LECTURE 4 · CSE 333 · Summer 2026

Types and Structs

Discussion: How long would it take you to paint the walls of this classroom blue?

Reminders

Assessments:

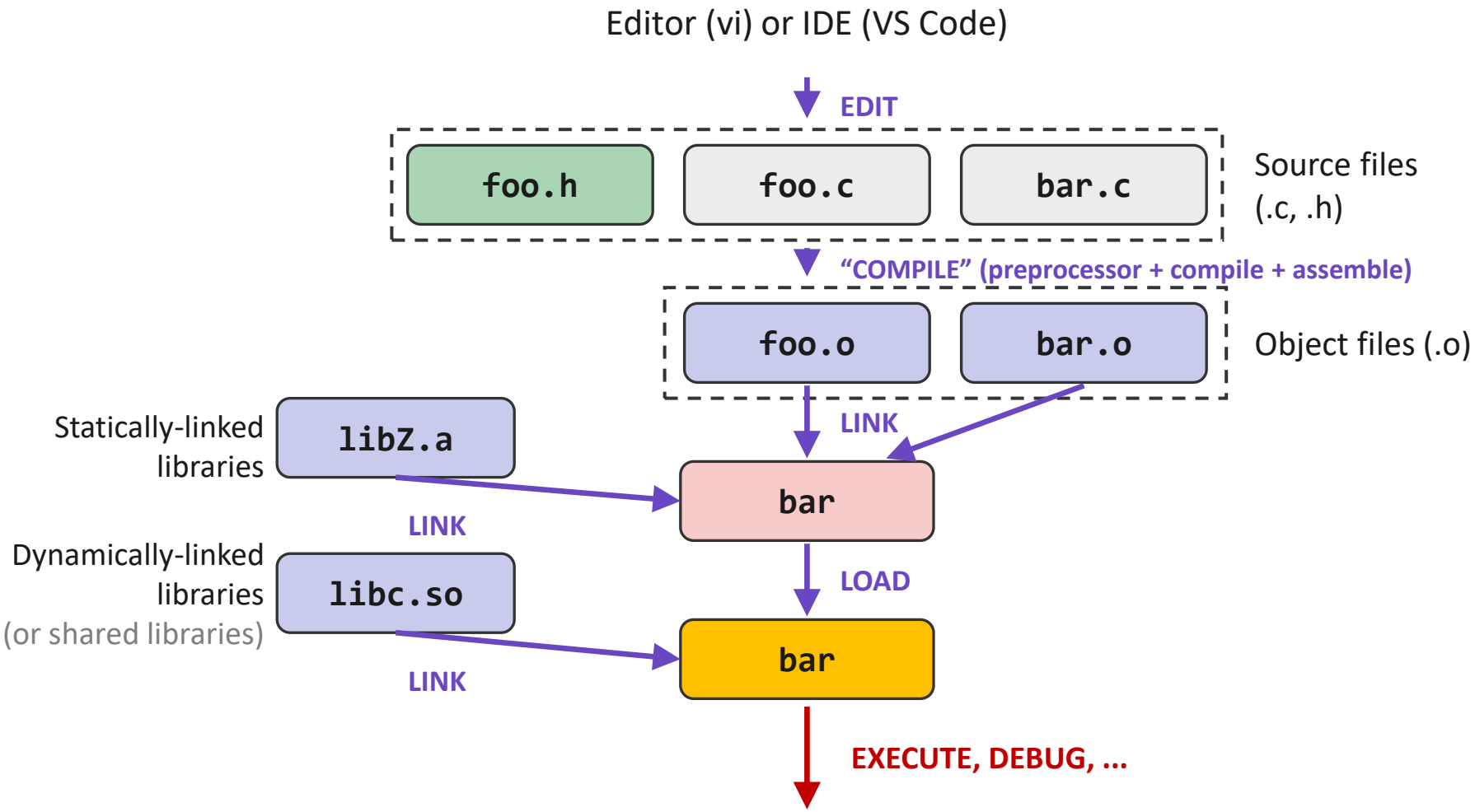
- Exercise 1 feedback is out!
- Exercise 2 due on Thursday at 11:59pm
- Homework 1 released today. Due next Thursday at 11:59pm
  - Much more challenging than the exercises!

General:

- Exams are scheduled
  - Midterm (7/27) from 9:40am to 10:40am in IEB G106
  - Final exam (8/18) from 1:30pm to 3:30pm in CSE2 G10
- I *think* I've scheduled all alternate exams. Let me know ASAP if you still need an alternate.

Review

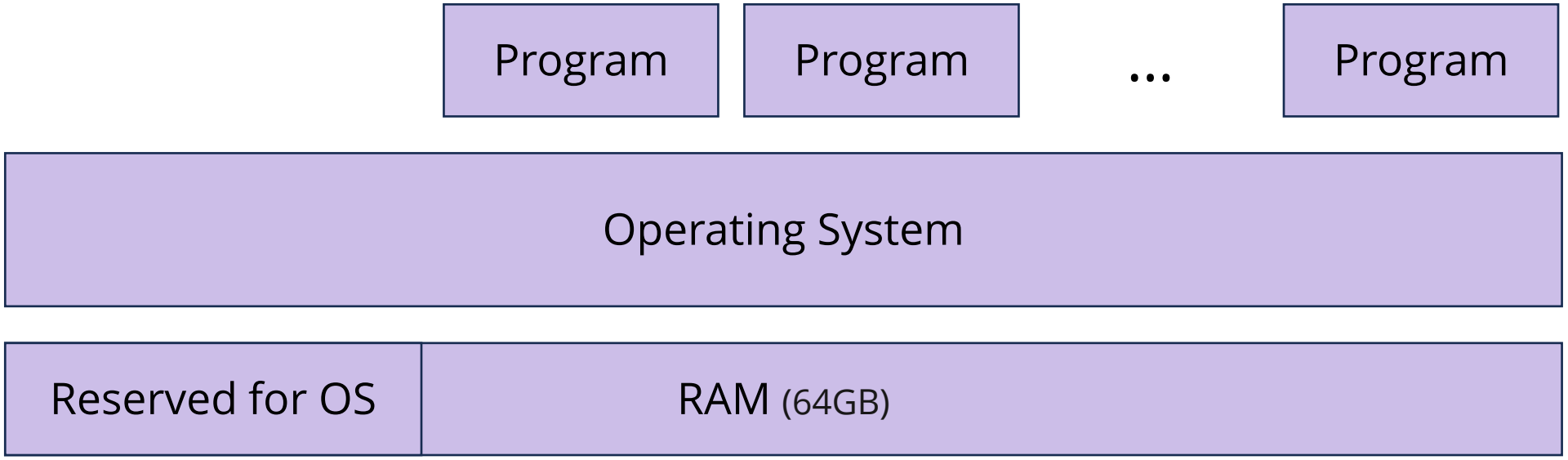
Compilation



REVIEW

OS manages many programs

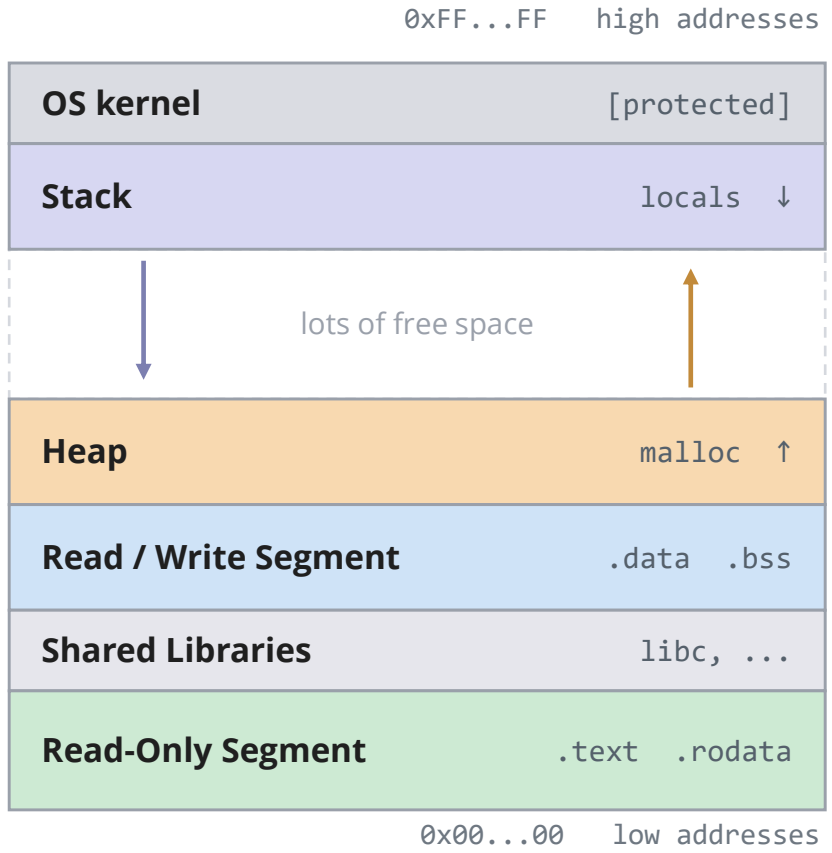
The operating system (OS) decides which programs get to run. It gives every process the illusion of its own private memory (i.e. a virtual address space).



REVIEW

Memory model during execution

```
1  int g_limit = 10;       // .data  (init global)
2  int g_count;            // .bss   (zero global)
3  const char* k = "hi";   // ptr .data, "hi" .rodata
4
5  int main(int argc, char* argv[]) {
6    int local = 5;        // stack
7    int* buf =            // buf -> stack
8        malloc(40);       // *buf -> heap
9    free(buf);
10   return EXIT_SUCCESS;
11 }
```

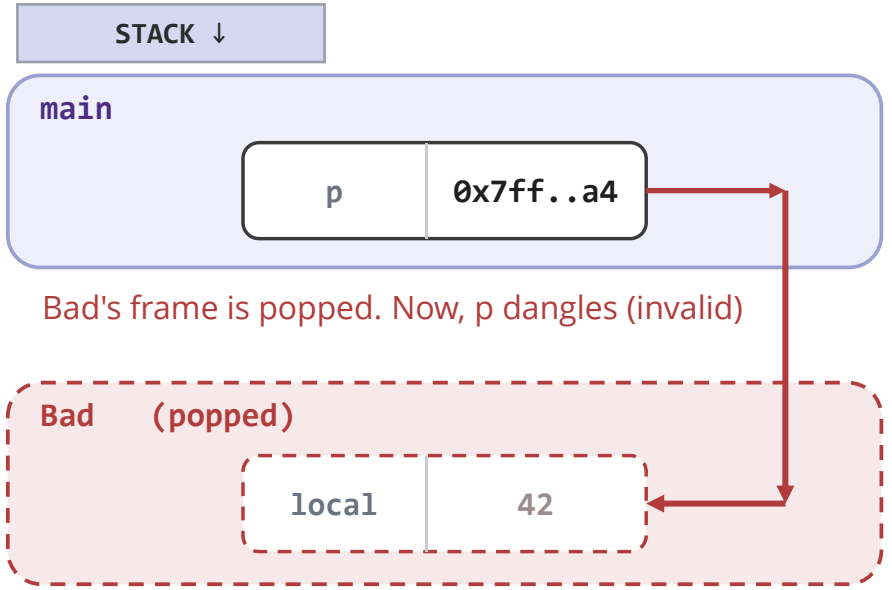


Summary: four regions, four lifetimes

Region	What lives there	Lifetime	Who manages it	Watch out for
`Stack`	locals, parameters, call bookkeeping	per function call	automatic (compiler)	dangling pointer to a local
`Heap`	malloc'd blocks	malloc → free	YOU, by hand	leaks, use-after-free
`.data / .bss`	globals & statics	whole program	automatic (loader)	shared mutable state
`.text / .rodata`	code + literals	whole program	read-only (loader)	writing a literal => segfault

Stack: dangling pointer to a local

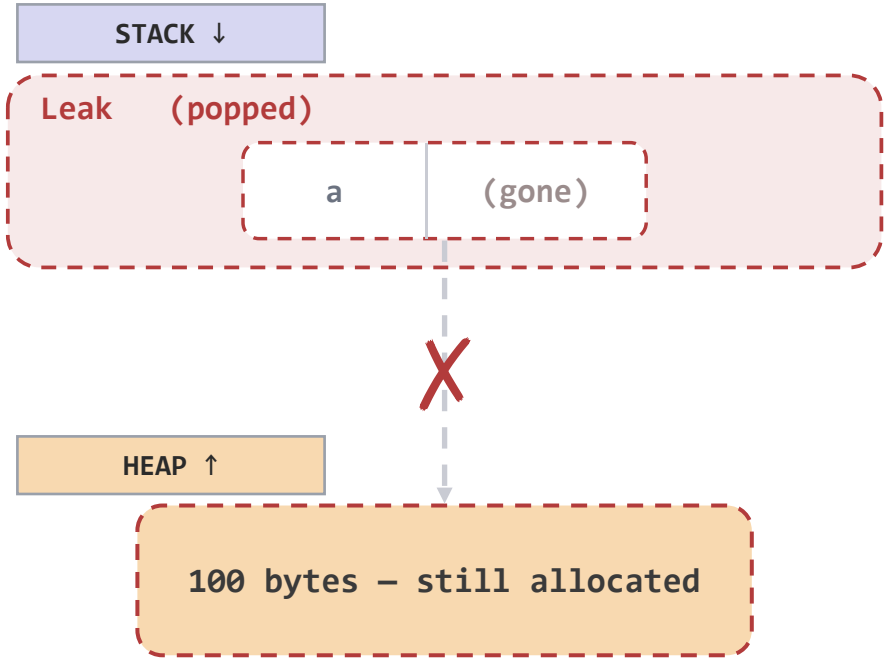
```
1  int* Bad(void) {
2    int local = 42;
3    return &local;   // address of a local!
4  }
5
6  int main(int argc, char* argv[]) {
7    int* p = Bad();
8    printf("%d\n", *p);  // reads dead memory
9    return EXIT_SUCCESS;
10 }
```



Takeaway: Never return the address of a local. Its frame is popped on return, so the pointer dangles. Return by value, or malloc the storage so it lives on the heap.

Heap: memory leak

```
1  void Leak(void) {
2    int* a = malloc(100);  // 100 bytes
3    // ... use a ...
4  }  // a disappears; bytes never freed
5
6  int main(int argc, char* argv[]) {
7    Leak();
8    // 100 bytes now unreachable
9    return EXIT_SUCCESS;
10 }
```

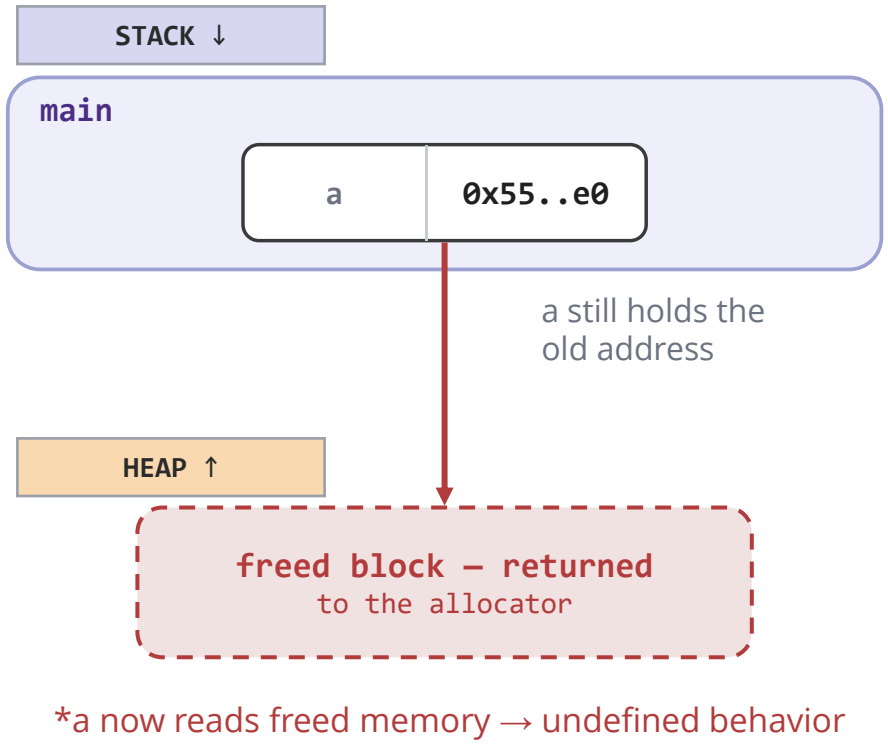


Leak returned: a is gone, block never freed → leak

Takeaway: A leak is heap memory you can no longer reach. When the last pointer to a block is lost without free(), those bytes are stuck until the program exits. valgrind finds these.

Heap: use-after-free

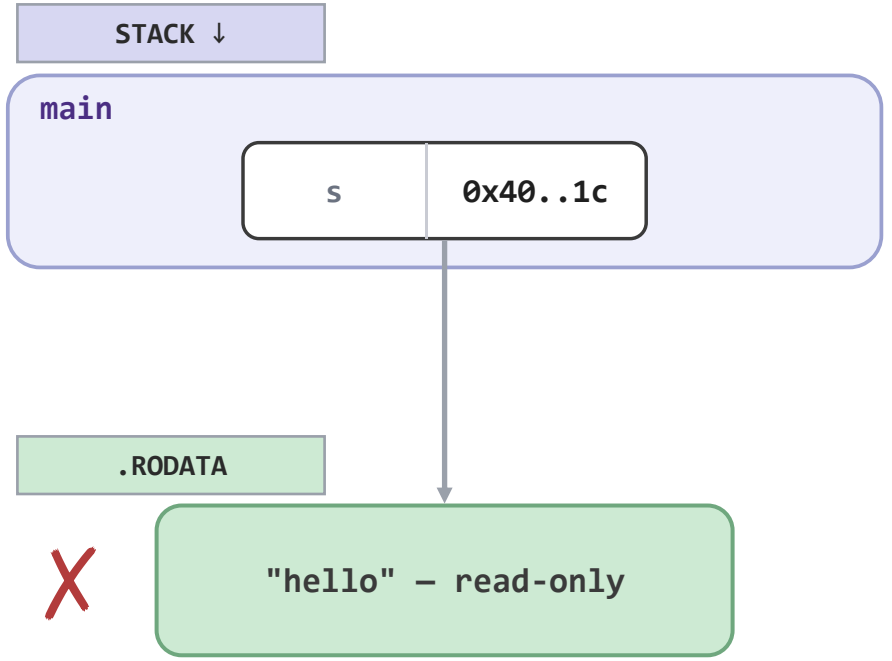
```
1  int main(int argc, char* argv[]) {
2    int* a = malloc(sizeof(int));
3    *a = 5;
4    free(a);             // block returned
5    printf("%d\n", *a);  // use after free!
6    // free(a);          // (double free!)
7    return EXIT_SUCCESS;
8  }
```



Takeaway: After free(a), the block is gone but a still points at it. Reading or writing *a (or freeing again) is undefined behavior. Set a = NULL right after freeing.

Read-only: writing a string literal

```
1  int main(int argc, char* argv[]) {
2    char* s = "hello";  // points into .rodata
3    printf("%s\n", s);  // reading is fine
4    s[0] = 'H';         // writing is NOT
5    return EXIT_SUCCESS;  // crash: SIGSEGV
6  }
7
8  // Fix: char s[] = "hello";  // writable copy
9  //                            // on the stack
```



s[0] = 'H' writes read-only memory → SIGSEGV

Takeaway: A string literal lives in read-only memory; s points at it, so s[0] = 'H' faults. Use char s[] = "hello" to get a writable copy on the stack.

Types

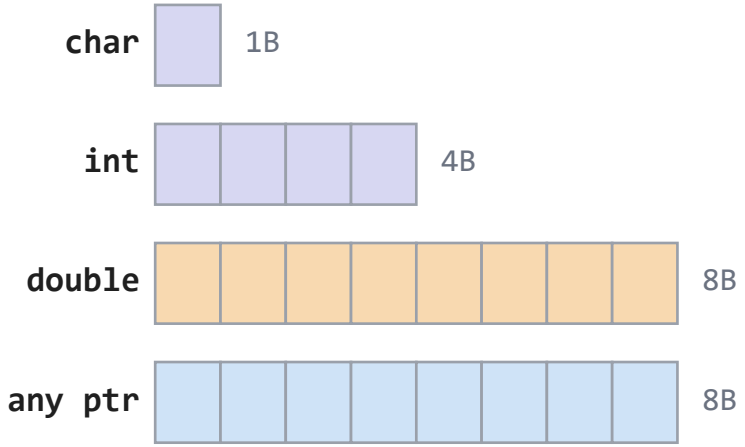
Primitive Types

```
1  int g = 1;                 // ?
2
3  int main(...) {
4    char*   x = "hi";        // ?
5    char    y = '!';         // ?
6    float*  z = malloc(4);   // ?
7
8    int*  a = &x;            // ?
9    char* d = z;             // ?
10 }
```

How big is each variable?

Sizes: pointers are all 8 bytes

```
1  int g = 1;                  // 4  (.data)
2
3  int main(...) {
4    char*   x = "hi";         // 8  ptr
5    char    y = '!';          // 1
6    float*  z = malloc(4);    // 8  ptr
7
8    int*  a = &x;             // 8  ptr
9    char* d = z;              // 8  ptr
10 }
```



Takeaway: Sizes vary, but every pointer is 8 bytes on 64-bit: it only holds an address. Measure anything with sizeof.

sizeof: measured at compile time

```
1  sizeof(int)        // 4
2  sizeof(double)     // 8
3  sizeof(p)          // size of p's type
4
5  int a[10];
6  sizeof(a)          // 40, the whole array
```

- Technically, its an operator, not a function!
- The compiler replaces it with a constant.
- Result type is **size_t**; print with %zu.

Fun Fact:

Since it is a compile time expression, the measured statement is not run so sizeof(x++) never increments x.

Sizes and printf at a glance

type	machine	size	printf
`char`	any	1	`%c / %d`
`int`	any	4	`%d`
`long`	Windows / Linux	4 or 8	`%ld`
`double`	any	8	`%f`
`T* (pointer)`	64-bit	8	`%p`
`size_t`	64-bit	8	`%zu`

yellow = varies by platform

Fixed widths: <stdint.h>

```c
#include <stdint.h>
#include <inttypes.h>  // PRI macros

uint8_t  b = 200;   // exactly 1 byte
int32_t  n = 42;    // exactly 4 bytes
uint64_t big = ...; // exactly 8 bytes

printf("%" PRId32 "\n", n);
printf("%u\n", b);  // promoted to int
```

- Names read as **[u]int<bits>_t**.
- Same size on every machine: files, networks, hardware.
- You meet **uint8_t** first in Exercise 2, not HW1.
- Print fixed widths with **PRId32 / PRIu64**.

Takeaway: stdint.h gives exact-size integers (uint8_t, int32_t, uint64_t). Use them when the byte count has to be guaranteed.

Sizes and printf at a glance

type	machine	size	printf
`char`	any	1	`%c / %d`
`int`	any	4	`%d`
`long`	Windows / Linux	4 or 8	`%ld`
`double`	any	8	`%f`
`T* (pointer)`	64-bit	8	`%p`
`size_t`	64-bit	8	`%zu`
`int32_t`	any	4	`%d or %ld`
`uint64_t`	any	8	`%lu or %llu`

yellow = varies by platform

Sizes and printf at a glance

type	machine	size	printf
`char`	any	1	`%c / %d`
`int`	any	4	`%d`
`long`	Windows / Linux	4 or 8	`%ld`
`double`	any	8	`%f`
`T* (pointer)`	64-bit	8	`%p`
`size_t`	64-bit	8	`%zu`
`int32_t`	any	4	`PRId32`
`uint64_t`	any	8	`PRIu64`

yellow = varies by platform

Signed vs unsigned: where do the bytes go?

```
1  uint8_t b = 250;      // 1 byte, 0..255
2  b = b + 10;
3  printf("%u\n", b);    // ?
4
5  int8_t s = 100;       // -128..127
6  s = s + 50;           // ?
   printf("%u\n", b);    // ?
```

- How many bits is one byte?
- What is the max a uint8_t can hold?
- What happens when you go past it?

Fixed width means values wrap

```c
uint8_t b = 250;
b = b + 10;           // wraps: 260 % 256
printf("%u\n", b);    // 4

int8_t s = 100;
s = s + 50;           // signed overflow = UB
```

8 bits = 256 values (0..255)

typedef: a nickname for a type

```c
// typedef gives an existing type a new name
typedef uint32_t  Color;     // alias
typedef char*     String;    // alias

Color bg = 0xFF00FF;
String s = "hi";

// most useful for structs / fn pointers
typedef struct ll LinkedList;  // HW1
typedef void (*FreeFn)(void* p);
```

- Renames a type; no new bytes, same size.
- Hides ugly syntax
- HW1 uses **LinkedList** and payload typedefs.
- Read it like a declaration: last word is the new name.

void*: address of anything

```c
void* p;                // 8 bytes, no type
int n = 5;
p = &n;                 // any pointer fits

// *p;                  // error: unknown size
int* ip = (int*) p;
printf("%d\n", *ip);    // 5

```

- Holds any address but has no element size.
- Cannot dereference until you cast it.
- Still 8 bytes, like all pointers.
- HW1 payload: **typedef void* LLPayload_t**.

void*: we've seen it before

```
void* malloc(size_t);
```

That's why we always cast malloc!

Using void*: not enough by itself

```
1  void Zero(void* p) {
2    // *p = 0;     error: how many bytes?
3    // p[1]  = 0;  error: step size?
4  }
5
6  int  a[4];
7  char b[100];
8  Zero(a);
9
```

- void* can point at a or b, any type.
- But the function has no element type.
- It cannot deref or index: unknown size.
- How many bytes should Zero clear?

Pair void* with a size (or count)

```
1  void Zero(void* p, size_t n) {
2    char* c = (char*) p;   // byte view
3    for (size_t i = 0; i < n; i++) {
4      c[i] = 0;
5    }
6  }
7  Zero(a, sizeof(a));    // 16 bytes
8  Zero(b, sizeof(b));    // 100 bytes
9
```

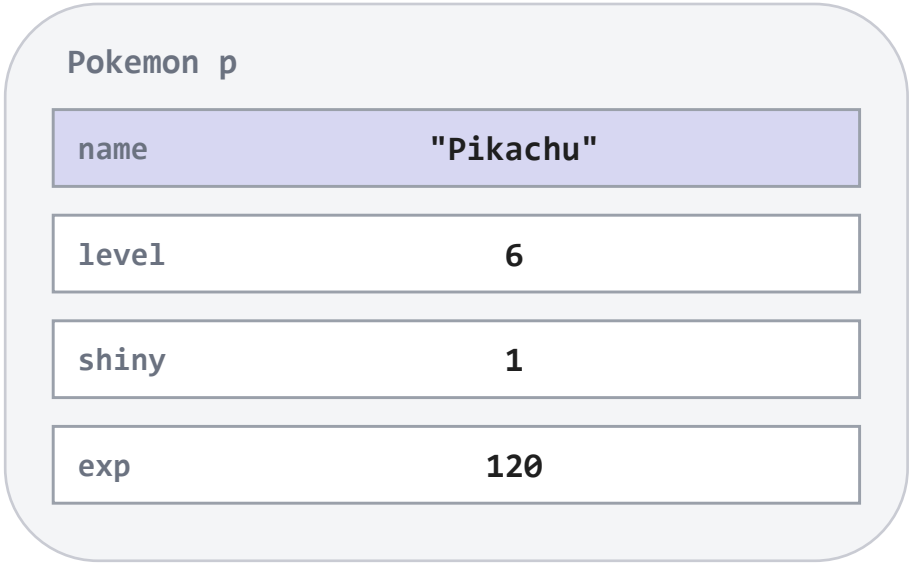
- Caller passes how many bytes.
- Cast to char* to step byte by byte.
- Now it works for any type.
- Real APIs do exactly this.

Structs

Group related fields into one type

```c
typedef struct {
  char* name;    // -> read-only str
  int   level;
  char  shiny;   // 0 or 1
  int   exp;
} Pokemon;

Pokemon p = {"Pikachu", 5, 1, 120};
p.level += 1;
```



- One value bundles four fields.
- Reach each with a `.`

How big is a Pokemon?

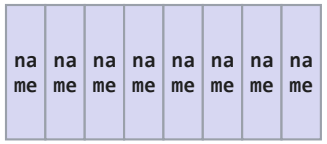
```
1  typedef struct {
2    char* name;   // ?
3    int   level;  // ?
4    char  shiny;  // ?
5    int   exp;    // ?
6  } Pokemon;
7
8  sizeof(Pokemon) == ?   // ?
```

- Add up the fields. What total?
- Could the compiler waste bytes?
- Why might that even help?

Alignment: why sizeof is 24 (1/4)

```
1  char* name;   // offset 0  (8)
2  int   level;  // offset 8  (4)
3  char  shiny;  // offset 12 (1)
4  int   exp;    // offset 16 (4)
5
6  sizeof(Pokemon) == 24;
```

bytes 0 .. 23

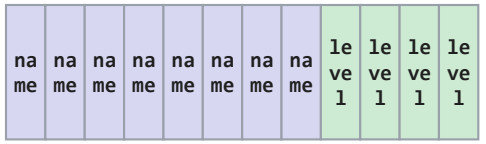


- name (pointer) takes offsets 0..7.

Alignment: why sizeof is 24 (2/4)

```
1  char* name;   // offset 0  (8)
2  int   level;  // offset 8  (4)
3  char  shiny;  // offset 12 (1)
4  int   exp;    // offset 16 (4)
5
6  sizeof(Pokemon) == 24;
```

bytes 0 .. 23

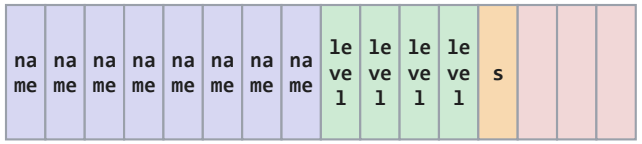


- name (pointer) takes offsets 0..7.
- level at 8 (already a multiple of 4).

Alignment: why sizeof is 24 (3/4)

```
1  char* name;   // offset 0  (8)
2  int   level;  // offset 8  (4)
3  char  shiny;  // offset 12 (1)
4  int   exp;    // offset 16 (4)
5
6  sizeof(Pokemon) == 24;
```

bytes 0 .. 23

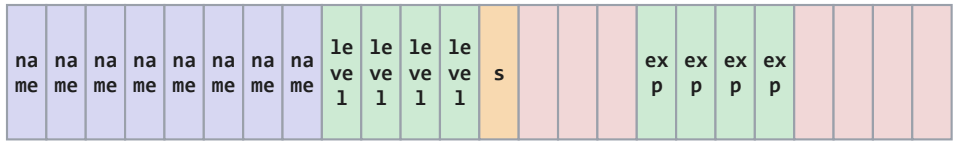


- name (pointer) takes offsets 0..7.
- level at 8 (already a multiple of 4).
- shiny at 12; exp needs a multiple of 4, so 3 pad bytes -> exp at 16.

Alignment: why sizeof is 24 (4/4)

```
1  char* name;   // offset 0  (8)
2  int   level;  // offset 8  (4)
3  char  shiny;  // offset 12 (1)
4  int   exp;    // offset 16 (4)
5
6  sizeof(Pokemon) == 24;
```

bytes 0 .. 23

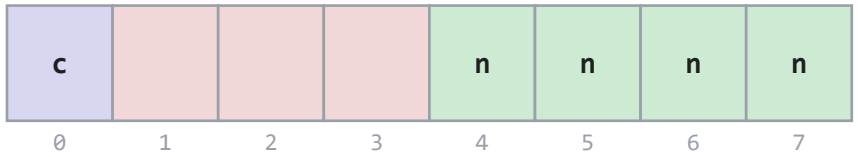


- ❖ name (pointer) takes offsets 0..7.
- ❖ level at 8 (already a multiple of 4).
- ❖ shiny at 12; exp needs a multiple of 4, so 3 pad bytes -> exp at 16.
- ❖ round total up to a multiple of 8: 24.

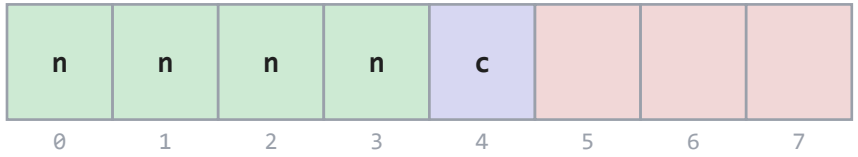
Takeaway: Two rules: (1) members must start at a multiple of their padding and (2) the overall struct must be a multiple of its largest member

Order changes where padding goes

```
{ char c; int n; }   // 8 bytes
```




```
{ int n; char c; }   // 8 bytes
```



- char c is always 1 byte; the int needs offset 0, 4, 8…
- Top: c at 0, 3 pad bytes, n at 4.
- Bottom: n at 0, c at 4, 3 pad bytes at the tail.
- Same fields, same 8 bytes, padding just moves.
  - With 3+ fields, order CAN change the total.

Will this level up the caller's Pokemon?

```
1  void LevelUp(Pokemon mon) {
2    mon.level += 1;
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(p);
6  // p.level == ?
7
```

Pass-by-value copies the whole struct (1/3)

```
1  void LevelUp(Pokemon mon) {
2    mon.level += 1;
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(p);
```

main: p

name	0x40..
level	5
shiny	1
exp	120

LevelUp: mon (copy)

name	0x40..
level	5
shiny	1
exp	120

Pass-by-value copies the whole struct (2/3)

```
1  void LevelUp(Pokemon mon) {
2    mon.level += 1;
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(p);
```

main: p

name	0x40..
level	5
shiny	1
exp	120

LevelUp: mon (copy)

name	0x40..	
level	5	→ 6
shiny	1	
exp	120	

Pass-by-value copies the whole struct (3/3)

```
1  void LevelUp(Pokemon mon) {
2    mon.level += 1;
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(p);
```

main: p

name	0x40..
level	5
shiny	1
exp	120

Takeaway: LevelUp changed its own local copy because everything in C is pass-by-value!

Pointer: one struct, edited in place (1/4)

```
1  void LevelUp(Pokemon* mon) {
2    mon->level += 1;   // -> not .
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(&p);
```

main: p

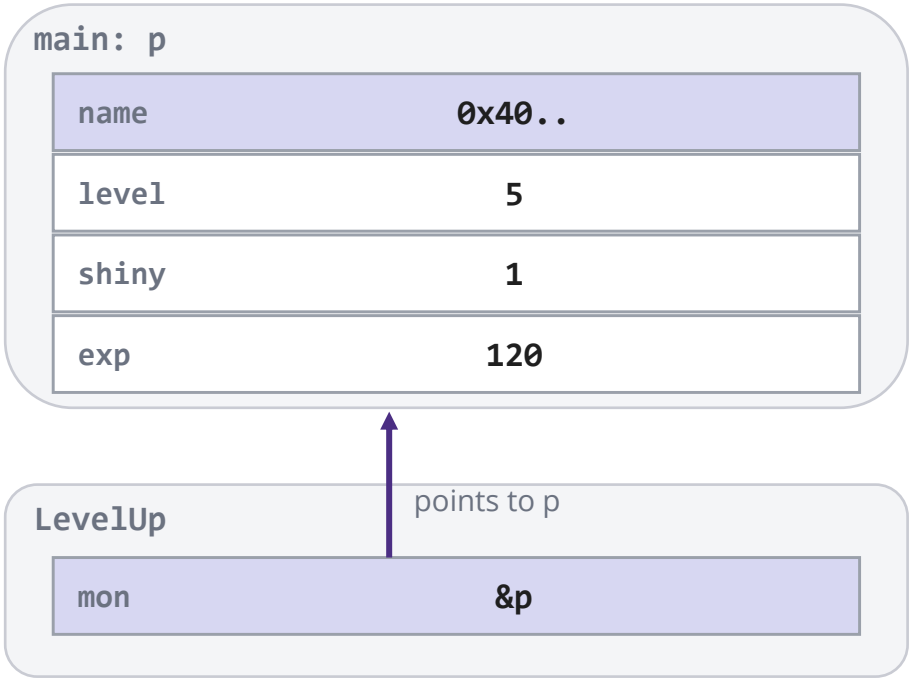
name	0x40..
level	5
shiny	1
exp	120

LevelUp

mon	?

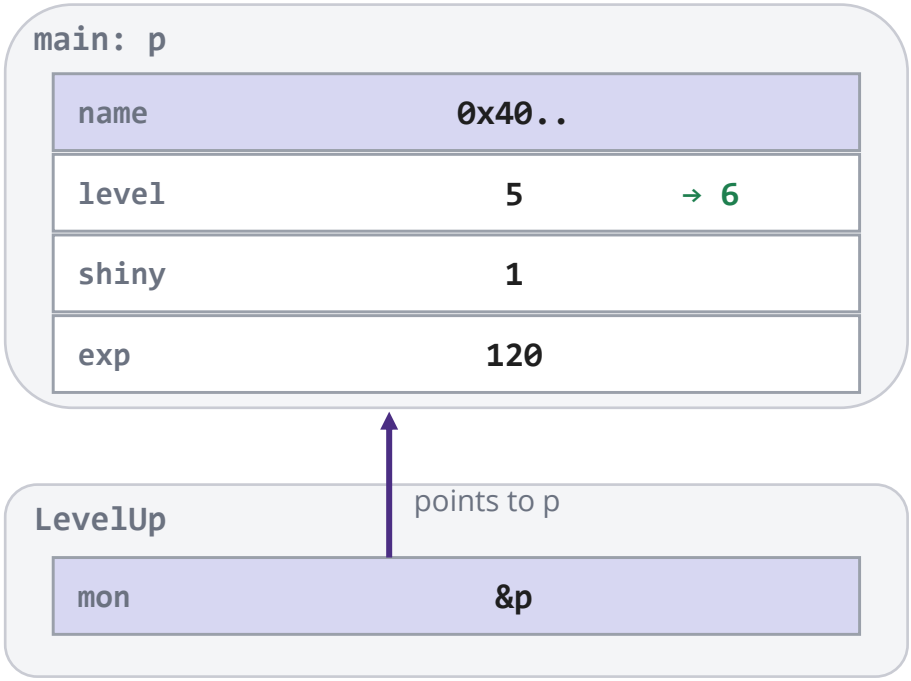
Pointer: one struct, edited in place (2/4)

```
1  void LevelUp(Pokemon* mon) {
2    mon->level += 1;   // -> not .
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(&p);
```



Pointer: one struct, edited in place (3/4)

```
1  void LevelUp(Pokemon* mon) {
2    mon->level += 1;   // -> not .
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(&p);
```



Pointer: one struct, edited in place (4/4)

```
1  void LevelUp(Pokemon* mon) {
2    mon->level += 1;   // -> not .
3  }
4  Pokemon p = {"Pikachu", 5, 1, 120};
5  LevelUp(&p);
```

main: p

name	0x40..
level	6
shiny	1
exp	120

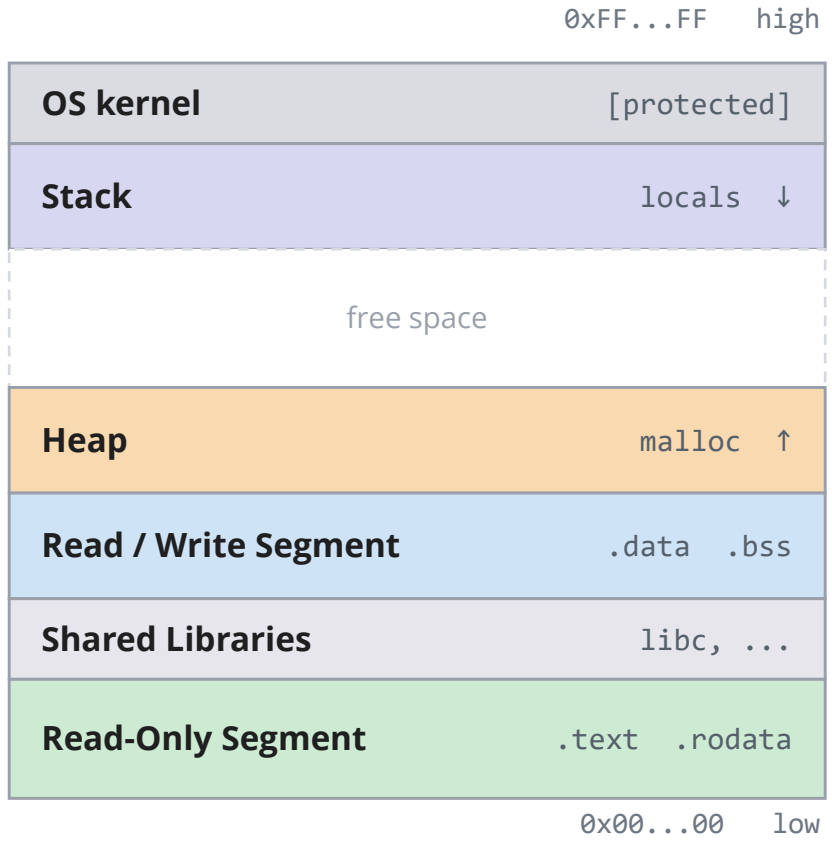
LevelUp

mon	?

Function Pointers

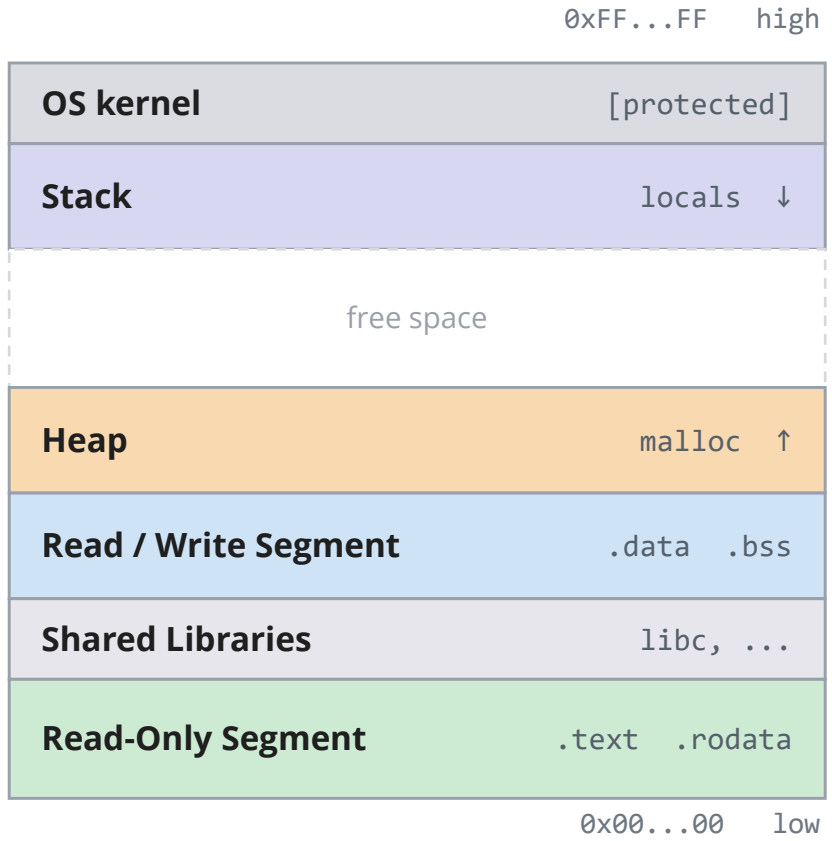
A pointer can point anywhere in memory

```
1  int g = 1;                    // ?
2
3  int main(...) {
4    char*    x = "hi";          // ?
5    uint8_t  y = 5;             // ?
6    float*   z = malloc(4);     // ?
7
8    int* a = &x;
9    int* b = y;
10   int* c = &g;
11   char* d = z;
12 }
```



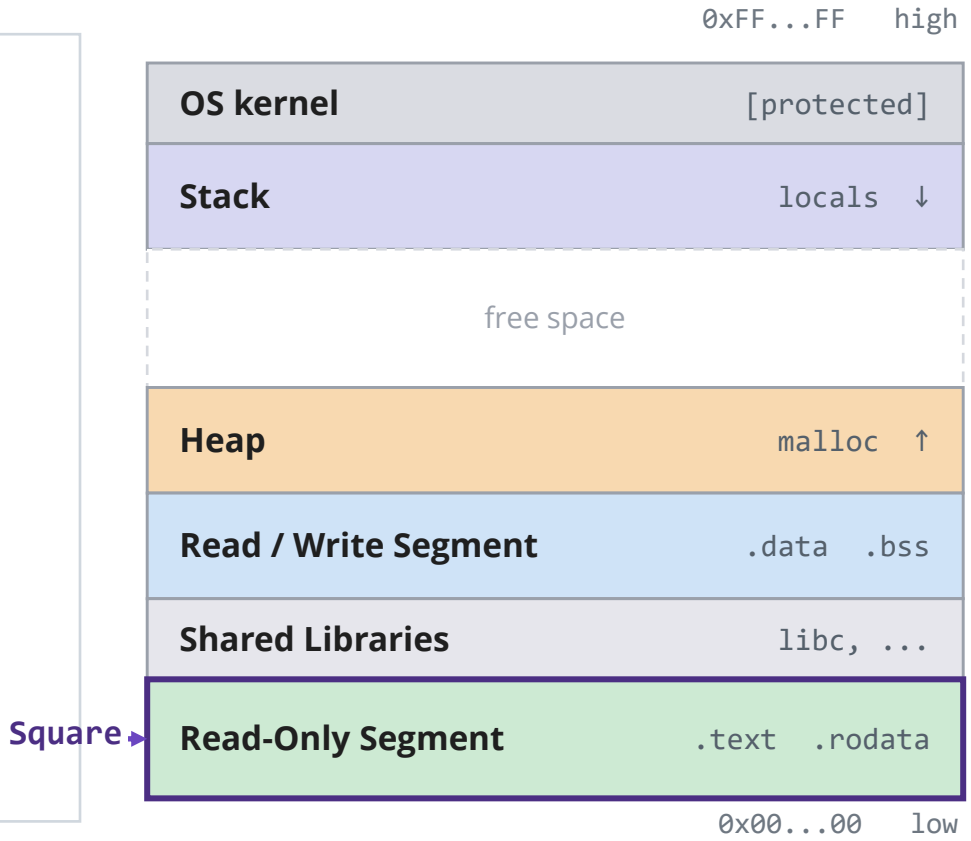
A pointer can point anywhere in memory

```
1  int g = 1;                    // read-write
2
3  int main(...) {
4    char*    x = "hi";          // read-only
5    uint8_t  y = 5;             // stack
6    float*   z = malloc(4);     // heap
7
8    int* a = &x;
9    int* b = y;
10   int* c = &g;
11   char* d = z;
12 }
```



Code lives in the read-only segment

```
1  int Square(int n) {
2    return n * n;
3  }
4
5  // Square names that block of code.
6  printf("%p\n", Square);   // 0x401136
```



Takeaway: A function is just compiled instructions in read-only memory. Its name is the address of that code, so we can take a pointer to it.

How to read the declaration

```
  4       2  1        3
int  (*op)(int, int)  = Add;
```

1) op is the name.

2) (*op): it is a pointer.

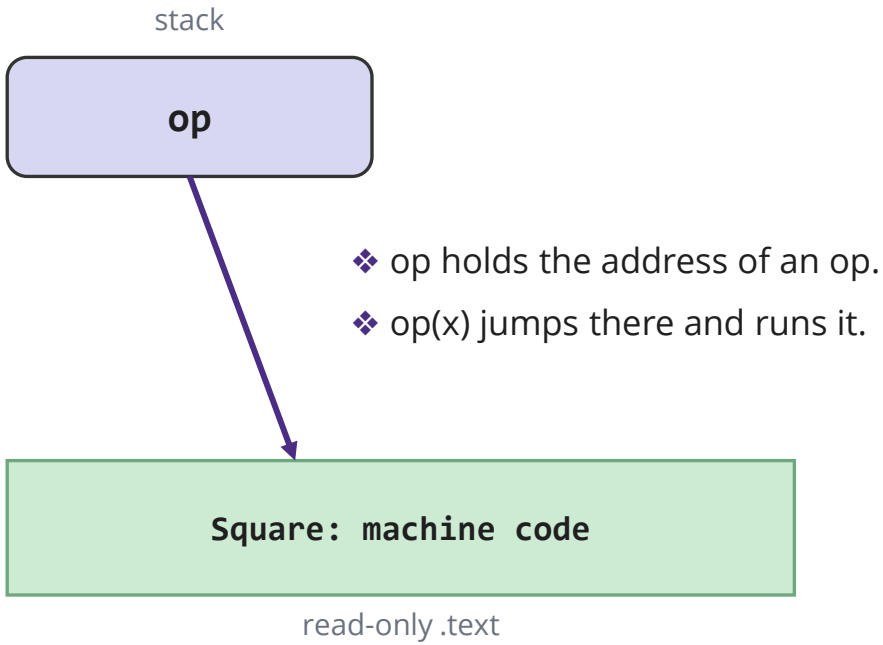
3) (int, int): points to a function taking two ints.

4) int: which returns int.

Note: Parentheses around *op are required. What happens if we don't have them?

Pass behavior in: one Map, many ops

```
1  int Square(int n) { return n * n; }
2  int Negate(int n) { return -n; }
3
4  void Map(int a[], int len, int (*op)(int)) {
5    for (int i = 0; i < len; i++) {
6      a[i] = op(a[i]);
7    }
8  }
9
10 Map(arr, 4, Square);  // {1,2,3,4}->{1,4,9,16}
   Map(arr, 4, Negate);  // same loop, new op
```



- op holds the address of an op.
- op(x) jumps there and runs it.

Pass behavior in: one Map, many ops

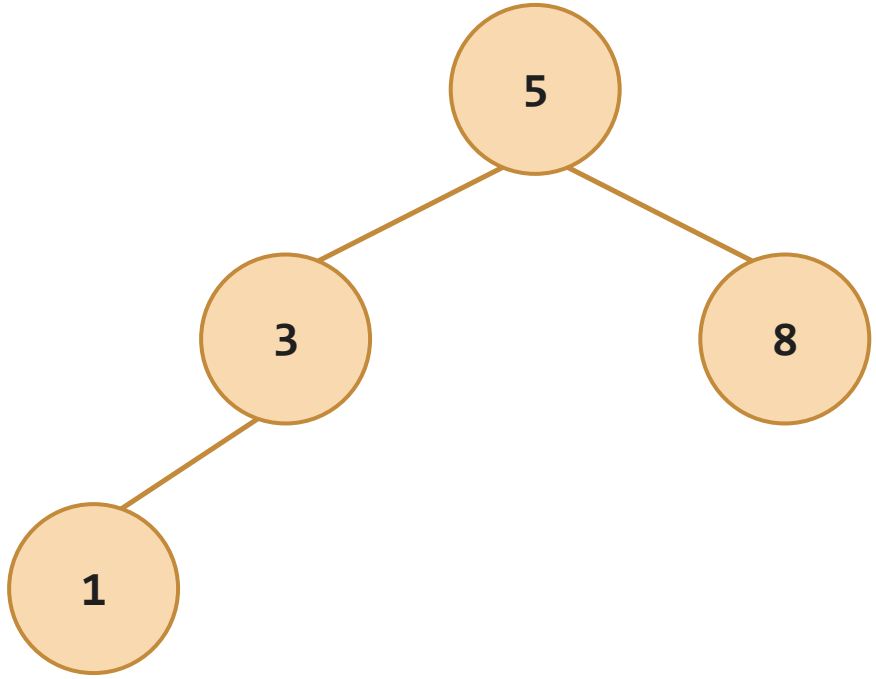
```
1  int Square(int n) { return n * n; }
2  int Negate(int n) { return -n; }
3
4  typedef int (*IntFunction)(int* x);
5
6  void Map(int a[], int len, IntFunction op) {
7    for (int i = 0; i < len; i++) {
8      a[i] = op(a[i]);
9    }
10 }
11
12 Map(arr, 4, Square);  // {1,2,3,4}->{1,4,9,16}
13 Map(arr, 4, Negate);  // same loop, new op
```

- Can use typedef to create a named function pointer type
- Syntax is a little weirder than other typedef – so be careful!

Complex Structs

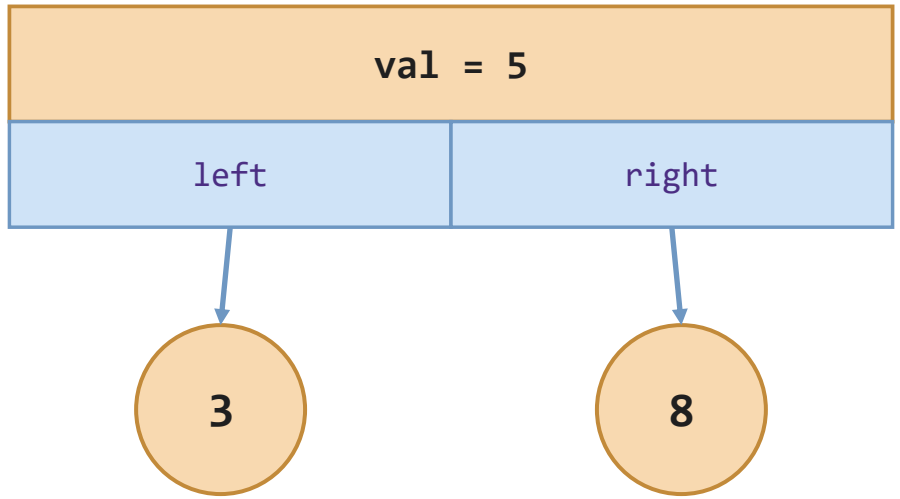
Why a binary search tree?

- We want ordered data with fast lookup.
- Each node keeps a value and two children.
- Smaller values go left, larger go right.
- Search halves the work at each step.
- Built from structs linked by pointers.



A node points to its own kind

```
1  typedef struct node {
2    int          val;
3    struct node* left;
4    struct node* right;
5  } TreeNode;
```



A node bundles a value with two pointers to more nodes. It cannot hold a TreeNode, but a TreeNode* is fine and fixed at 8 bytes.

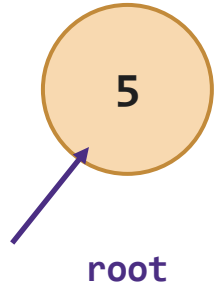
NULL means empty or no child

```
1  #include <stdlib.h>   // defines NULL
2
3  TreeNode* root = NULL;    // empty tree
4  if (root == NULL) ... // nothing here
5  leaf->left  = NULL;   // no child
6
```

- NULL is the address 0: points at nothing.
- Not built in: comes from a header.
- Empty tree and missing children are NULL.
- Check before you **->**: NULL deref crashes.

Build one node on the heap

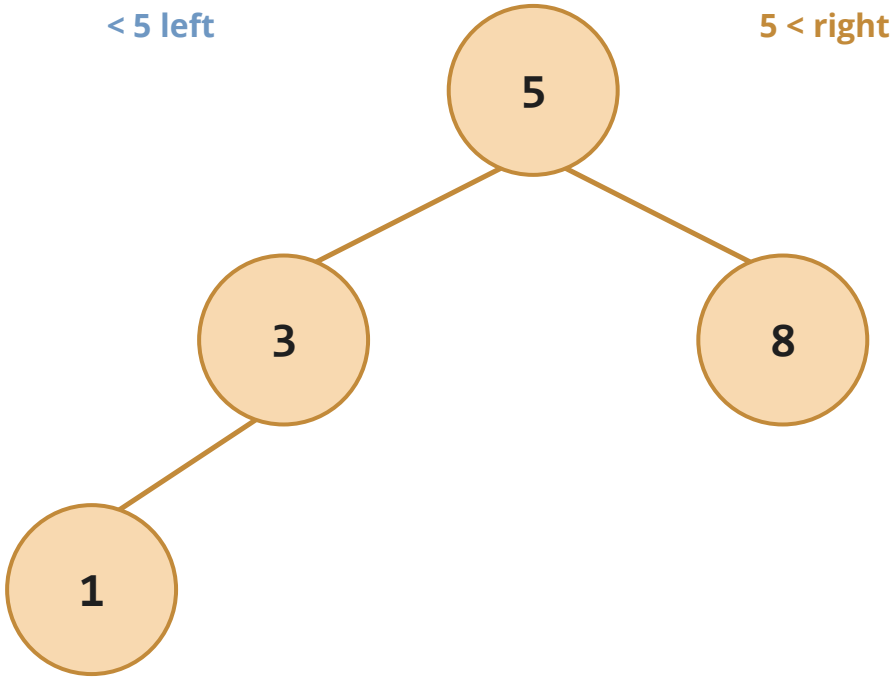
```
1  TreeNode* MakeNode(int v) {
2    TreeNode* n = malloc(sizeof(TreeNode));
3    n->val   = v;
4    n->left  = NULL;
5    n->right = NULL;
6    return n;            // outlives call
7  }
8
9  // client:
10 TreeNode* root = MakeNode(5);
```



Takeaway: Allocate nodes on the heap so they outlive the call, reach fields with ->, and pair every malloc with a free.

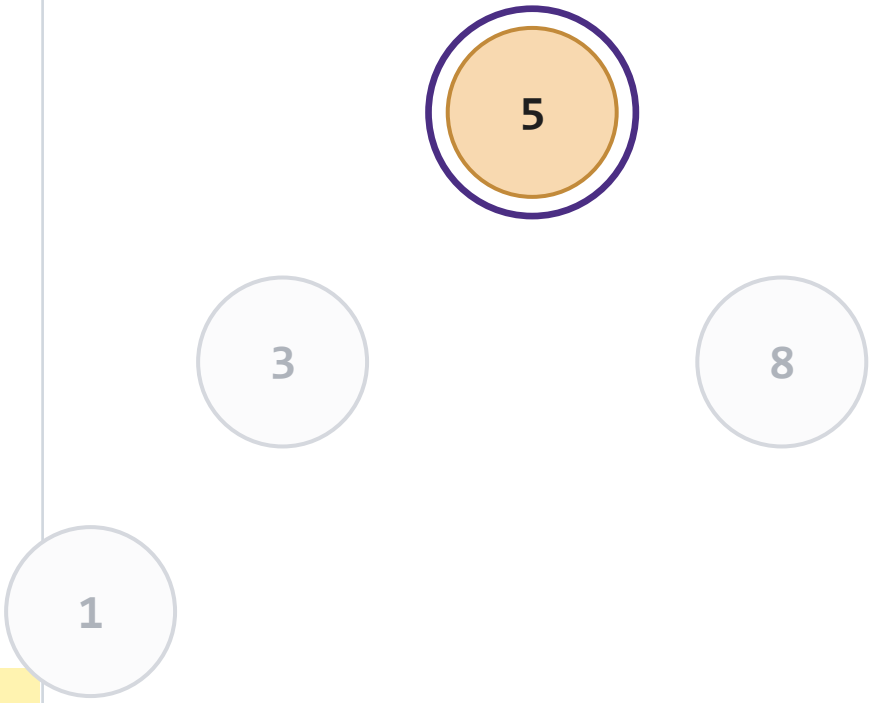
The ordering rule

- For every node: left subtree < node < right subtree.
- So smaller keys live left, larger keys live right.
- To find a key, compare and step one way.
- To insert, do the same walk, then attach.



Insert 5, 3, 8, 1 (1/4)

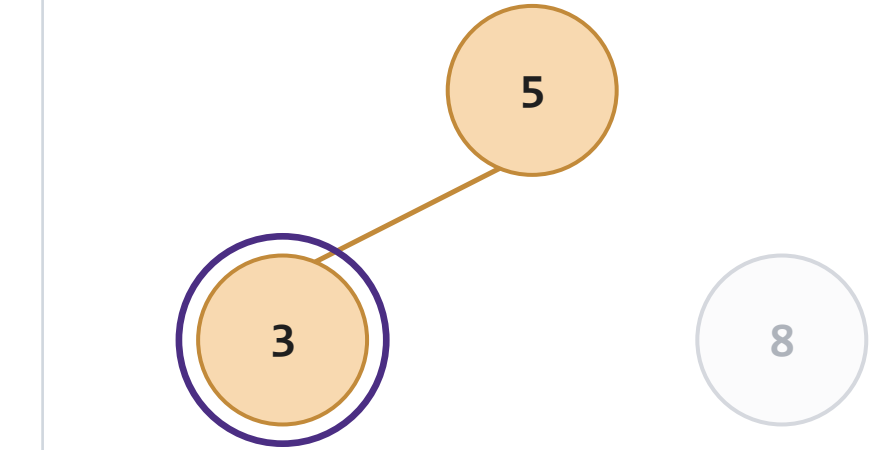
```
1  TreeNode* Insert(TreeNode* root, int v) {
2    if (root == NULL) {
3      return MakeNode(v);          // empty spot
4    }
5    if (v < root->val) {
6      root->left  = Insert(root->left,  v);
7    } else {
8      root->right = Insert(root->right, v);
9    }
10   return root;
11 }
12
13 root = Insert(root, 5);  // reassign root
```



5: tree empty -> new root

Insert 5, 3, 8, 1 (2/4)

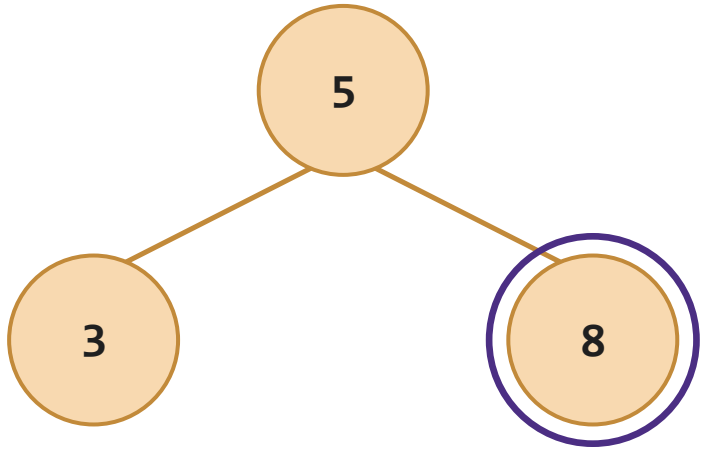
```
1  TreeNode* Insert(TreeNode* root, int v) {
2    if (root == NULL) {
3      return MakeNode(v);          // empty spot
4    }
5    if (v < root->val) {
6      root->left  = Insert(root->left,  v);
7    } else {
8      root->right = Insert(root->right, v);
9    }
10   return root;
11 }
12
13 root = Insert(root, 5);  // reassign root
```



3 < 5 -> left of 5

Insert 5, 3, 8, 1 (3/4)

```
1  TreeNode* Insert(TreeNode* root, int v) {
2    if (root == NULL) {
3      return MakeNode(v);          // empty spot
4    }
5    if (v < root->val) {
6      root->left  = Insert(root->left,  v);
7    } else {
8      root->right = Insert(root->right, v);
9    }
10   return root;
11 }
12
13 root = Insert(root, 5);  // reassign root
```

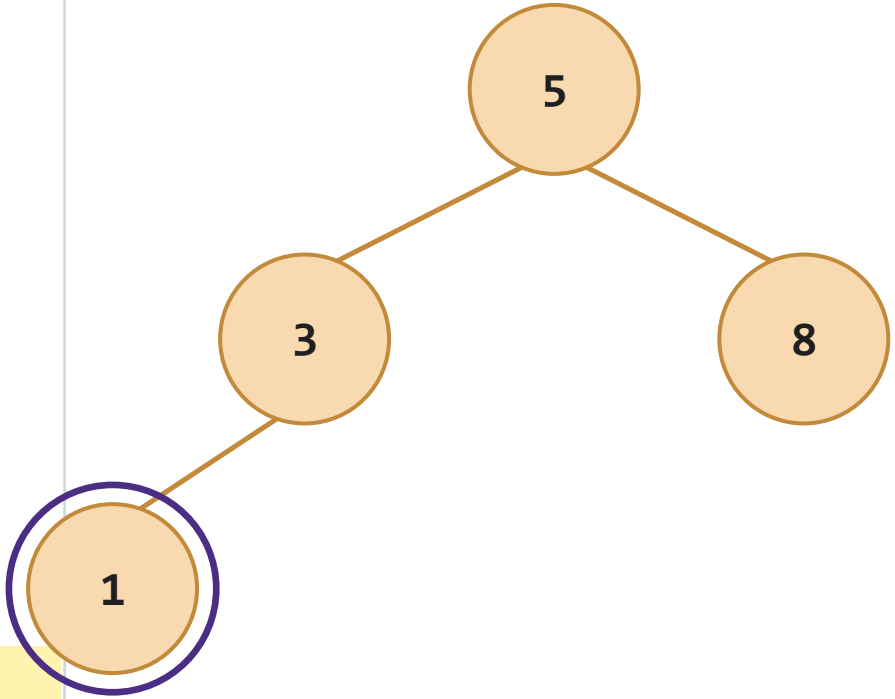


1

8 > 5 -> right of 5

Insert 5, 3, 8, 1 (4/4)

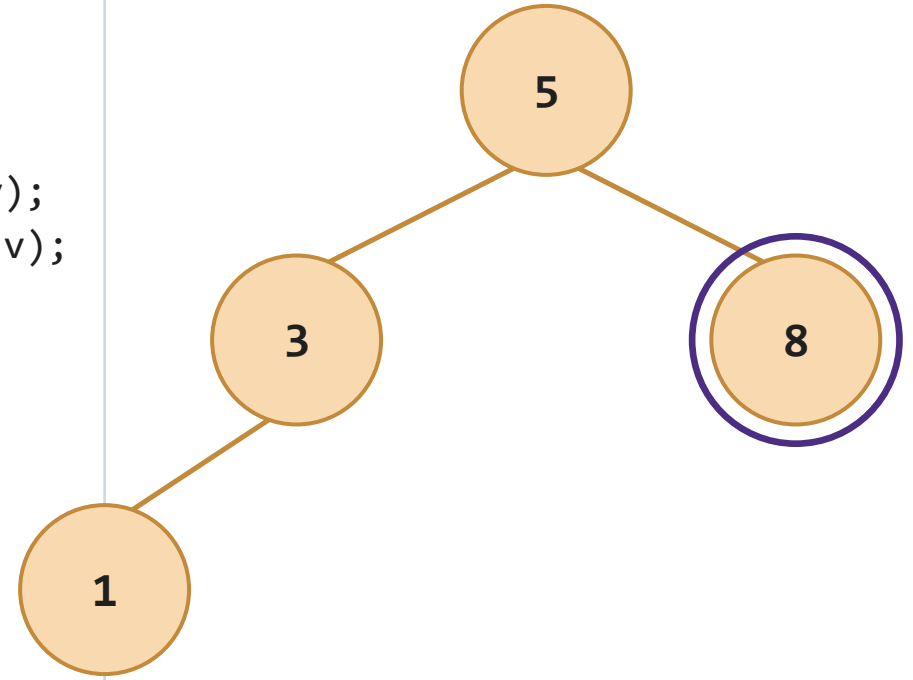
```
1  TreeNode* Insert(TreeNode* root, int v) {
2    if (root == NULL) {
3      return MakeNode(v);          // empty spot
4    }
5    if (v < root->val) {
6      root->left  = Insert(root->left,  v);
7    } else {
8      root->right = Insert(root->right, v);
9    }
10   return root;
11 }
12
13 root = Insert(root, 5);  // reassign root
```



1 < 5, 1 < 3 -> left of 3

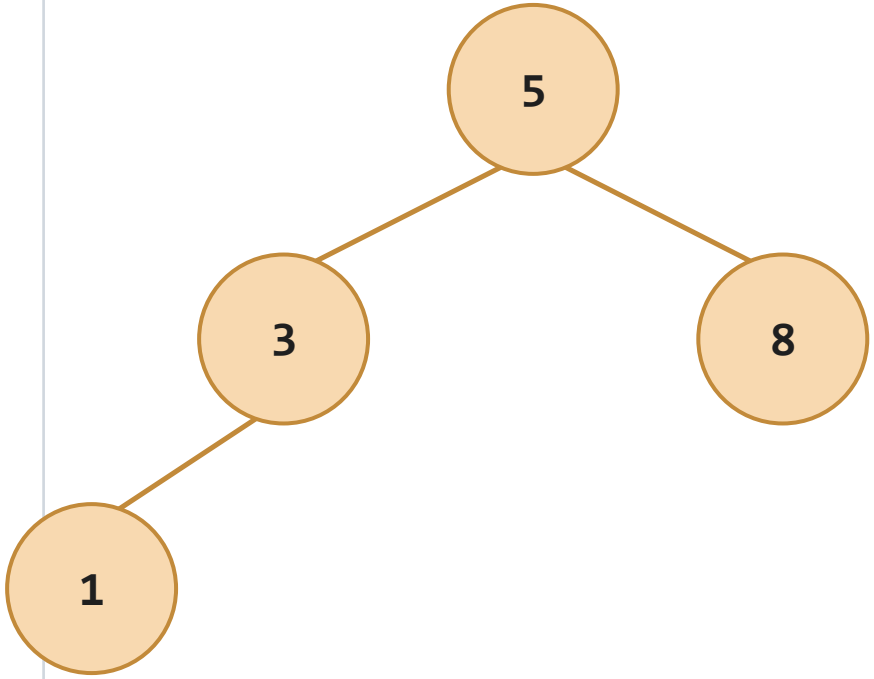
Searching a BST

```
1  bool Contains(TreeNode* n, int v) {
2    if (n == NULL)     return false;
3    if (v == n->val)   return true;
4    if (v <  n->val)   return Contains(n->left, v);
5    else               return Contains(n->right, v);
6  }
7
8  // client:
10 Contains(root, 8);   // returns true
```



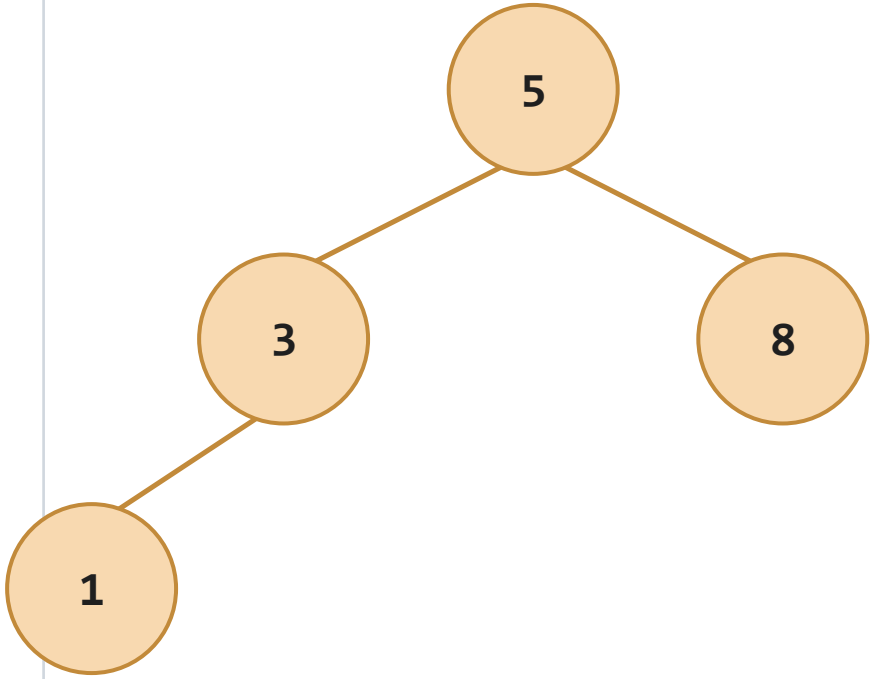
Free the tree! (uh oh…)

```
1  void Free(TreeNode* n) {
2    if (n == NULL) return;
3    free(n);
4    Free(n->left);
5    Free(n->right);
6  }
7
8  // client:
9  Free(root);
10 root = NULL;
```



Free the tree!

```
1  void Free(TreeNode* n) {
2    if (n == NULL) return;
3    Free(n->left);
4    Free(n->right);
5    free(n);
6  }
7
8  // client:
9  Free(root);
10 root = NULL;
```



Generic: the HW1 LinkedList pattern

```
1  typedef void*  Payload;          // any type
2  typedef int  (*Cmp)(Payload, Payload);
3
4  typedef struct node {
5    Payload      data;             // void*
6    struct node* left, * right;
7  } Node;
8
9  Insert(&root, item, cmp);  // caller's order
10 Free(root, free_fn);       // caller frees
```

- **void*** payload stores any type.
- Caller passes compare + free fns.
- Same nodes hold ints or strings.
- This is HW1's **LinkedList**.

Takeaway: Structs hold data, pointers link them, void* and function pointers make it work for any type. That is HW1's LinkedList.

Reminders

Assessments:

- Exercise 1 feedback is out!
- Exercise 2 due on Thursday at 11:59pm
- Homework 1 released today. Due next Thursday at 11:59pm
  - Much more challenging than the exercises!

General:

- Exams are scheduled
  - Midterm (7/27) from 9:40am to 10:40am in IEB G106
  - Final exam (8/18) from 1:30pm to 3:30pm in CSE2 G10
- I *think* I've scheduled all alternate exams. Let me know ASAP if you still need an alternate.

Questions?

Thanks for coming - see you next lecture!