Computer Systems Principles

Dynamic Data Structures

UMASSCS

SCHOOL OF COMPUTER SCIENCE

Announcement

- **Midterm**
  - Time: Feb 25 (Thur), 7pm to 9pm
  - Location: ILCN room 151
  - Material covered: Up through pointers (last Thursday)
  - Style of exam: Similar to the quizzes, but with some short answer programming questions as well.
  - Allowed resources: open book && close notes

Announcement

- **Quiz 5 released, due Feb 28 (Sun)**
- **HW4 released, due Feb 29 (Mon)**

Learning Objectives

- Understand stack allocation
- Learn about dynamic/heap allocation
- Learn about dynamic arrays
- Learn about pointer to pointer

Memory layout and variable declaration order

- **Does the compiler always layout the memory in declaration order or in reverse declaration order？**
  - No. It depends on the behavior of particular compilers.
- **stack_address.c**

THREE POINTER OPERATIONS

C Pointers

Imagine we have the following declarations…

```
int x;
int *ptr = &x;
```

`x` is located “somewhere” in memory

`ptr` is also located “somewhere” in memory

Three pointer operations

- **Referencing**
  - v = address-of(x)
  - Create location l
  - Introduce v->l

Three pointer operations

- **Referencing**
  - v = address-of(x)
  - Create location l
  - Introduce v->l
- **Dereferencing**
  - x=*v or *v= x
  - Access existing location pointed by v

Three pointer operations

- **Referencing**
  - v = address-of(x)
  - Create location l
  - Introduce v->l
- **Dereferencing**
  - x=*v or *v= x
  - Access existing location pointed by v
- **Aliasing**
  - Pointer variable v1, v2
  - v2 = v1
  - v1->l ⇔v2->l

PARAMETER PASSING

C Parameter Passing

- **Pass-by-value**
  - Same as Java (all references/primitives)
  - The parameter is evaluated and bound to the corresponding variable in the function

```
void foo(int i) {
  i = 10; // Does not change i outside of function
}

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

C Parameter Passing

- **Pass-by-value (pointer)**
  - The parameter is a pointer
  - The referenced object can be manipulated

```
void bar(int *i) {
  *i = 20; // Does change *i outside of function
}

int main() {
  int x = 5;
  bar(&x);  // will change x
}
```

i-clicker question

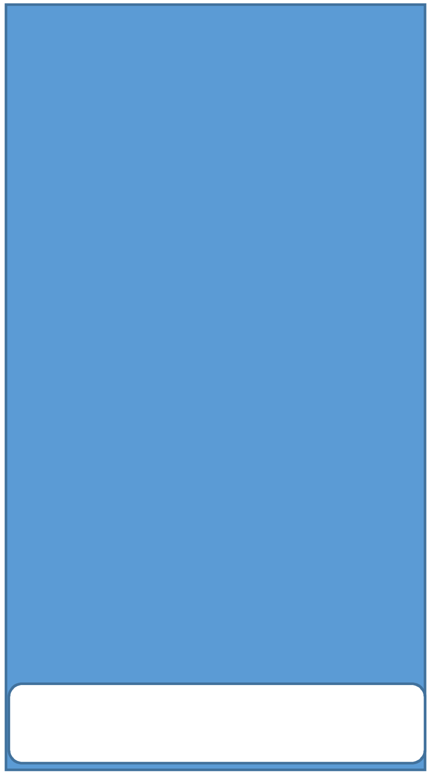
What is the output?

```
void foo(int i) {
  i = 30;}

void bar(int* i) {
  *i = 20;}

int main() {
  int x = 5;
  foo(x); bar(&x);
  printf("%d\n", x);
}
```

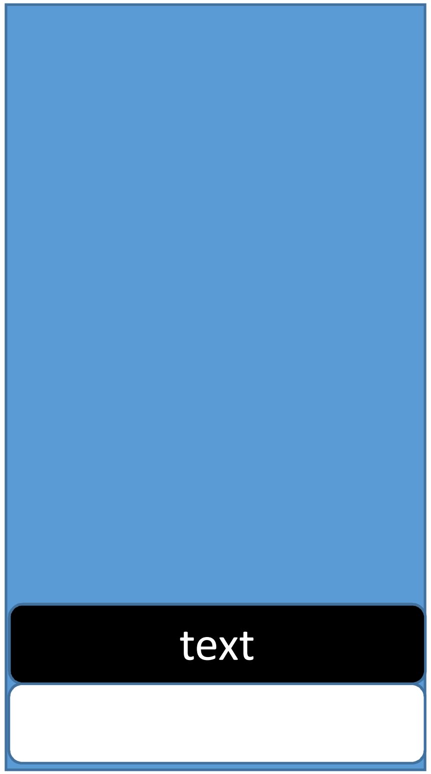
A. 30
B. 20
C. 5;
D. none of the above

Memory allocation

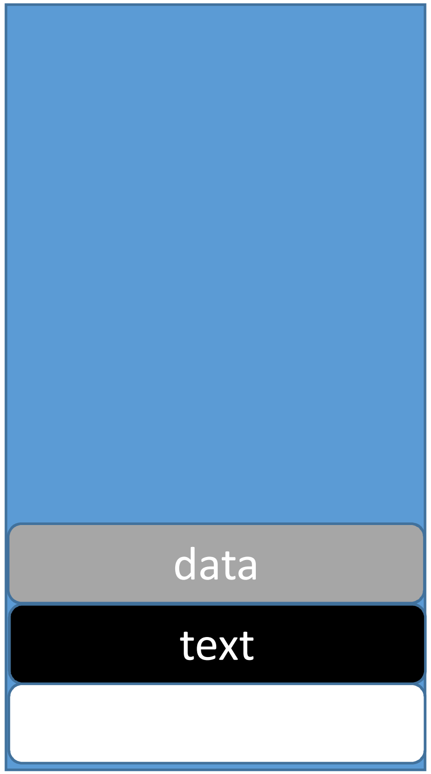
Memory layout for a Linux Process



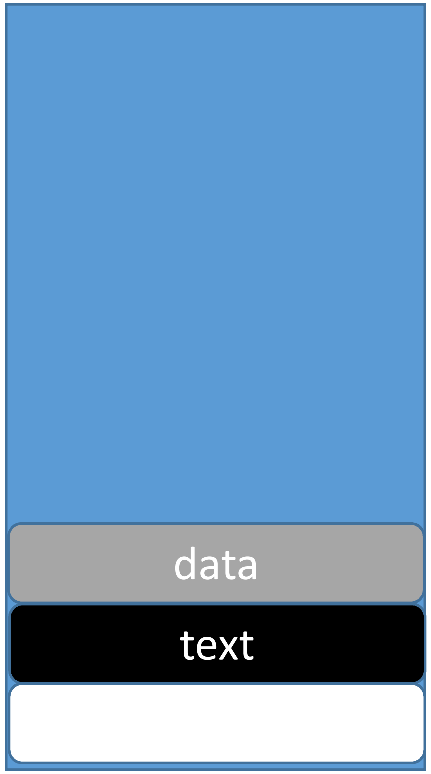
Memory layout for a Linux Process



Memory layout for a Linux Process

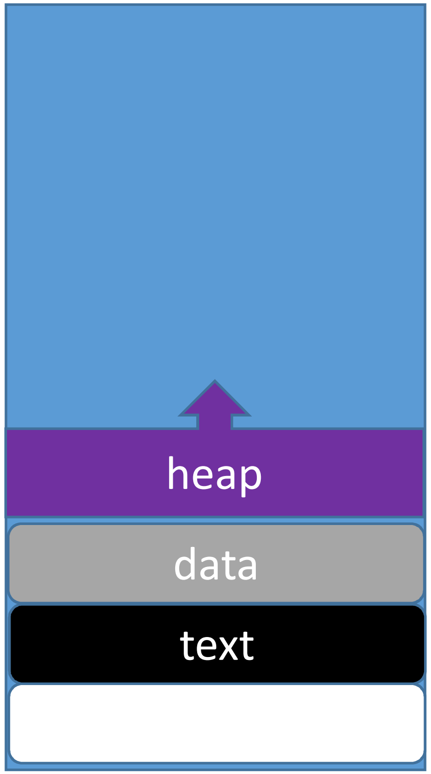


Memory layout for a Linux Process



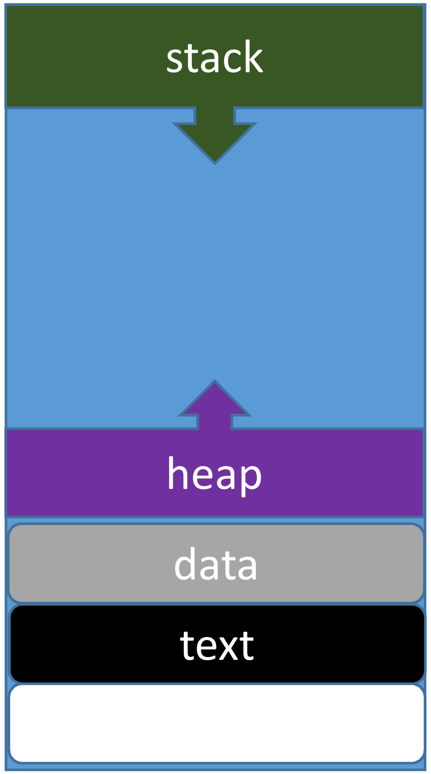
```
/* global variable declaration */
int a[2] ;

int main () {
    …
```

Memory layout for a Linux Process



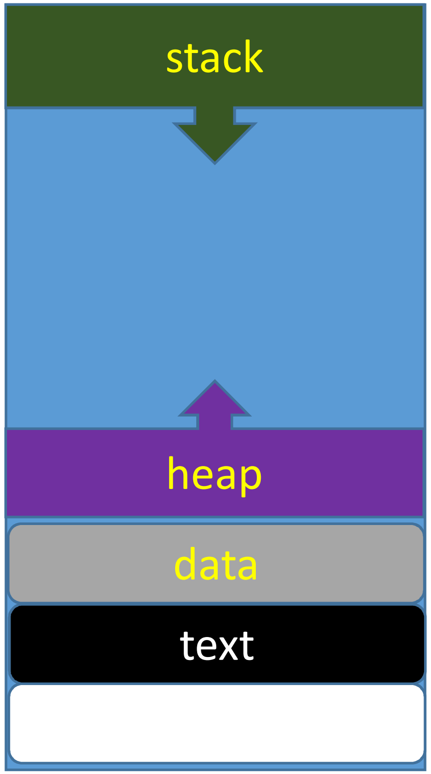
```
int main () {
    char *str;
    str = (char *) malloc(15);
    …
```

Memory layout for a Linux Process

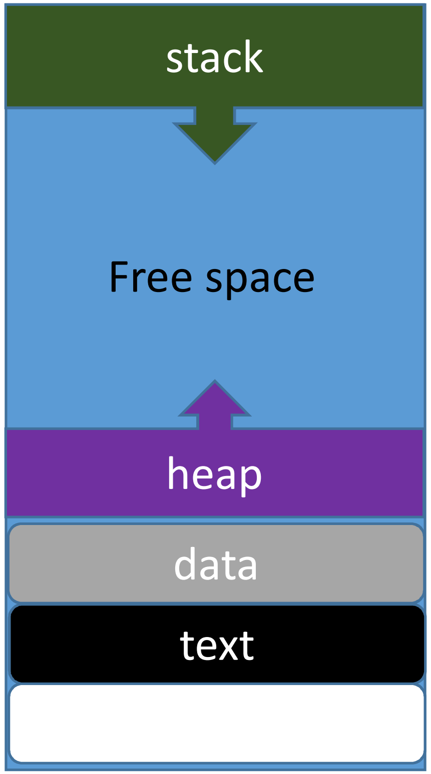


```
f() {
    int a[4];
    …
}
```

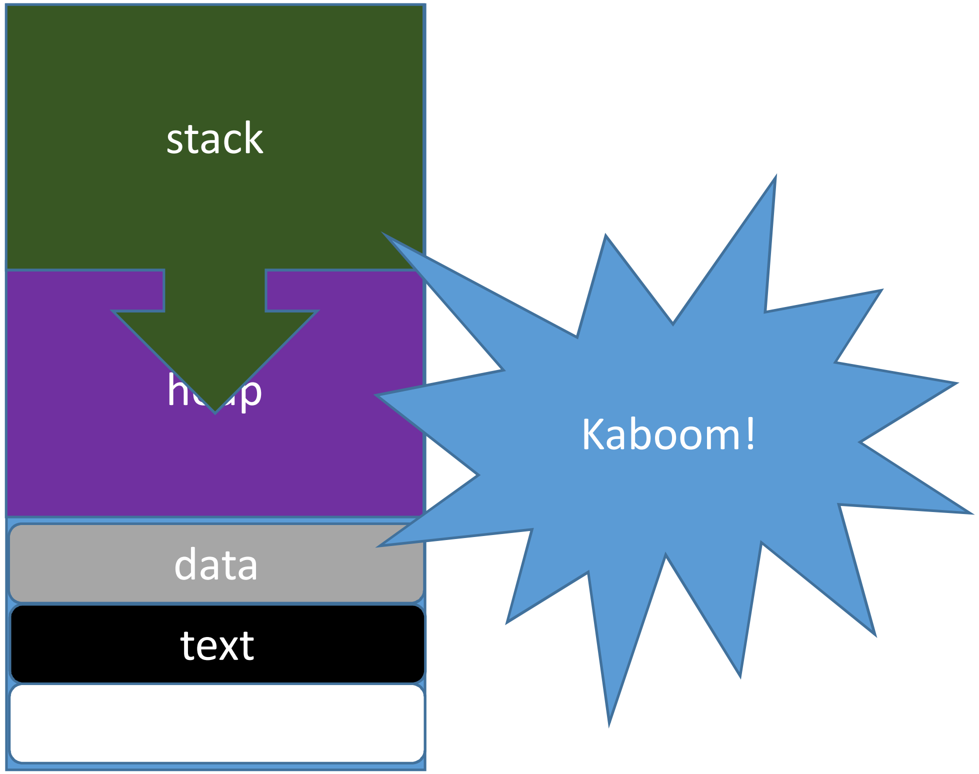
Memory layout for a Linux Process



Memory layout for a Linux Process



Memory layout for a Linux Process



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

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

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

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

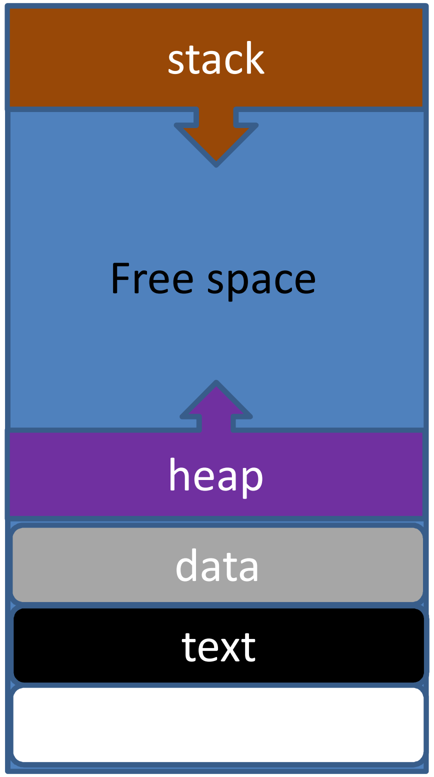
- **How to Allocate:**
  - the `malloc` function

C Heap Allocation

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

C Heap Allocation

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



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* x = (int*)malloc(sizeof(int));

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* x = (int*)malloc(sizeof(int));

x is allocated on stack.

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

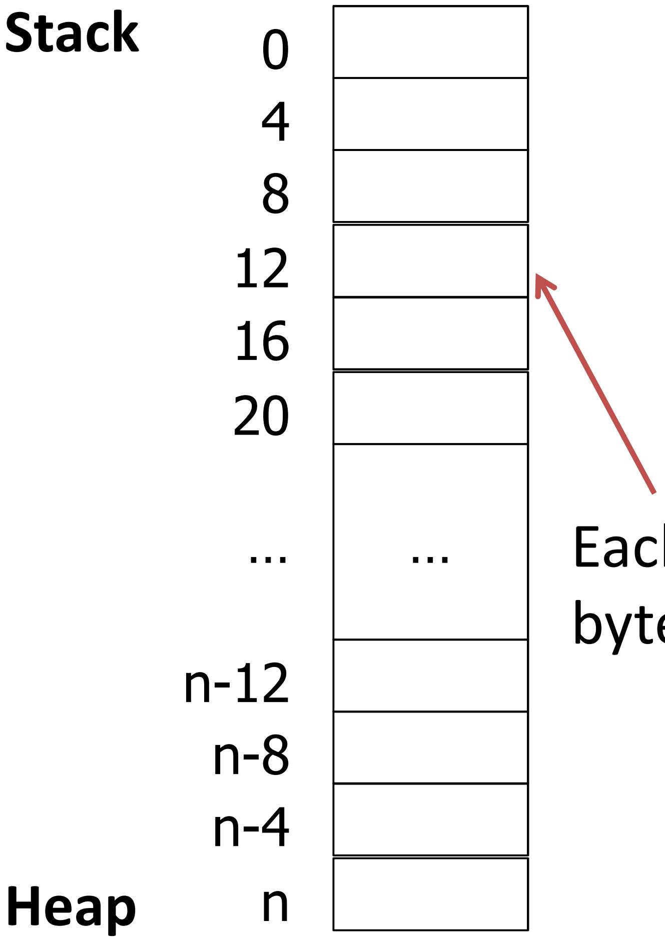
```
int* foo() {
  int b = 10; // Allocated from stack
  return &b;  // This is bad!
}

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

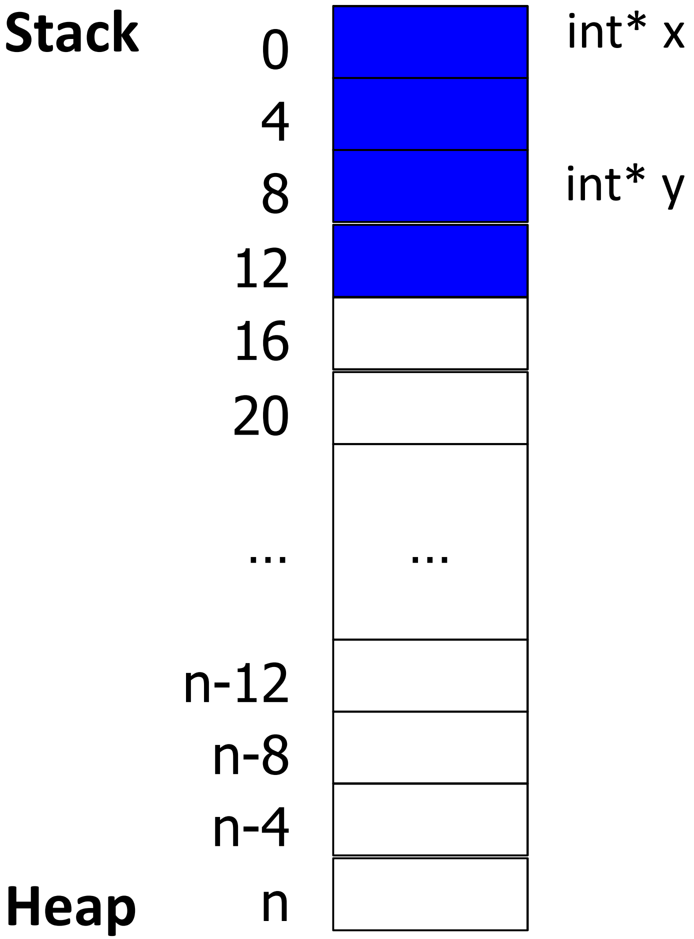
```
int* foo() {
  int b = 10;
  return &b;
}

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

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



Each box is 4 bytes

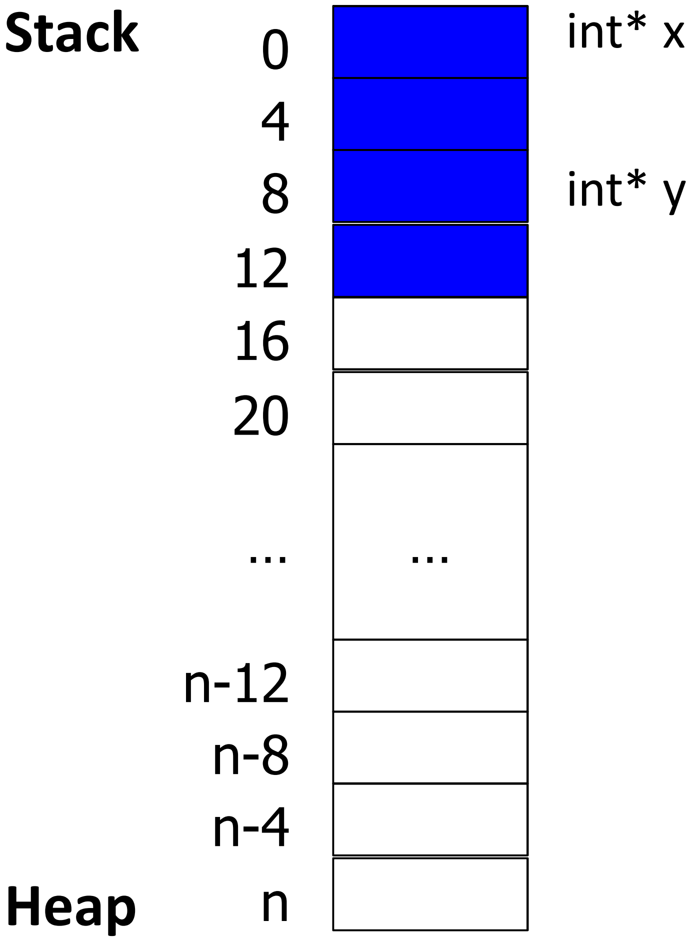
C Heap Allocation

```c
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();
}
```


Stack

0

4

8

12

16

20

…

…

n-12

n-8

n-4

Heap

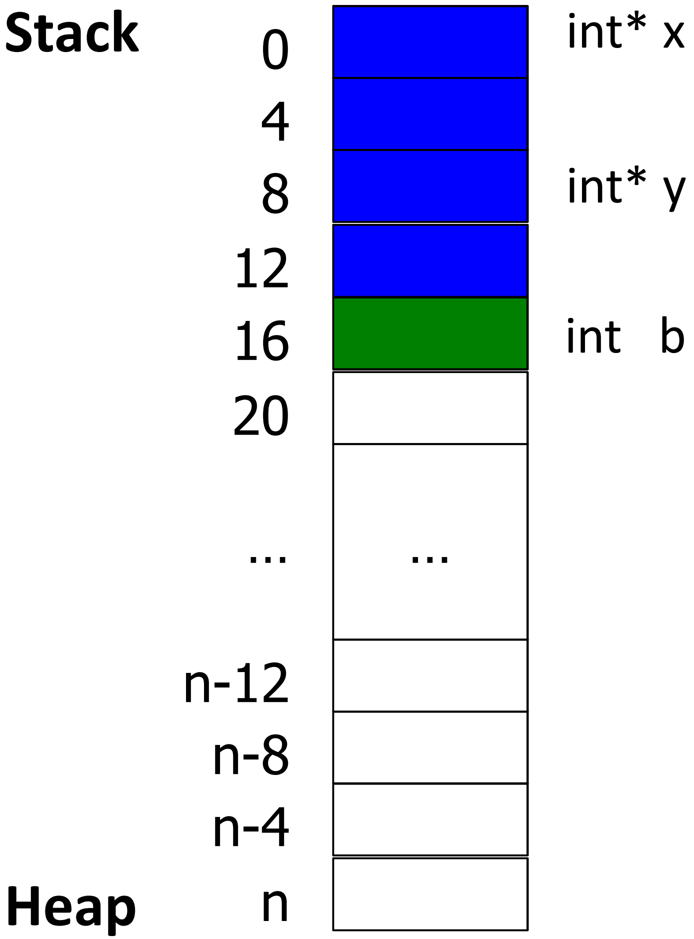
n

int* x

int* y

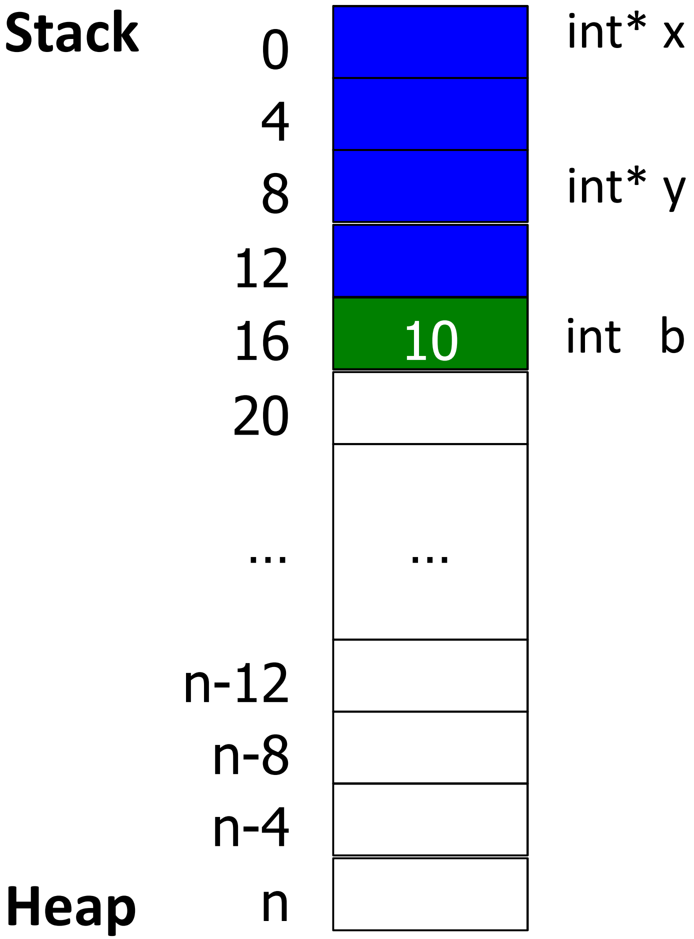
C Heap Allocation

```
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();
}
```



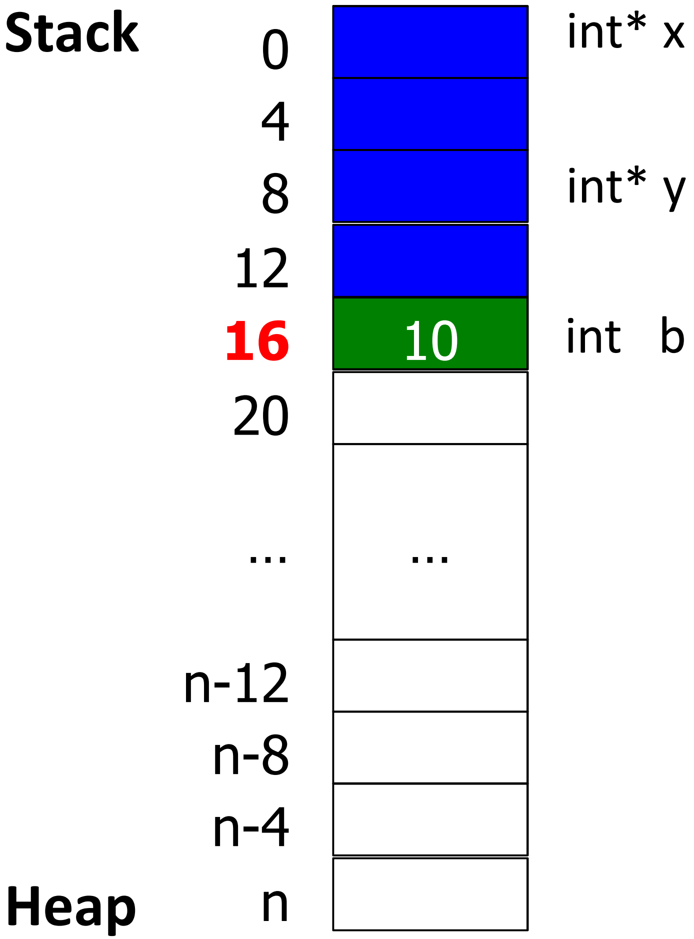
C Heap Allocation

```
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();
}
```



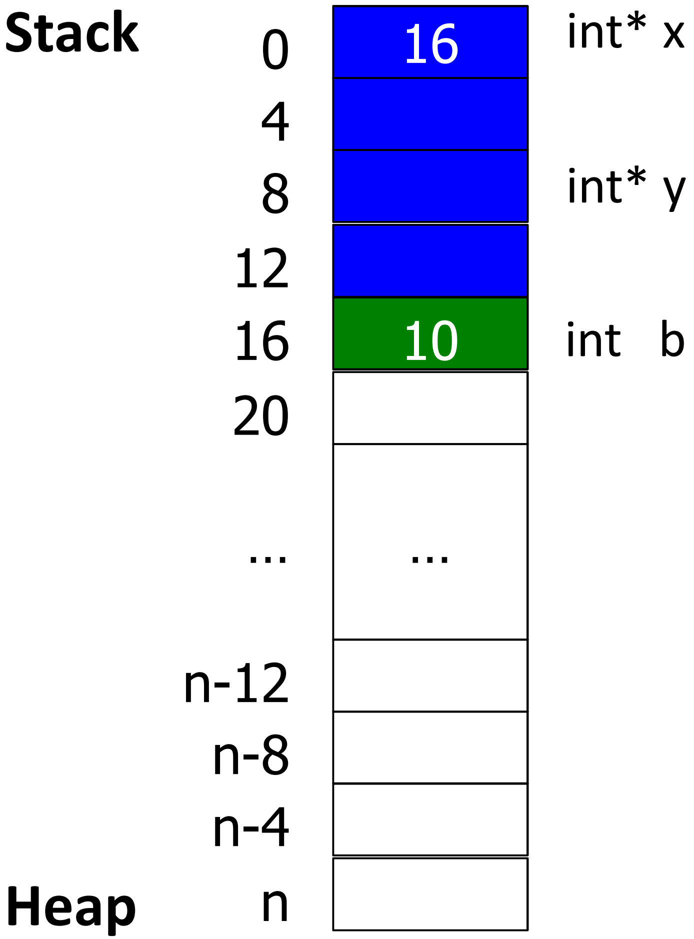
C Heap Allocation

```
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

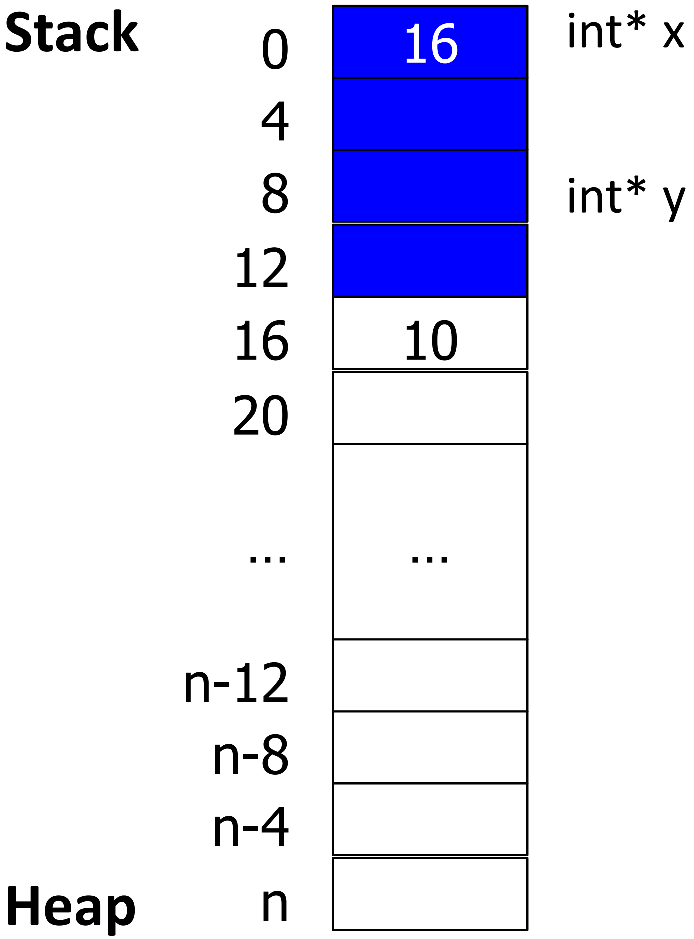
```
int* foo() {
  int b = 10;
  return &b;
}

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

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



0	16	int* x
4		
8		int* y
12		
16	10	int b
20		
…	…	
n-12		
n-8		
n-4		
n		

C Heap Allocation

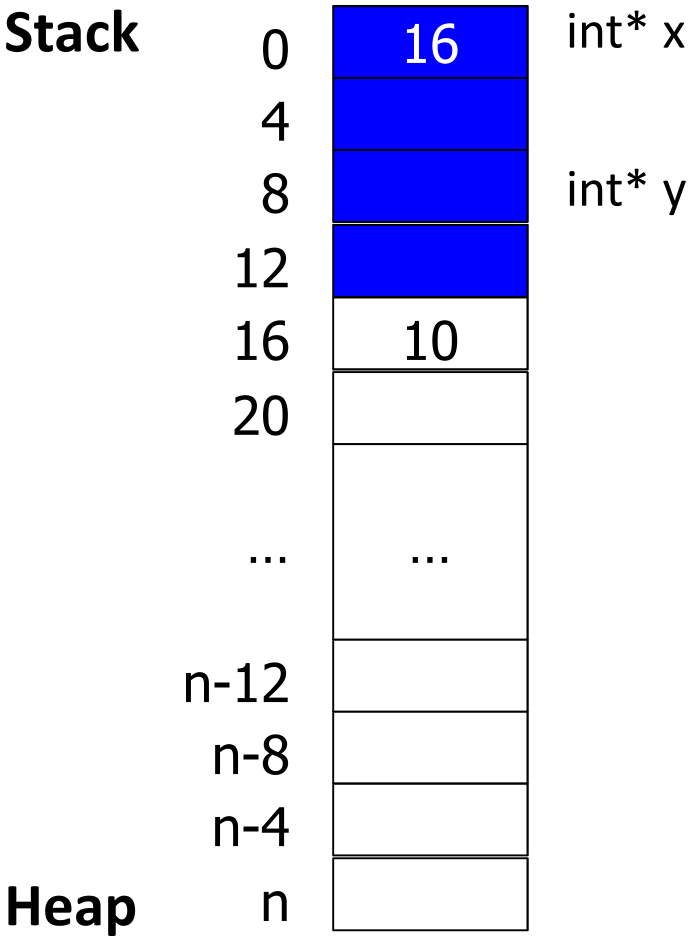
```
int* foo() {
  int b = 10;
  return &b;
}

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

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



Stack	0	16	int* x
	4		
	8		int* y
	12		
	16	10	
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n		

C Heap Allocation

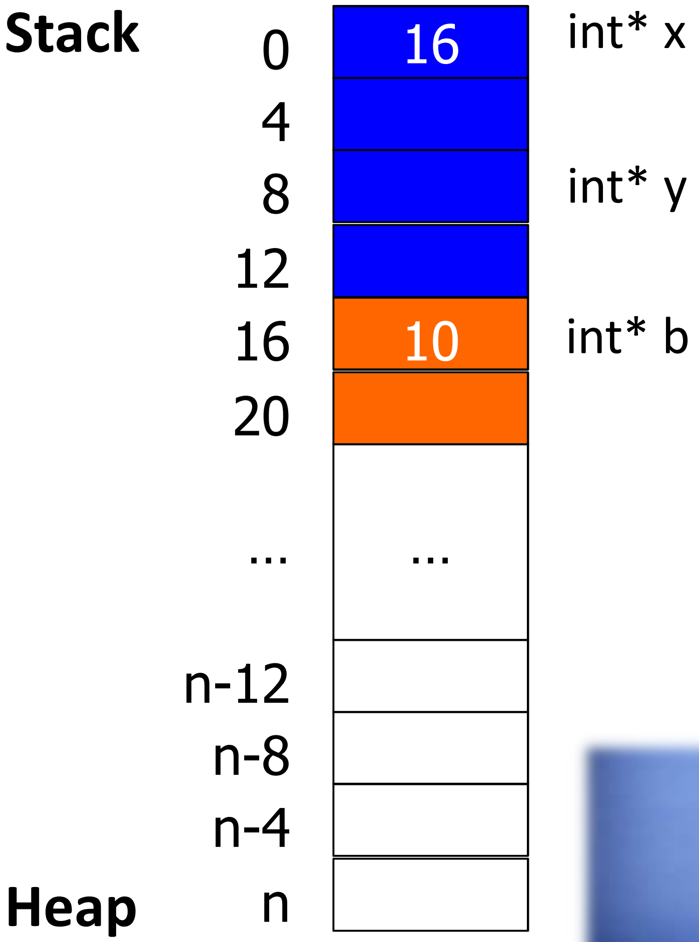
```
int* foo() {
  int b = 10;
  return &b;
}

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

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



Stack	0	16	int* x
	4		
	8		int* y
	12		
	16	10	
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n		

C Heap Allocation

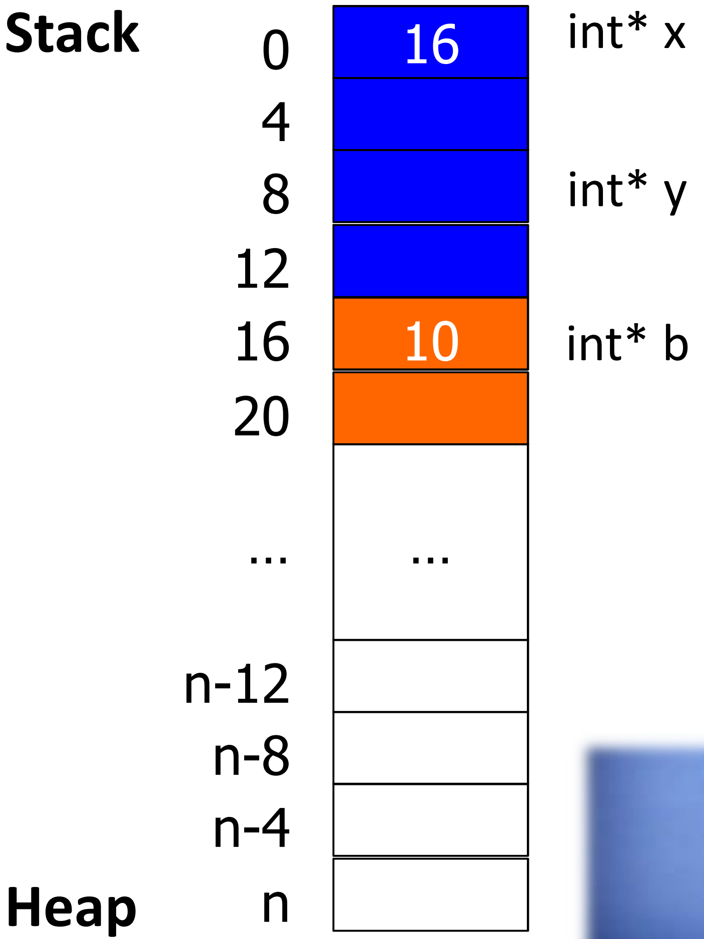
```
int* foo() {
  int b = 10;
  return &b;
}

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

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



Stack	0	16	int* x
	4		
	8		int* y
	12		
	16	10	int* b
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n		

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();
}
```



Stack	0	16	int* x
	4		
	8		int* y
	12		
	16	10	int* b
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n		

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();
}
```

Stack	0	16	int* x
	4		
	8		int* y
	12		
	16	n	int* b
	18		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n	?	

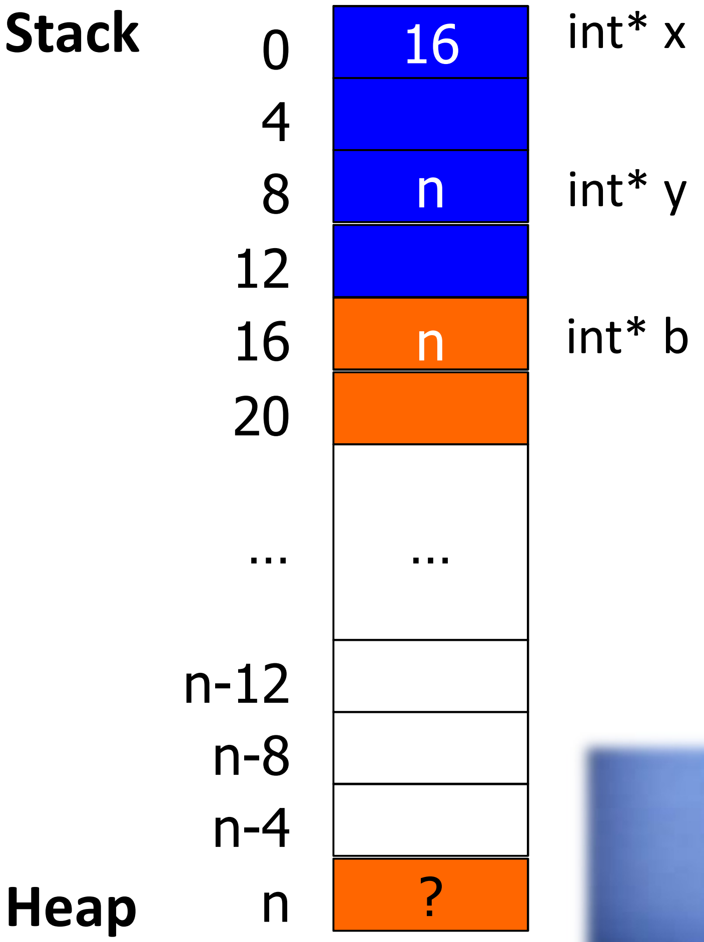
C Heap Allocation

```
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();
}
```

Stack	0	16	int* x
	4		
	8		int* y
	12		
	16	n	int* b
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n	?	

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();
}
```



Stack	0	16	int* x
	4		
	8	n	int* y
	12		
	16	n	int* b
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n	?	

C Heap Allocation

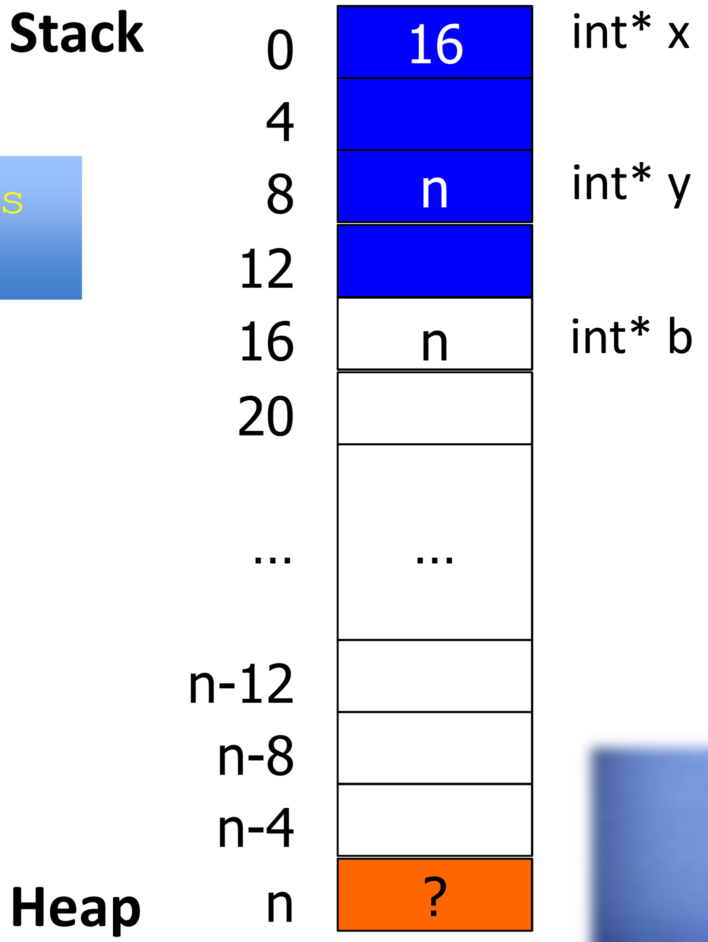
```
int* foo() {
  int b = 10;
  return &b;
}

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

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

Stack	0	16	int* x
	4		
	8	n	int* y
	12		
	16	n	int* b
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n	?	

C Heap Allocation

```c
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();
}
```

This is bad!



Stack	Address	Value	Variable
Stack	0	16	int* x
	4		
	8	n	int* y
	12		
	16	n	int* b
	20		
	…	…	
	n-12		
	n-8		
	n-4		
Heap	n	?	

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

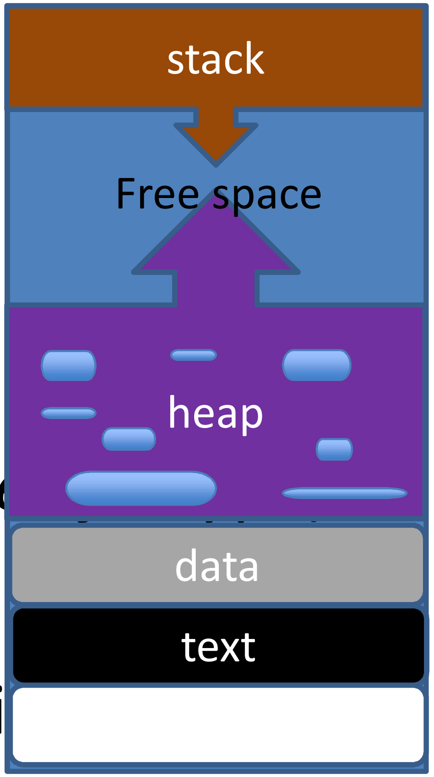
- **How to De-Allocate:**
  - The `free` function
  - Releases memory back to heap

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);

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 instanc*
- **Example:**
  - int* int_ptr = (int*)malloc(sizeof(i
  - free(int_ptr);



Stack vs Heap

- **Lifetime**
  - **Stack :** lifetime of a function (static)
  - **Heap :** lifetime of a program (dynamic)

Stack vs Heap

- **Lifetime**
  - **Stack :** lifetime of a function (static)
  - **Heap :** lifetime of a program (dynamic)
- **Memory Placement**

Stack vs Heap: Do we need both?

- **Yes**
- **Stack allocation is**
  - Simpler: Automatically deallocated

Stack vs Heap: Do we need both?

- **Yes**
- **Stack allocation is**
  - Simpler: Automatically deallocated
  - Faster

Stack vs Heap: Do we need both?

- **Yes**
- **Stack allocation is**
  - Simpler: Automatically deallocated
  - Faster
- **Heap allocation is used if**
  - you want to control the lifecycle of a variable

i-clicker question

Let’s Define the structure of linked list node as follows

```
struct node
{
   int data;
   struct node* next;
};
```

What is the best way to create a linked list node using malloc?

A. struct node* new_node = (struct node*) malloc(sizeof(struct node));

B. struct node* new_node = malloc(sizeof(struct node));

C. struct node new_node = malloc(sizeof(struct node));

D. struct node* new_node = (struct node*) malloc(10000);

POINTER TO POINTER

Pointer to pointer

int ** q;

int *p;

Before:

q->r->t, p->y

p=*q

After:

q->r->t, p->t

Pointer to pointer

int ** q;

int *p;

Before:

q->r->t, p->y

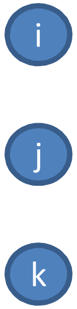
*q=p

After:

q->r->y, p->y

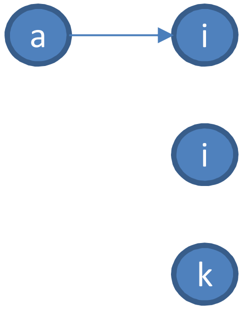
Example

```
int i, j, k; <=
int *a = &i;
int *b = &k;
a = &j;
int **p = &a;
int **q = &b;
p = q;
int *c = *q;
```



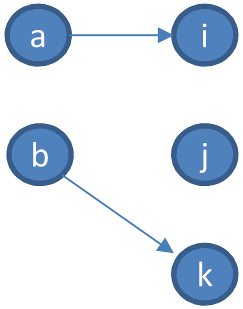
Example

```
int i, j, k;
int *a = &i; <=
int *b = &k;
a = &j;
int **p = &a;
int **q = &b;
p = q;
int *c = *q;
```



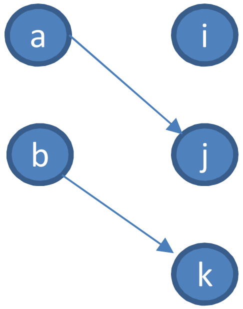
Example

```
int i, j, k;
int *a = &i;
int *b = &k; <=
a = &j;
int **p = &a;
int **q = &b;
p = q;
int *c = *q;
```



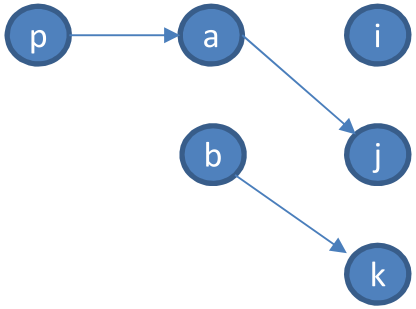
Example

```
int i, j, k;
int *a = &i;
int *b = &k;
a = &j; <=
int **p = &a;
int **q = &b;
p = q;
int *c = *q;
```



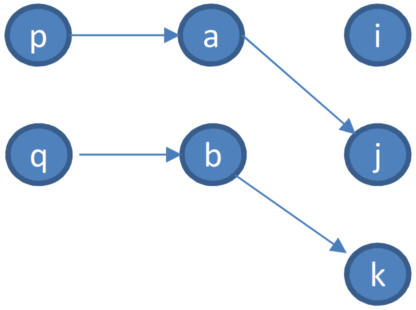
Example

```
int i, j, k;
int *a = &i;
int *b = &k;
a = &j;
int **p = &a; <=
int **q = &b;
p = q;
int *c = *q;
```



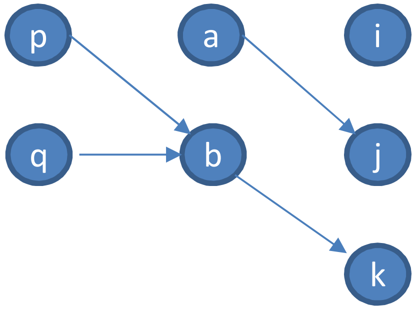
Example

```
int i, j, k;
int *a = &i;
int *b = &k;
a = &j;
int **p = &a;
int **q = &b; <=
p = q;
int *c = *q;
```



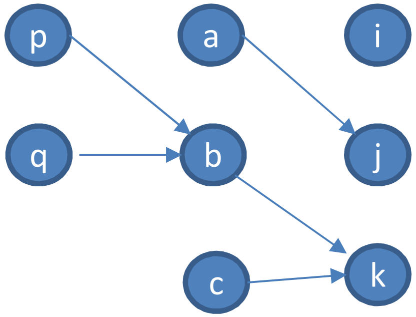
Example

```
int i, j, k;
int *a = &i;
int *b = &k;
a = &j;
int **p = &a;
int **q = &b;
p = q; <=
int *c = *q;
```



Example

```
int i, j, k;
int *a = &i;
int *b = &k;
a = &j;
int **p = &a;
int **q = &b;
p = q;
int *c = *q; <=
```



Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable “a” points to in the end?)**

```
p = &a;
q= &b;
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```

Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable “a” points to in the end?)**

p = &a; <=

q= &b;

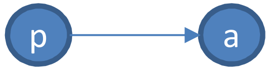
***p = q;**

r = &c;

s = p;

t = *p;

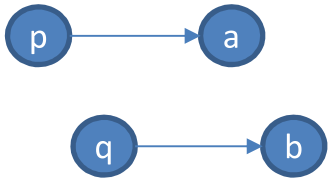
***s = r;**



Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable "a" points to in the end?)**

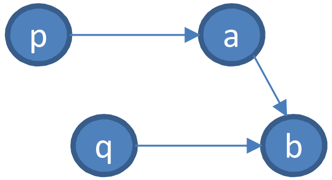
```
p = &a;
q= &b; <=
*p = q;
r = &c;
s = p;
t = *p;
*s = r;
```



Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable "a" points to in the end?)**

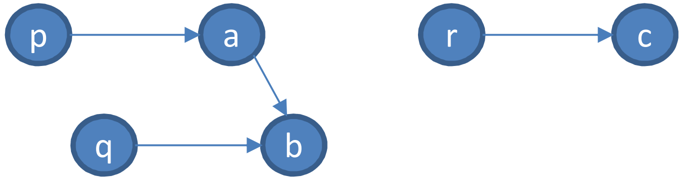
```
p = &a;
q= &b;
*p = q; <=
r = &c;
s = p;
t = *p;
*s = r;
```



Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable "a" points to in the end?)**

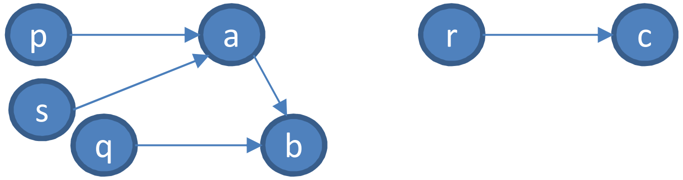
```
p = &a;
q= &b;
*p = q;
r = &c; <=
s = p;
t = *p;
*s = r;
```



Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable “a” points to in the end?)**

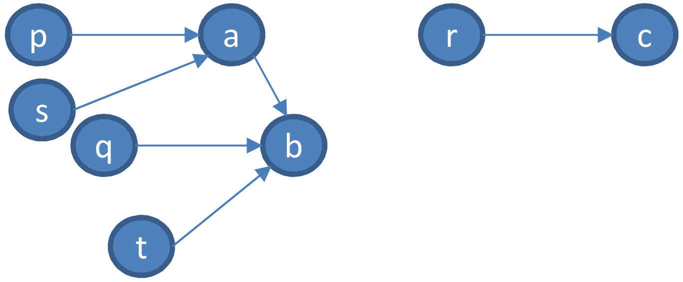
```
p = &a;
q= &b;
*p = q;
r = &c;
s = p; <=
t = *p;
*s = r;
```



Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable “a” points to in the end?)**

```
p = &a;
q= &b;
*p = q;
r = &c;
s = p;
t = *p; <=
*s = r;
```



Group Activity

- **Assume program consists of statements of form, draw the points-to graph. (e.g. what does the variable "a" points to in the end?)**

p = &a;

q= &b;

***p = q;**

r = &c;

s = p;

t = *p;

***s = r; <=**