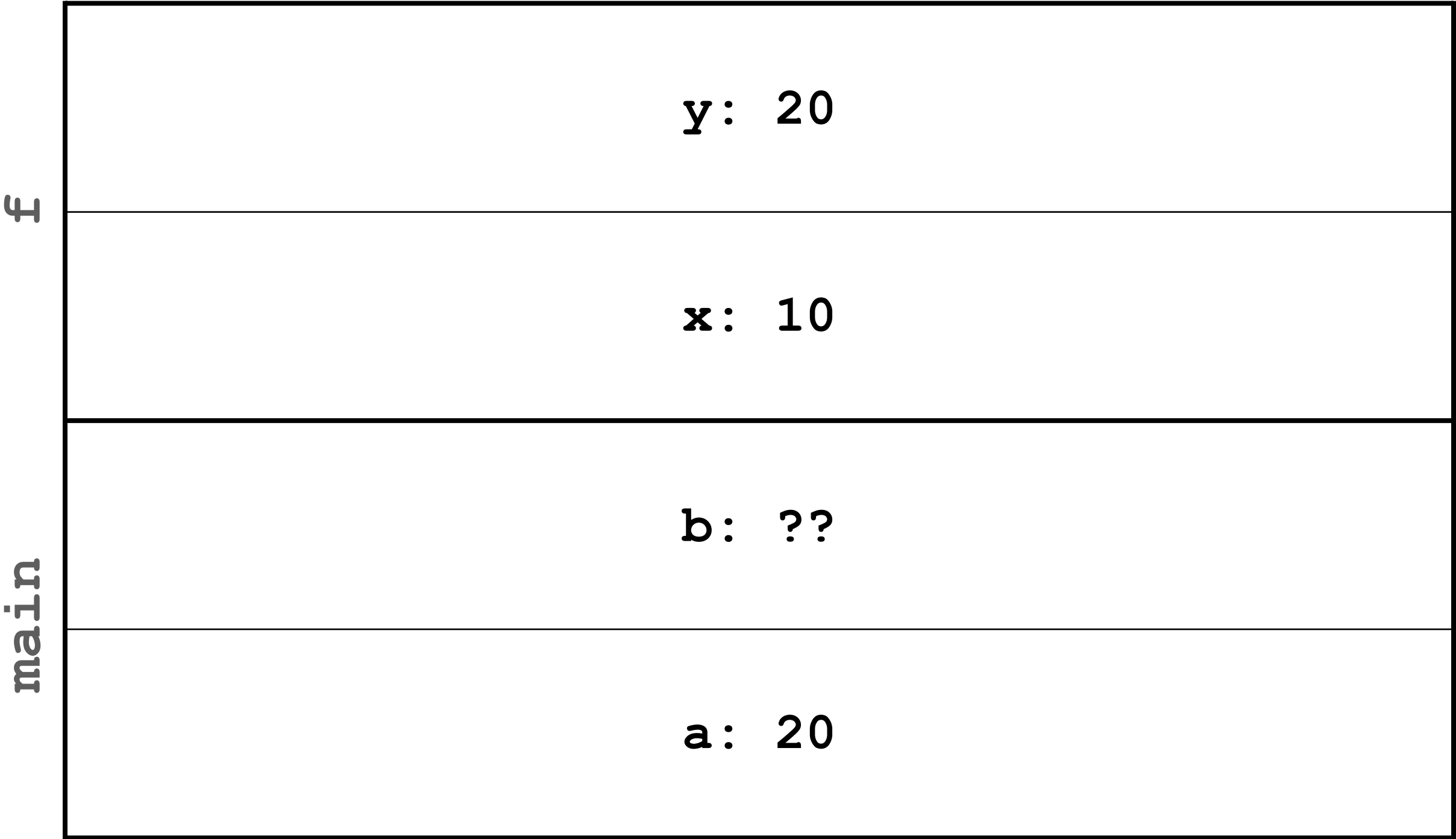


# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```



# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```



main

b: ??

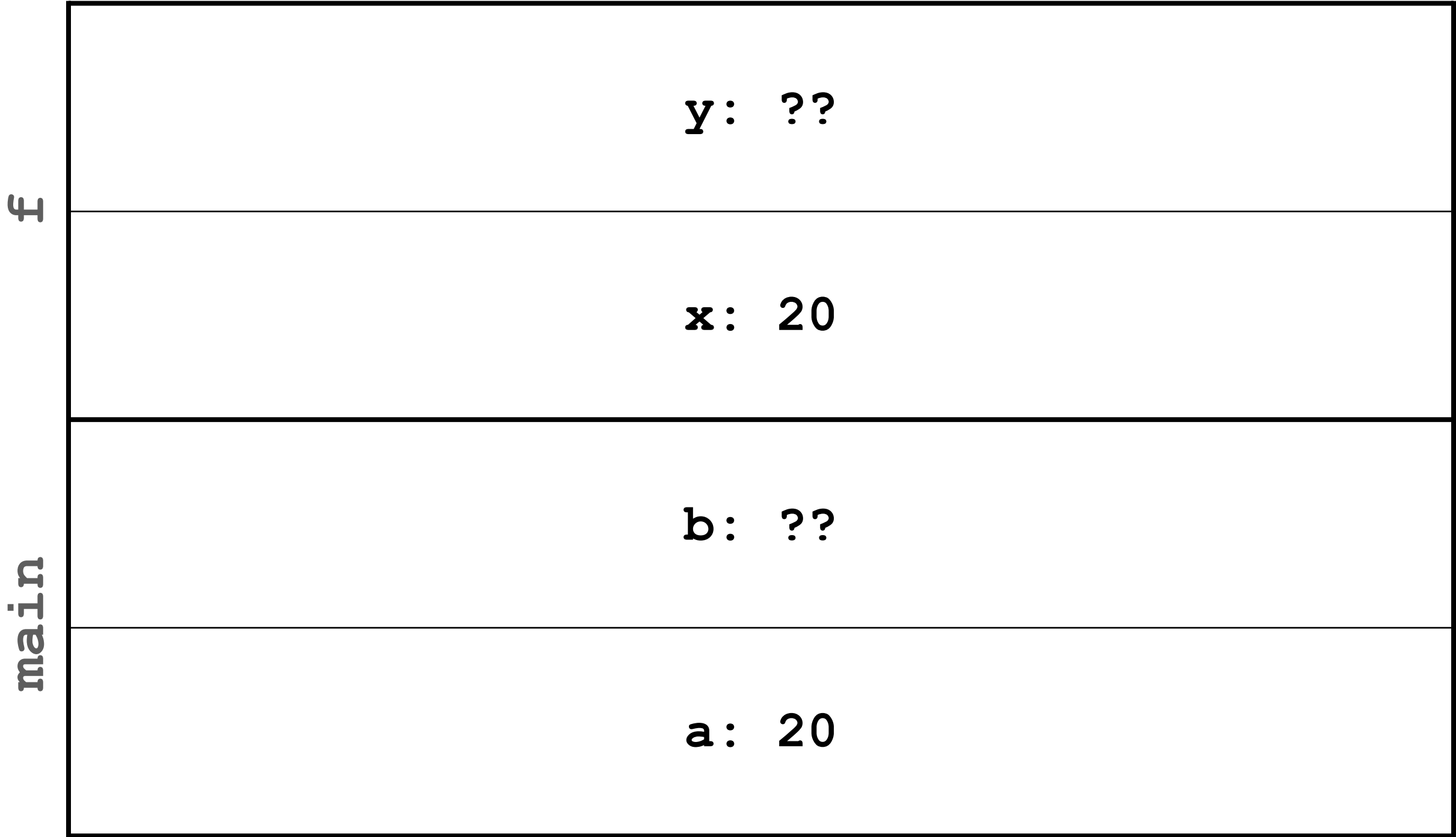
a: 20

# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```



# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```

|      |       |
|------|-------|
| f    | y: 40 |
|      | x: 20 |
| main | b: ?? |
|      | a: 20 |

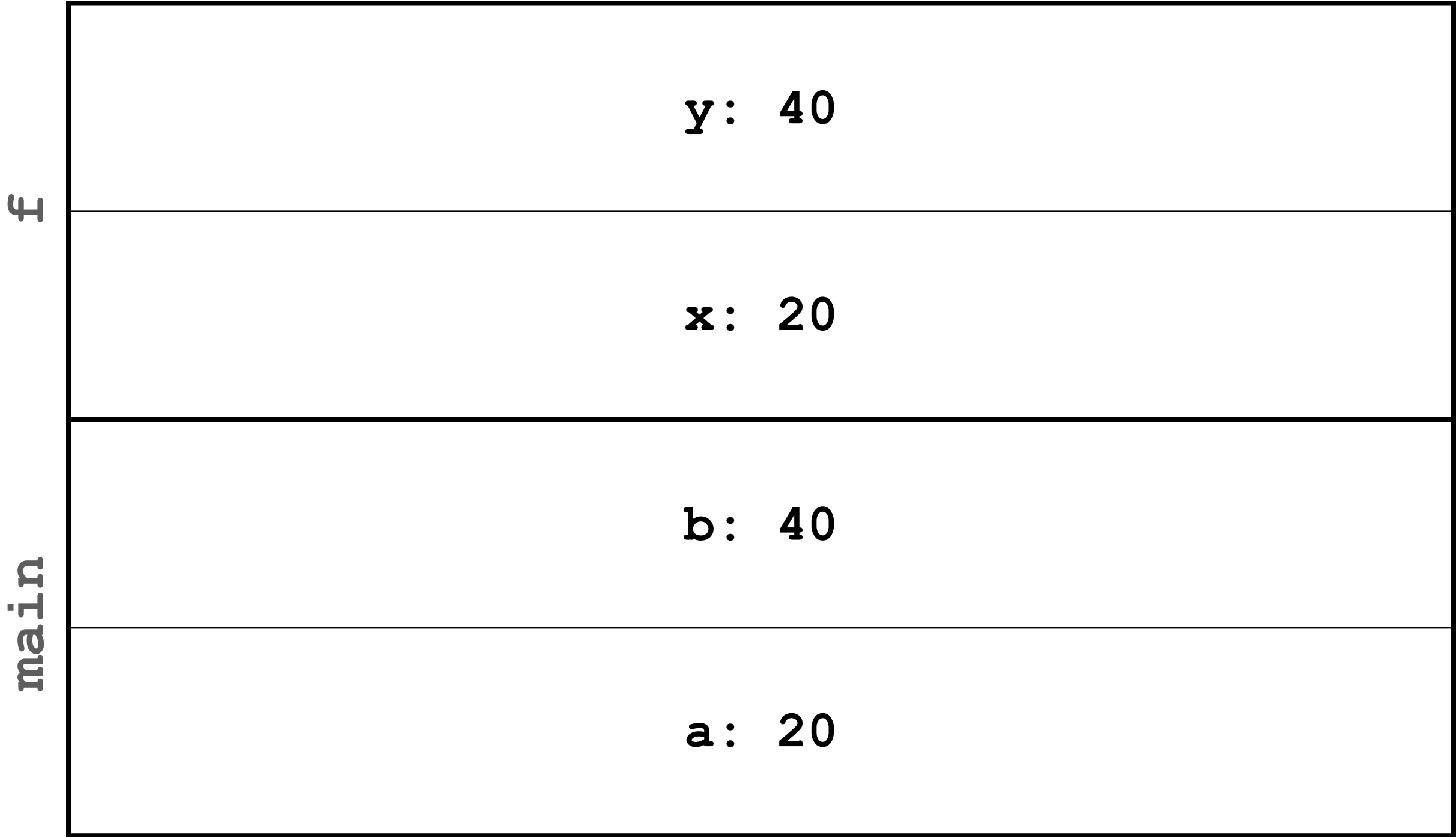


# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```



# Variable Lifetime

```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```



main

b: 40

a: 20

# Variable Lifetime

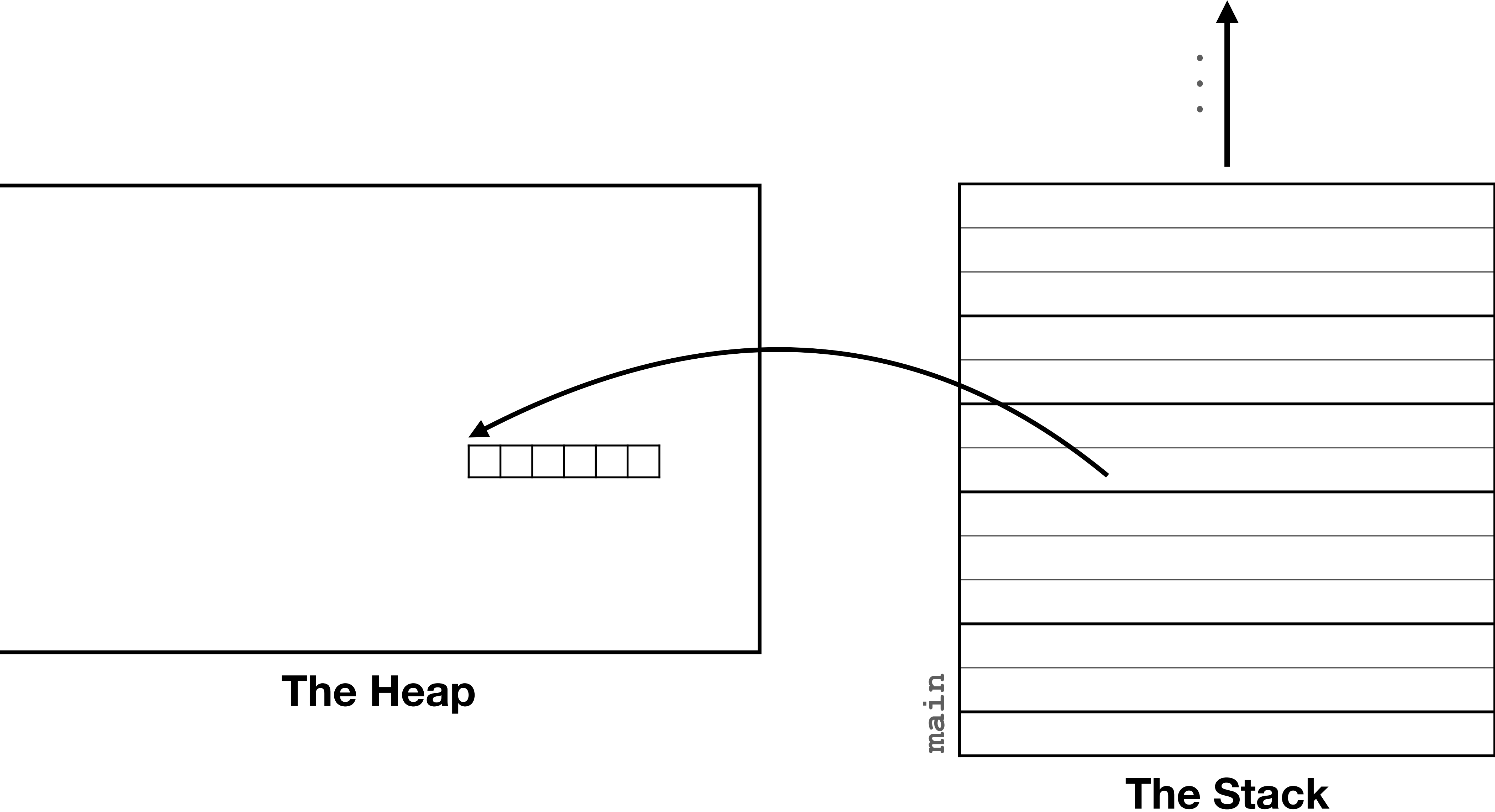
```
int f(int x)
{
    int y = x * 2;
    return y;
}

int main(void)
{
    int a = f(10);
    int b = f(a);
    printf("%d\n", b);

    return 0;
}
```



# The Heap



# The Heap

## Stack vs Heap

### Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing
  - You can't forget to release

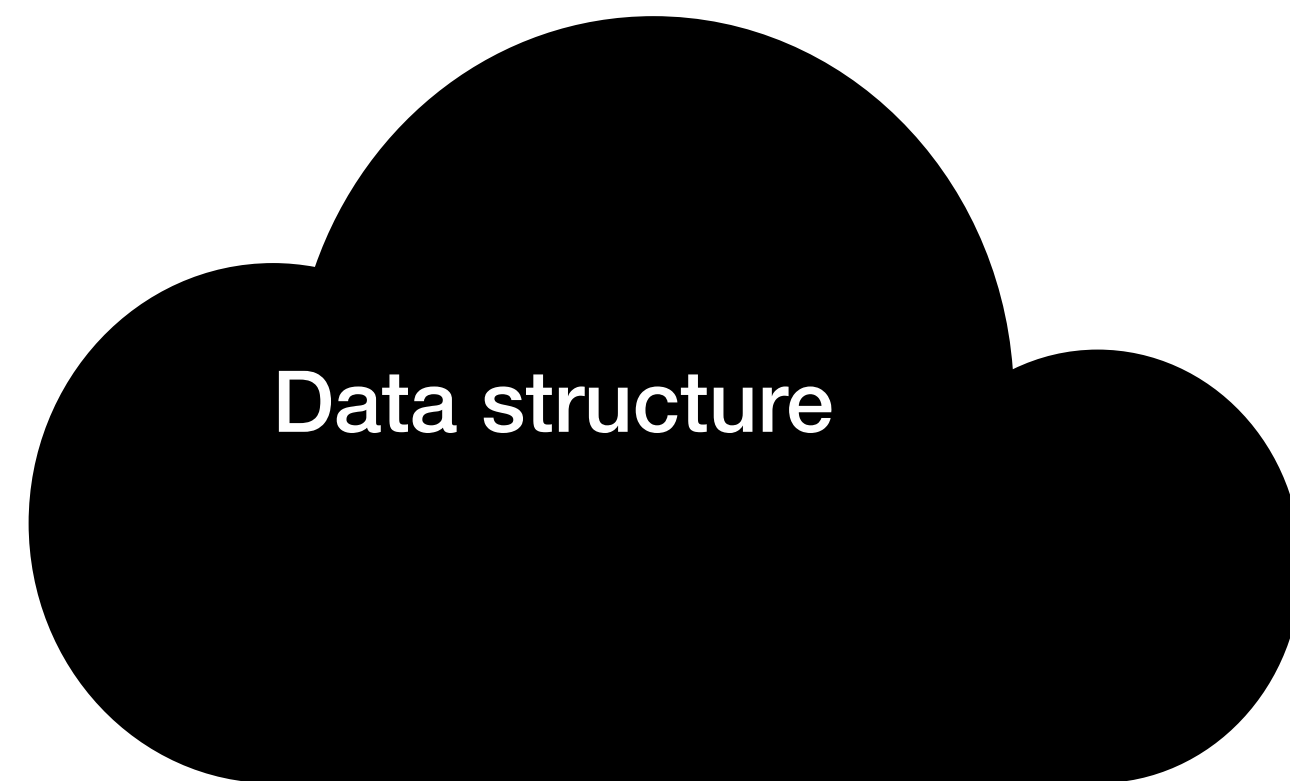
### Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:
  - `free(ptr)`
  - You can forget to release;  
*memory leak*

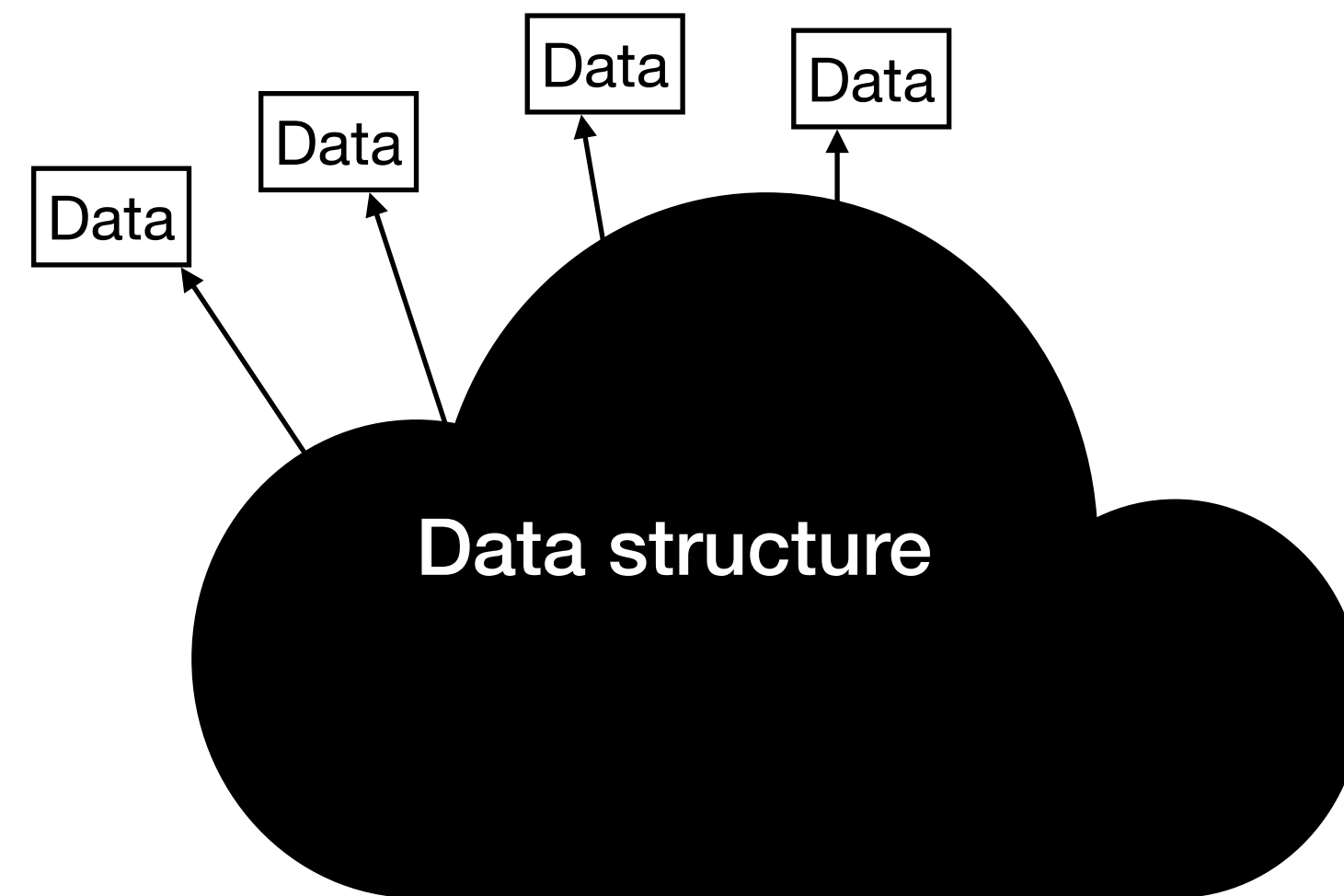
- Accessing released memory is bad;  
*memory error*

# Data Structures

Week 4 onwards



Unboxed



Boxed

- **Boxed:** Nodes store pointers to client-managed data. (Polymorphic)
- **Unboxed:** Data would be stored directly in the nodes. (Faster access)

# Data Structures

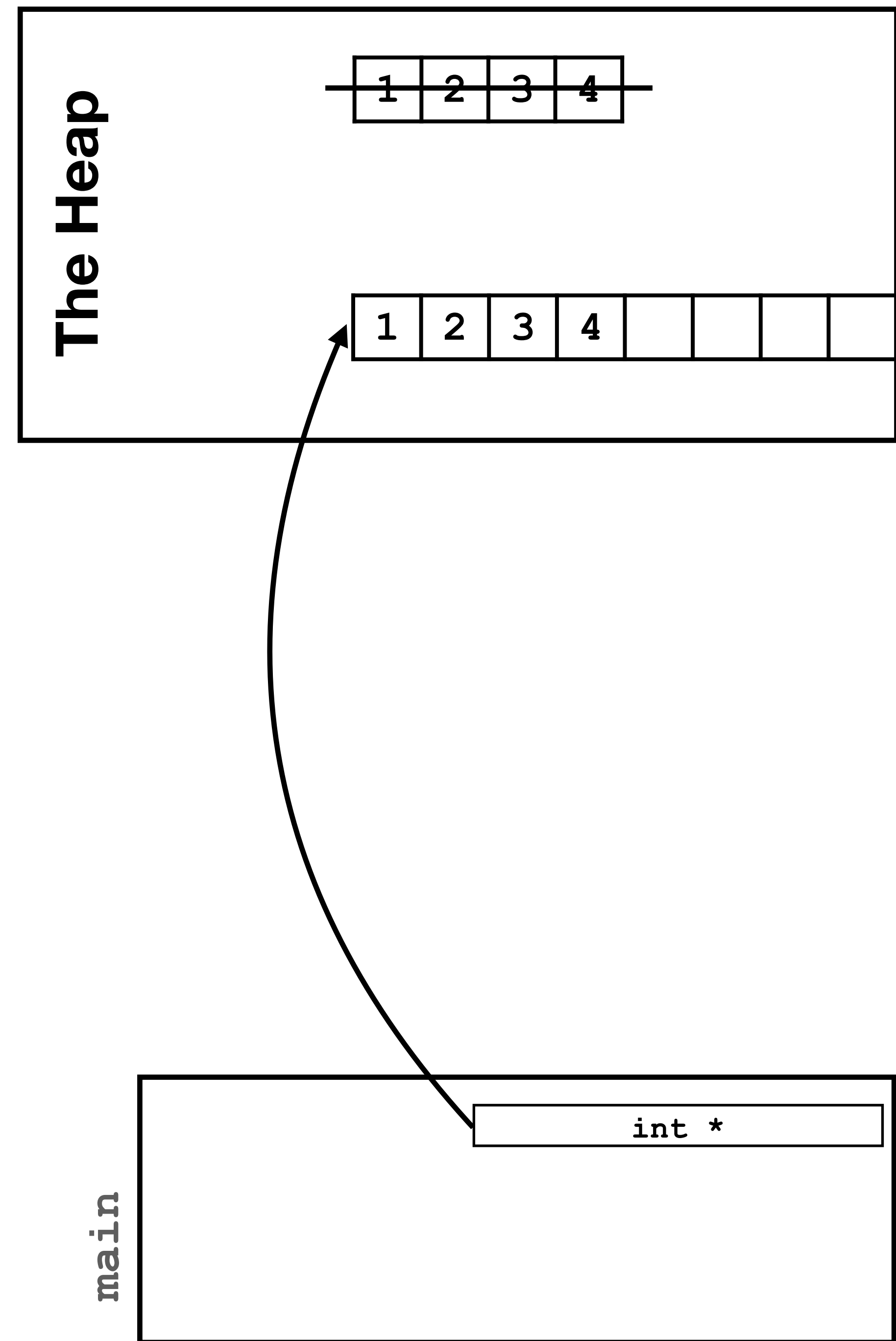
- Establishing structures on the heap:
  - Indices: contiguous
    - $O(1)$  random access
    - difficult to reorder and reallocate
  - Pointer: scattered
    - sequential access
    - easy to reorder and reallocate

|      | Indices    | Pointers    |
|------|------------|-------------|
| List | Array List | Linked List |
| Map  | Hash Table | BST         |

# Array

## Growing an array

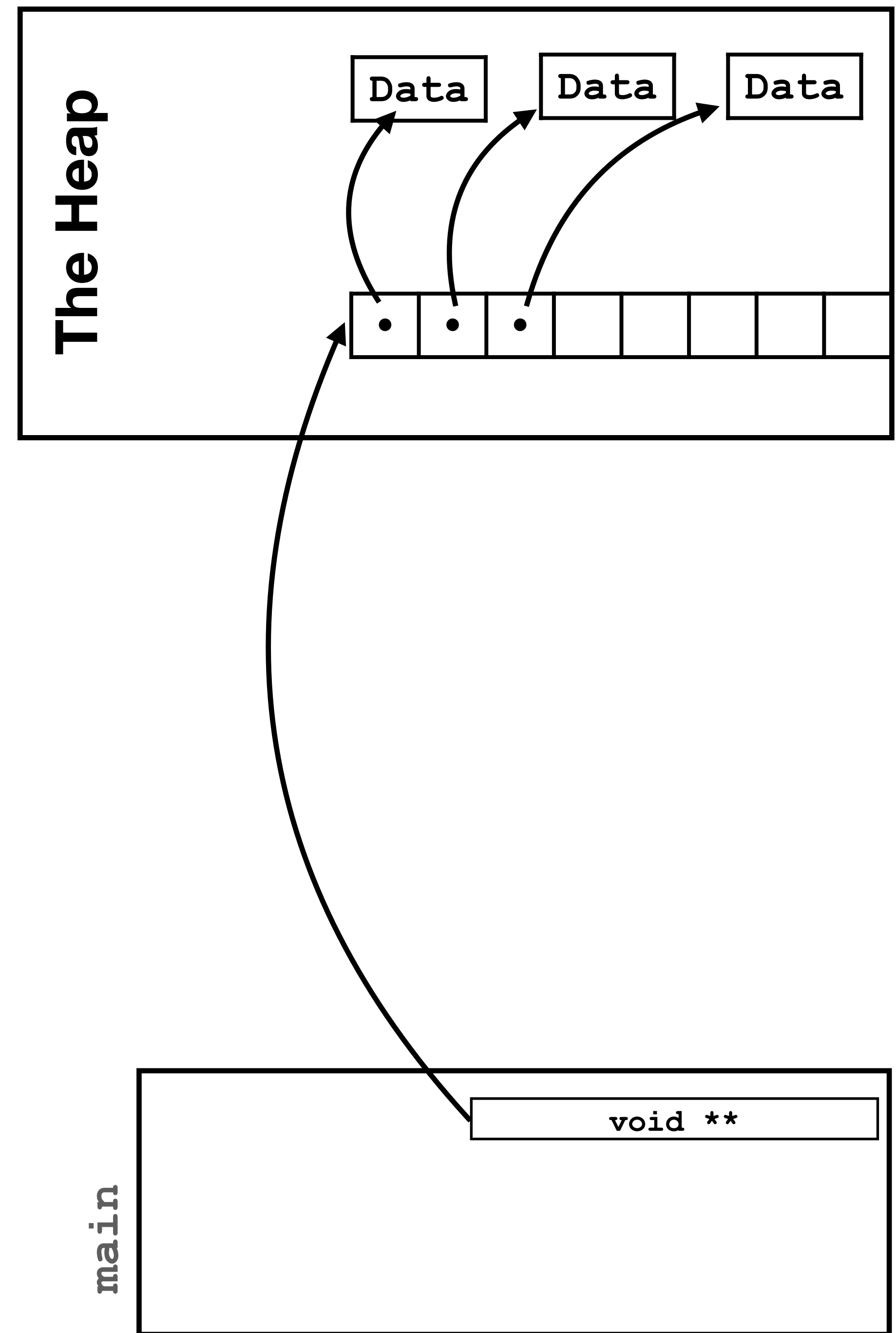
- Pointers serve as an indirection.
  - We aren't changing the size of the array; we are changing which array the pointers point to.
  - By changing the address of the pointer, it seems to the user that we have changed the size of the array.
- We create and delete memory however we want thanks to the heap.



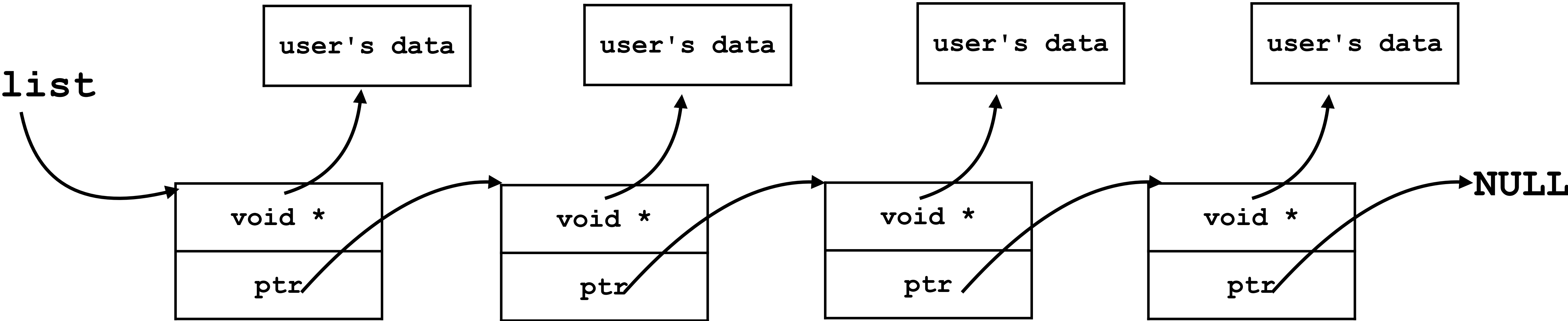


# Array

## Boxed Array

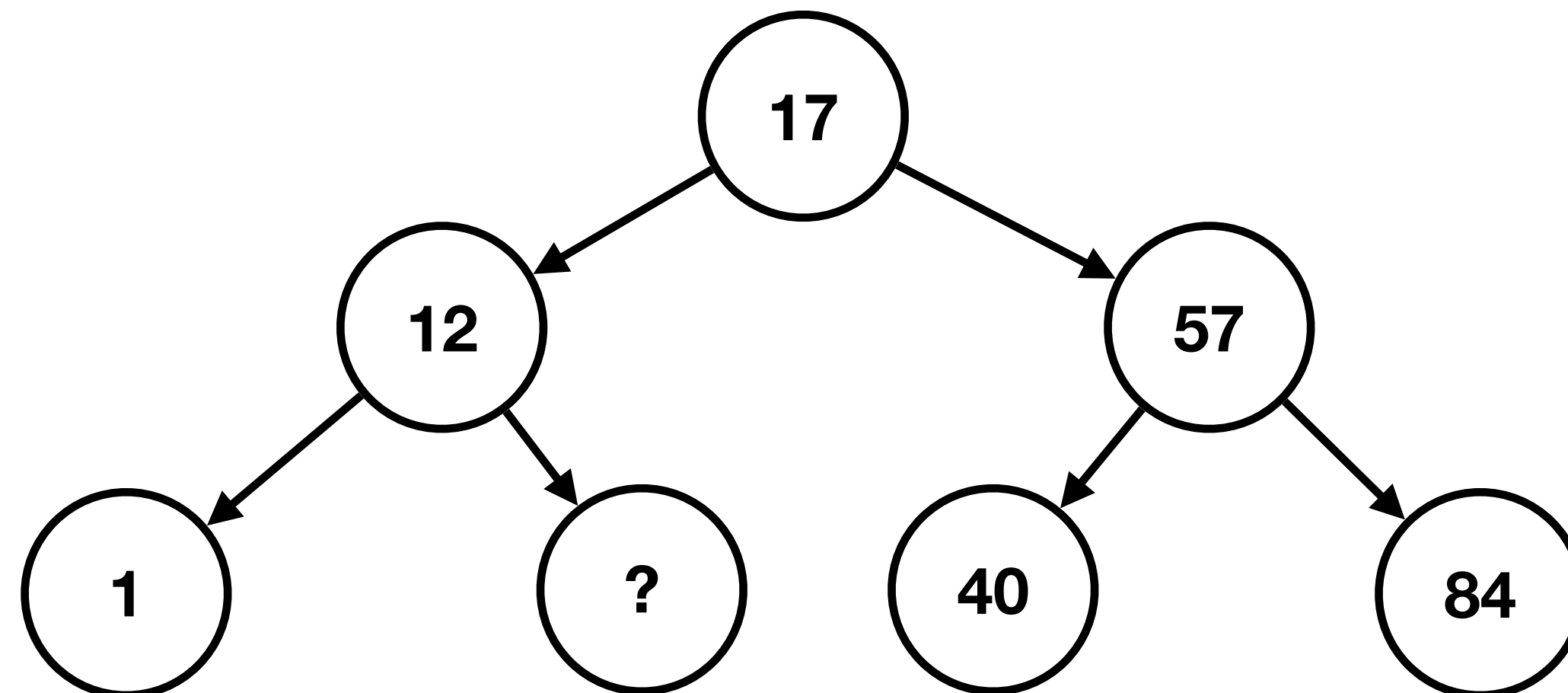


# Linked Lists



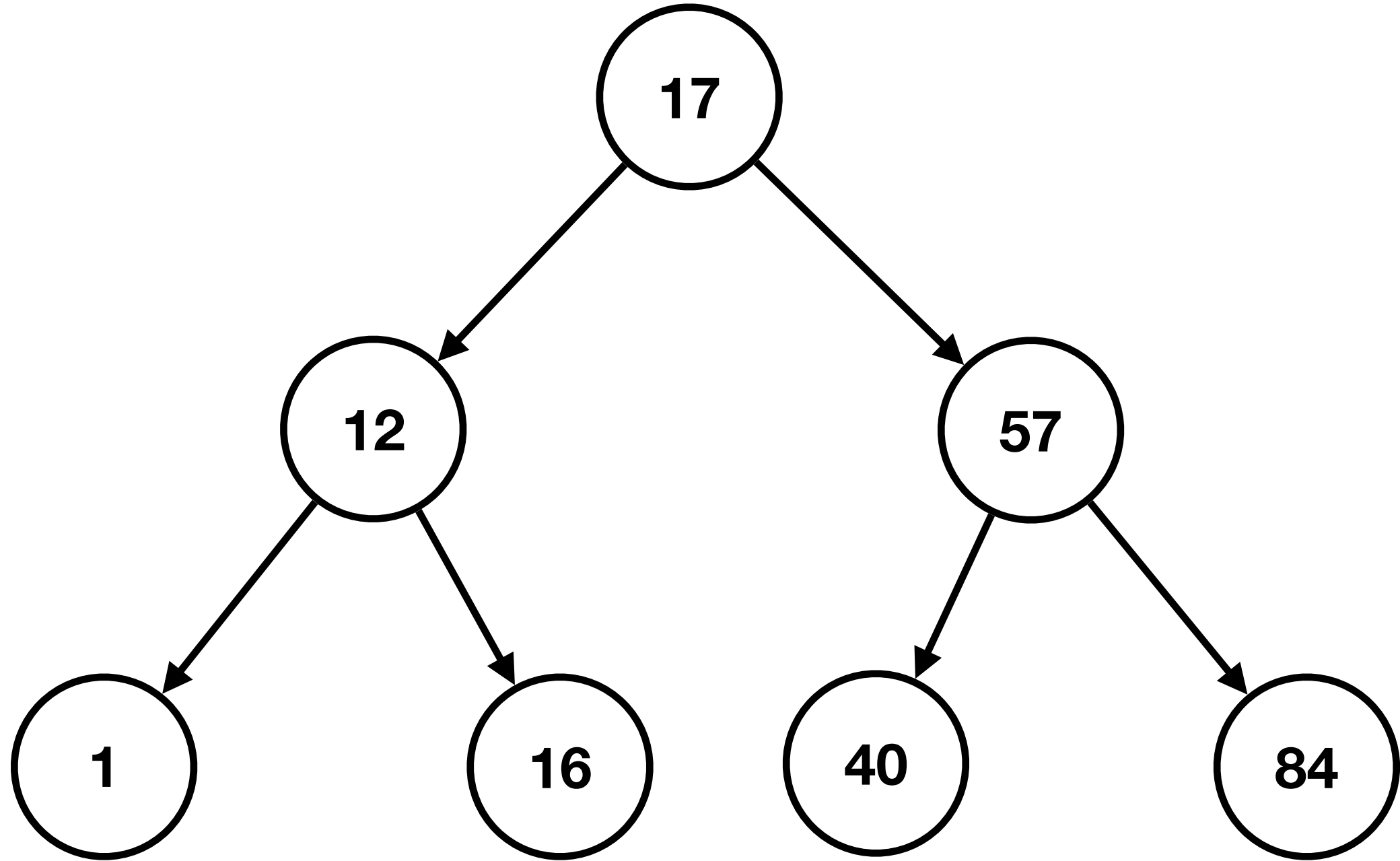
# Binary Search Tree

- A binary search tree is a binary tree where
- For a given node  $n$  with key  $k$ ,
  - All nodes with keys less than  $k$  are in  $n$ 's left subtree.
  - All nodes with keys greater than  $k$  are in  $n$ 's right subtree.

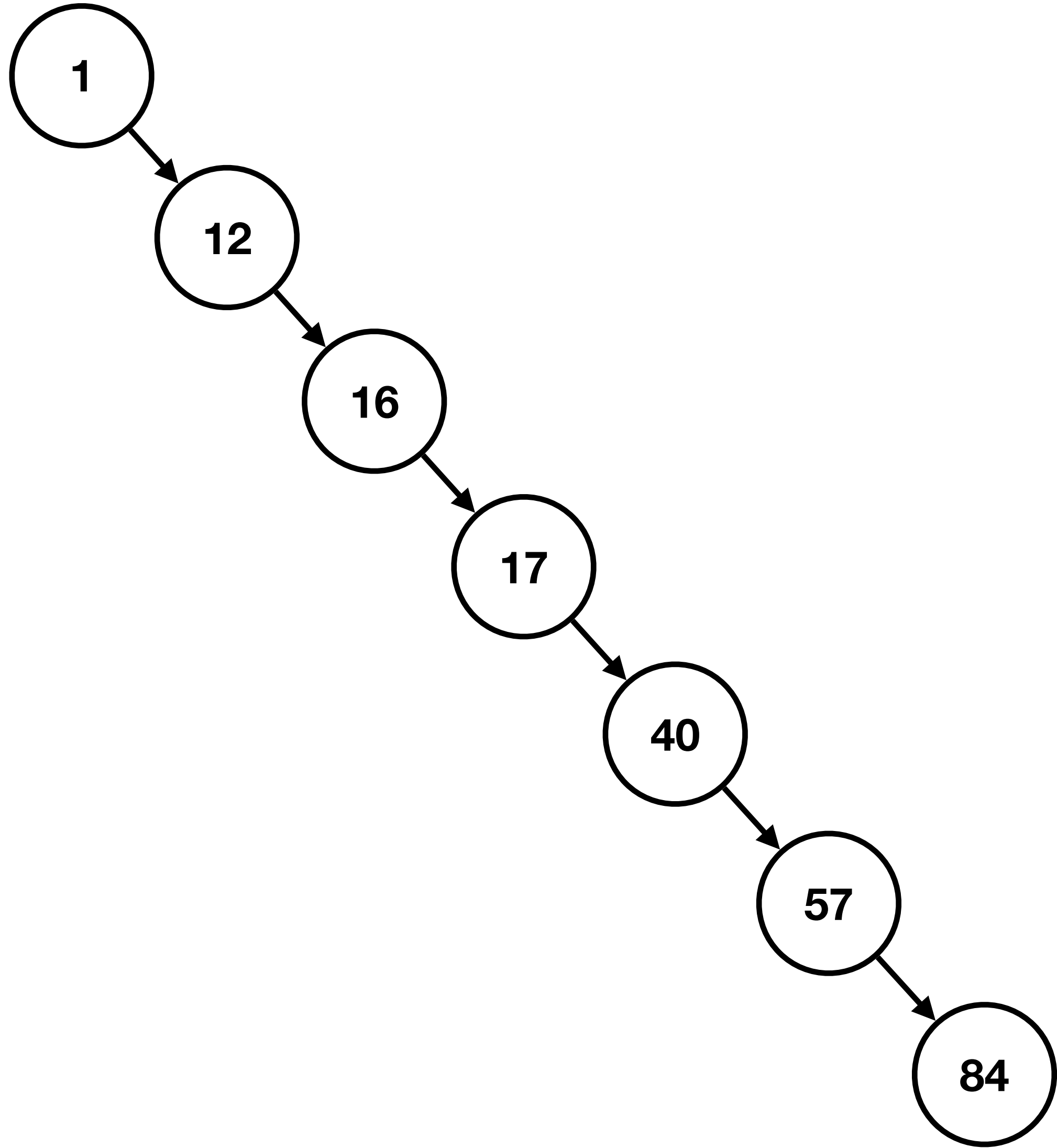


# BST

## Height



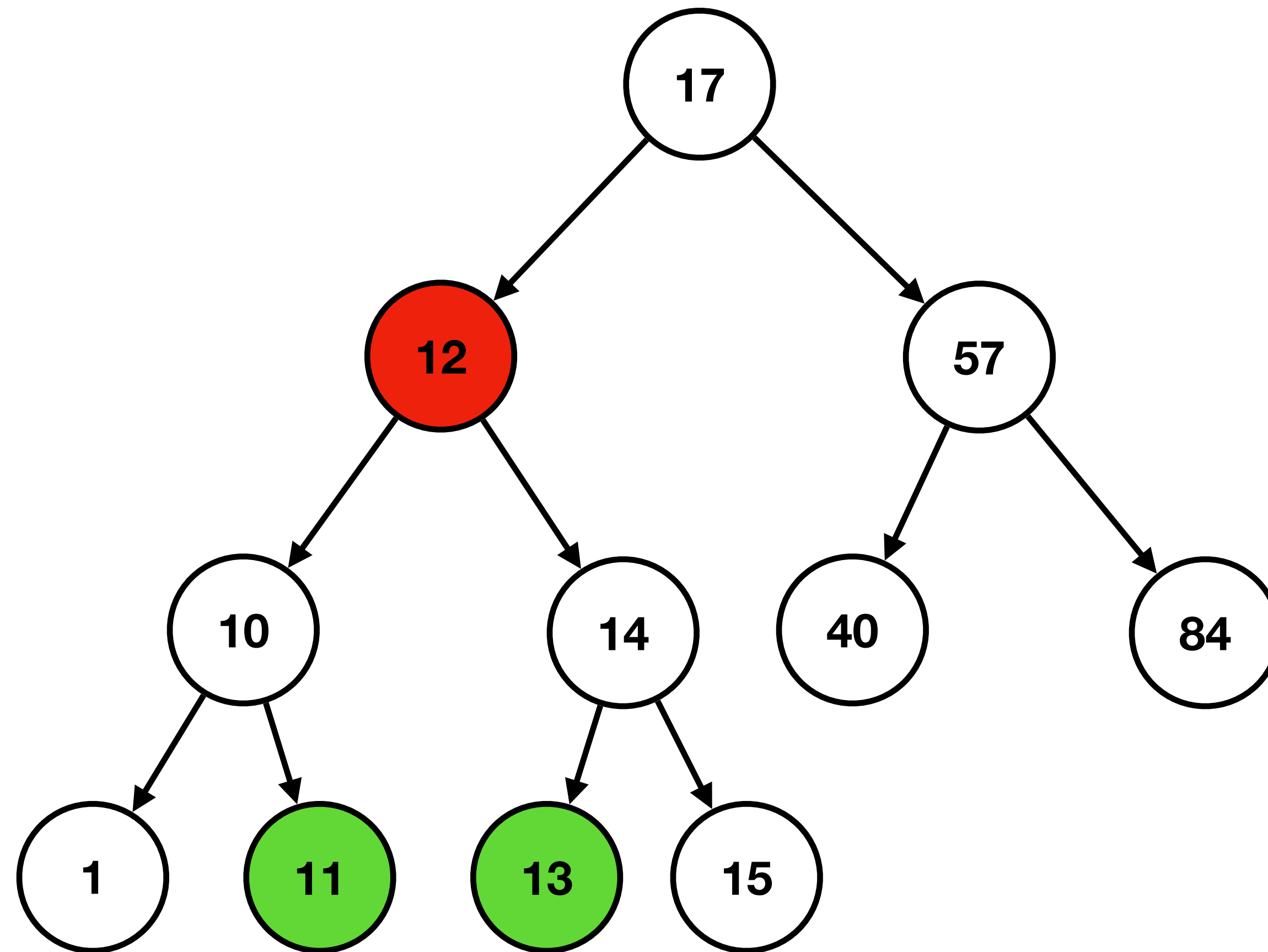
Balanced



Unbalanced

# BST

## Remove



# Hash Table

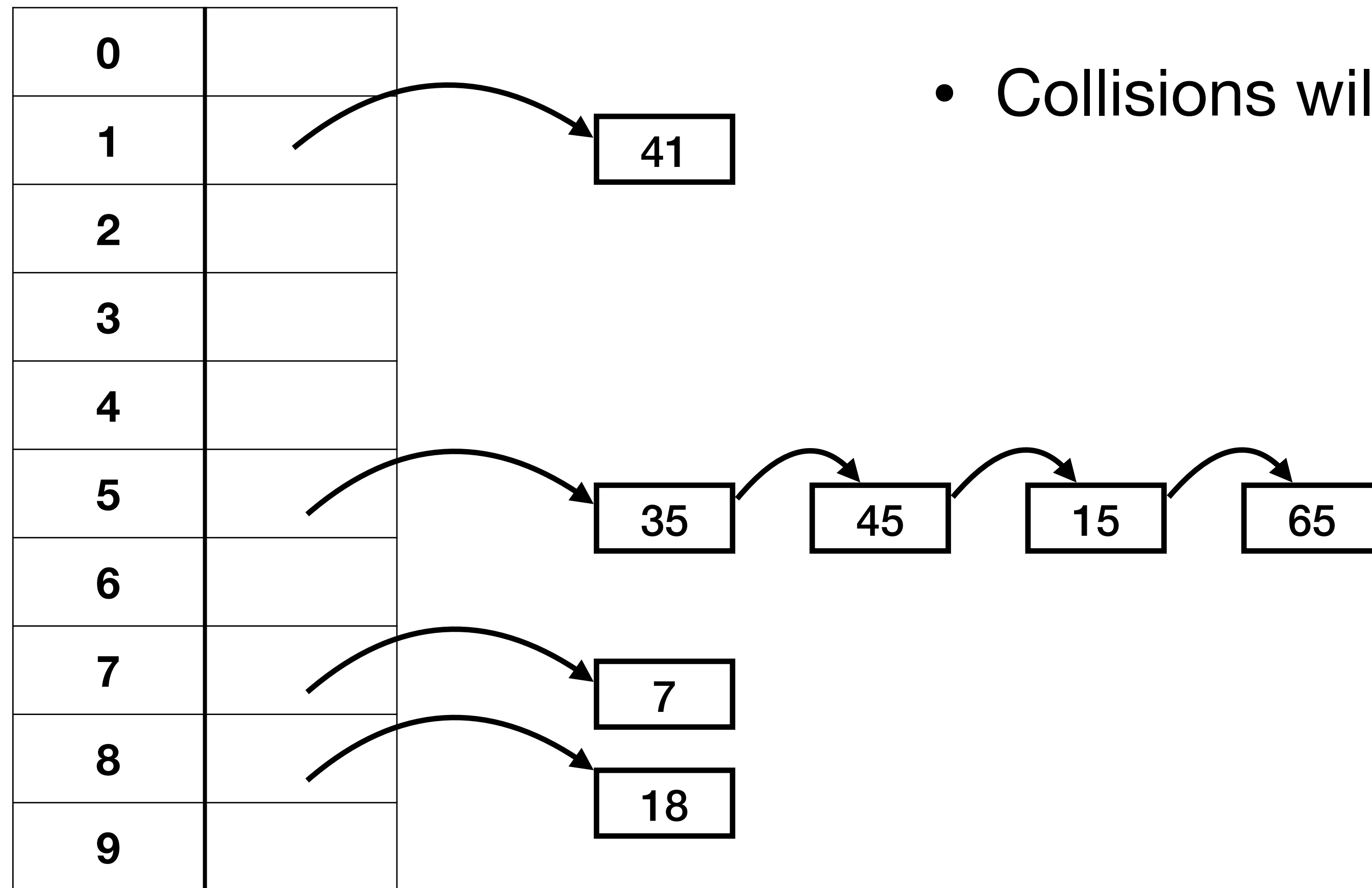
## Review

- Nice  $O(1)$  complexity because we can index into an array instead of chasing pointers
- We have a way to turn anything into an integer -- hash function
- We have a way to force any integers into a reasonable range -- compression (usually modulus)
- We need to handle collisions:
  - Collisions can be the result of the hash function
  - ... of compression

# Hash Table

## Chaining

- Each slot is a *list* of key-value pairs, called a *bucket*




- Collisions will be prepended into the list

# Hash Table

## Linear probing

true when  
previously occupied

```
struct bucket {  
    bool removed;  
    void *key;  
    void *value;  
};
```

|   |   |
|---|---|
| 0 |   |
| 1 |   |
| 2 |   |
| 3 | ("bob", 30)   |
| 4 | ("carl", 50)  |
| 5 |  |
| 6 | ("eve", 100)  |
| 7 | ("david", 60)   |
| 8 |   |
| 9 |   |

- Find/Remove:
  - Move down until first empty bucket
  - If tombstone is encountered, continue searching
- Insert:
  - Move down until first empty bucket
  - If tombstone is encountered, we can reuse that bucket
  - But to avoid inserting duplicate keys, we need to continue searching until an unremoved bucket



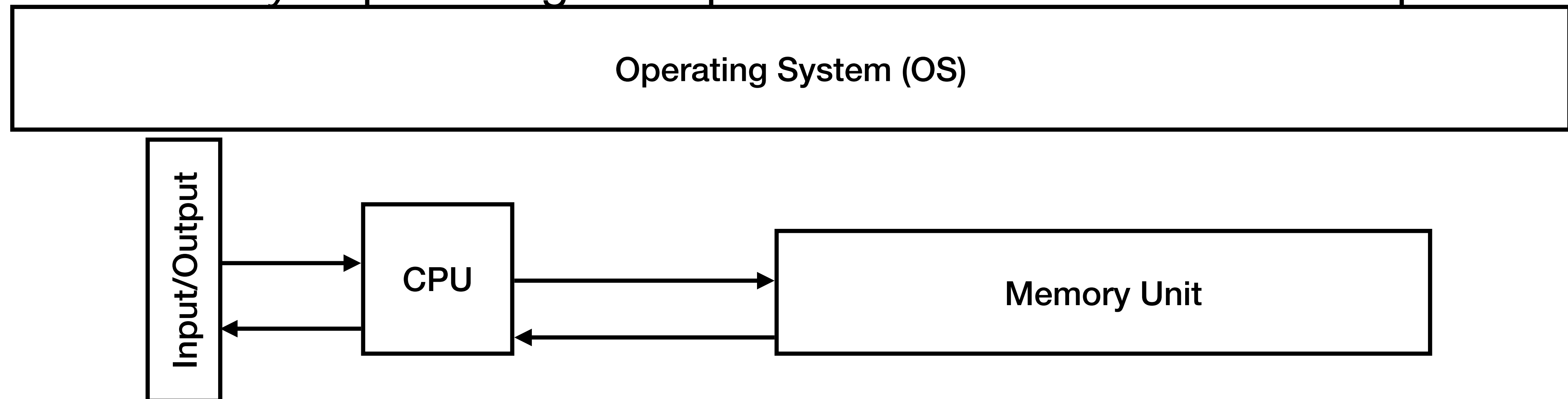
# Sorting

- $O(n^2)$ : Selection, Insertion, Bubble
- $O(n \log n)$ : Tree, Merge, Quick
- $O(n \log n)$  without extra space (not even a stack): Heap sort
  - Heap sort is "selection sort with the right data structure."
- <https://github.com/uchicago-cmsc14300-smr24/starter/blob/main/sort/sort.c>

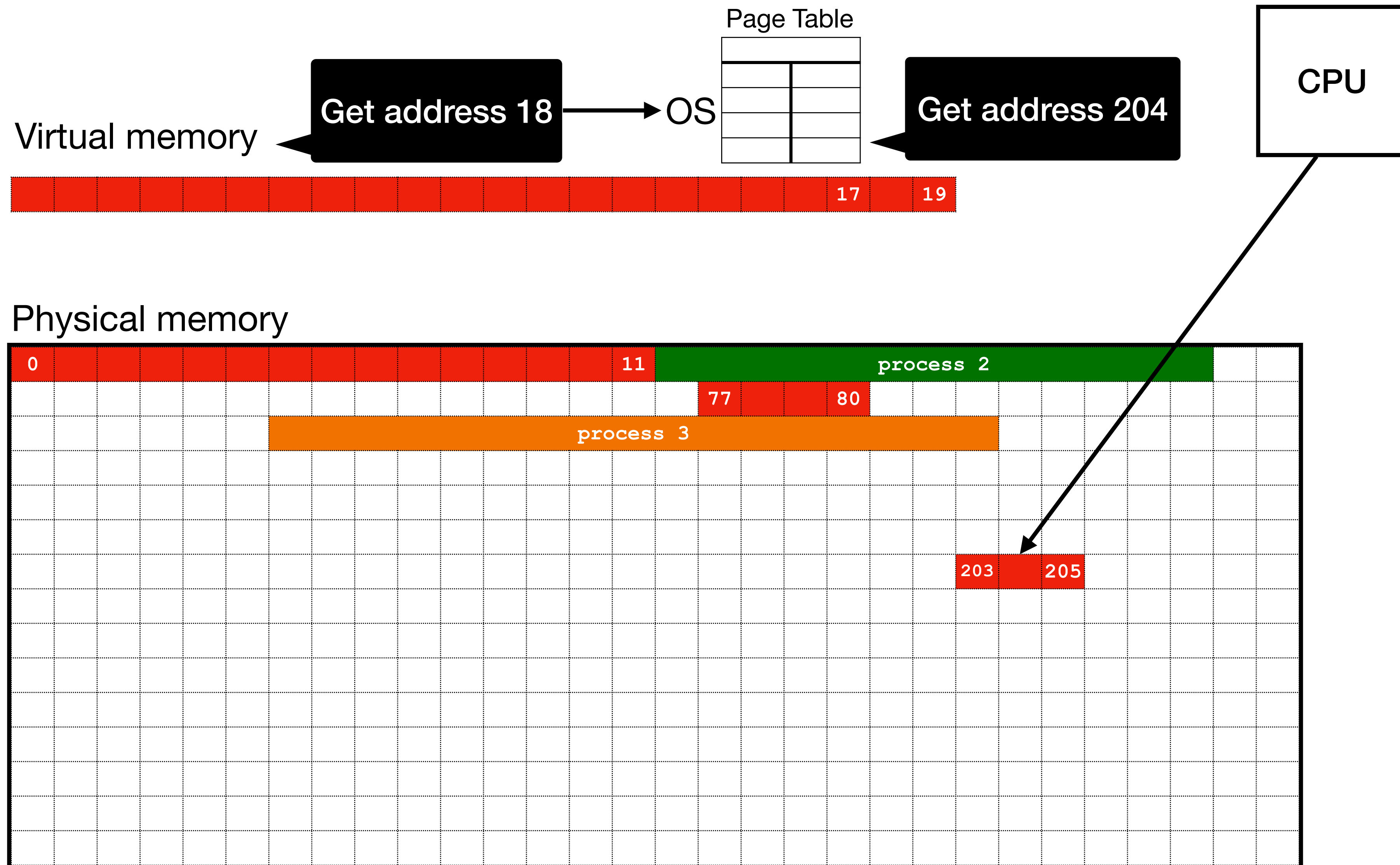
# Machine

**Your computer can do many things at the same time...**

- The operating system creates an illusion that each process is running by itself by:
  - *Context switching* -- rapidly switching which process has control over the CPU
  - *Virtual memory* -- providing each process with its own address space



# Virtual Memory

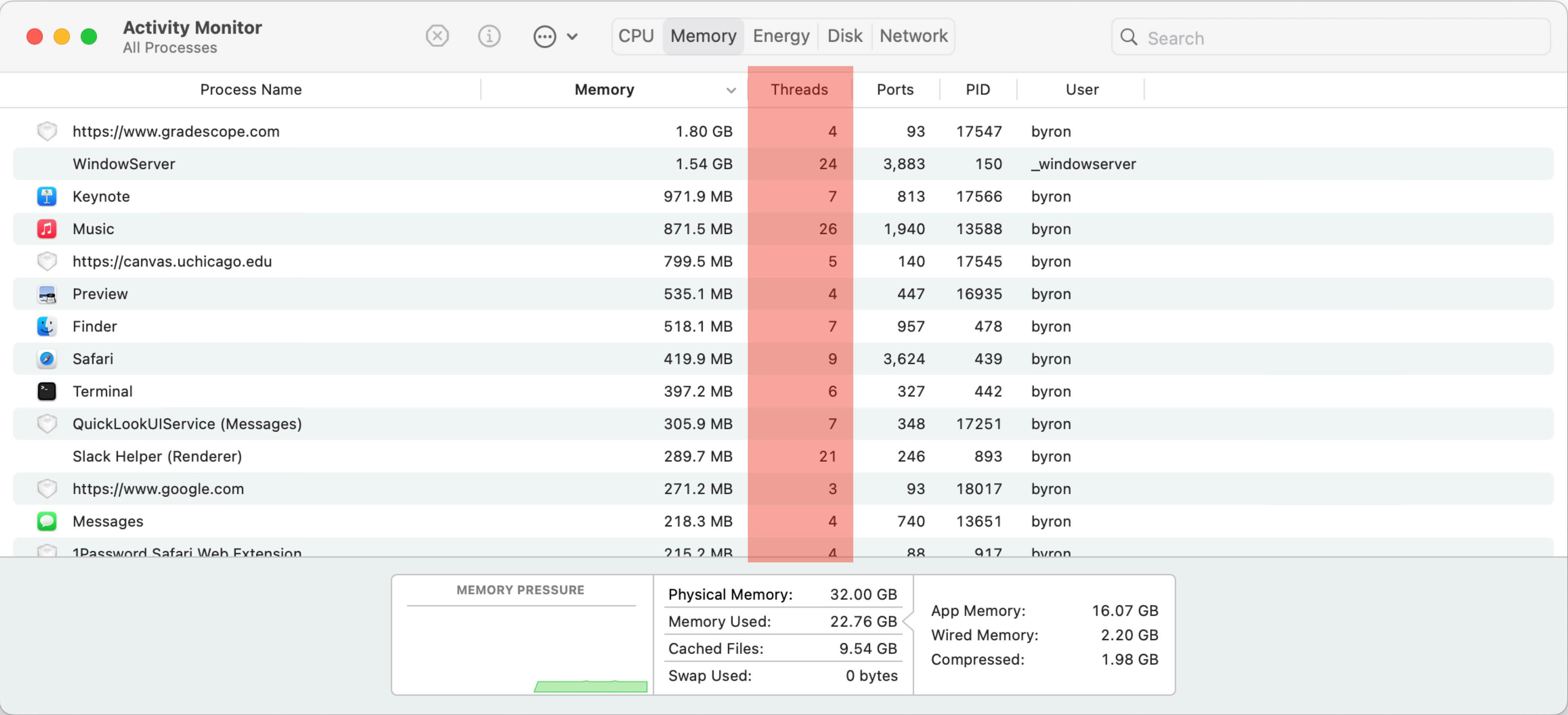


- CPU can do this translation very efficiently
- The chunks of memory used to be called *segments*.
- segmentation fault!

# Context Switching

- Each process has its own
  - Virtual memory
  - Registers
  - Program counter
  - ...
- OS keeps track of these data in its internal data structure.

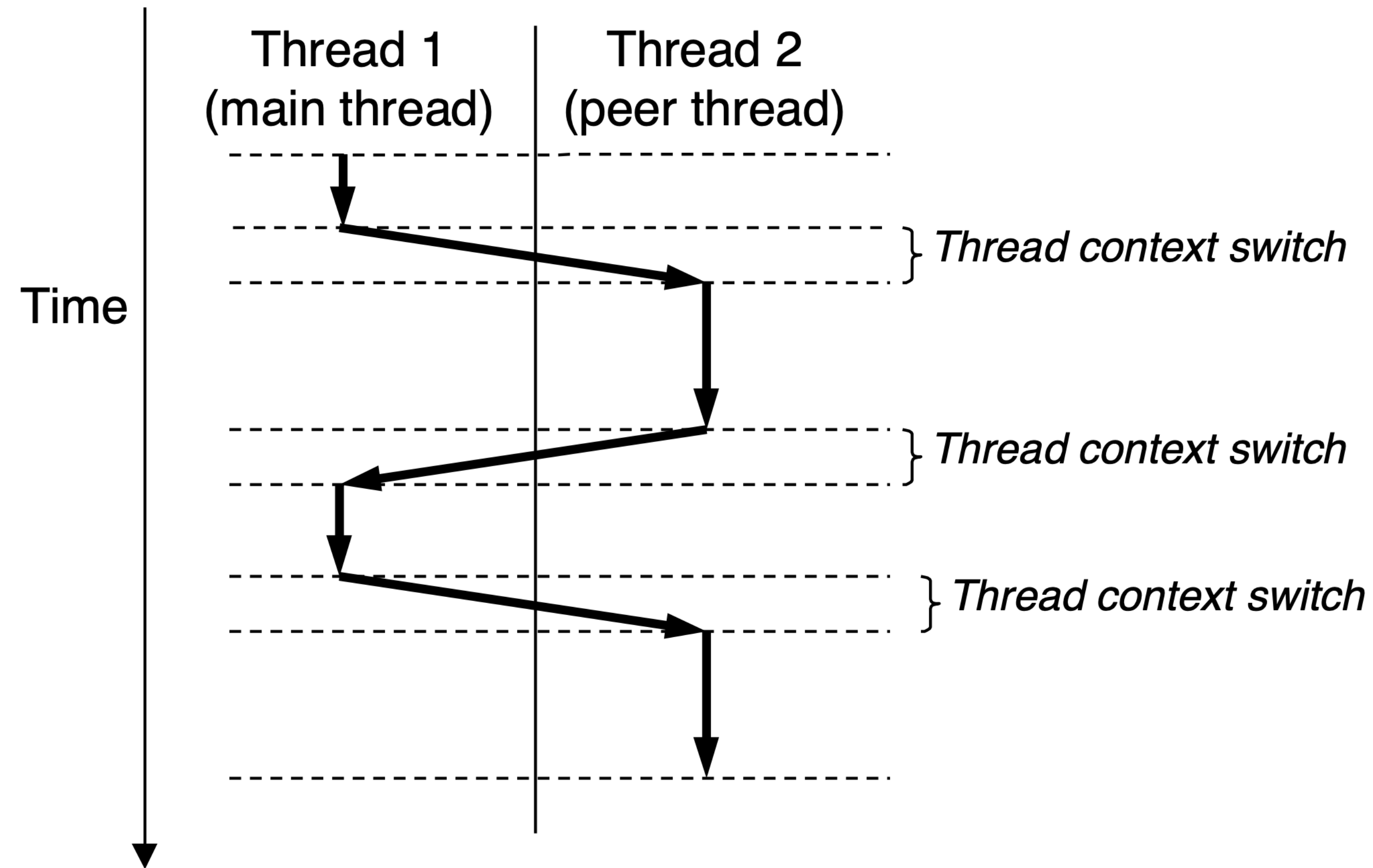
# Threads



# Threads

- A thread is a unit of execution. Each thread has its own:
  - Thread ID
  - Stack
  - Program counter (pc)
  - Registers
- A process contains a number of threads. Threads within a process share:
  - Code, data
- Threads are executed *concurrently*.

# Threads





# What next?

- Data structure, complexity, sorting:
  - CMSC 27200. Theory of Algorithms
- File, permanent storage, bits:
  - CMSC 23500. Introduction to Database Systems
- Memory, instructions, language:
  - CMSC 14400 Systems Programming II
  - CMSC 22200. Computer Architecture
  - CMSC 22600. Compilers for Computer Languages
- Communication, bits, systems:
  - CMSC 23300. Networks and Distributed Systems
- Concurrency, threads, scheduling:
  - CMSC 23000. Operating Systems
  - CMSC 23010. Parallel Computing

... and many more!



# Study for Final

- Binary, hex, decimal conversion (both signed and unsigned)
- Your homework solutions
- Tagged union
  - Write a tagged union called Car with variants SUV, Sedan, Truck
- Array List
  - Malloc and realloc
- Linked List
  - Write a traversal by hand
- BST
  - What are the properties of a BST? Draw a binary tree that is not a BST.
  - Write a "map\_get" by hand

# Study for Final

## Cont.

- Sorting
  - Insertion, Selection, Bubble: In each iteration, where do we look? What is swapped?
  - Merge sort: How to merge two sorted lists?
  - Quick sort: Why partitioning sorts the list?
  - Heap sort: Visually, how do insertion and removal look like?
- Hash table
  - What is a good hash function? What is a *problematic* hash function?
  - Chaining
  - Probing -- why do we need tombstones?