Computer Systems Principles

Dynamic Data Structures

UMASSCS
SCHOOL OF COMPUTER SCIENCE

Learning Objectives

- Understand stack allocation
- Learn about dynamic/heap allocation
- Learn about dynamic arrays
- Learn about header files and how to create them
- Understand two implementations for a stack

But first: learnings from HW 3

- **Offer this for your insight (no recording please!)**
- **Do not expect code quite this "tight" from programmers new to C**
- **Idioms help to convey intent and make code easier to grasp**
- **'for' loops group together loop information that makes a loop easier to understand**

C Stack Allocation

- **All values need to be allocated somewhere**
- **Two Options**
  - **Stack :** lifetime of a function (static)
  - **Heap :** lifetime of a program (dynamic)
- **Do we need both?**
  - No, but stack allocation is…
  - **Simpler**
  - **Faster**
  - **Automatically deallocated**

C Stack Allocation

- **What is allocated on the stack?**
  - Local (function) variables
  - Function return values
  - Function parameters

```
void foo(int i) {
  i = 30; // i is allocated on stack.
}

int main() {
  int x = 5; // x is allocated on stack.
  foo(x);
}
```

C Stack Allocation

```
void foo(int i) {
  i = 30; // i is allocated on stack.
}

void bar(int* i) {
  *i = 20; // Is i on the stack? What about *i?
}

int main() {
  int x = 5; // x is allocated on stack.
  foo(x); bar(&x);
}
```

C Stack Allocation

```
int inc(int j) {
  return j+1;
}

int main() {
  int x = 5;
  x = inc(x);
}
```

What is allocated on the stack?

C Stack Allocation

```
int inc(int j) {
  return j+1;
}

int main() {
  int x = 5;
  x = inc(x);
}
```

What is allocated on the stack?

C Structs

- **The Land Before OOP and C:**
  - C **only** has a basic aggregate type (record)
  - It is called a *structure* or **struct**

C Structs

```
struct foo {
  int a;
  char b;
};

int main() {
  foo x;        // x is a struct allocated on stack
  foo *y = &x; // y points to a struct
}
```

C Structs

- **Member Access:**
  - **Non-pointer:** use '.' notation
  - **Pointer:** use '->' (arrow) notation

```
int main() {
  foo x;        // x is a struct allocated on stack
  foo *y = &x; // y points to a struct

  x.a = 10;
  y->a = 20;  // y->a is exactly the same as (*y).a

  printf("%d\n", x.a); // prints 10
}
```

C Heap Allocation

- **Dynamic Memory Allocation**
  - *Manually* Allocated
  - *Manually* 'Destroyed' (Deallocated)
  - **No** Garbage Collector (unlike Java)
- **Where:**
  - Large pool of unused memory *(heap/free store)*
  - Accessed indirectly by a **pointer**

C Heap Allocation

- **How to Allocate:**
  - the **`malloc`** function
- **Basic Syntax:**
  - p = (**type***) malloc(sizeof(**type**));
  - Where p is a *pointer to type*
- **Example:**
  - int* int_ptr = (int*)malloc(sizeof(int));

Pointers & NULL

- **NULL Pointers**
  - A pointer that has been explicitly set to the special value called NULL (which is 0).

```
int* p = NULL;
```

Pointers & NULL

- **NULL Pointers**
  - A pointer that has been explicitly set to the special value called NULL.

```
int* p = NULL;
```

All pointers should be explicitly assigned NULL before they are allocated storage and NULL when you deallocate the storage they point to! (Good software engineering.)

C Heap Allocation

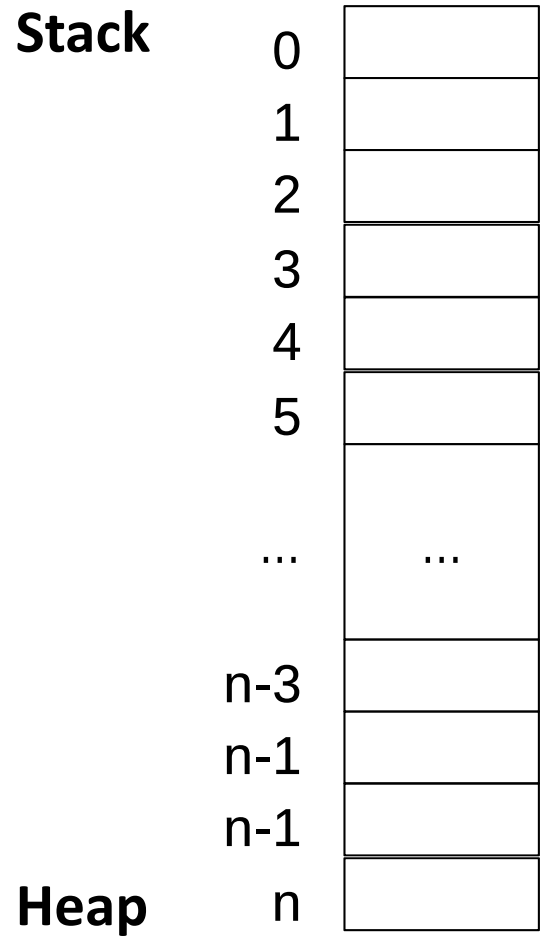
```
int* foo() {
  int b = 10; // Allocated from stack
  return &b;  // This is bad!
}

int* bar() {
  int* b = (int*) malloc(sizeof(int)); // from heap
  return b; // This is good!
}

int main() {
  int* x = foo();
  int* y = bar();
}
```

C Heap Allocation

```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```

C Heap Allocation

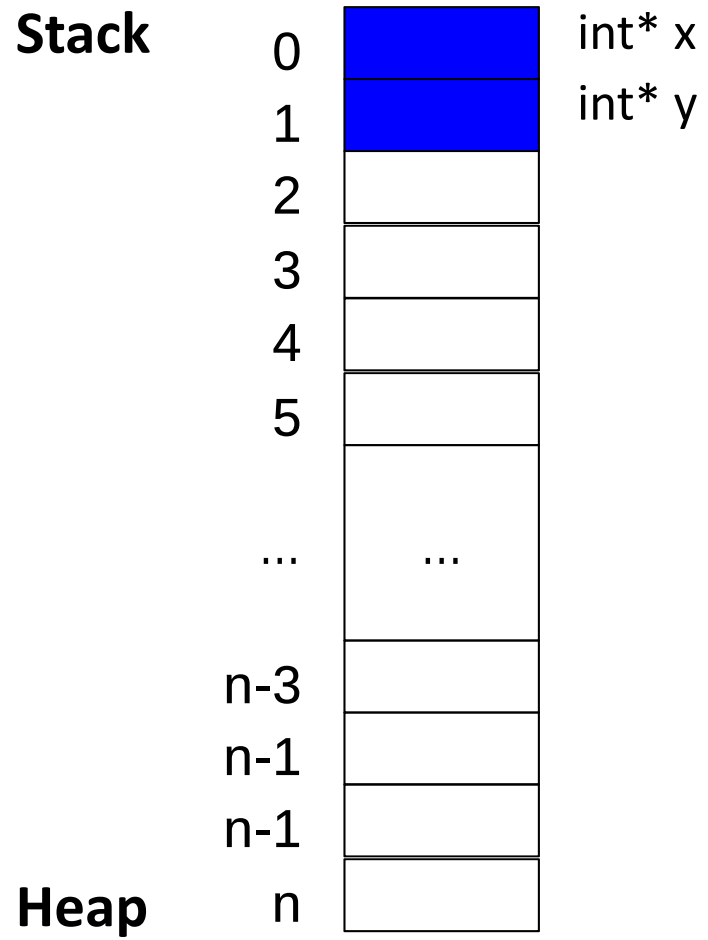
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Start program execution

C Heap Allocation

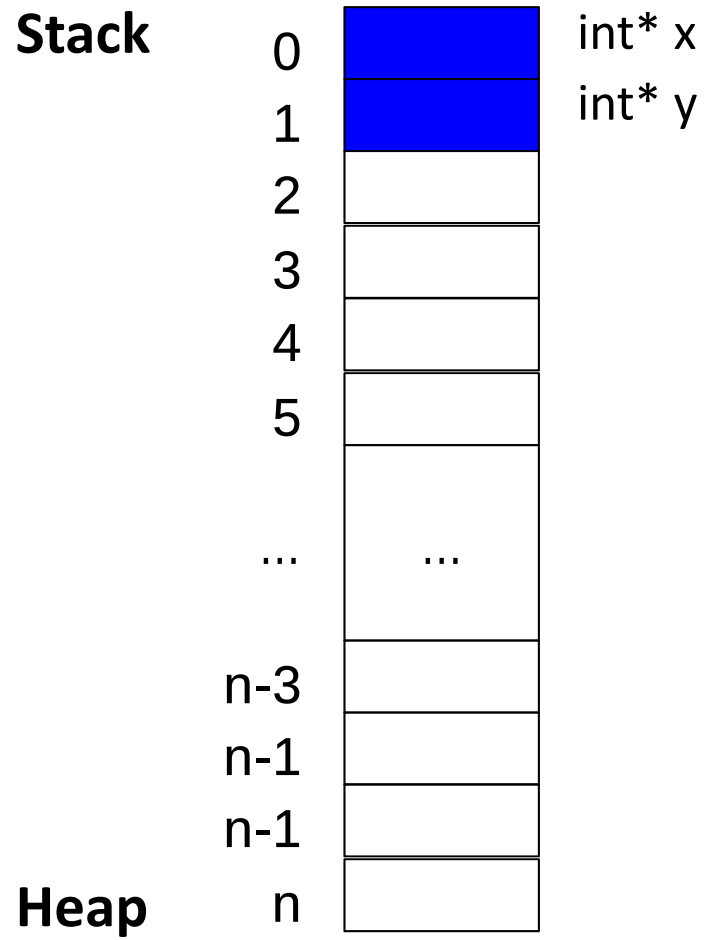
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Allocate space on stack for main…

C Heap Allocation

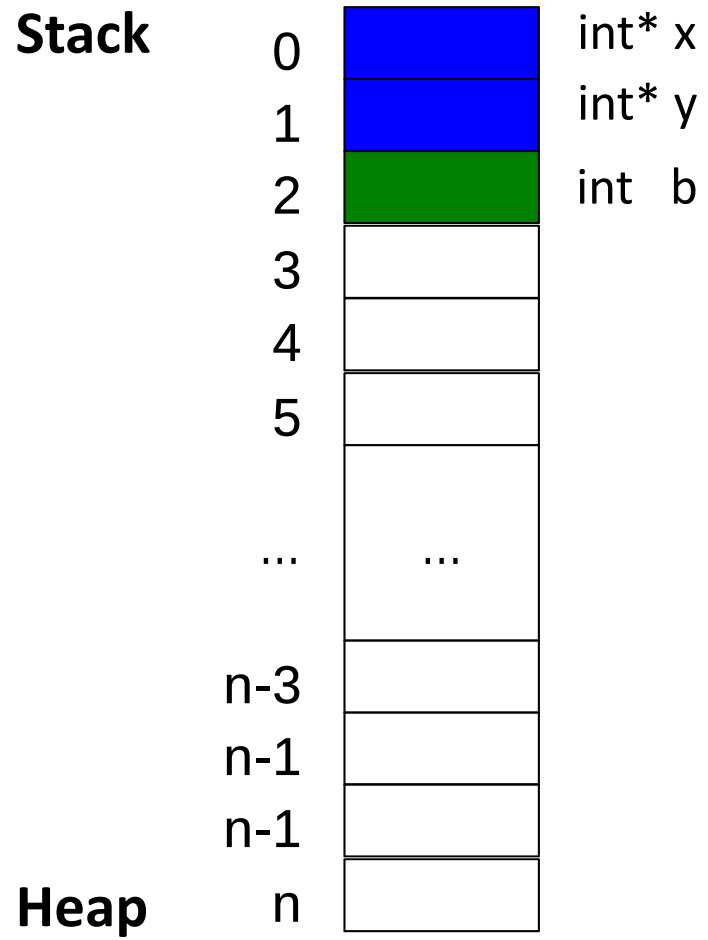
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Call the function foo…

C Heap Allocation

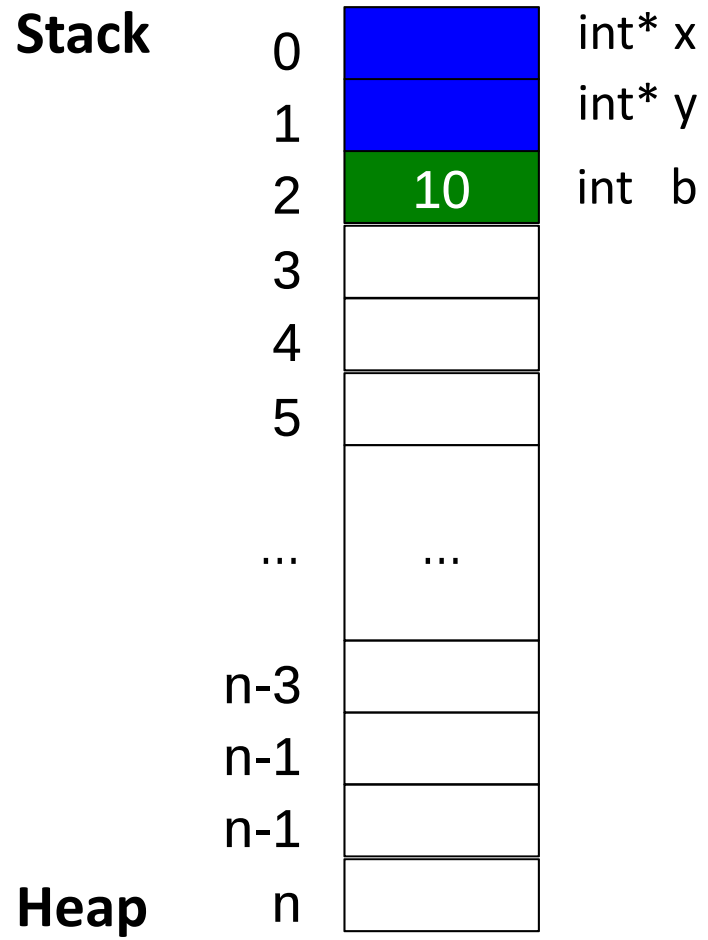
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Allocate space on stack for foo…

C Heap Allocation

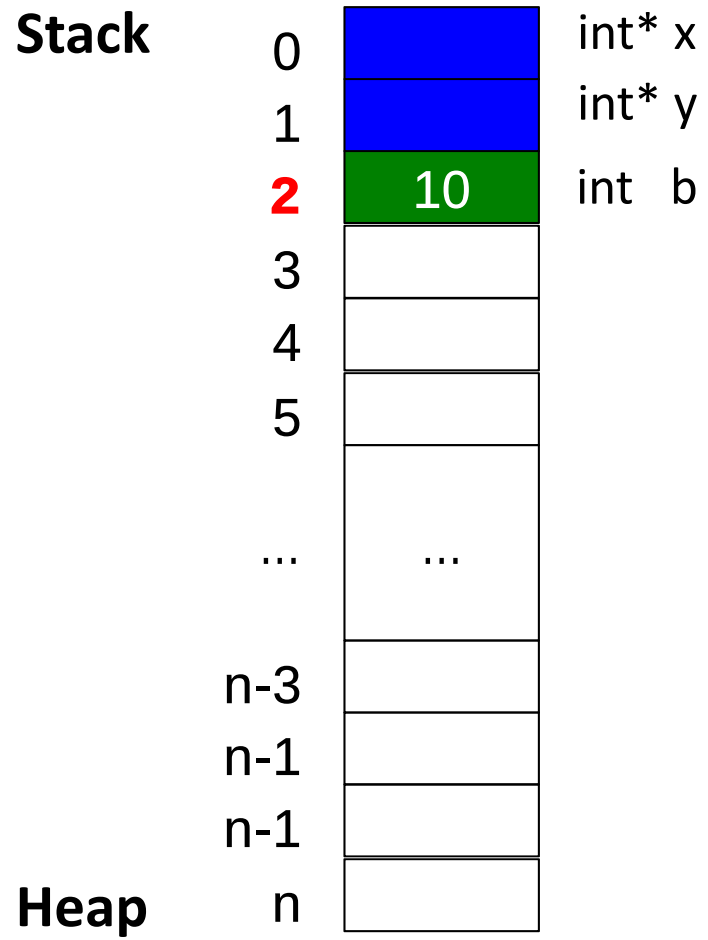
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Assignment 10 to variable b…

C Heap Allocation

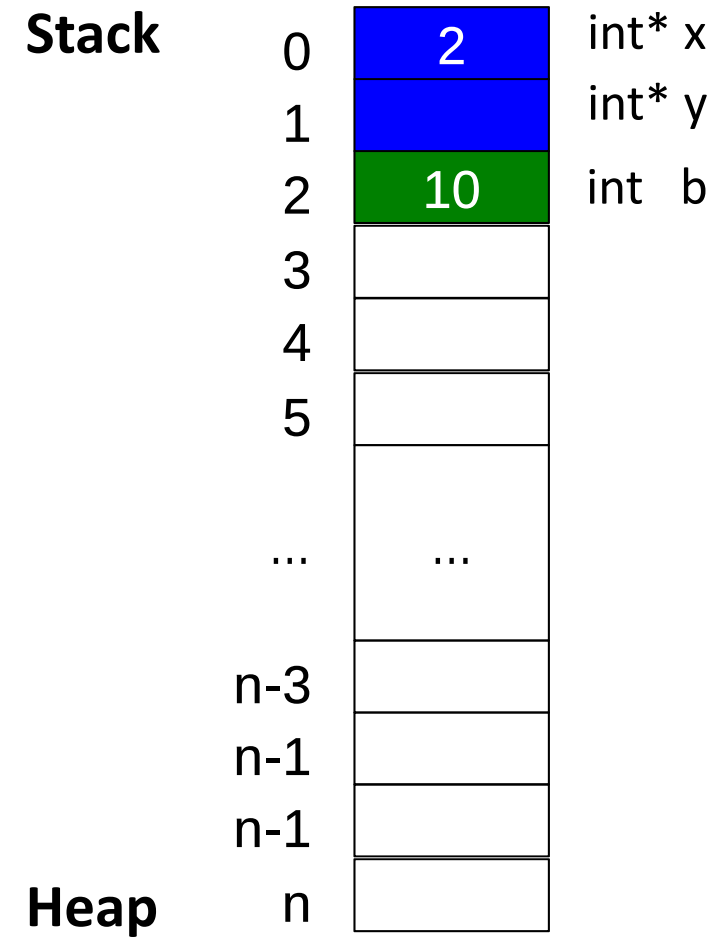
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Return the “address of” b (which is 2)…

C Heap Allocation

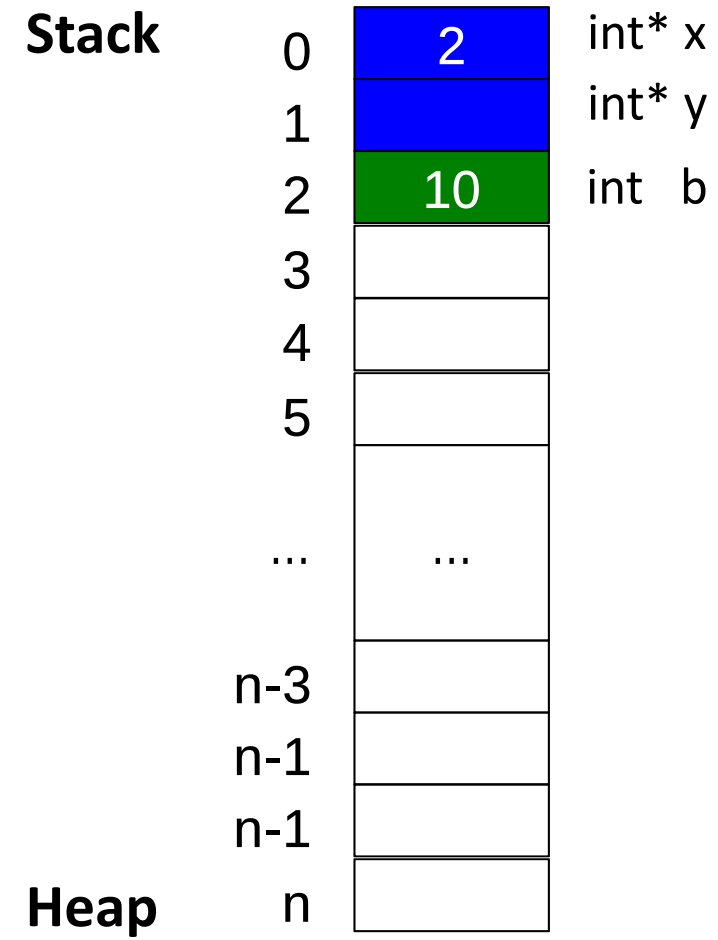
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And store it in the variable x (which “holds” a pointer to int)…

C Heap Allocation

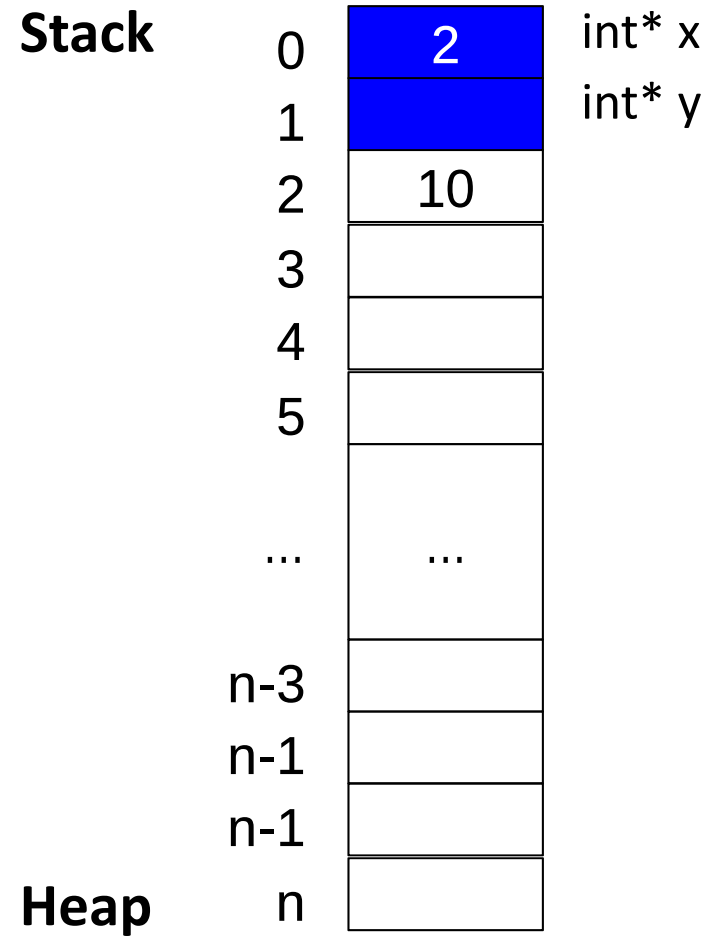
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And “pop” the space (frame) for function foo off stack…

C Heap Allocation

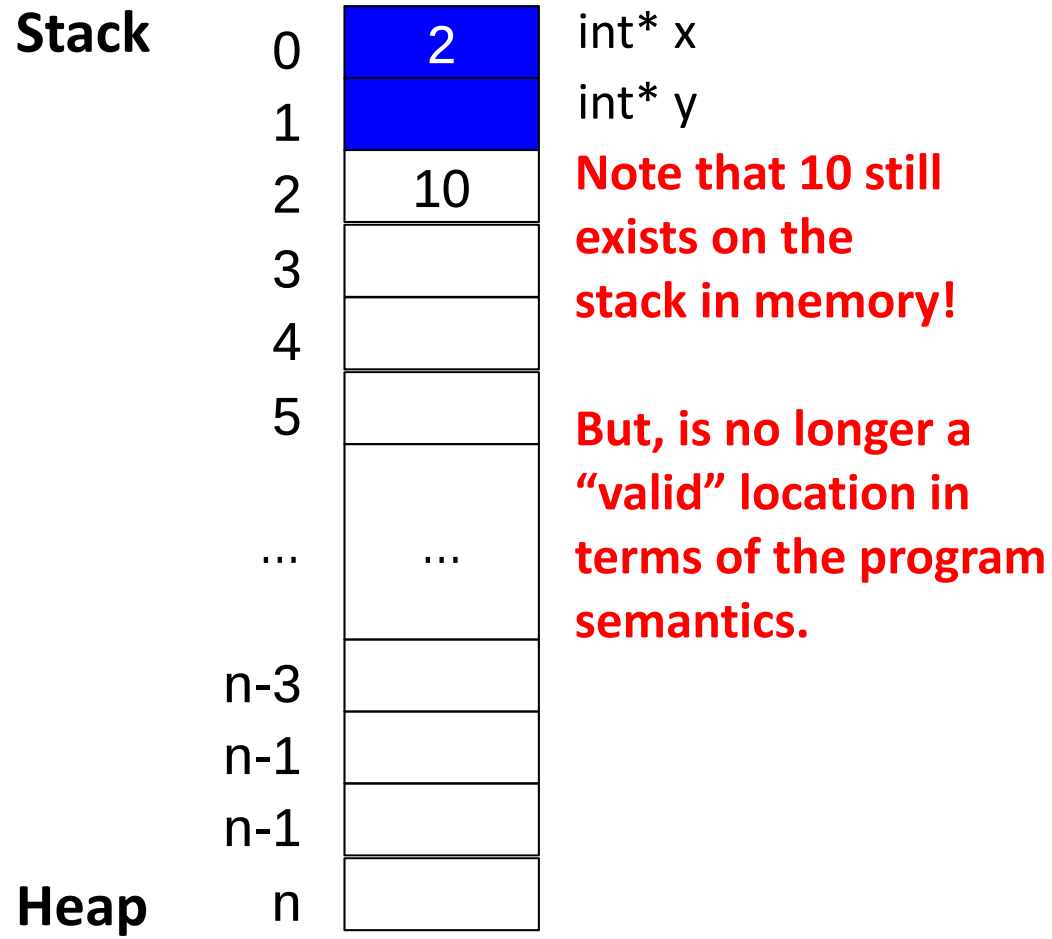
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And “pop” the space (frame) for function foo off stack…

C Heap Allocation

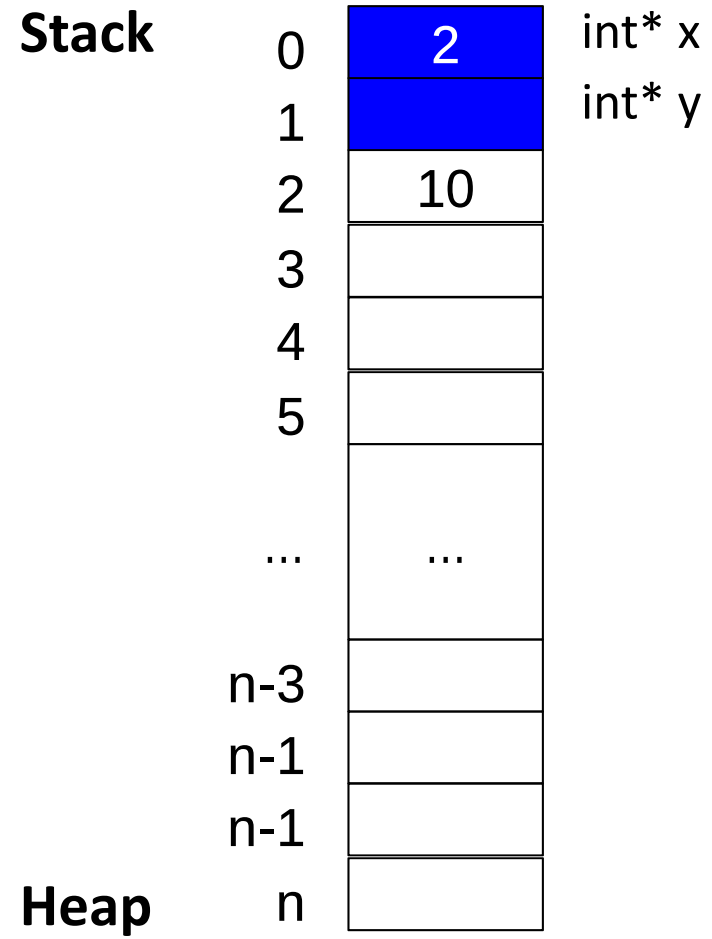
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And "pop" the space (frame) for function foo off stack…

C Heap Allocation

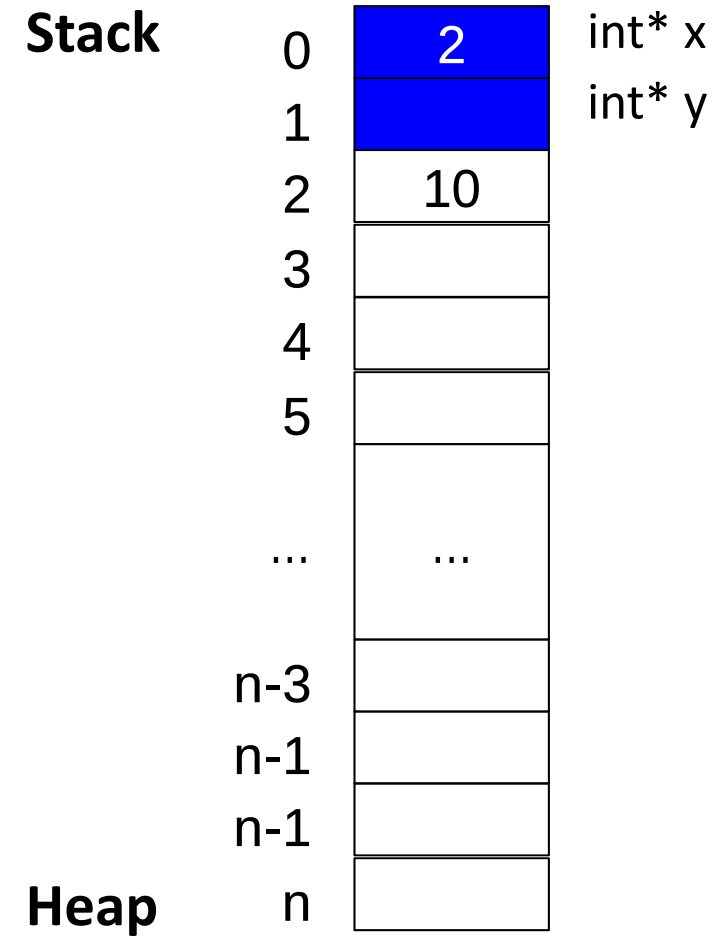
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Next, call function bar…

C Heap Allocation

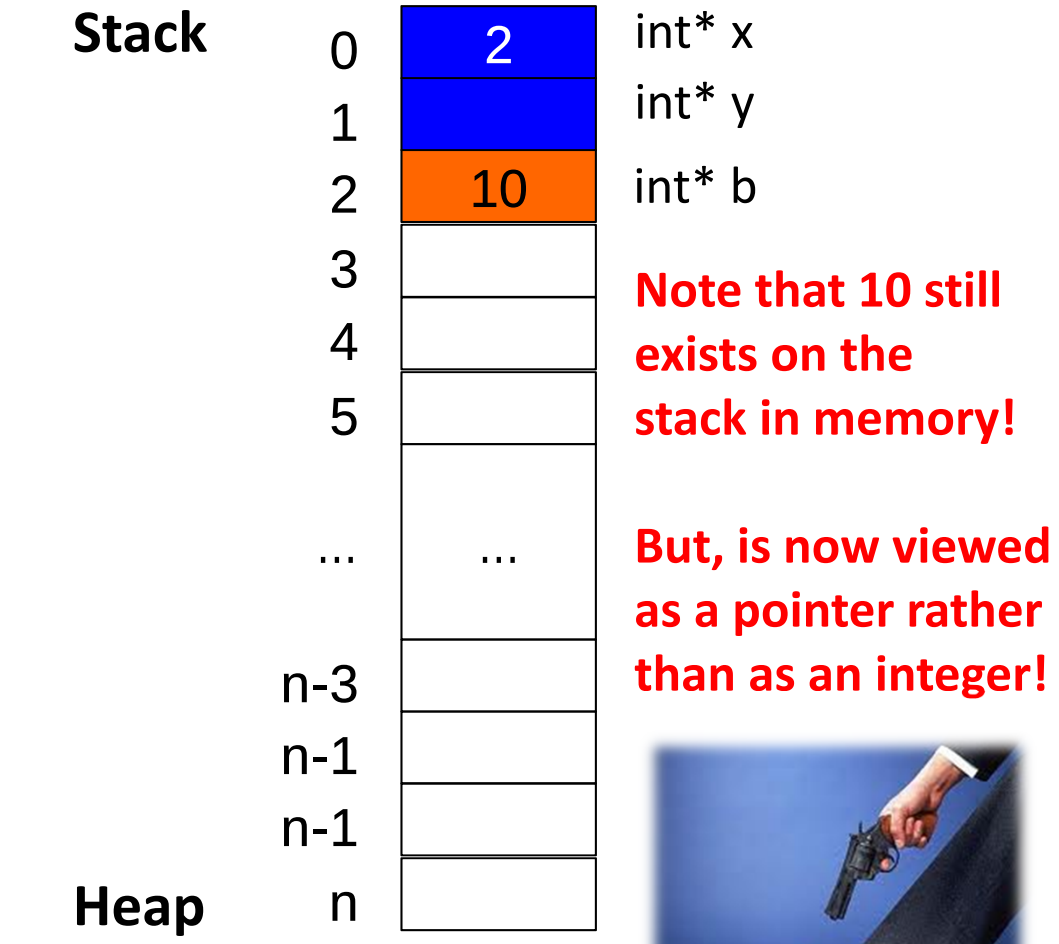
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



Allocate space on stack for bar…

C Heap Allocation

```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



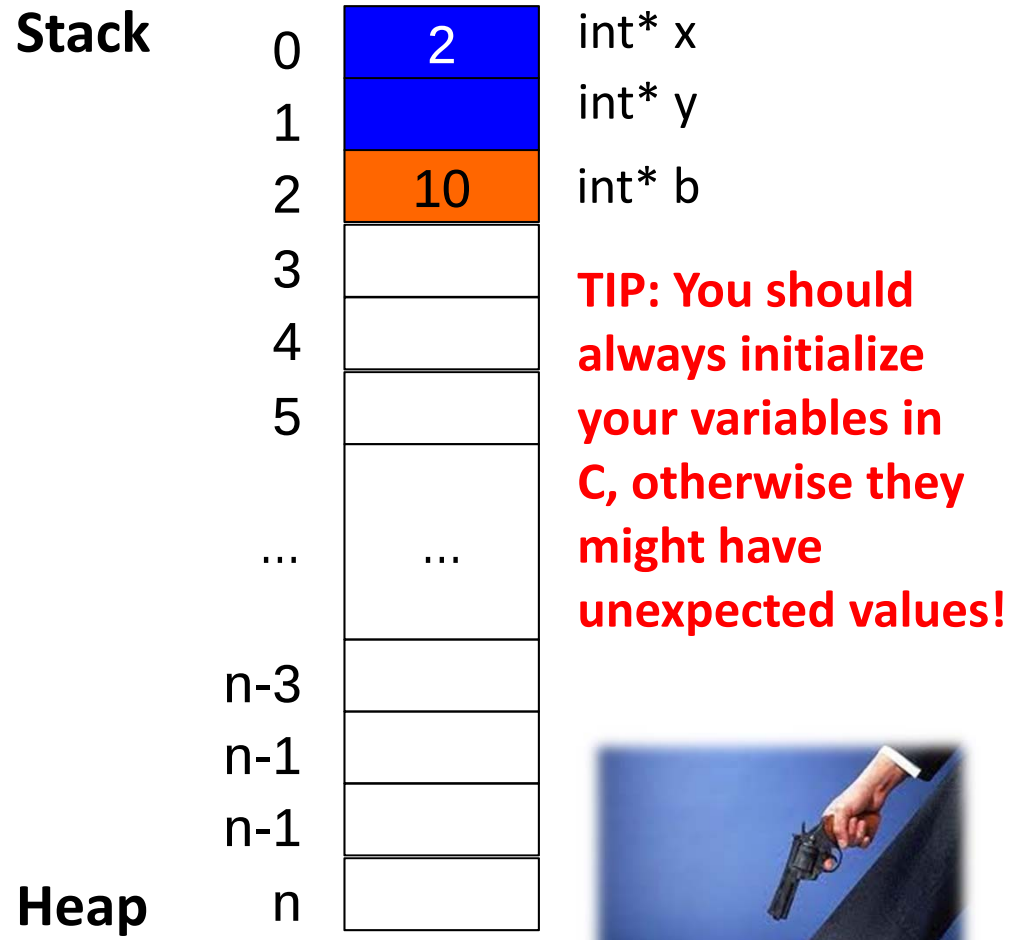
Note that 10 still exists on the stack in memory!

But, is now viewed as a pointer rather than as an integer!

Allocate space on stack for bar…

C Heap Allocation

```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```

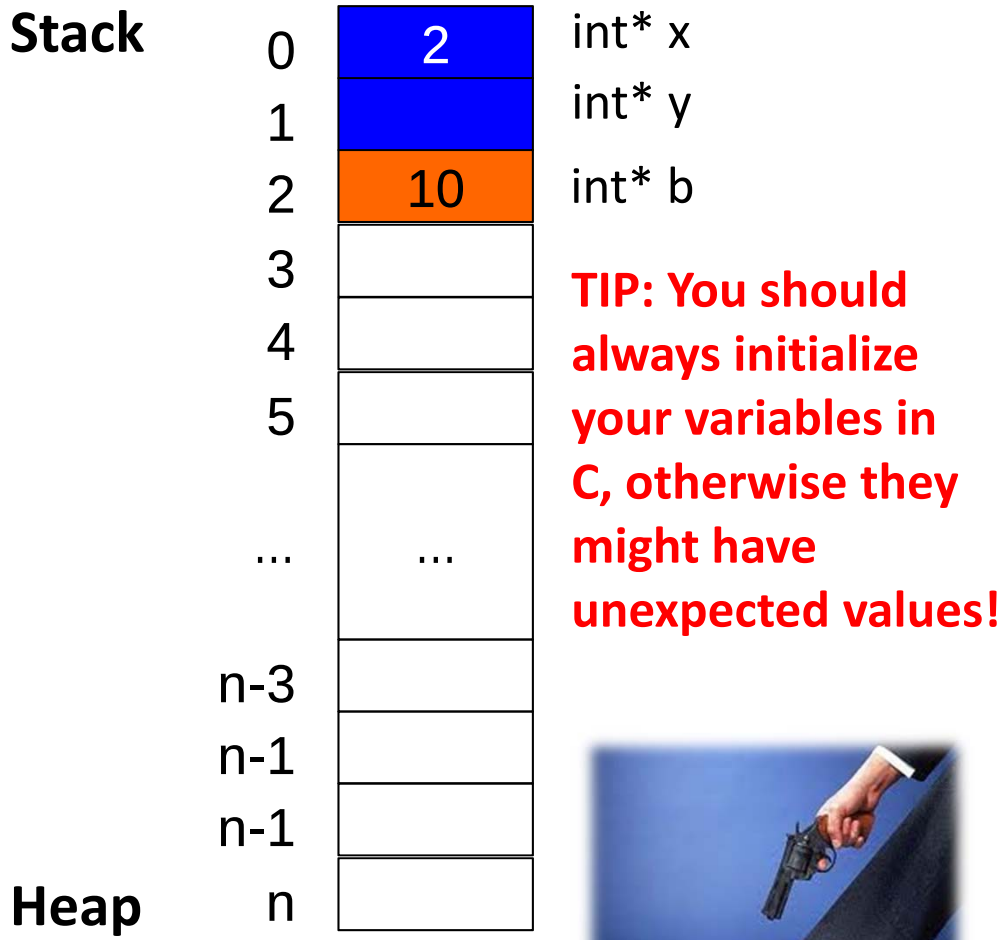


TIP: You should always initialize your variables in C, otherwise they might have unexpected values!

Allocate space on stack for bar…

C Heap Allocation

```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```

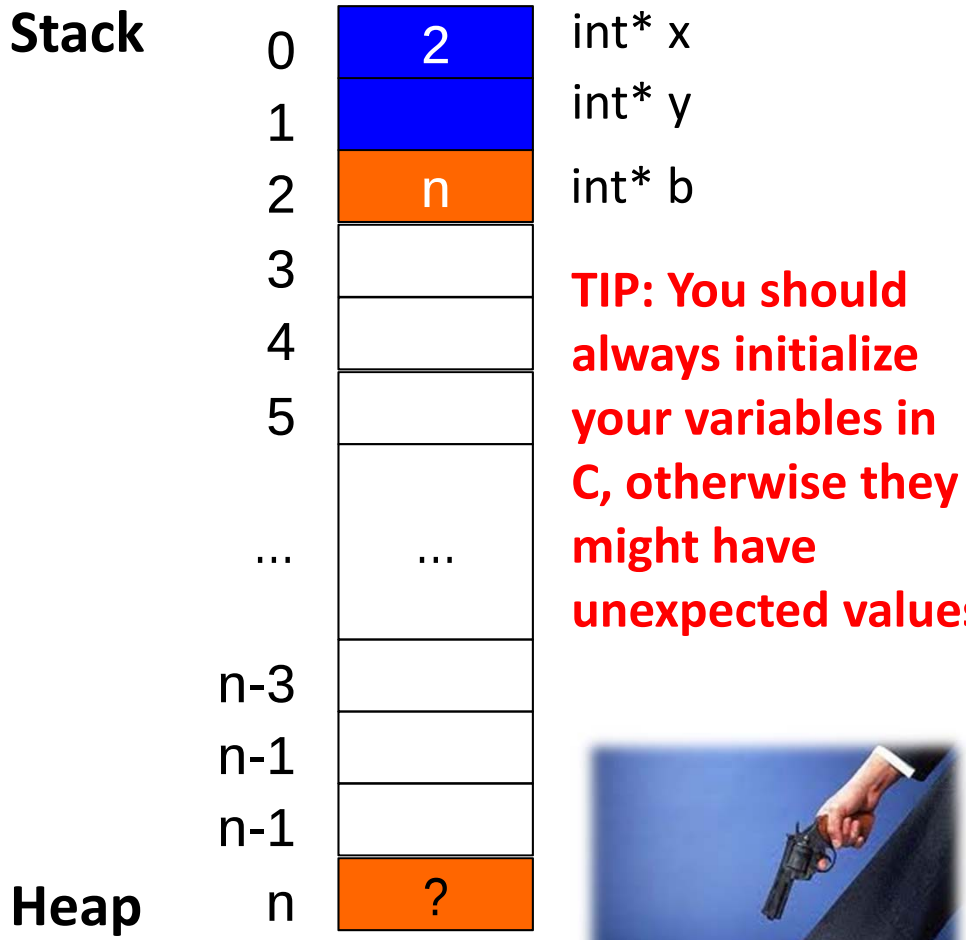


TIP: You should always initialize your variables in C, otherwise they might have unexpected values!

We do! Allocate a new int from Heap…

C Heap Allocation

```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```

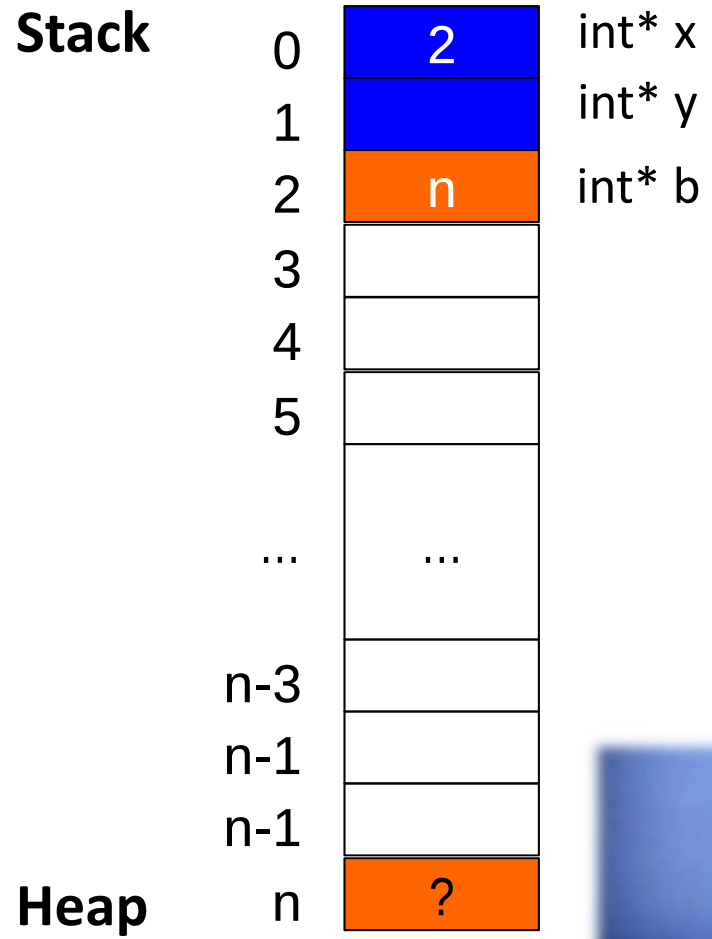


TIP: You should always initialize your variables in C, otherwise they might have unexpected values!

We do! Allocate a new int from Heap…

C Heap Allocation

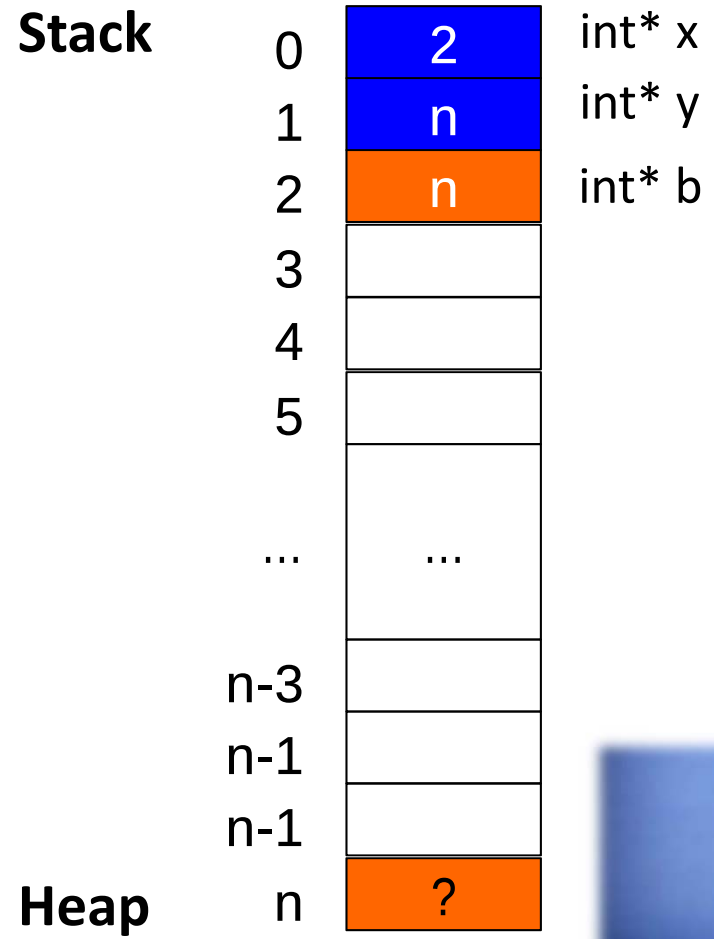
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And return the value stored in variable b…

C Heap Allocation

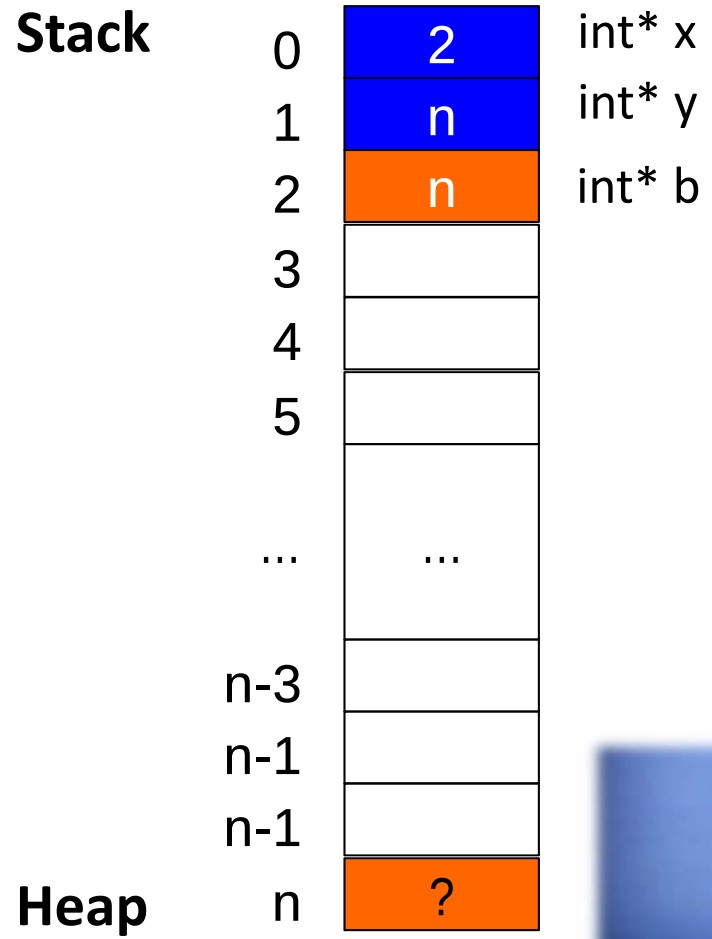
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And return the value stored in variable b…

C Heap Allocation

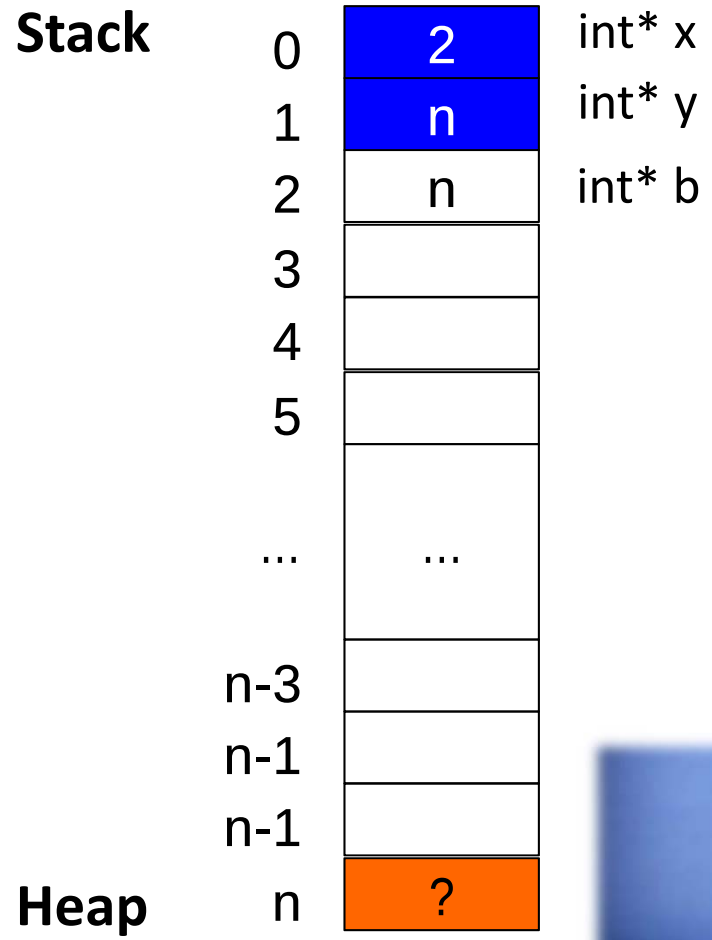
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And “pop” bar off the stack…

C Heap Allocation

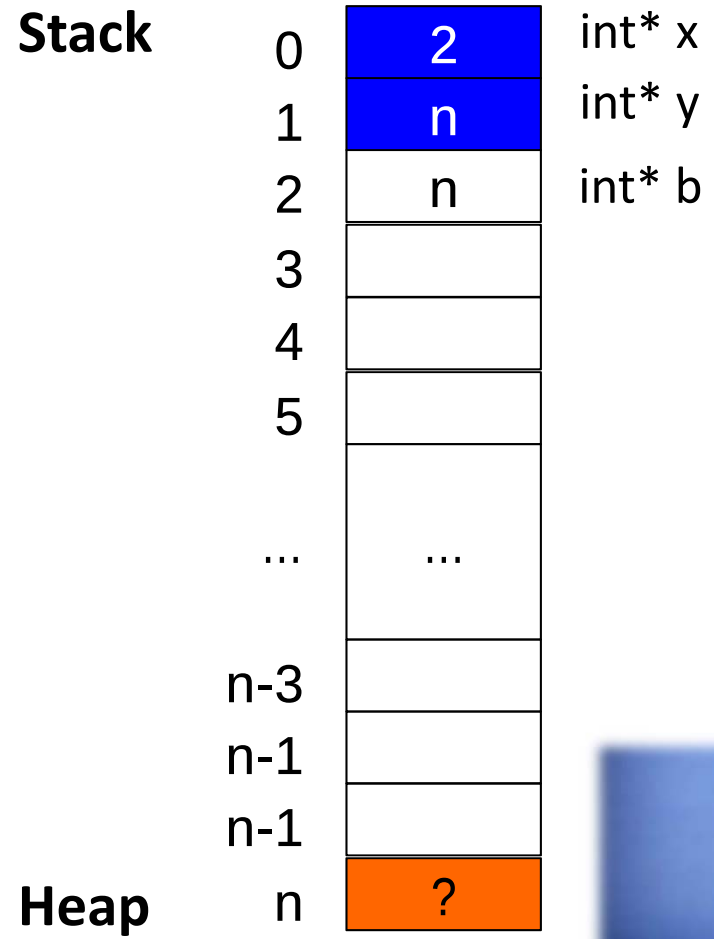
```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



And “pop” bar off the stack…

C Heap Allocation

```
int* foo() {
  int b = 10;
  return &b;
}

int* bar() {
  int* b = (int*)
malloc(sizeof(int));
  return b;
}

int main() {
  int* x = foo();
  int* y = bar();
}
```



We have a “new” “object” from the heap!

Stack Fun!

- **allocate.c**

Dynamic Arrays

- **Fixed Sized Array:**
  - char buffer[25];
  - Creates an array of characters of length 25
- **Dynamically Sized Arrays**
  - *type** buffer = (*type**) malloc(x * sizeof(*type*));
  - Creates an array of *type* objects of length:

    *x * sizeof(type)*
  - Arrays are very efficient, so it is often useful to use an array even if you need to re-allocate it!

C Heap Allocation

- **Dynamic Memory Allocation**
  - *Manually* Allocated
  - ***Manually* 'Destroyed' (Deallocated)**
  - **No** Garbage Collector (unlike Java)
- **Where:**
  - Large pool of unused memory *(heap/free store)*
  - Accessed indirectly by a **pointer**

C Heap De-Allocation

- **How to De-Allocate:**
  - The **`free`** function
  - Releases memory back to heap
- **Basic Syntax:**
  - free (p);
  - Where p is a *pointer (to a instance of a type)*
- **Example:**
  - int* int_ptr = (int*)malloc(sizeof(int));
  - free(int_ptr);

Exercise

- **exercise.c**

Stack Data Structure

- **array_stack.c**
- **int_stack.c**