# CS 241: Systems Programming
# Lecture 14. Pointers and Arrays

Spring 2020

Prof. Stephen Checkoway

# Arrays in Java

Arrays in Java are normal `Objects` created with new

```
int[] arr = new int[100];
```

They're indexed from `0` to `arr.length-1`

Attempts to access out of bounds elements leads to `ArrayIndexOutOfBoundsExceptions`

They can be passed to functions or returned from function

# Arrays in C

```c
int arr1[100];                  // Fixed-size array
double arr2[x];                 // Variable-sized array
unsigned char arr3[x][y][z];    // Multi-dimensional array
```

Arrays are indexed from 0 to one less than their bound

- Arrays don't keep track of their length
- Accessing an array outside its bound is undefined behavior:

> — An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression **`a[1][7]`** given the declaration **`int a[4][5]`**) (6.5.6).

Arrays cannot be returned from functions (but can sort of be passed to them)

# Initializing arrays

Like all other variables in C, arrays need to be initialized

- Exception: global variables are initialized to all zeros

Fixed-sized arrays can be initialized with an **initializer**

- `int a[5] = { 0 };` // same as { 0, 0, 0, 0, 0 }
- `int b[5] = { 1, 2, 3 };` // same as { 1, 2, 3, 0, 0 }
- `int c[] = { 1, 2, 3 };` // b has length 3
- `int d[5] = { [3] = 1, [4] = 2, [0] = 3 };`
  // same as { 3, 0, 0, 1, 2 }
- `int e[] = { [3] = 1, [0] = 3 };` // same as { 3, 0, 0, 1 }

Variable-sized arrays cannot be initialized with an initializer

Which of the following defines an array of four integers with the 0th element set to 5?

A. `int arr[4] = { 5, 4, 3, 2, 1 };`

B. `int arr[] = { 5 };`

C. `int arr[4] = { [5] = 0 };`

D. `int arr[4] = { [0] = 5, [4] = 3 };`

E. `int arr[4] = { [0] = 5, [3] = 2 };`

# Aside about style

# Aside about style

Using multiple lines can improve readability

- But do it only when it does (it probably doesn't here)

```
int a[] = {
  37,
  42, // Trailing commas are fine
};
```

# Aside about style

Using multiple lines can improve readability

- But do it only when it does (it probably doesn't here)

```c
int a[] = {
  37,
  42, // Trailing commas are fine
};
```

Explicit indices in the initializer, like [3] = 5, can help

- Use them when readability is improved

```c
int a[] = {
  [0] = 37,
  [1] = 42,
};
```

# Initializing a variable sized array

```c
// Option 1. Loop over each element and assign it a value
void foo(size_t count) {
  int arr[count];
  for (size_t idx = 0; idx < count; ++idx)
    arr[idx] = 0;
  // ...
}

// Option 2. Use memset() from string.h
#include <string.h>
void bar(size_t count) {
  int arr[count];
  memset(arr, 0, sizeof arr);
  // ...
}
```

# Size and length of an array

# Size and length of an array

For arrays that are **not** function parameters, e.g.,

```
int a[5];
int b[x];
```

we can use `sizeof` to get the size (in bytes) and length

# Size and length of an array

For arrays that are **not** function parameters, e.g.,

```
int a[5];
int b[x];
```

we can use `sizeof` to get the size (in bytes) and length

- Size

```
size_t size1 = sizeof a; // 5 * sizeof(int)
size_t size2 = sizeof b; // x * sizeof(int)
```

# Size and length of an array

For arrays that are **not** function parameters, e.g.,

```
int a[5];
int b[x];
```

we can use `sizeof` to get the size (in bytes) and length

- Size

```
size_t size1 = sizeof a; // 5 * sizeof(int)
size_t size2 = sizeof b; // x * sizeof(int)
```

- Length

```
size_t len1 = sizeof a / sizeof a[0];
  // size1 / sizeof(int) = 5
size_t len2 = sizeof b / sizeof b[0];
  // size2 / sizeof(int) = x
```

# Function parameters

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

void make_identity(size_t n, double arr[n][n]) {
  for (size_t row = 0; row < n; ++row) {
    for (size_t col = 0; col < n; ++col) {
      arr[row][col] = (row == col ? 1.0 : 0.0);
    }
  }
}
int main(int argc, char *argv[argc]) {
  size_t dim = (argc > 1 ? atoi(argv[1]) : 2);
  double ident[dim][dim]; // Danger of crashing with large dim!

  make_identity(dim, ident);
  for (size_t row = 0; row < dim; ++row) {
    for (size_t col = 0; col < dim; ++col) {
      printf("%.1f ", ident[row][col]);
    }
    putchar('\n');
  }
  return 0;
}
```

# Function parameters

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

void make_identity(size_t n, double arr[n][n]) {
  for (size_t row = 0; row < n; ++row) {
    for (size_t col = 0; col < n; ++col) {
      arr[row][col] = (row == col ? 1.0 : 0.0);
    }
  }
}
int main(int argc, char *argv[argc]) {
  size_t dim = (argc > 1 ? atoi(argv[1]) : 2);
  double ident[dim][dim]; // Danger of crashing with large dim!

  make_identity(dim, ident);
  for (size_t row = 0; row < dim; ++row) {
    for (size_t col = 0; col < dim; ++col) {
      printf("%.1f ", ident[row][col]);
    }
    putchar('\n');
  }
  return 0;
}
```

Array syntax for main

# Function parameters

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

void make_identity(size_t n, double arr[n][n]) {
  for (size_t row = 0; row < n; ++row) {
    for (size_t col = 0; col < n; ++col) {
      arr[row][col] = (row == col ? 1.0 : 0.0);
    }
  }
}
int main(int argc, char *argv[argc]) {
  size_t dim = (argc > 1 ? atoi(argv[1]) : 2);
  double ident[dim][dim]; // Danger of crashing with large dim!

  make_identity(dim, ident);
  for (size_t row = 0; row < dim; ++row) {
    for (size_t col = 0; col < dim; ++col) {
      printf("%.1f ", ident[row][col]);
    }
    putchar('\n');
  }
  return 0;
}
```

Not passed by value!
There are no array *values* in C

Array syntax for main

# Function parameters

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

void make_identity(size_t n, double arr[n][n]) {
  for (size_t row = 0; row < n; ++row) {
    for (size_t col = 0; col < n; ++col) {
      arr[row][col] = (row == col ? 1.0 : 0.0);
    }
  }
}
int main(int argc, char *argv[argc]) {
  size_t dim = (argc > 1 ? atoi(argv[1]) : 2);
  double ident[dim][dim]; // Danger of crashing with large dim!

  make_identity(dim, ident);
  for (size_t row = 0; row < dim; ++row) {
    for (size_t col = 0; col < dim; ++col) {
      printf("%.1f ", ident[row][col]);
    }
    putchar('\n');
  }
  return 0;
}
```

Not passed by value!
There are no array *values* in C

Array syntax for main

```
$ ./matrix 3
1.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
```

# C's memory model: Objects

C has a bunch of "objects" (not at all like the Java notion of an object!)

- Each object is a collection of bytes
- Every variable definition creates a new, distinct object
- Literals (e.g., the string literal "foo") are objects
- `sizeof object` — gives the size of an object
- `sizeof(type)` — gives the size of an object with type type

```
int x;
assert(sizeof x == sizeof(int));
```

# Object lifetimes

Objects have a lifetime

- Local variables live as long as they are in scope
- Global variables (including file and function static) live the whole program
- Temporary objects (returned from functions) live only until the end of the expression with the function call (we can mostly ignore these)
- We can dynamically create objects and manage their lifetimes (later)
- **Accessing an object outside its lifetime is undefined behavior**

# Object example

```
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**

```
"%u\n"
```

# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**

`"%u\n"`

`x: 2`

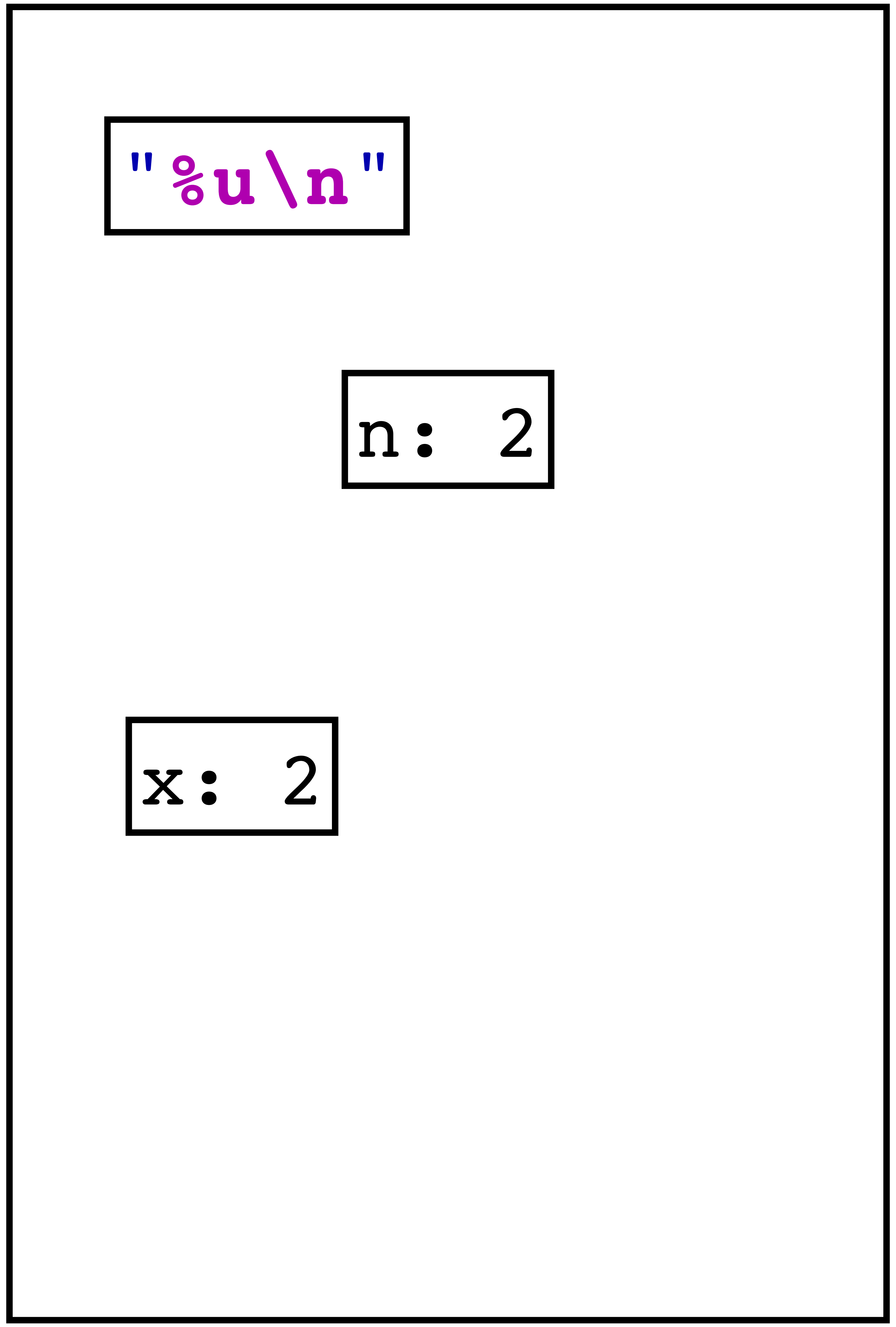
# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**



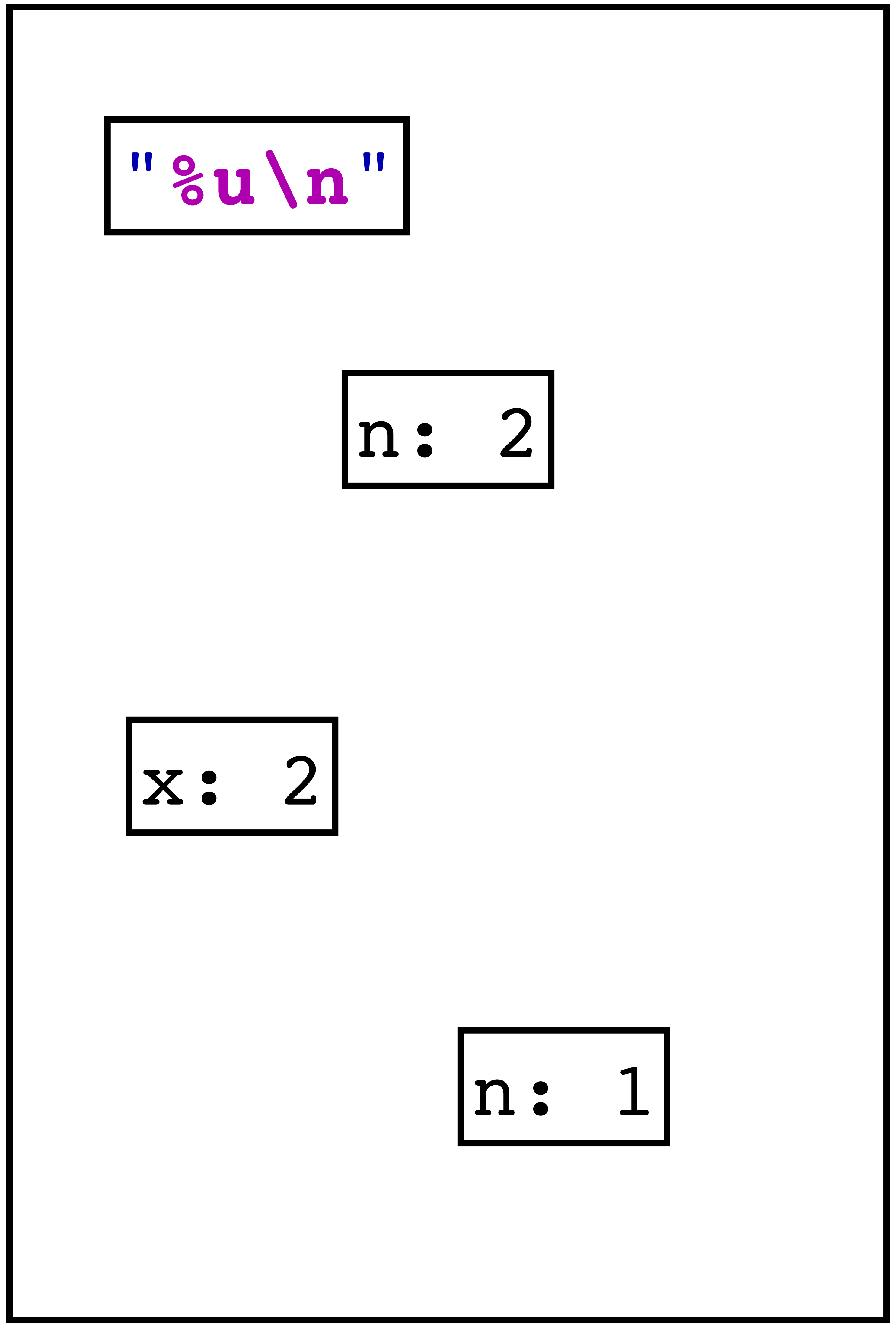
# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**



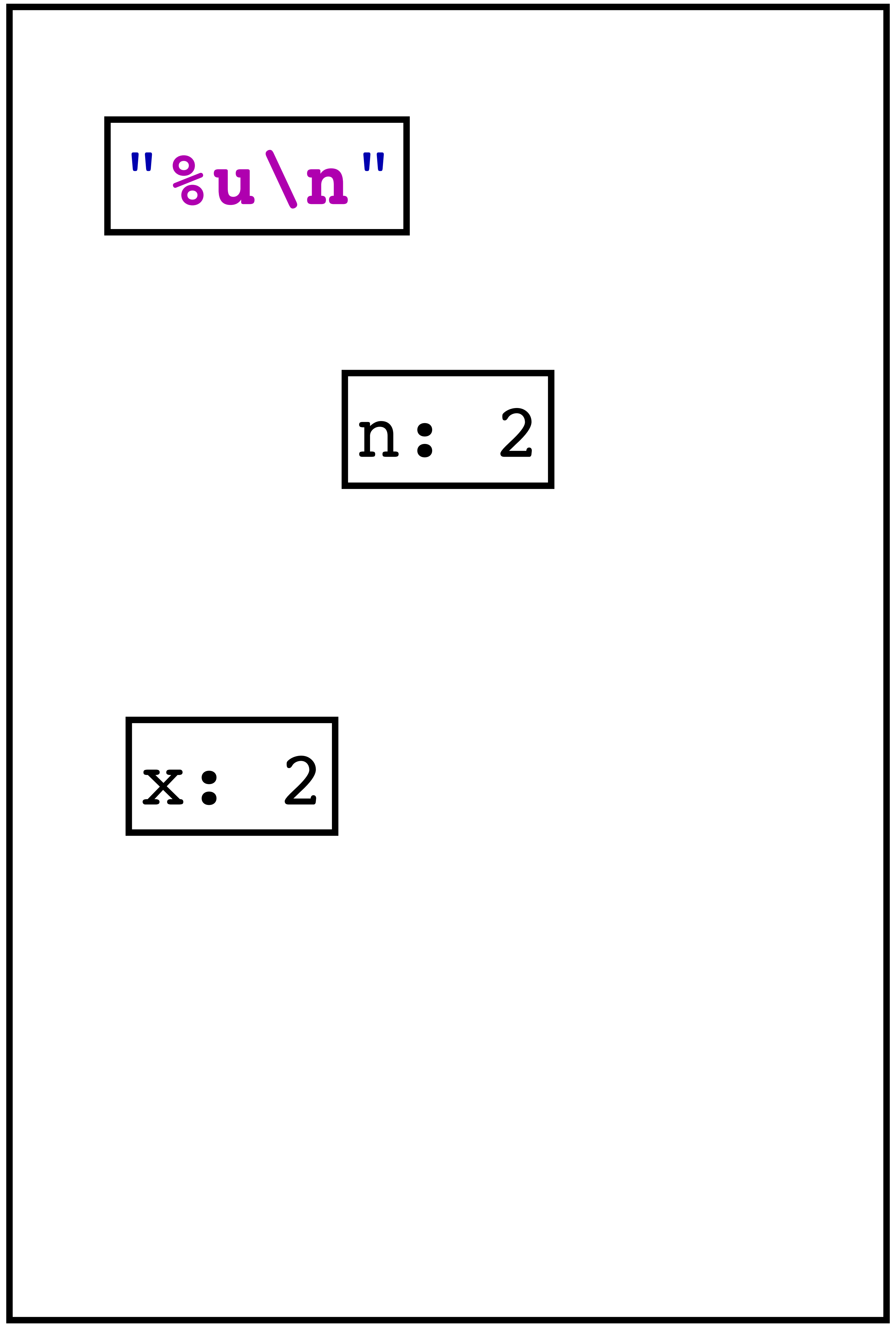
# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**



# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**

```
"%u\n"

n: 2

n: 0

x: 2
```

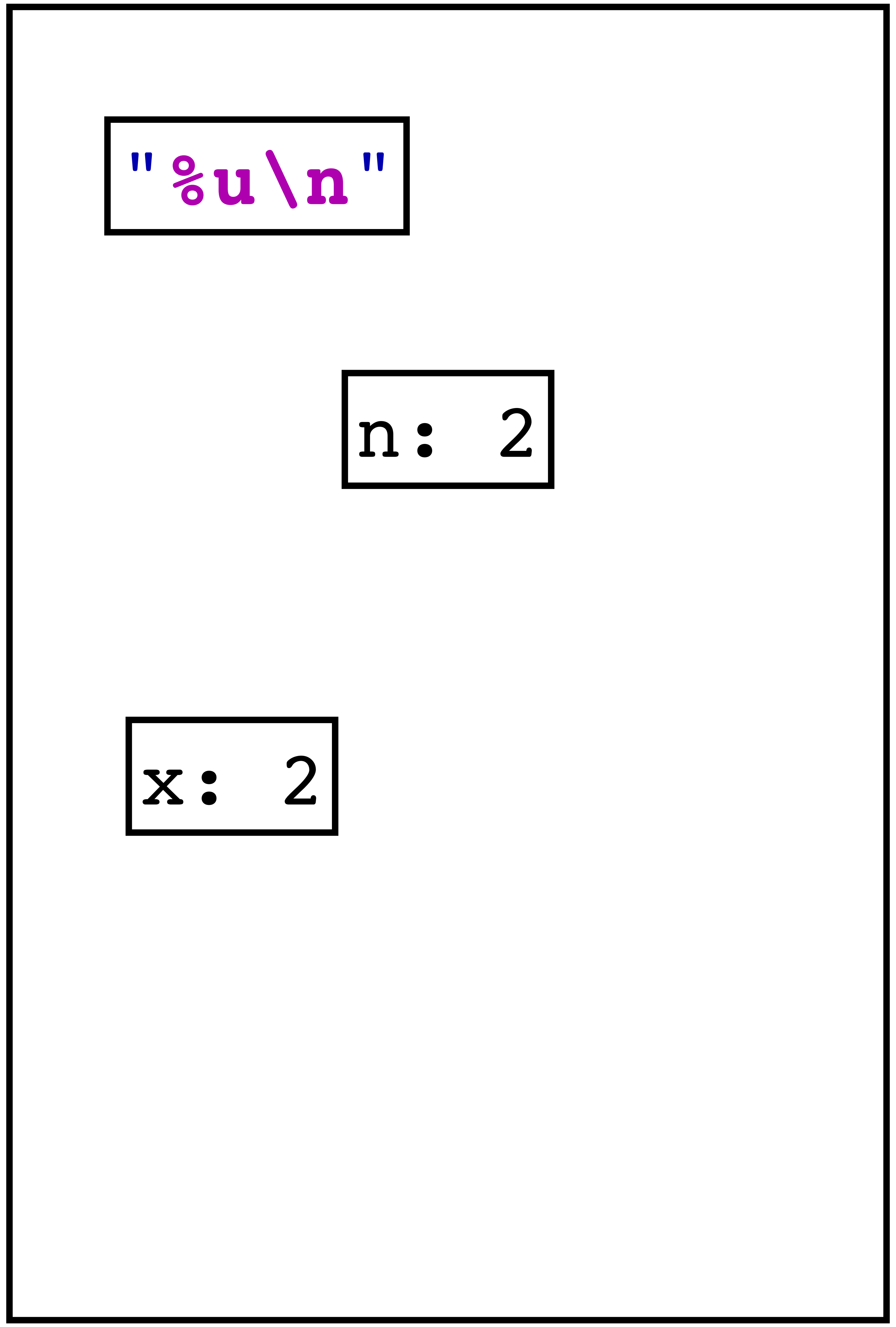
# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**



# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**

```
"%u\n"

x: 2
```

# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**

`"%u\n"`

`x: 2`

`fx: 1`

# Object example

```c
#include <stdio.h>

unsigned int slow_fib(unsigned int n) {
  if (n <= 1)
    return n;
  return slow_fib(n-1) + slow_fib(n-2);
}

int main(void) {
  unsigned int x = 2;
  unsigned int fx = slow_fib(x);
  printf("%u\n", fx);
  return 0;
}
```

**Objects**

"%u\n"

# What most machines do

Memory is a giant array of bytes (this is a lie the OS presents to applications)

- Each object lives in some contiguous sequence of bytes in this array

Some of this memory is filled with program and library code

A region of the memory, the **stack**, stores the local variables for functions

- Each function call allocates more space on the stack (called a stack frame) for its local variables

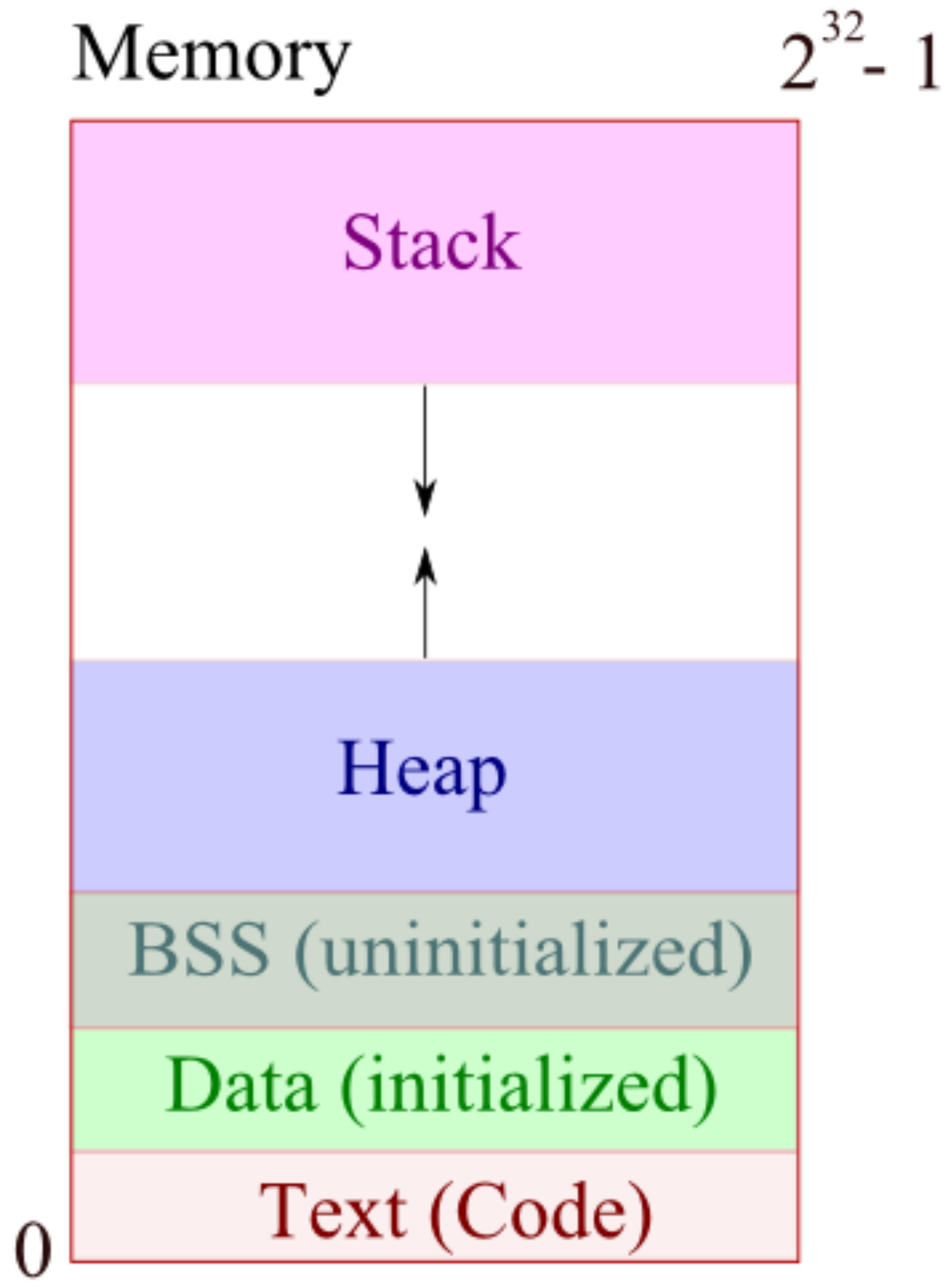
A region of the memory, the **heap**, stores dynamically created data (we'll talk more about this later)

# Memory Layout x86 (simplified)

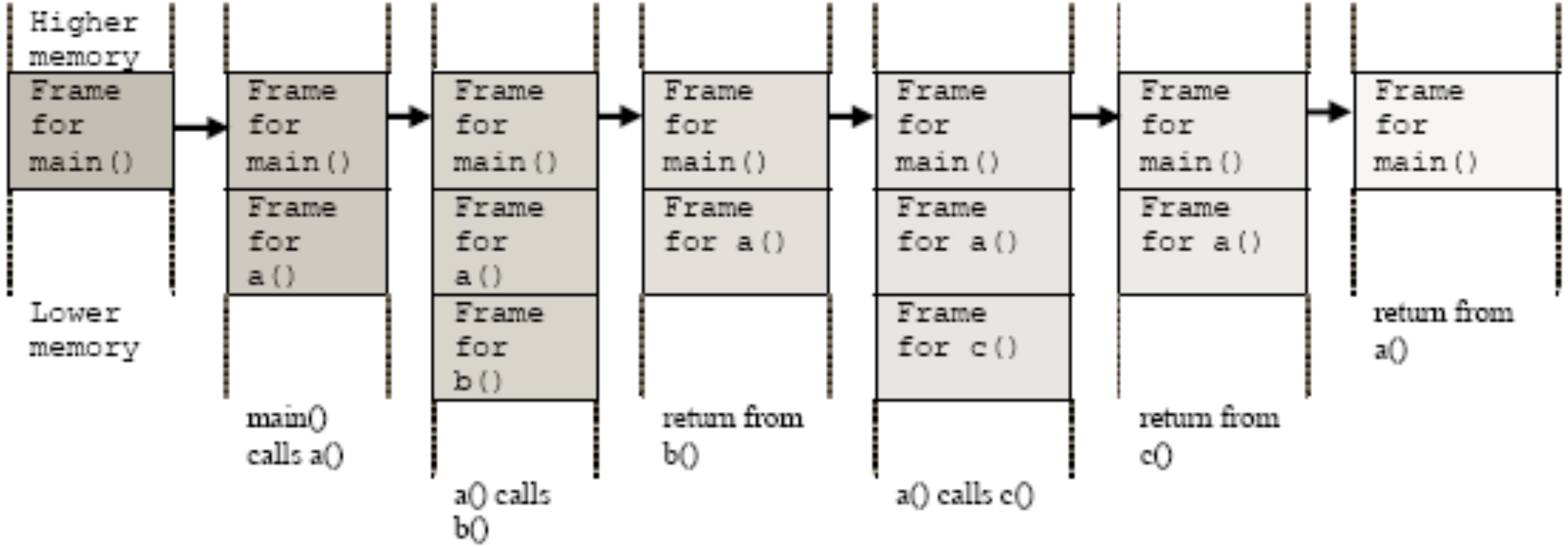
Stack and Heap grow towards each other

- Efficient use of space

Stacks grow "down" in x86 (not all do)



# Stack frames



# Object addresses

# Object addresses

Each object has an address

# Object addresses

Each object has an address

- In C, an address is just a way to refer to an object

# Object addresses

Each object has an address

- In C, an address is just a way to refer to an object
- In reality, an address is just the index into the array-of-bytes-that-is-all-of-memory of the first byte of the object

# Object addresses

Each object has an address

- In C, an address is just a way to refer to an object
- In reality, an address is just the index into the array-of-bytes-that-is-all-of-memory of the first byte of the object
- The address-of unary operator, `&`, gives the address of the object

```
int x = 37;
int y = 42;
printf("x has value %d and address %p\n", x, &x);
printf("y has value %d and address %p\n", y, &y);

$ ./addr
x has value 37 and address 0x7ffee11d21b8
y has value 42 and address 0x7ffee11d21b4
```

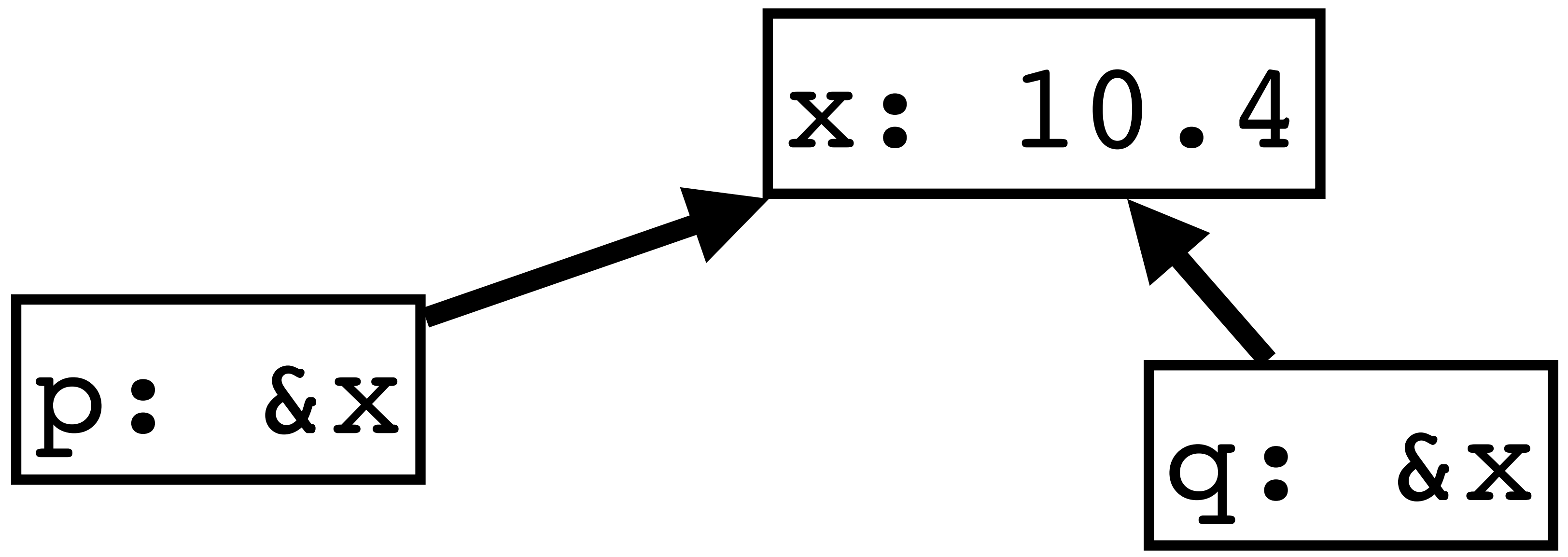
# Pointers

A pointer is an object whose value is the **address of some object**

- If `x` is an object (say a `double`), and `p` is a pointer whose value is `&x`, then we say "`p` points to `x`"

Every pointer has a type that tells you what the type of the pointed-to object is

- ```
  double x = 10.4;
  double *p = &x;
  double *q = p;
  ```



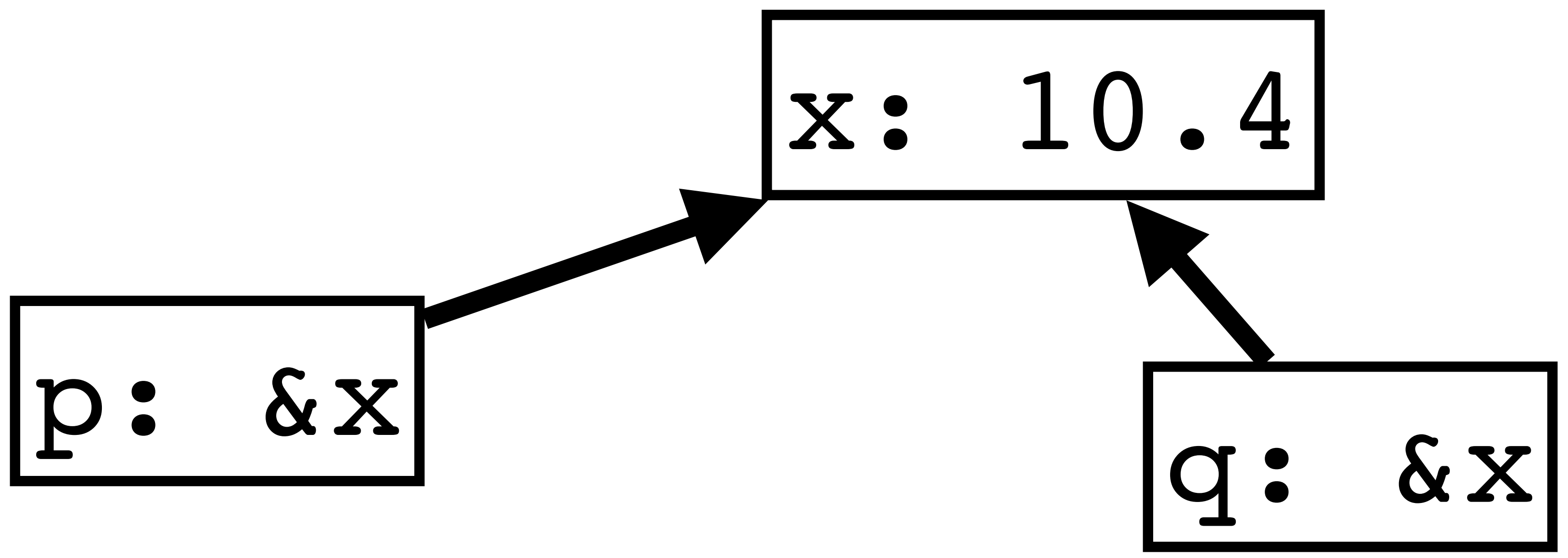

`0` (or `NULL`) is a special pointer value used to indicate that the pointer points at no object
  ```

# Dereferencing a pointer

To read or write the value of the object pointed to by the pointer, we need to **dereference** the pointer

```c
double x = 10.4;
double *p = &x;
double *q = p;

*p = 23.5; // stores 23.5 in x
printf("%.2f\n", x);   // prints 23.50
printf("%.2f\n", *p);  // prints 23.50
printf("%.2f\n", *q);  // prints 23.50
```

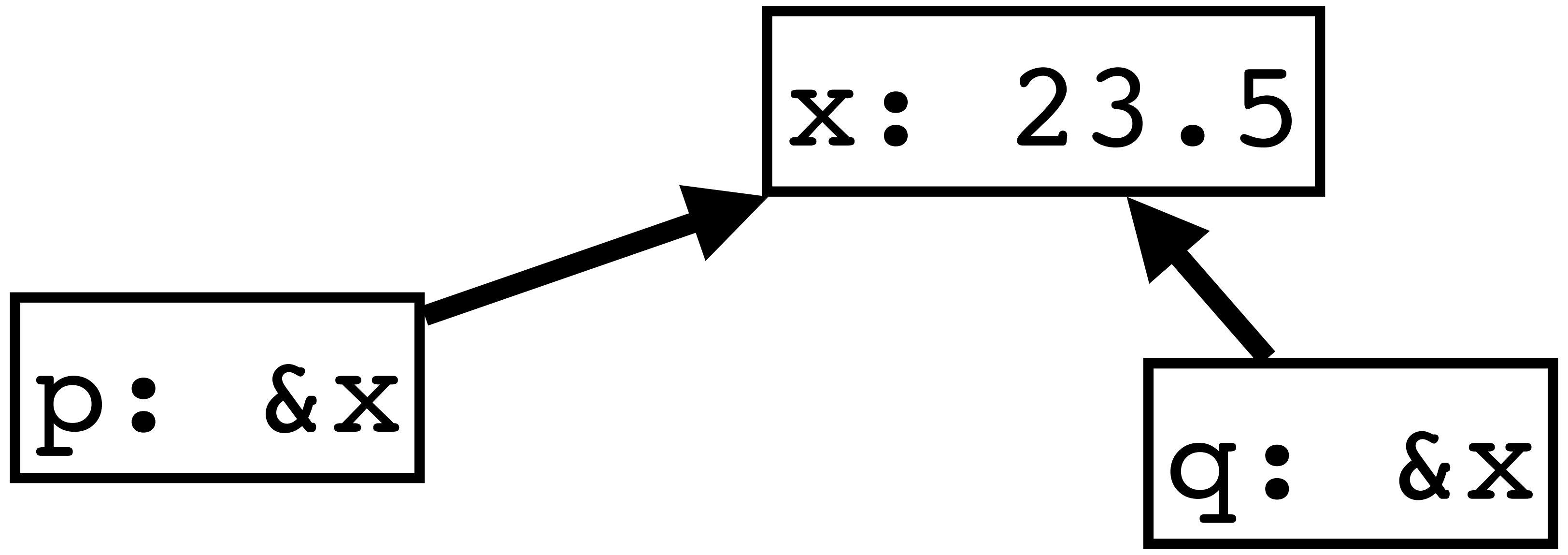


# Dereferencing a pointer

To read or write the value of the object pointed to by the pointer, we need to **dereference** the pointer

```
double x = 10.4;
double *p = &x;
double *q = p;

*p = 23.5; // stores 23.5 in x
printf("%.2f\n", x);   // prints 23.50
printf("%.2f\n", *p);  // prints 23.50
printf("%.2f\n", *q);  // prints 23.50
```



```
What is printed by this?
int x = 5;
void foo(int *p) {
  p = &x;
}
int main(void) {
  int z = 3;
  int *p = &z;
  foo(p);
  *p = 0;
  printf("%d\n", z);
}
```

A. 0

B. 3

C. 5

D. Undefined behavior

E. Implementation-defined behavior