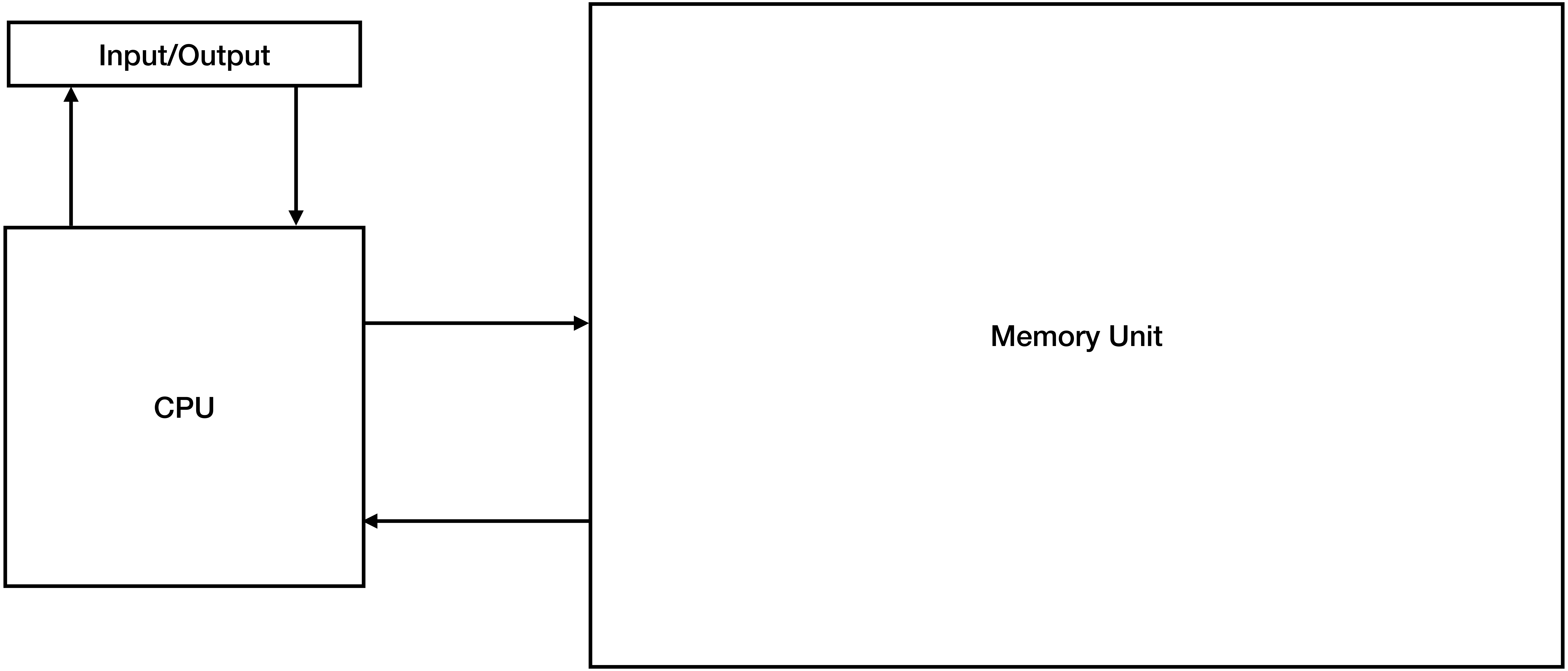
Wrap Up

CS143: lecture 18

Konstantinos Ameranis, August 4

A Von Neumann Machine



A new perspective...

A new perspective...

- Variables (data) and functions (code) live in memory

A new perspective...

- Variables (data) and functions (code) live in memory
  - Memory is a contiguous storage of bytes

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

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:

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

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

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, ...

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 new perspective...

- A process's memory is partitioned into

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

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

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);`

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);`

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

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 ...

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 -- FFFFFFF...)

A new perspective...

A new perspective...

- Data and code are just bits

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

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

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

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)

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...

A new perspective...

- Statically, we can organize data...

A new perspective...

- Statically, we can organize data...
  - ... of different types into a `struct`

A new perspective...

- Statically, we can organize data...
  - ... of different types into a `struct`
    - to represent a real-world object

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)

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

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

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

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...

A new perspective...

- Dynamically, we can organize data as:

A new perspective...

- Dynamically, we can organize data as:
  - `list`: an ordered sequence

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

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, ...

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

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

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

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

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

Topics Covered

Memory:

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

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

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

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

Review

C

- All operators:

Review

C

- All operators:
  - Unary: `!x` `~x` `-x` `x++` `++x` `x--` `--x`

Review
C

- All operators:
  - Unary: `!x ~x -x x++ ++x x-- --x`
  - Binary:

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`,

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`

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

&x

type : int *
value: 100

type : int *
value: 100

&x_p

type : int **
value: 108

type : int
value: 25

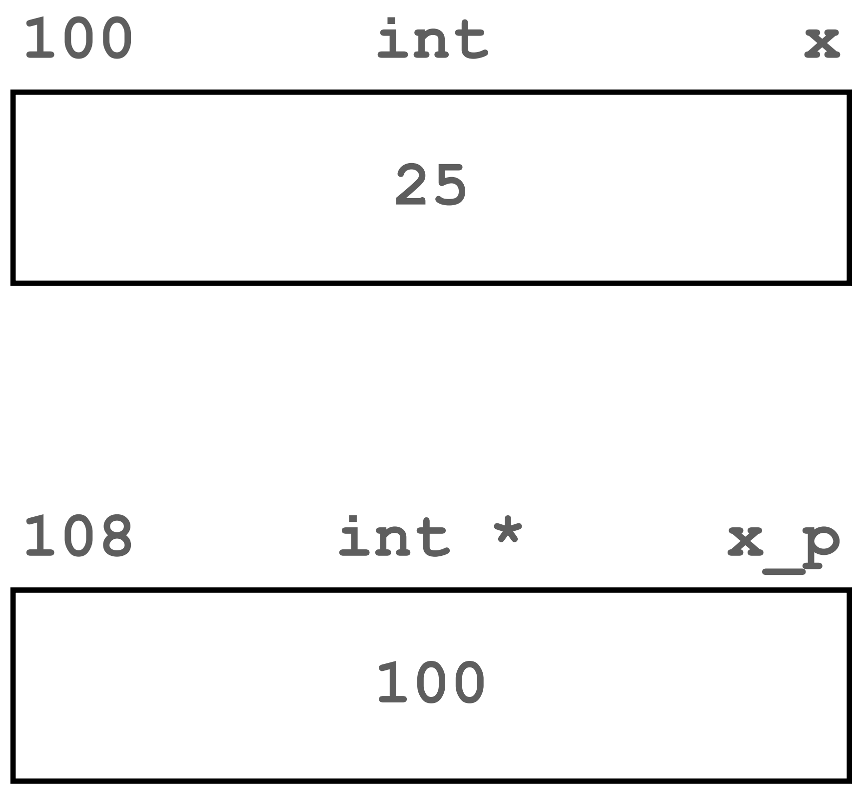
***x**

error

type : int *
value: 100

***x_p**

type : int
value: 25



Review

C

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

Review
C

- All operators: (Cont.)
  - Pointers: `*a`, `&a`
  - Subscript and member:

Review

C

- All operators: (Cont.)
  - Pointers: `*a`, `&a`
  - Subscript and member:
    - `a.field`

Review
C

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

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`

Review

C

- All operators: (Cont.)
  - Pointers: `*a`, `&a`
  - Subscript and member: `a.field`, `a[b]`, `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`)

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

Bits

Endian

- We think of an integer as one atomic value:

Bits

Endian

- We think of an integer as one atomic value:
  - `int x = 0x1A2B3C4D;`

Bits

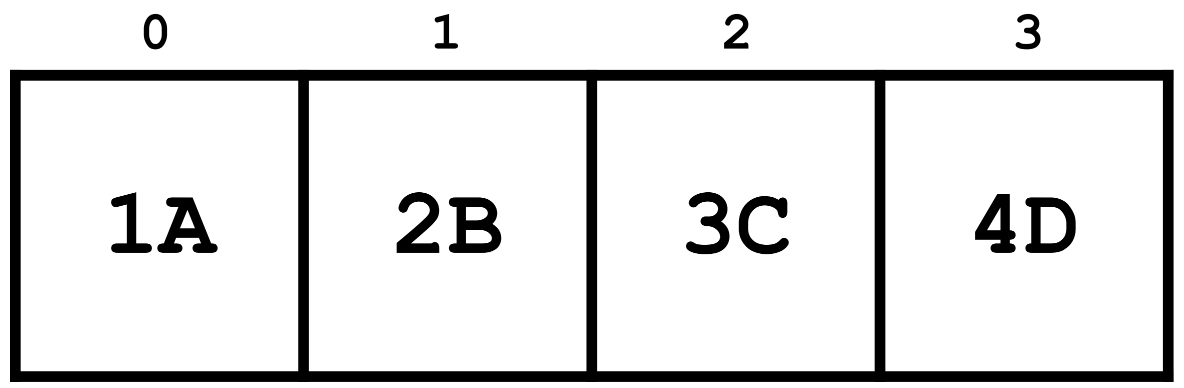
Endian

- 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

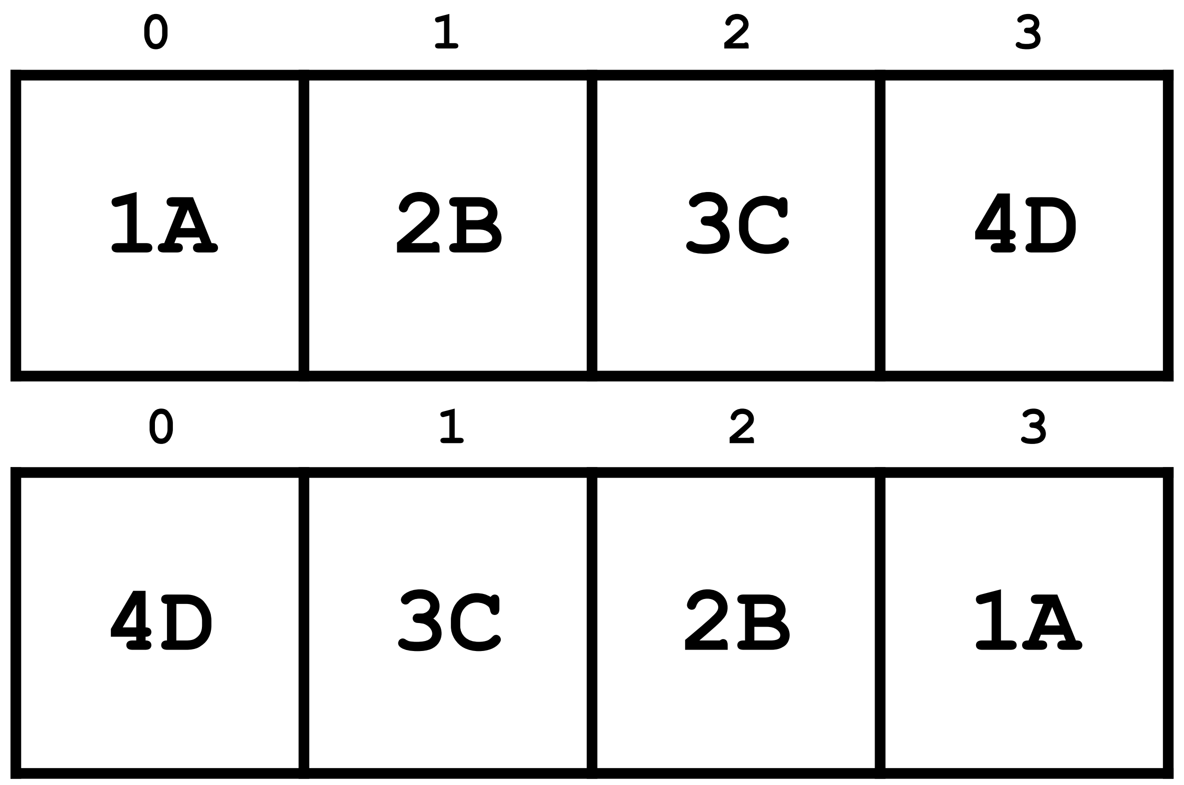
- We think of an integer as one atomic value:
  - `int x = 0x1A2B3C4D;`
- But if an integer has 4 bytes and each byte is addressable, which of the 4 bytes is stored first?



Bits

Endian

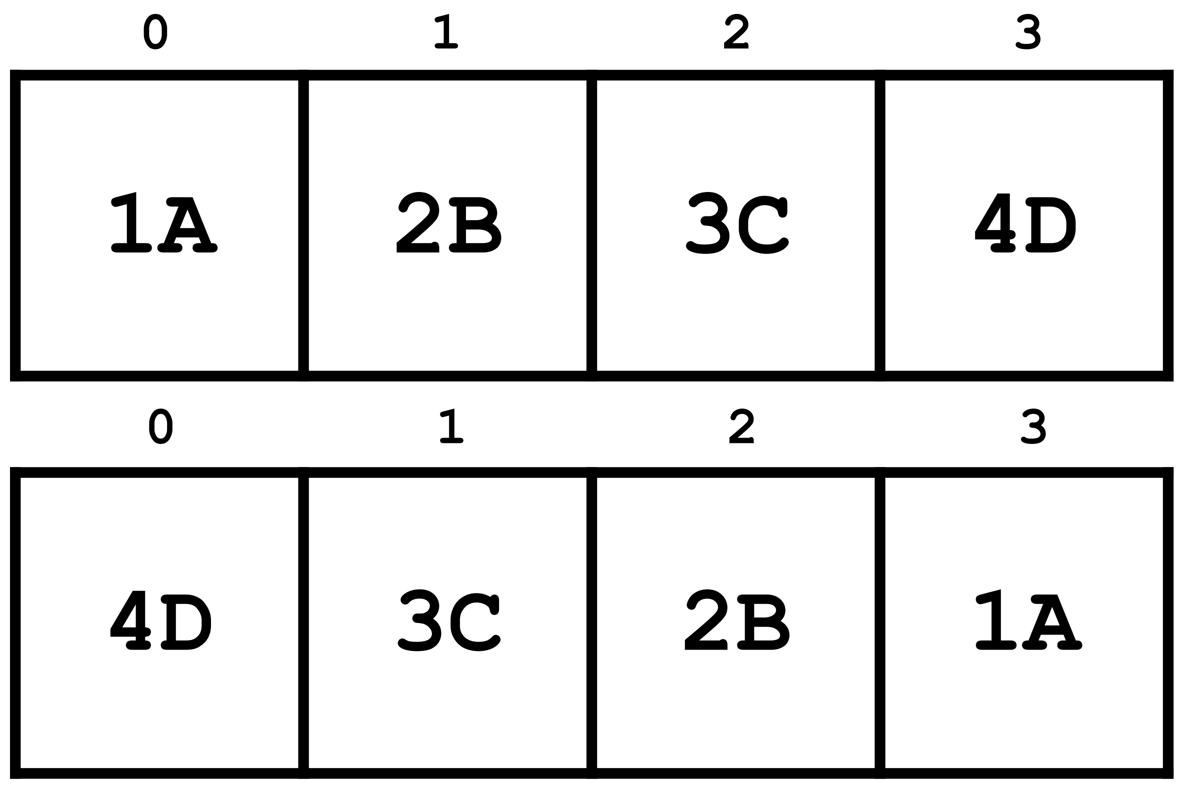
- We think of an integer as one atomic value:
    - `int x = 0x1A2B3C4D;`
- But if an integer has 4 bytes and each byte is addressable, which of the 4 bytes is stored first?



Bits

Endian

- 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?



Most significant byte first

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?

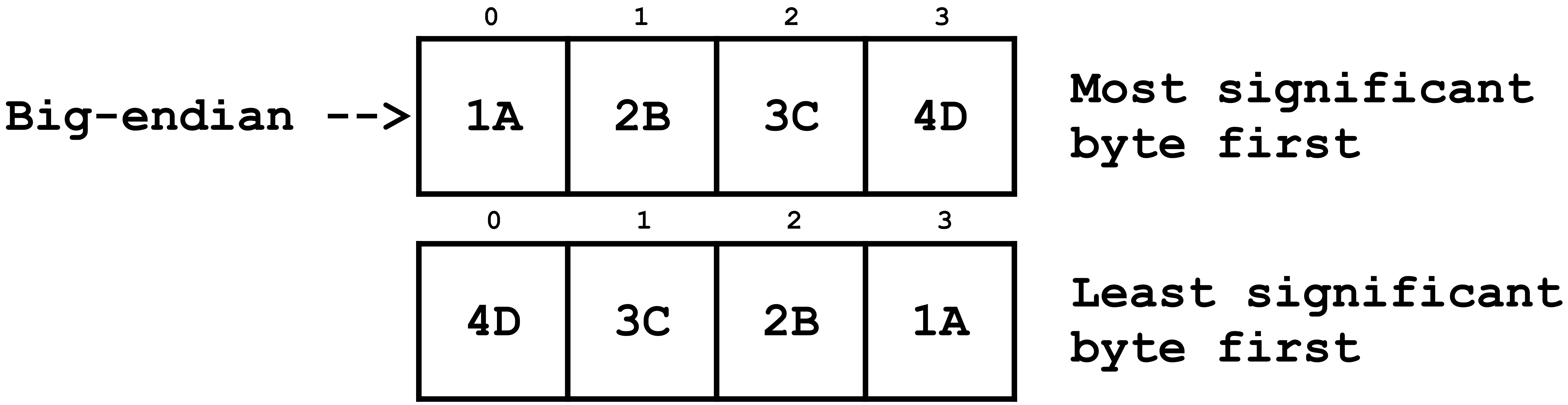
0	1	2	3	
1A	2B	3C	4D	**Most significant byte first**

0	1	2	3	
4D	3C	2B	1A	**Least significant byte first**

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

- 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?

	0	1	2	3	
`Big-endian -->`	1A	2B	3C	4D	`Most significant byte first`
`Little-endian -->`	4D	3C	2B	1A	`Least significant byte 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)

Review

Function Frames

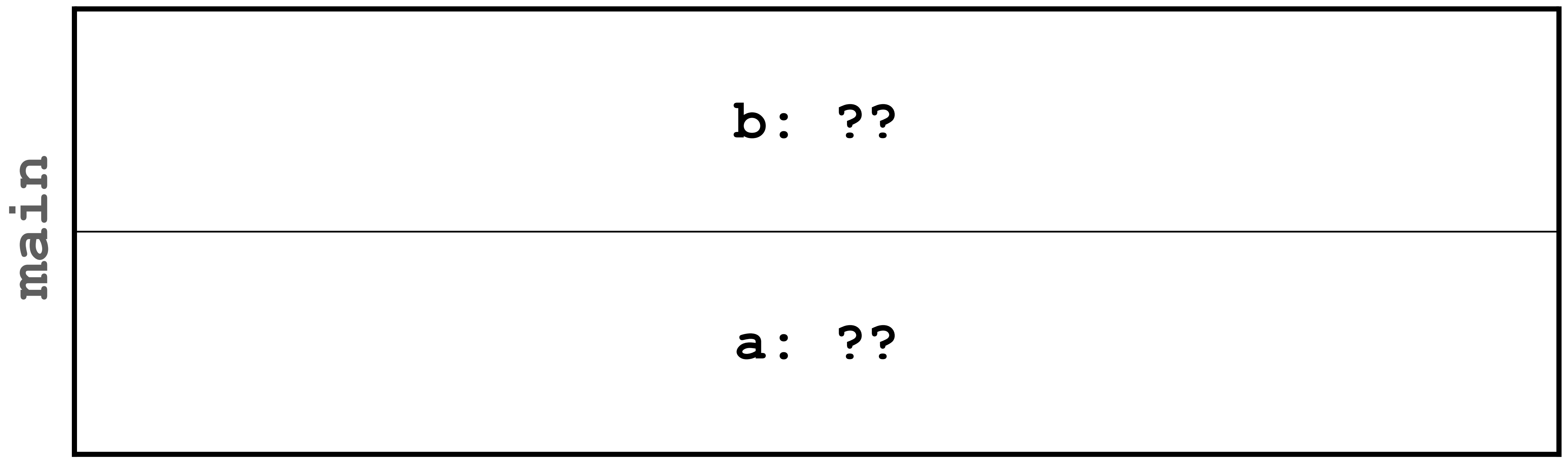
- When a function returns, we can recycle the memory used by the variables declared inside the function.
    - Variables declared in `{ .. }` can only be accessed in `{ .. }` (Scope)
- Local variables and arguments live in a *frame*.

Variable Lifetime

```c
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;
}
```

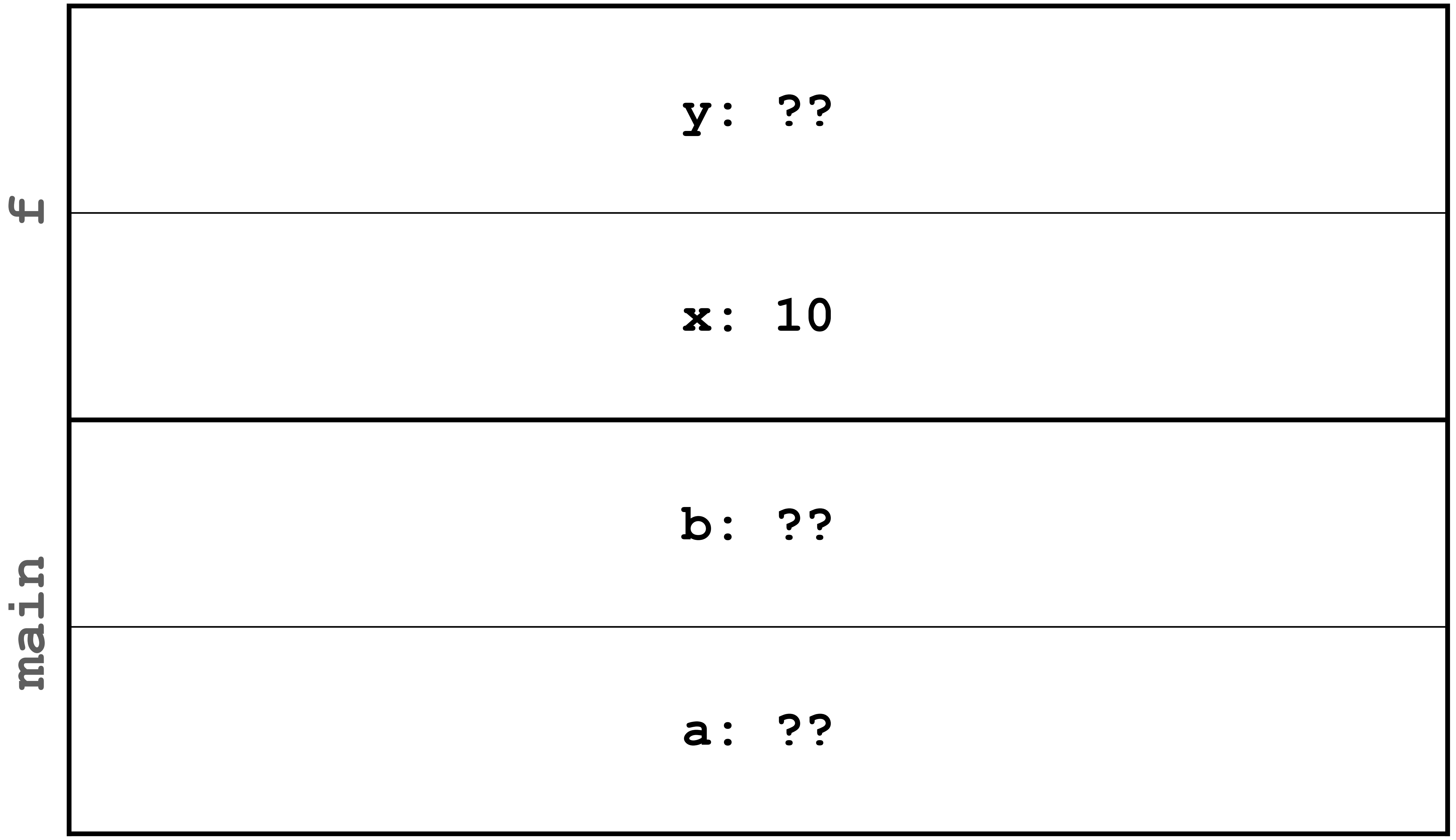
Variable Lifetime

```
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;
}
```



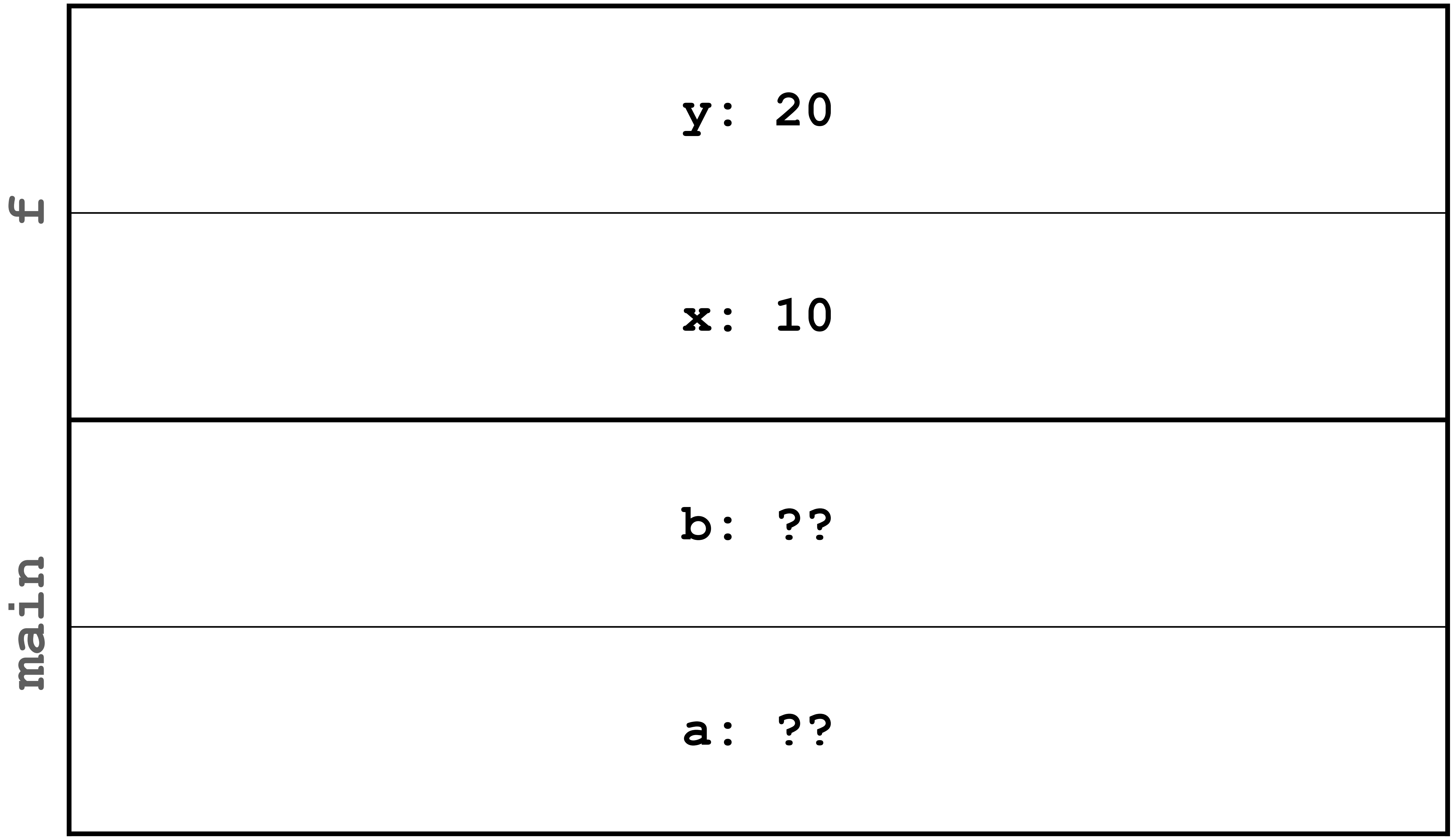
Variable Lifetime

```
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;
}
```



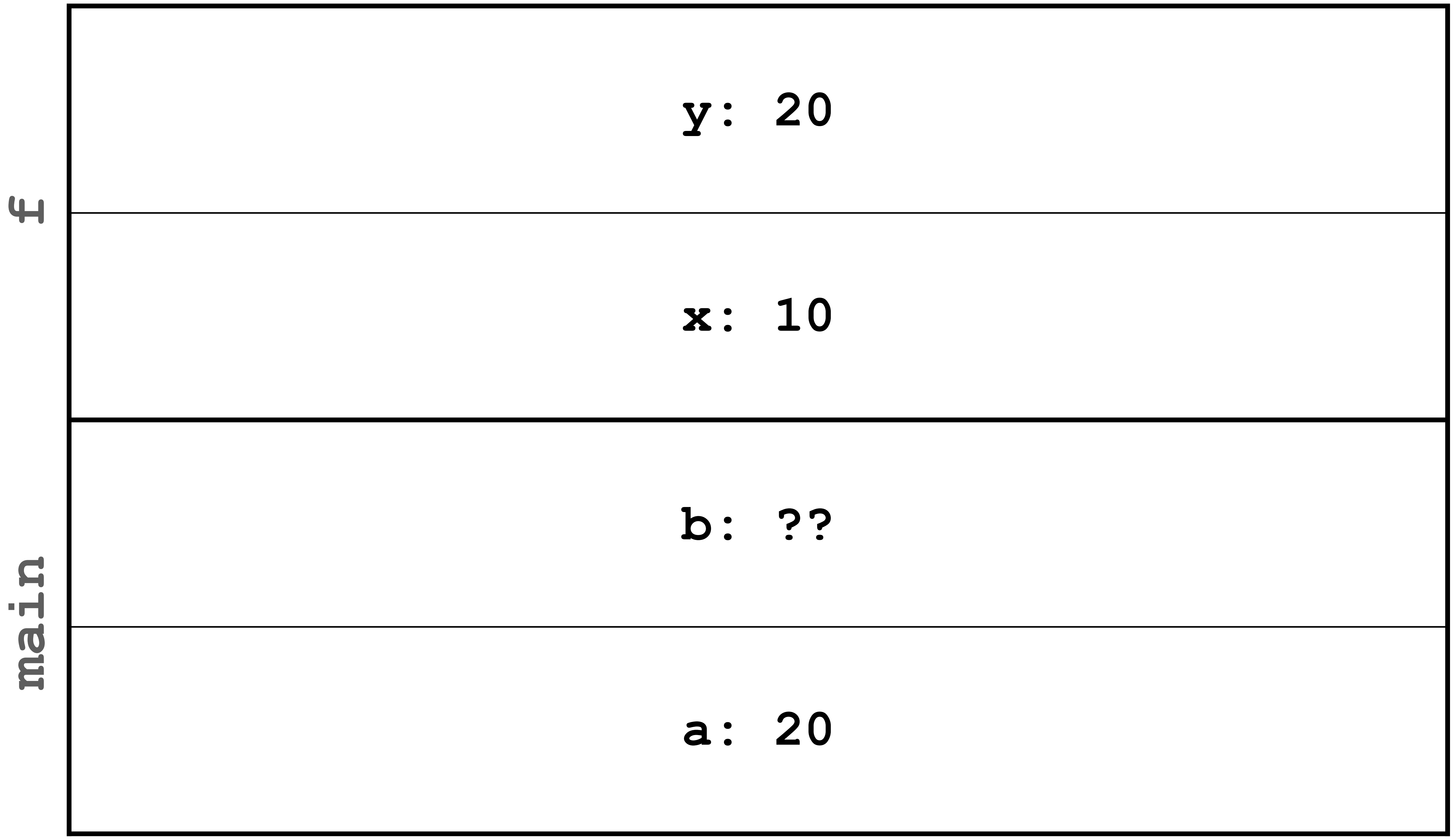
Variable Lifetime

```
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;
}
```



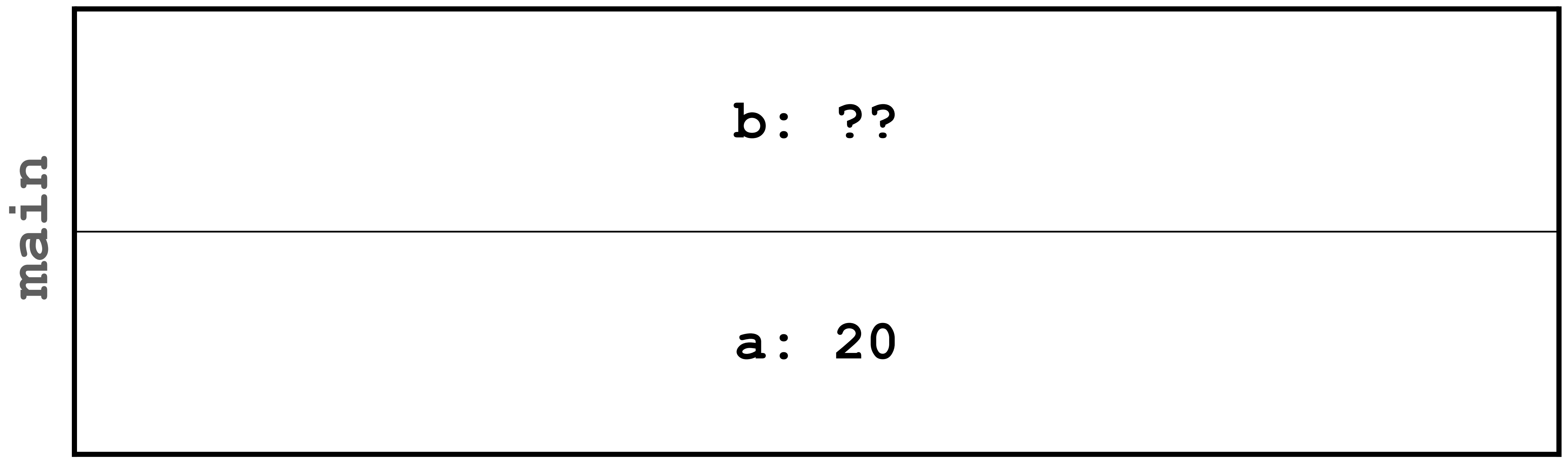
Variable Lifetime

```
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;
}
```



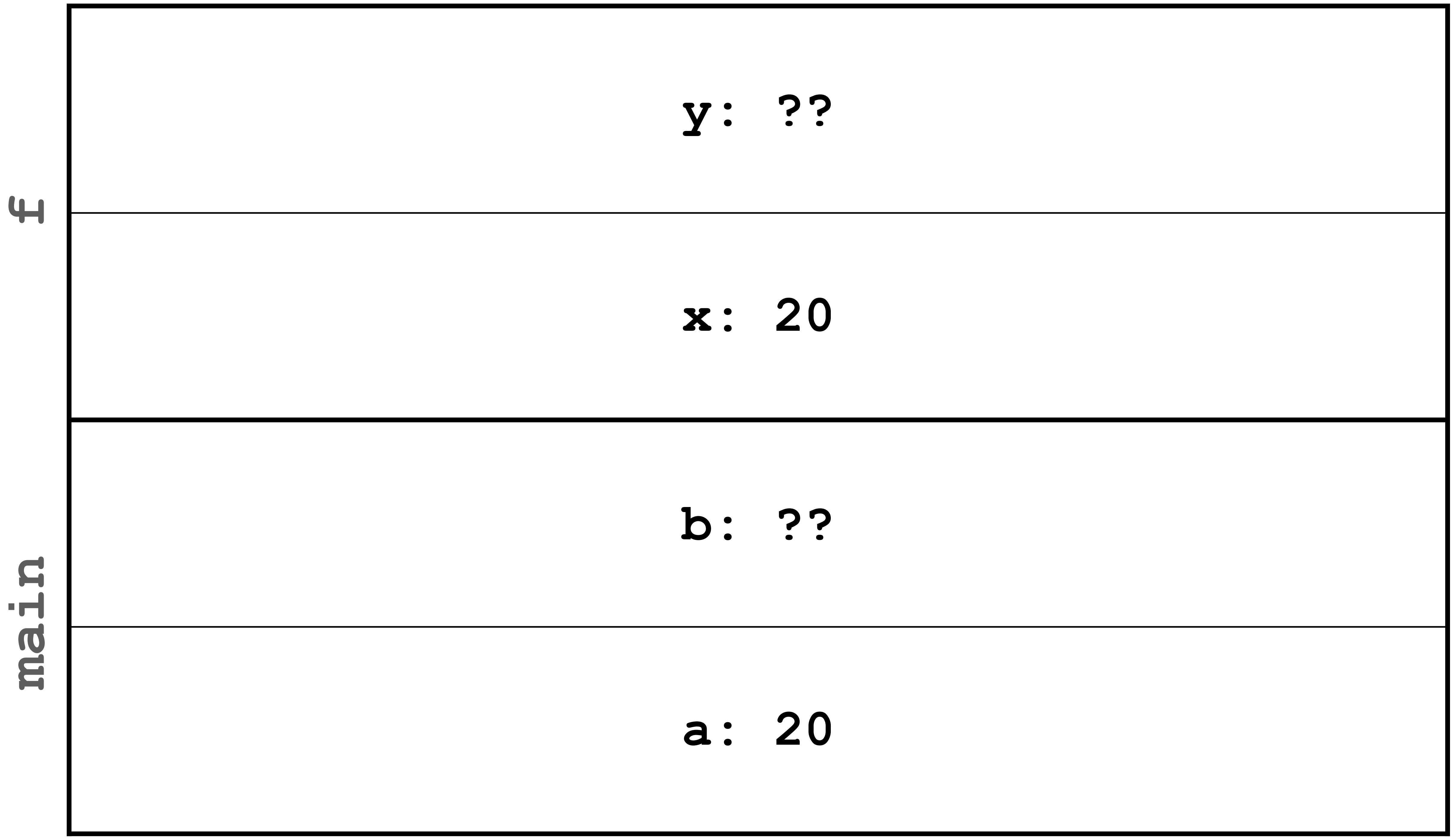
Variable Lifetime

```
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;
}
```



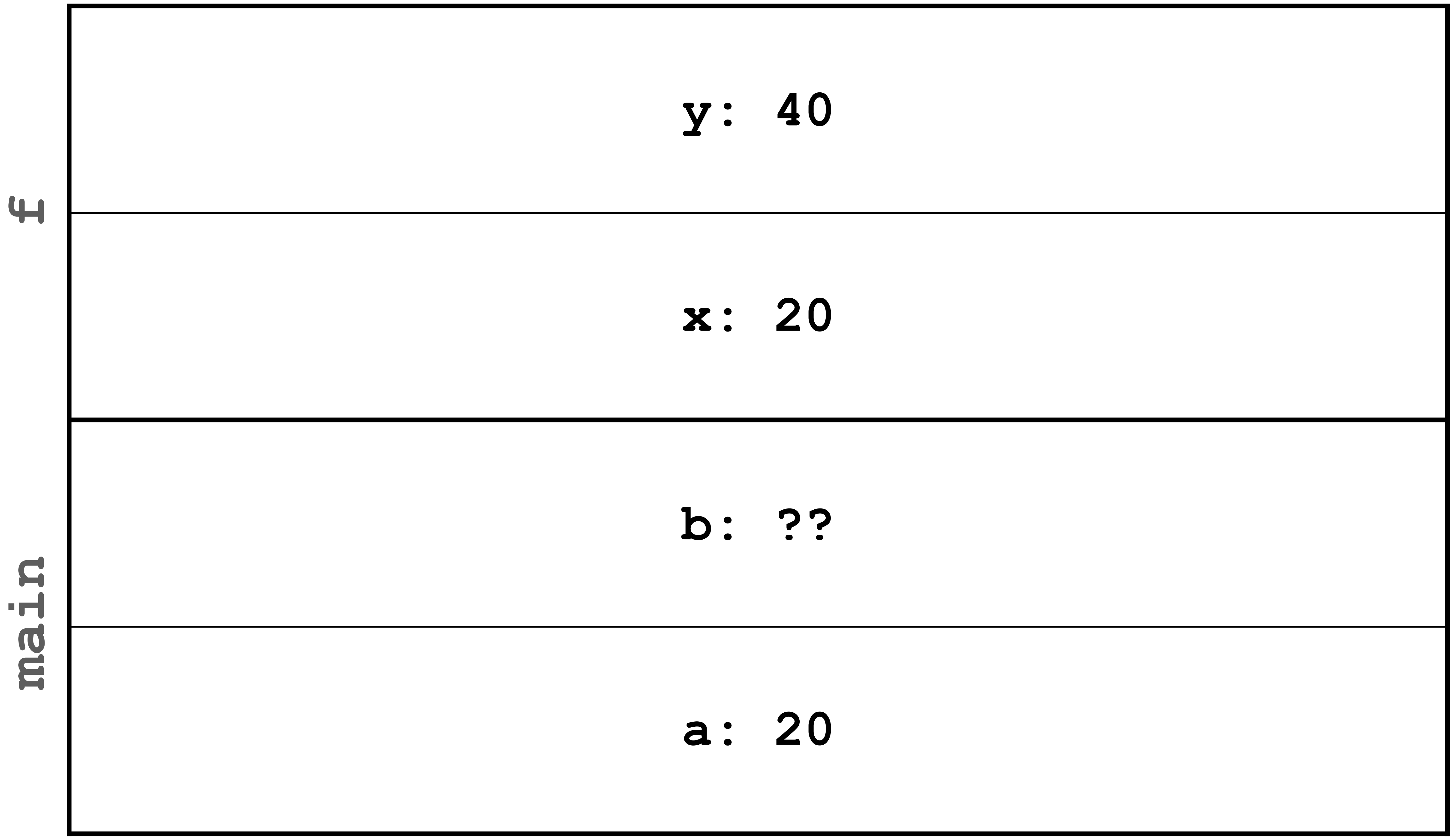
Variable Lifetime

```c
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;
}
```



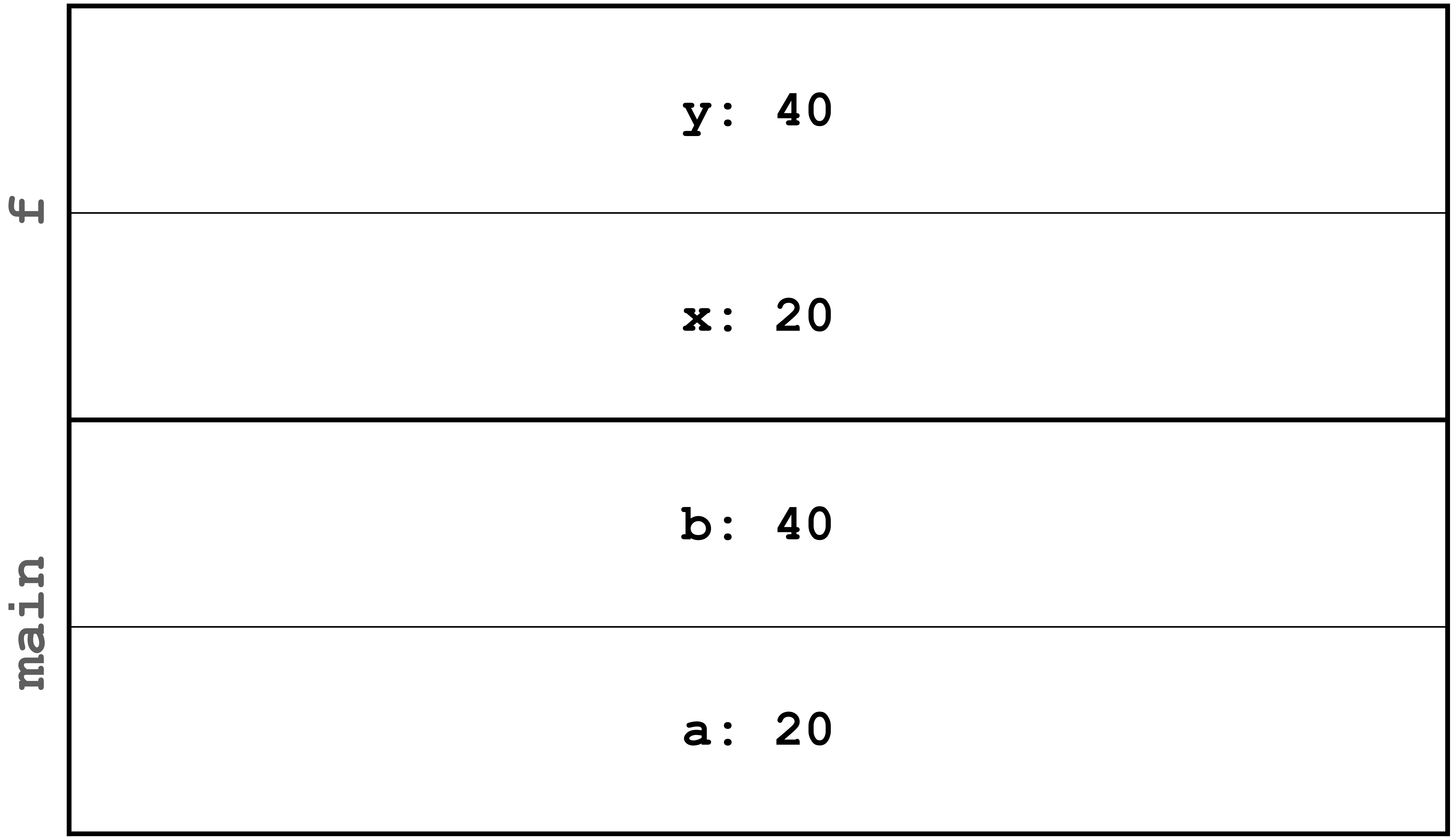
Variable Lifetime

```c
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;
}
```



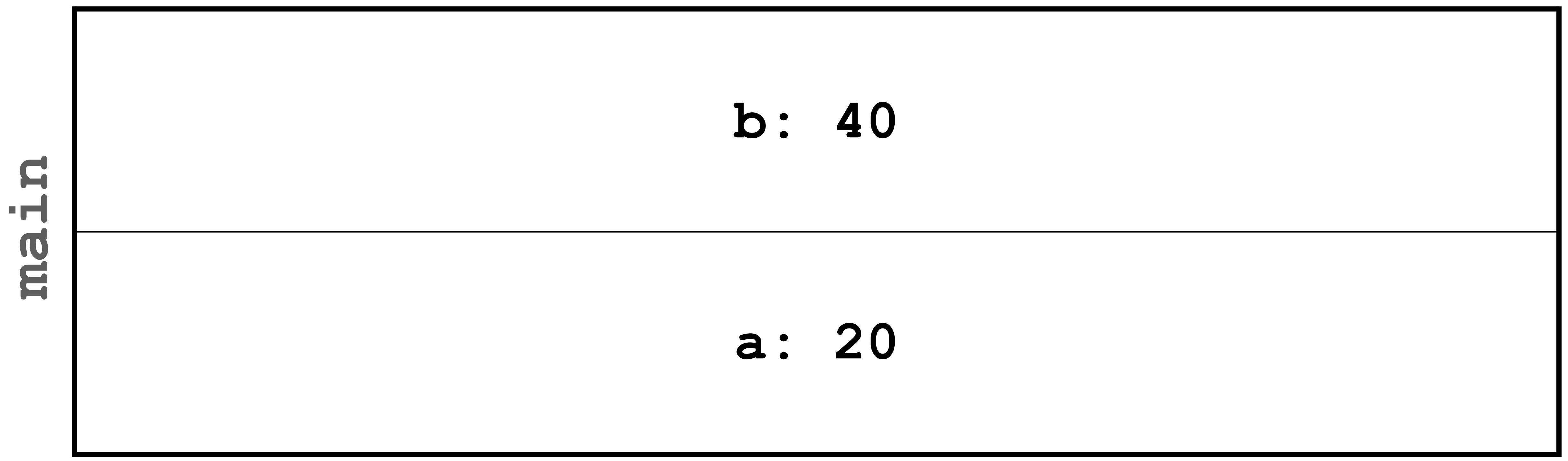
Variable Lifetime

```c
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

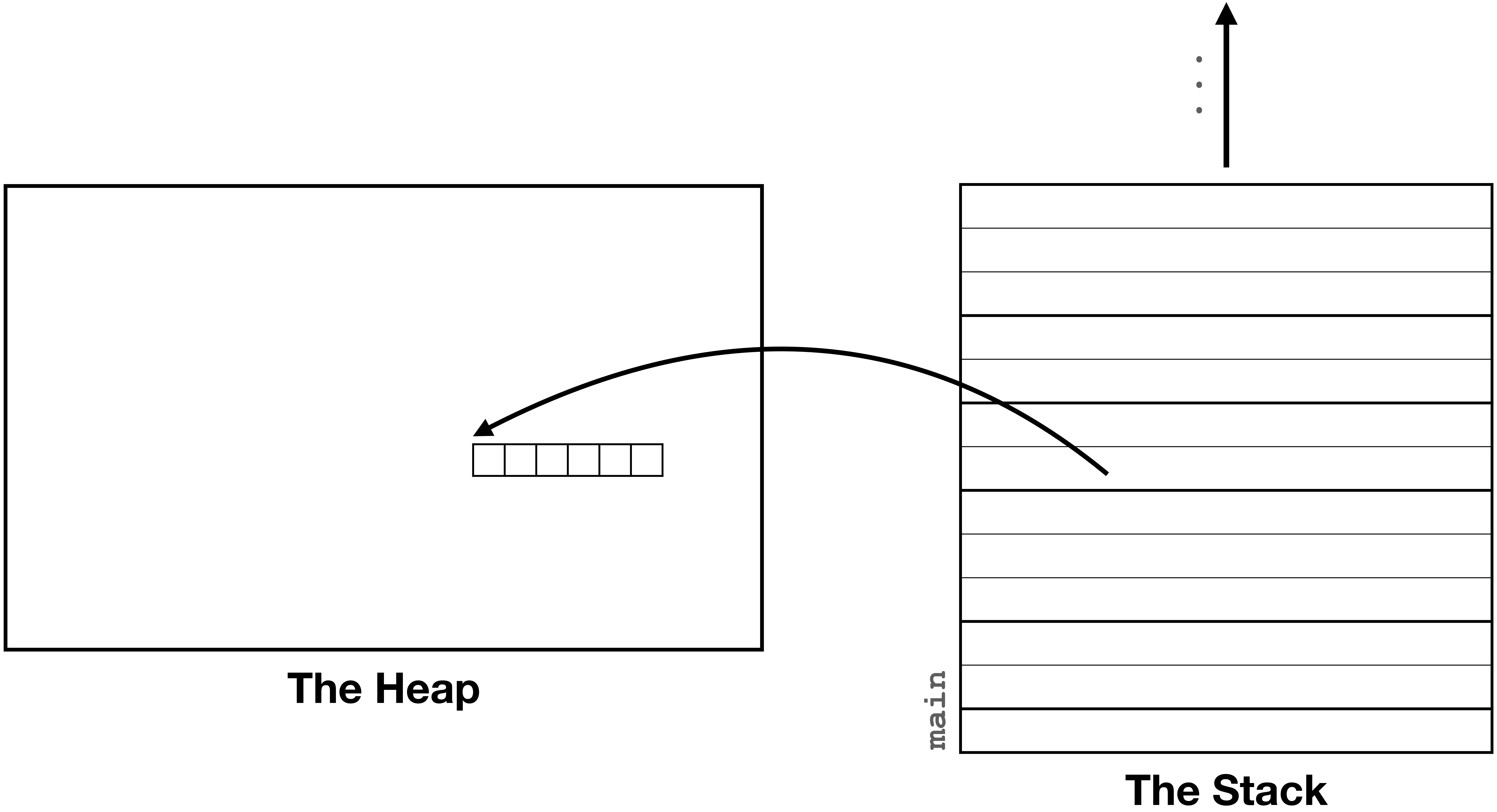
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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;
}
```

The Heap



The Heap

Stack vs Heap

Stack	Heap

The Heap

Stack vs Heap

Stack	Heap
• Acquire memory:	• Acquire memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables

Heap

- Acquire memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables

Heap

- Acquire memory:
  - `ptr = malloc(n)`

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)

Heap

- Acquire memory:
  - `ptr = malloc(n)`

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:
  - `free(ptr)`

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing

Heap

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

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*

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

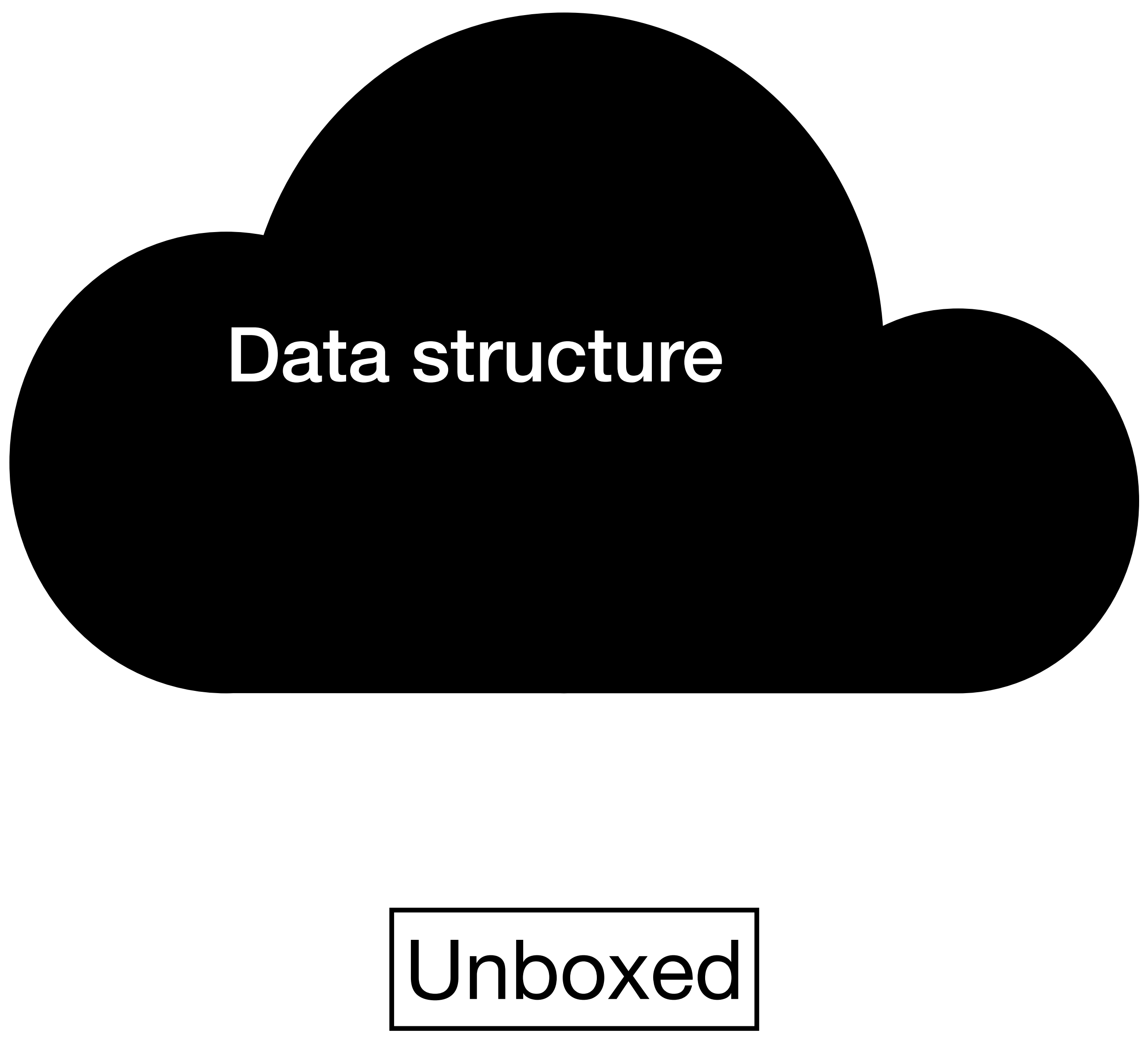
Heap

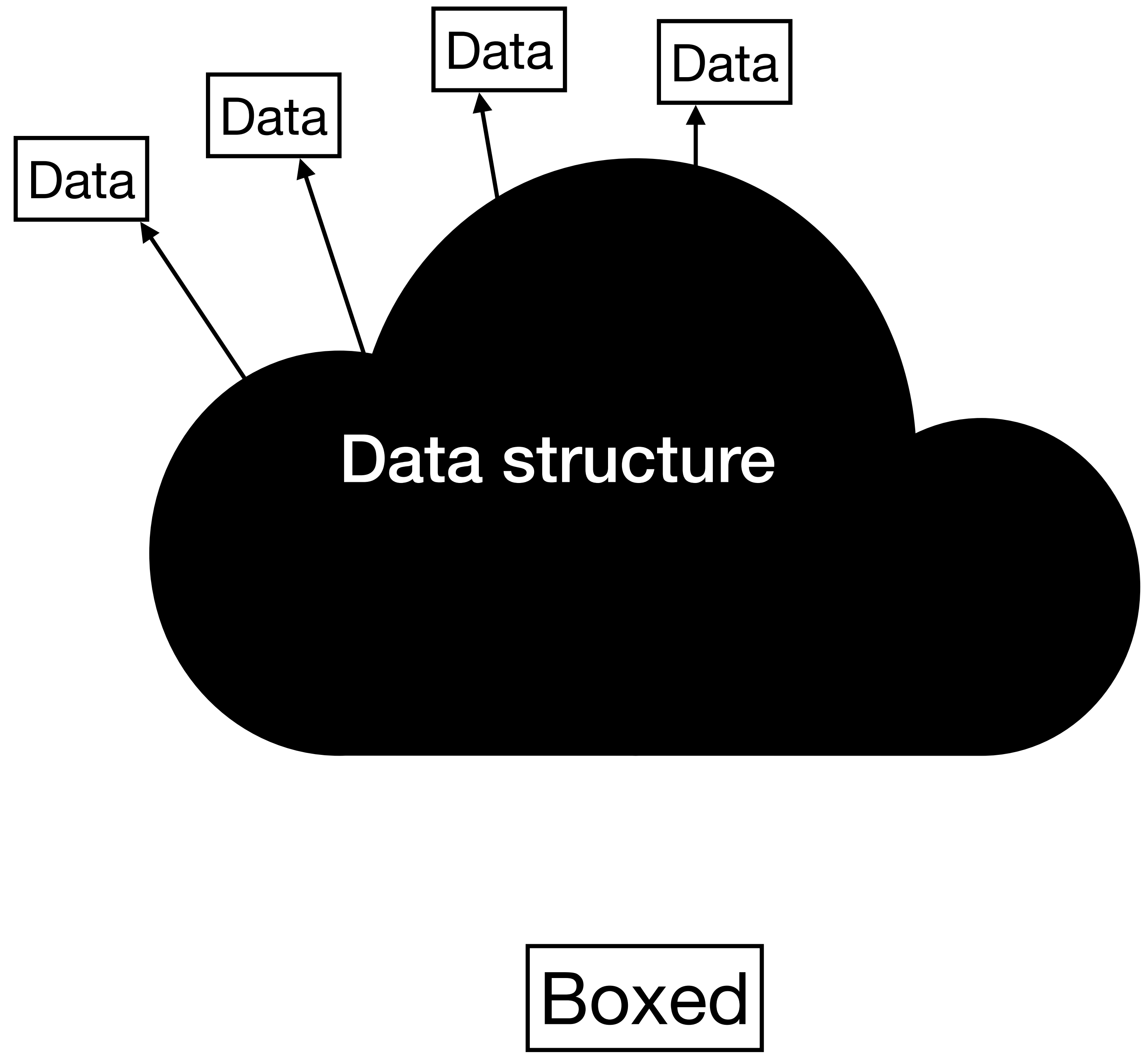
- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:
  - `free(ptr)`
  - You can forget to release; *memory leak*

- Accessing released memory is bad; *memory error*

Data Structures

Week 4 onwards





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

Data Structures

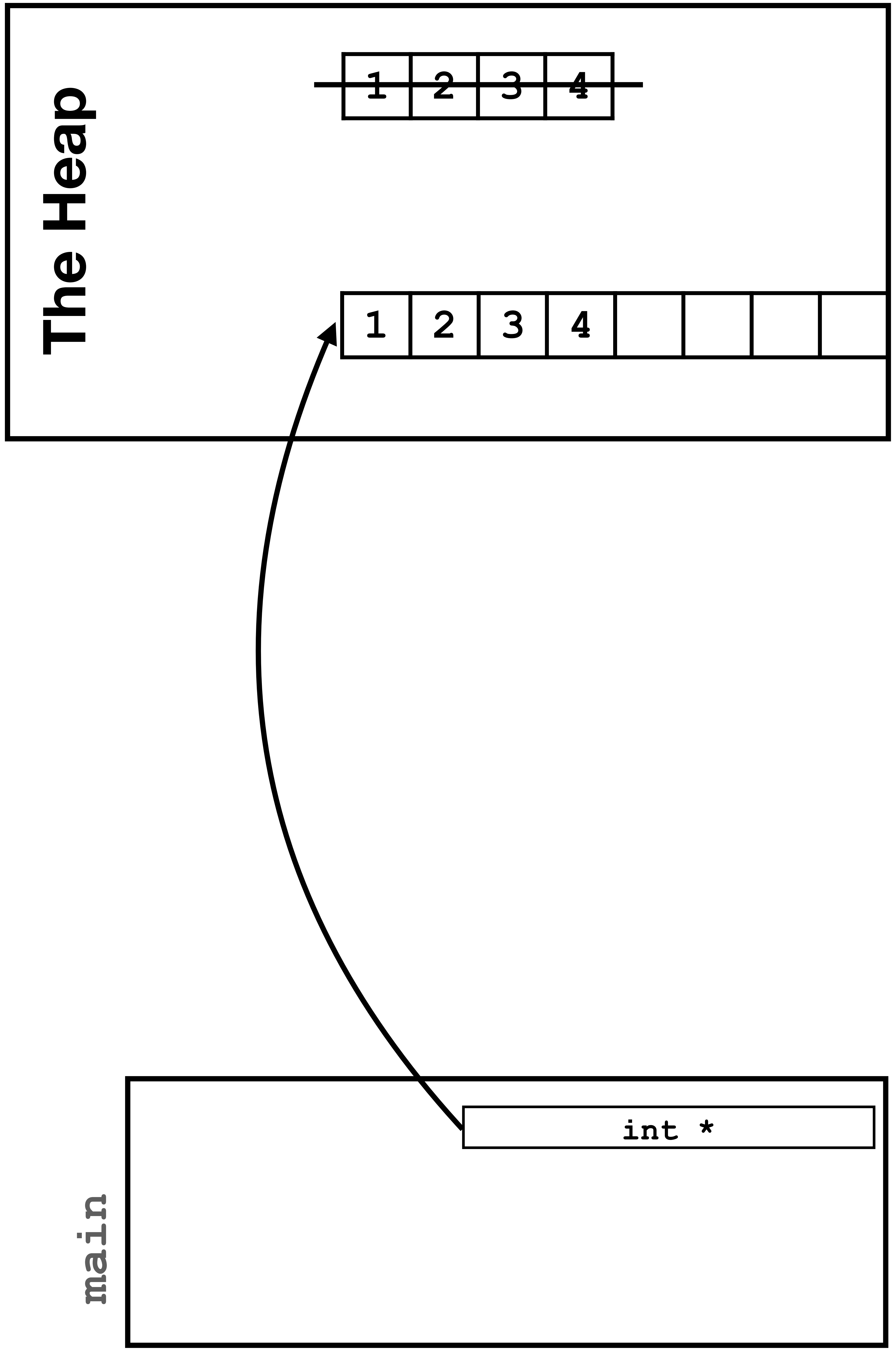
- Establishing structures on the heap:
  - Indices: contiguous
    - $O(1)$ random access
    - difficult to reorder and reallocate
  - Pointer: scattered
    - sequential access
    - easy to reorder and reallocate

	Indices	**Pointers**
List	Array List	Linked List
Map	Hash Table	BST

Array

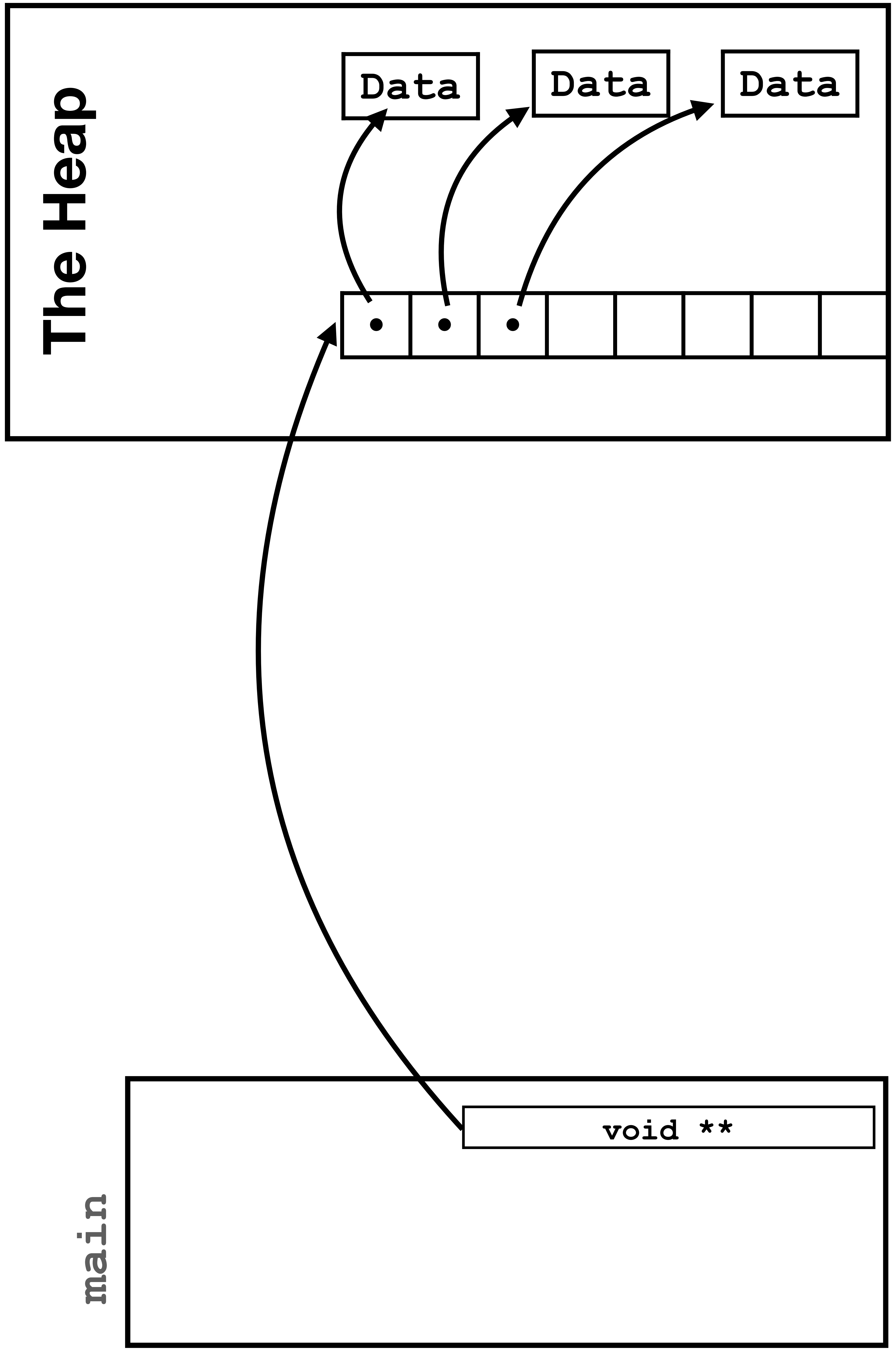
Growing an array

- 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

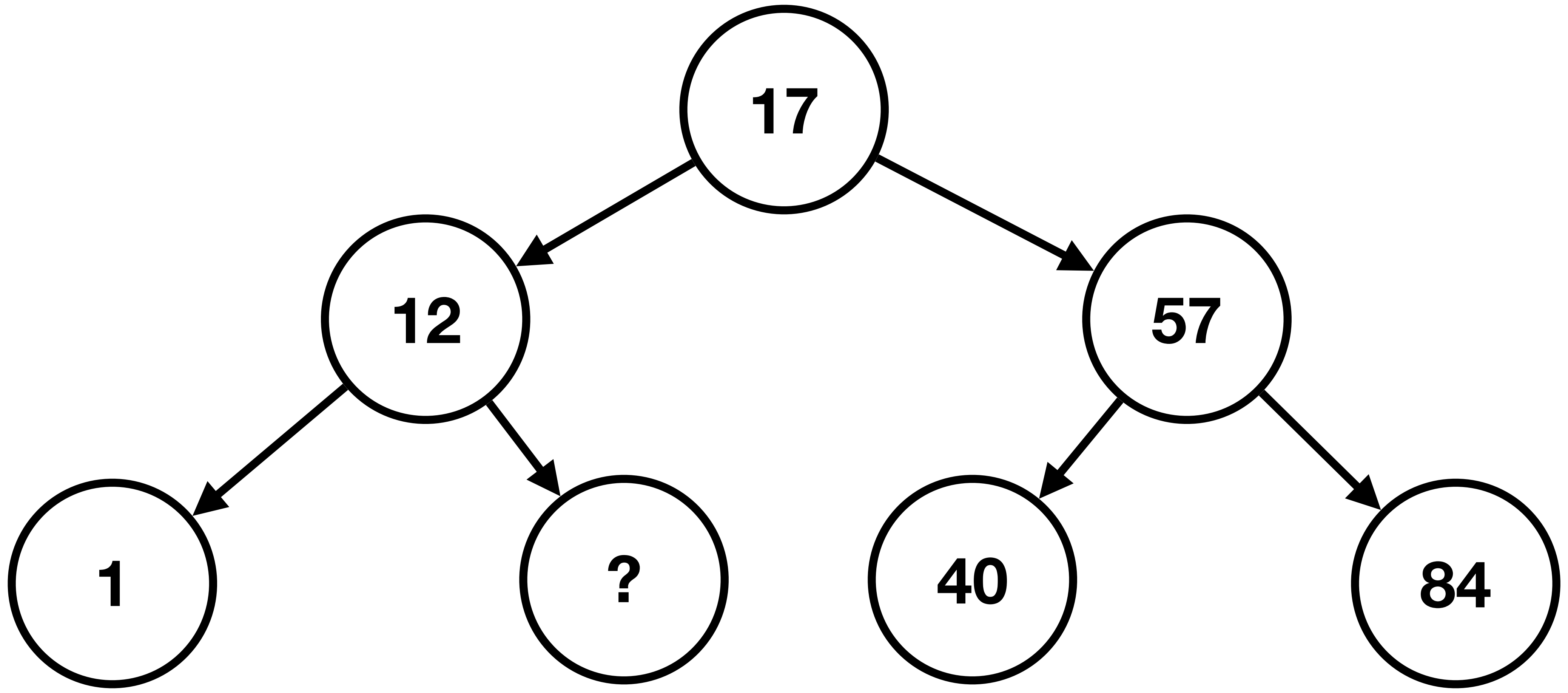
Boxed Array



Linked Lists

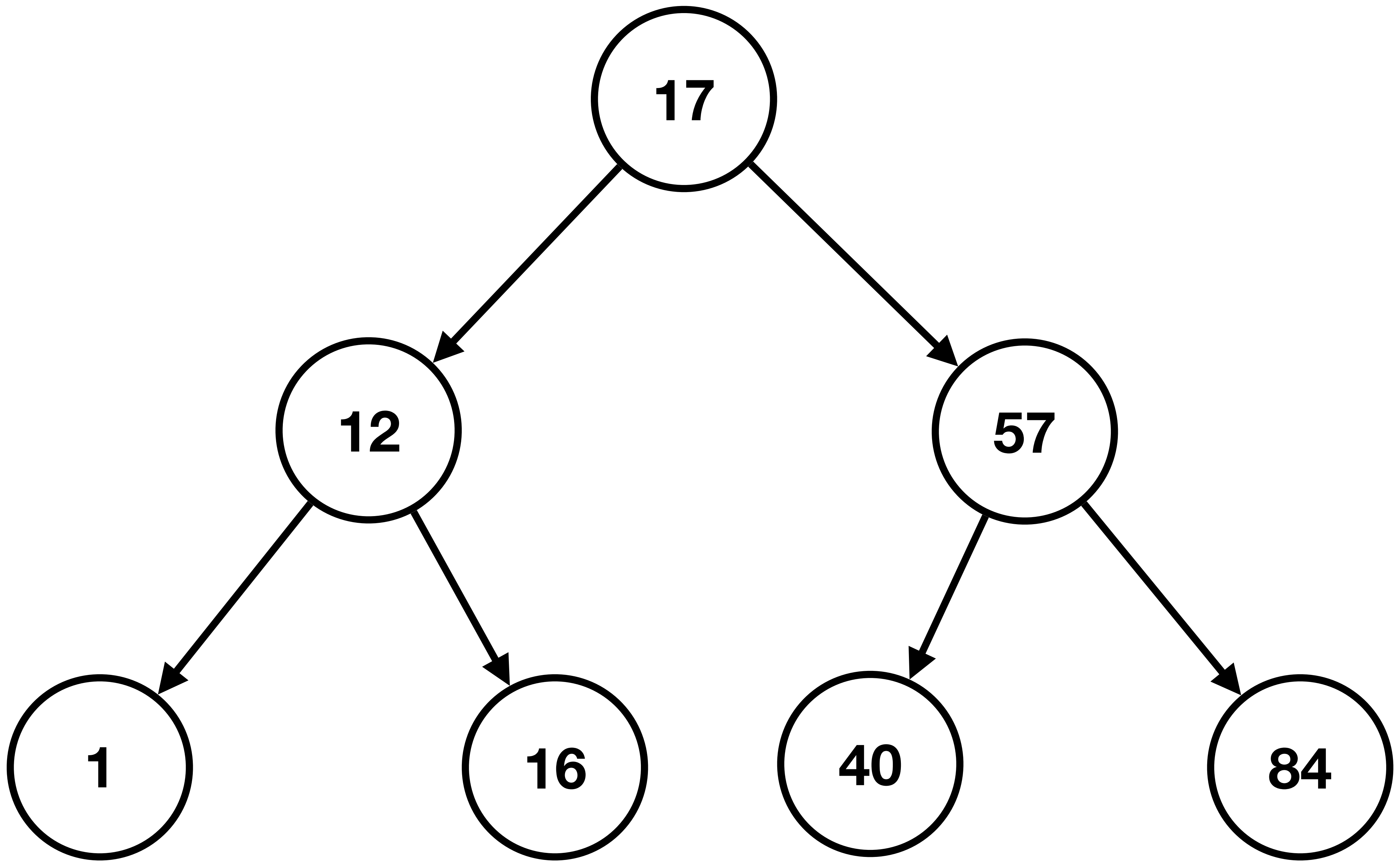
Binary Search Tree

- 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.

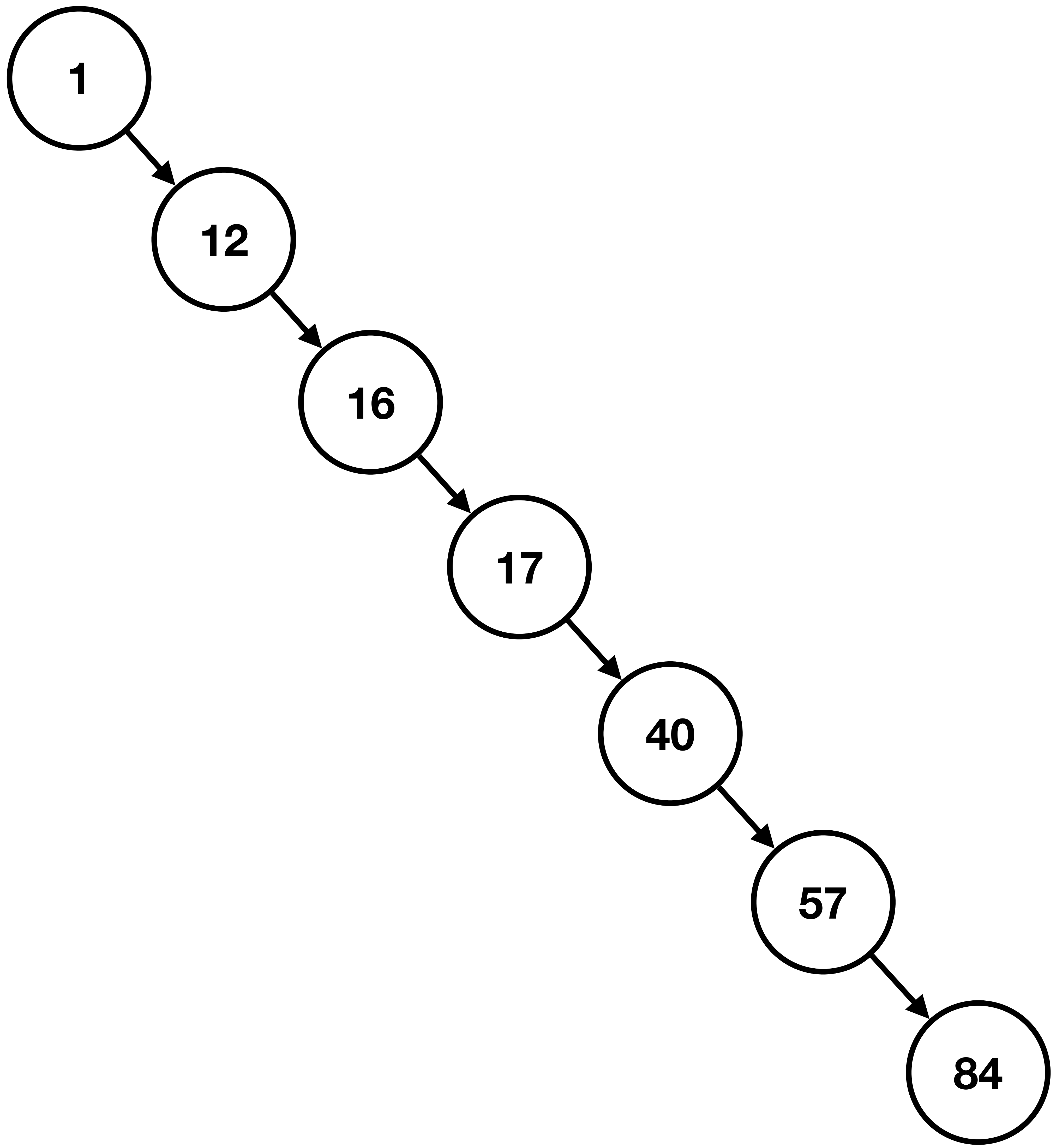


BST

Height



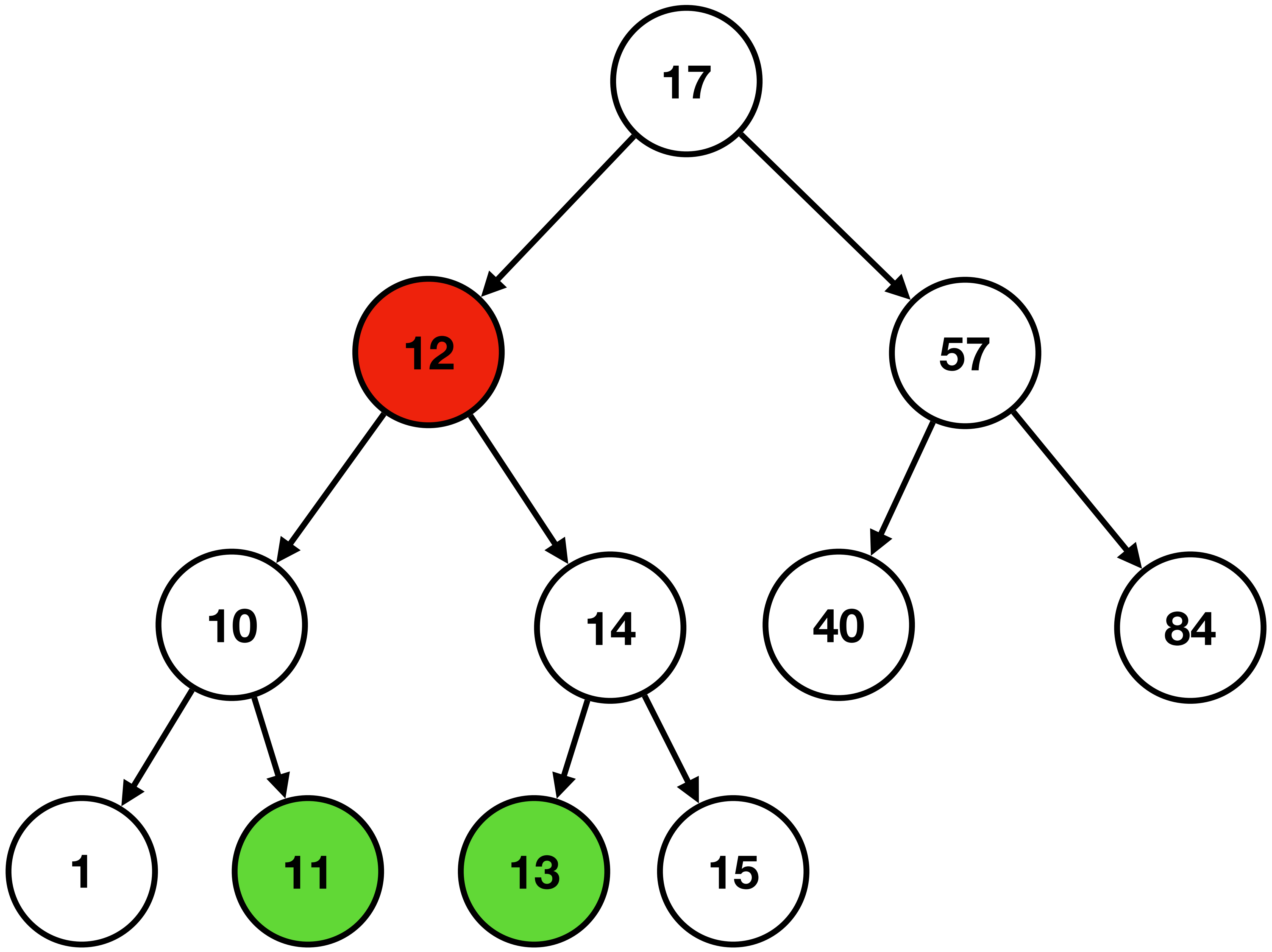
Balanced



Unbalanced

BST

Remove



Hash Table

Review

Hash Table

Review

- Nice $O(1)$ complexity because we can index into an array instead of chasing pointers

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

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)

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:

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

Hash Table

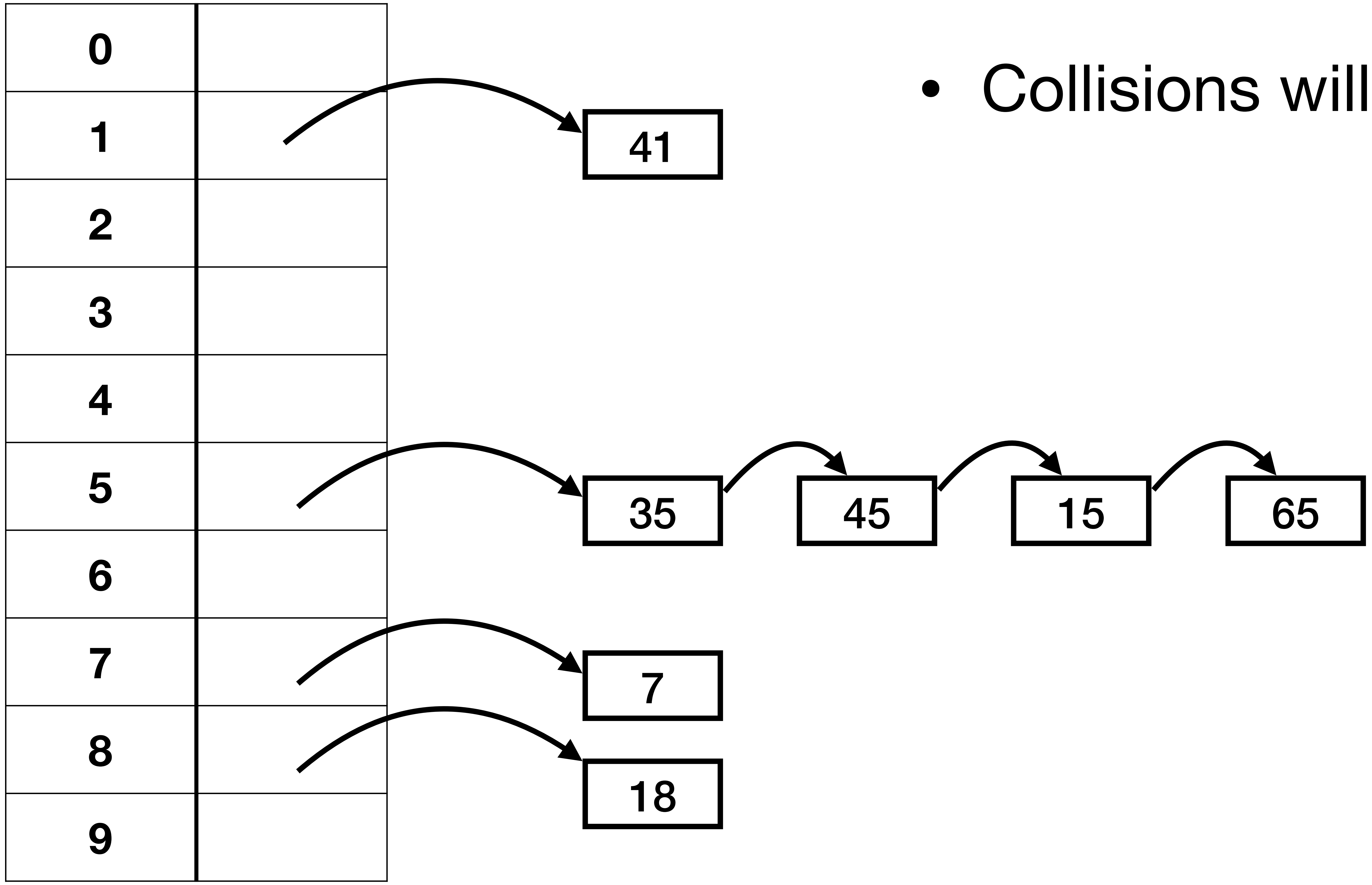
Review

- 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

```c
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	

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:

Hash Table

Linear probing

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

true when
previously occupied

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

Hash Table

Linear probing

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

true when previously occupied

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

Hash Table

Linear probing

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

true when previously occupied

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:

Hash Table

Linear probing

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

true when previously occupied

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

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

Hash Table

Linear probing

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

true when previously occupied

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

Hash Table

Linear probing

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

true when previously occupied

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

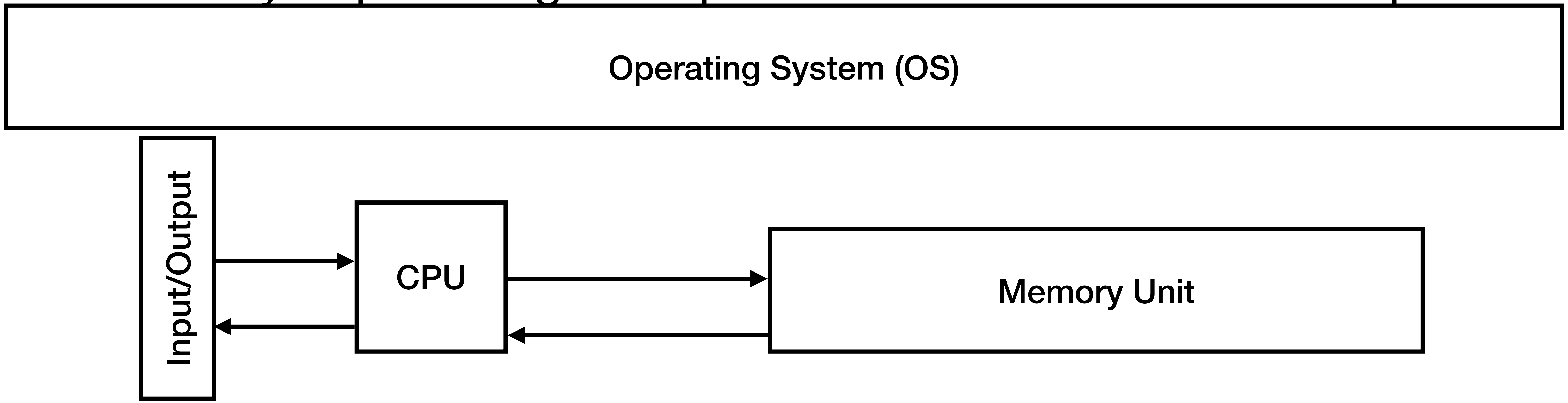
Sorting

- $O(n^2)$: 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.

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



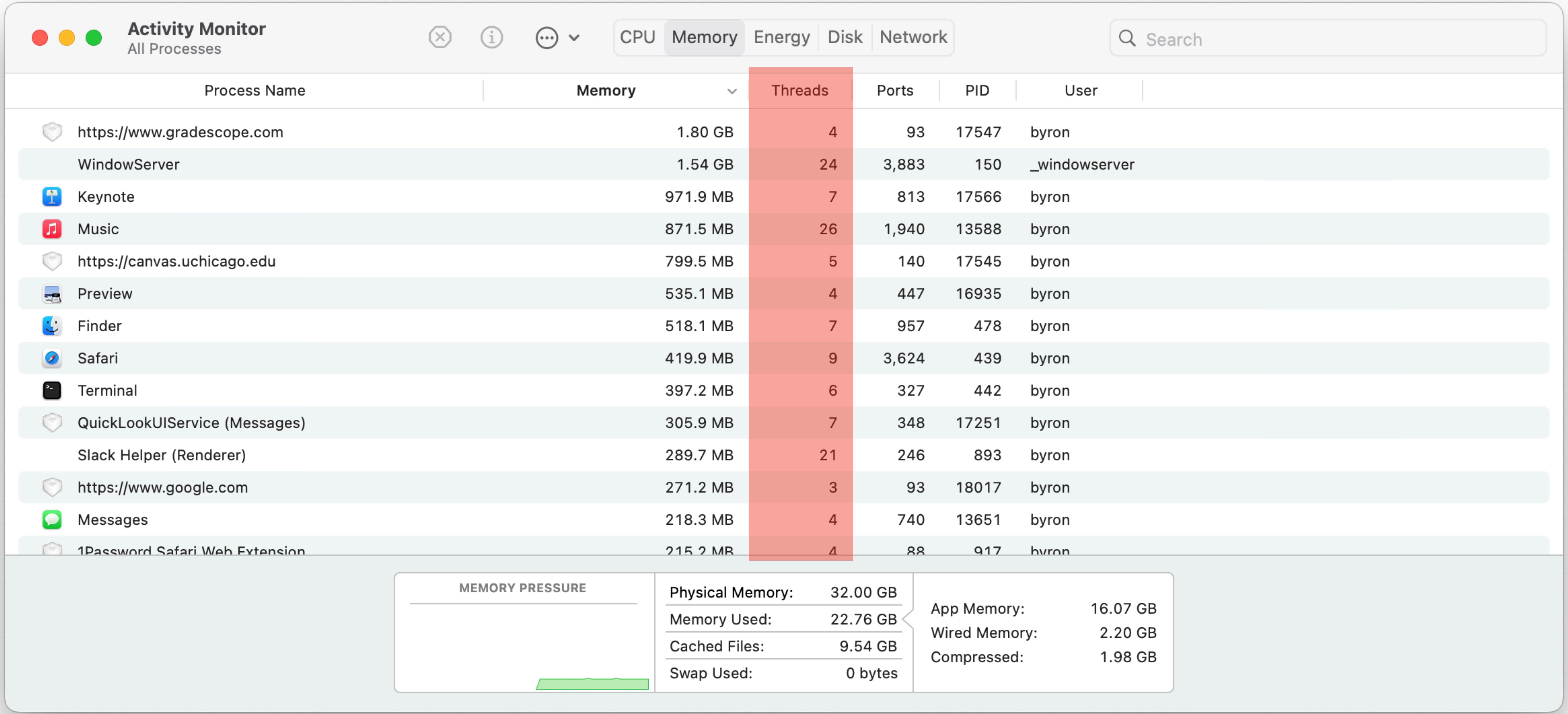
Virtual Memory

- 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



Activity Monitor
All Processes

CPU | Memory | Energy | Disk | Network

Search

Process Name	Memory	Threads	Ports	PID	User
https://www.gradescope.com	1.80 GB	4	93	17547	byron
WindowServer	1.54 GB	24	3,883	150	_windowserver
Keynote	971.9 MB	7	813	17566	byron
Music	871.5 MB	26	1,940	13588	byron
https://canvas.uchicago.edu	799.5 MB	5	140	17545	byron
Preview	535.1 MB	4	447	16935	byron
Finder	518.1 MB	7	957	478	byron
Safari	419.9 MB	9	3,624	439	byron
Terminal	397.2 MB	6	327	442	byron
QuickLookUIService (Messages)	305.9 MB	7	348	17251	byron
Slack Helper (Renderer)	289.7 MB	21	246	893	byron
https://www.google.com	271.2 MB	3	93	18017	byron
Messages	218.3 MB	4	740	13651	byron
1Password Safari Web Extension	215.2 MB	4	88	917	byron

MEMORY PRESSURE

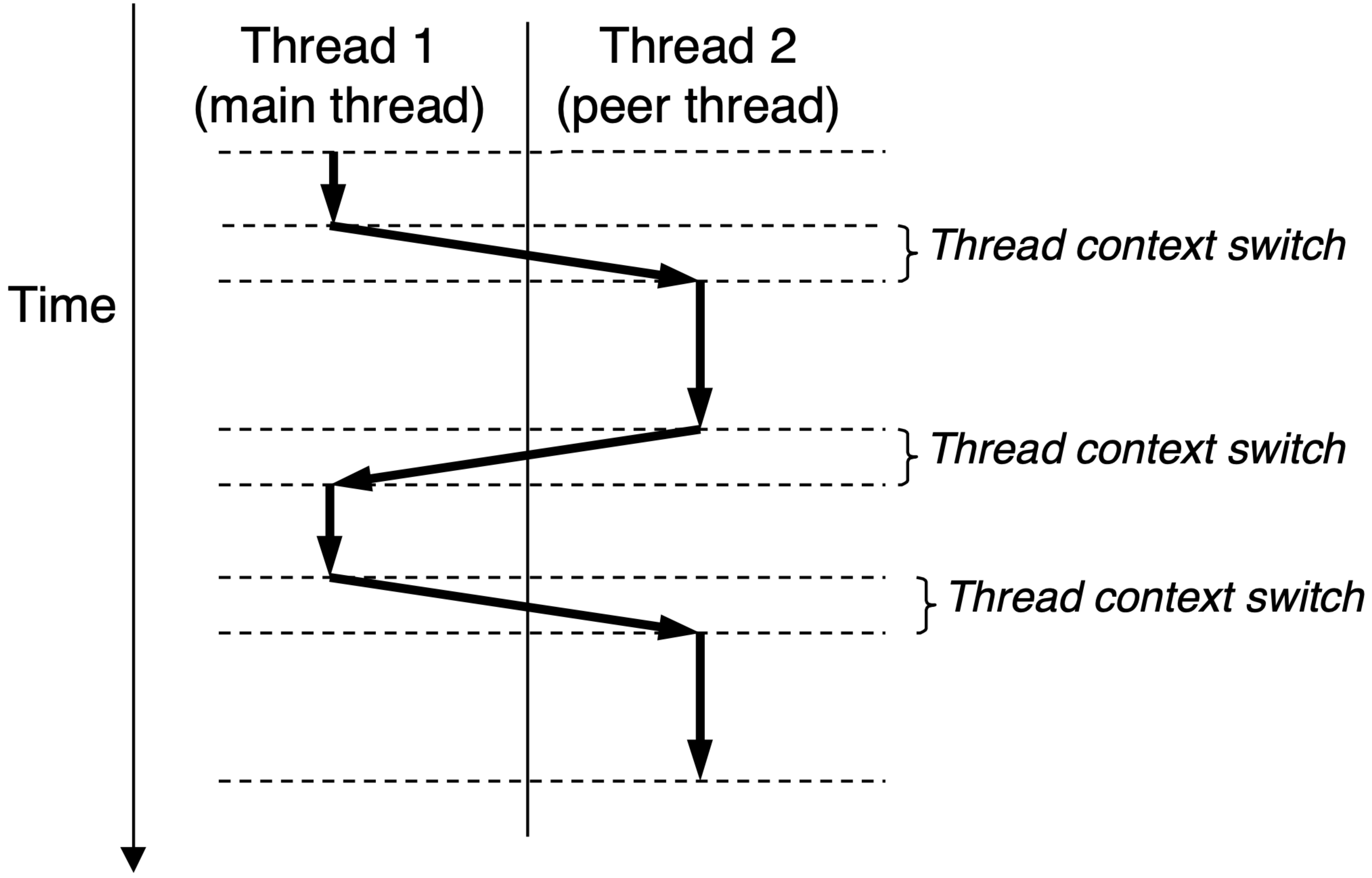
Physical Memory:	32.00 GB
Memory Used:	22.76 GB
Cached Files:	9.54 GB
Swap Used:	0 bytes

App Memory:	16.07 GB
Wired Memory:	2.20 GB
Compressed:	1.98 GB

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


Thread 1
(main thread)
Thread 2
(peer thread)
Time
Thread context switch
Thread context switch
Thread context switch

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?

Course Evaluation

https://go-stage.blueja.io/NNGQqaei9UKjhM4zcOWDGg