

# Runtime Environments

---

Ronghui Gu

Spring 2020

Columbia University

\* Course website: <https://www.cs.columbia.edu/~rgu/courses/4115/spring2019>

\*\* These slides are borrowed from Prof. Edwards.

# Storage Classes

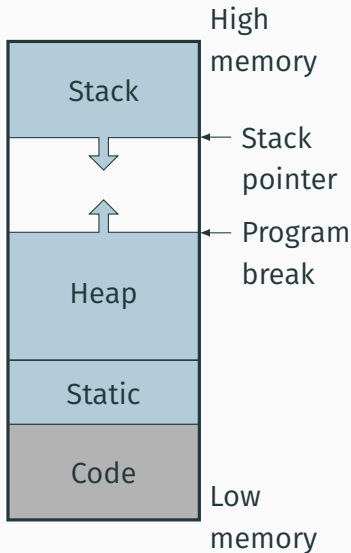
---

## Storage Classes and Memory Layout

**Stack:** objects created/destroyed in last-in, first-out order

**Heap:** objects created/destroyed in any order; automatic garbage collection optional

**Static:** objects allocated at compile time; persist throughout run



# Static Objects

```
class Example {  
    public static final int a = 3;  
  
    public void hello() {  
        System.out.println("Hello");  
    }  
}
```

## Examples

Static class variable

String constant "Hello"

Information about the  
Example class

## Advantages

Zero-cost memory  
management

Often faster access (address a  
constant)

No out-of-memory danger

## Disadvantages

Size and number must be  
known beforehand

Wasteful

# The Stack and Activation Records

---

# Stack-Allocated Objects

**Idea:** some objects persist from when a procedure is called to when it returns.

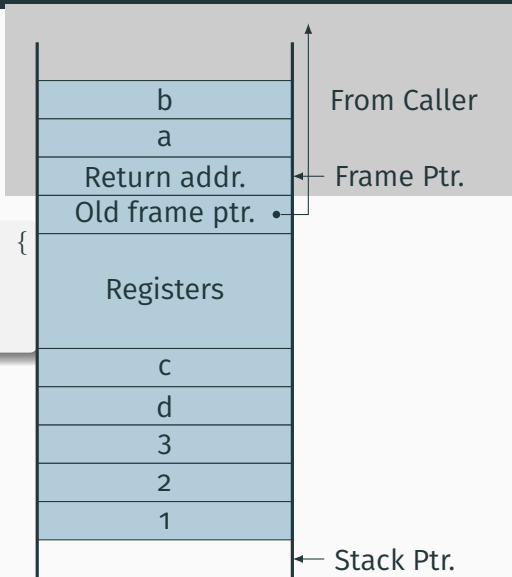
Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

Natural for supporting recursion.

Each invocation of a procedure gets its own *frame* (*activation record*) where it stores its own local variables and bookkeeping information.

## An Activation Record: The State Before Calling *bar*

```
int foo(int a, int b) {  
    int c, d;  
    bar(1, 2, 3);  
}
```



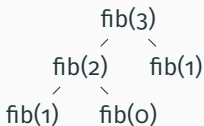
# Recursive Fibonacci

(Real C)

```
int fib(int n) {  
    if (n < 2)  
        return 1;  
    else  
        return  
            fib(n-1)  
            +  
            fib(n-2);  
}
```

(Assembly-like C)

```
int fib(int n) {  
    int tmp1, tmp2, tmp3;  
    tmp1 = n < 2;  
    if (!tmp1) goto L1;  
    return 1;  
L1: tmp1 = n - 1;  
    tmp2 = fib(tmp1);  
L2: tmp1 = n - 2;  
    tmp3 = fib(tmp1);  
L3: tmp1 = tmp2 + tmp3;  
    return tmp1;  
}
```

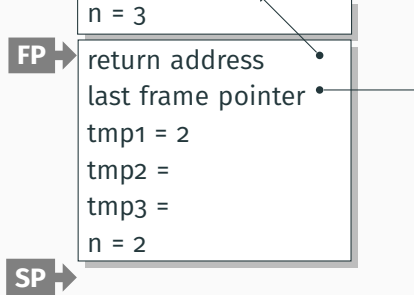


Executing fib(3)



```
int fib(int n) {  
    int tmp1, tmp2, tmp3;  
    tmp1 = n < 2;  
    if (!tmp1) goto L1;  
    return 1;  
L1: tmp1 = n - 1;  
    tmp2 = fib(tmp1);  
L2: tmp1 = n - 2;  
    tmp3 = fib(tmp1);  
L3: tmp1 = tmp2 + tmp3;  
    return tmp1;  
}
```

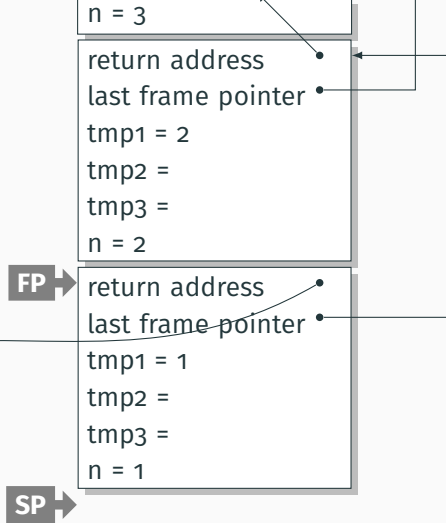
```
int fib(int n) {  
    int tmp1, tmp2, tmp3;  
    tmp1 = n < 2;  
    if (!tmp1) goto L1;  
    return 1;  
L1: tmp1 = n - 1;  
    tmp2 = fib(tmp1);  
L2: tmp1 = n - 2;  
    tmp3 = fib(tmp1);  
L3: tmp1 = tmp2 + tmp3;  
    return tmp1;  
}
```



```

int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}

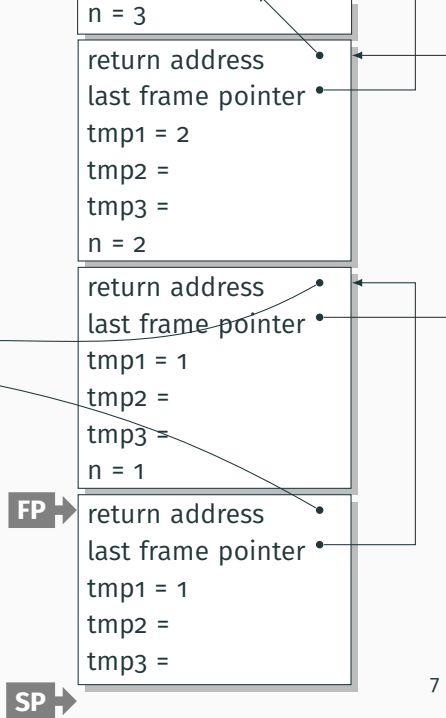
```



```

int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}

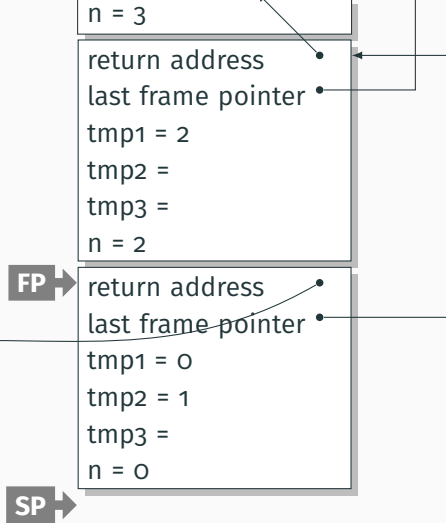
```



```

int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}

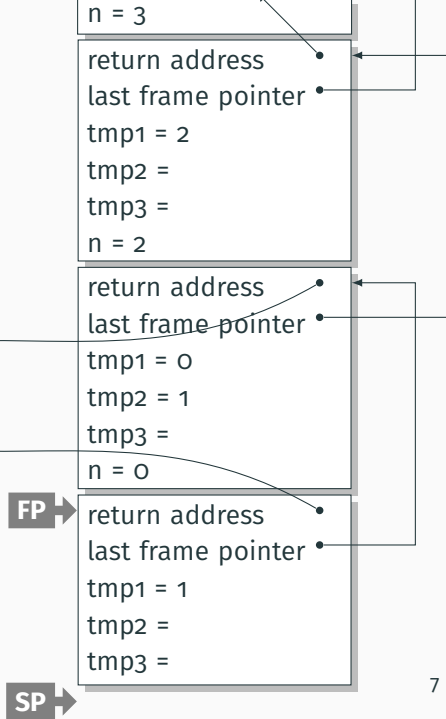
```



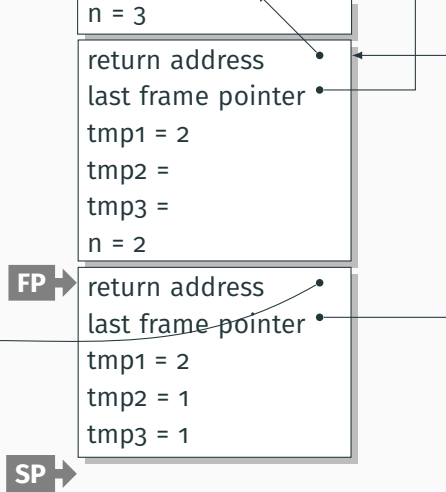
```

int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}

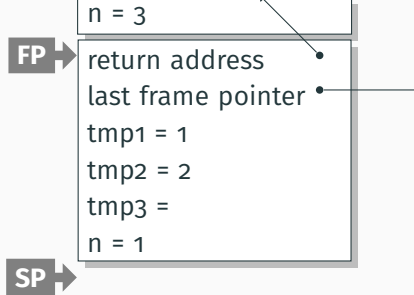
```



```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



```
int fib(int n) {  
    int tmp1, tmp2, tmp3;  
    tmp1 = n < 2;  
    if (!tmp1) goto L1;  
    return 1;  
L1: tmp1 = n - 1;  
    tmp2 = fib(tmp1);  
L2: tmp1 = n - 2;  
    tmp3 = fib(tmp1);  
L3: tmp1 = tmp2 + tmp3;  
    return tmp1;  
}
```

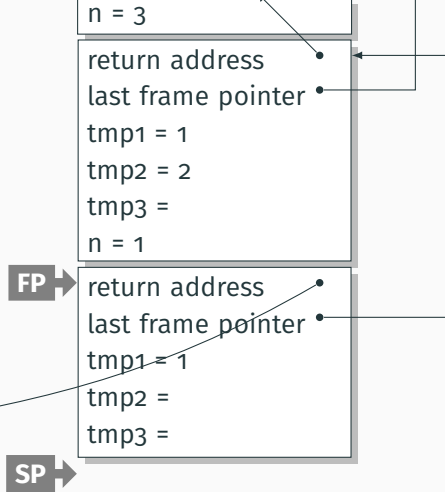




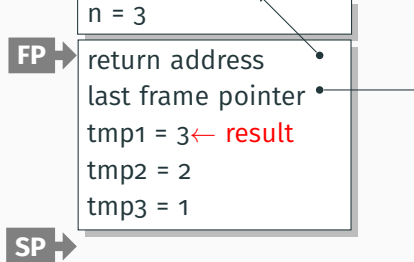
```

int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}

```



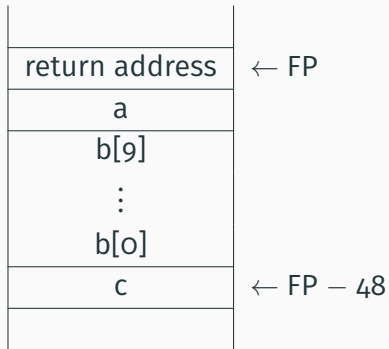
```
int fib(int n) {  
    int tmp1, tmp2, tmp3;  
    tmp1 = n < 2;  
    if (!tmp1) goto L1;  
    return 1;  
L1: tmp1 = n - 1;  
    tmp2 = fib(tmp1);  
L2: tmp1 = n - 2;  
    tmp3 = fib(tmp1);  
L3: tmp1 = tmp2 + tmp3;  
    return tmp1;  
}
```



## Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

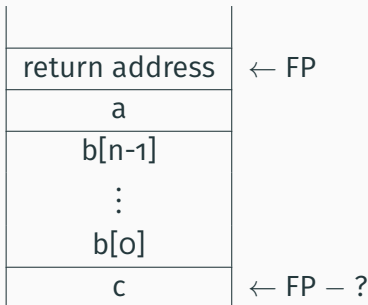
```
void foo()  
{  
    int a;  
    int b[10];  
    int c;  
}
```



## Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

```
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```

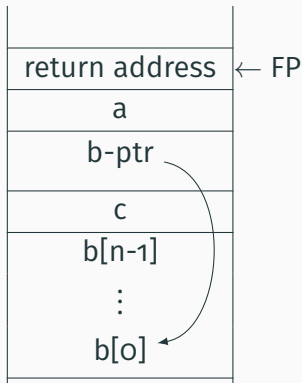


Doesn't work: generated code expects a fixed offset for c. Even worse for multi-dimensional arrays.

## Allocating Variable-Sized Arrays

As always:  
add a level of indirection

```
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```



Variables remain constant offset from frame pointer.

## Implementing Nested Functions with Access Links

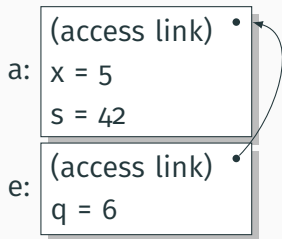
```
let a x s =  
  let b y =  
    let c z = z + s in  
    let d w = c (w+1) in  
    d (y+1) in (* b *)  
  let e q = b (q+1) in  
e (x+1) (* a *)
```

(access link) •  
a: x = 5  
s = 42

What does “a 5 42” give?

## Implementing Nested Functions with Access Links

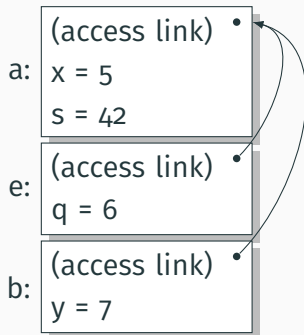
```
let a x s =  
  let b y =  
    let c z = z + s in  
    let d w = c (w+1) in  
    d (y+1) in (* b *)  
  let e q = b (q+1) in  
e (x+1) (* a *)
```



What does “a 5 42” give?

## Implementing Nested Functions with Access Links

```
let a x s =  
  let b y =  
    let c z = z + s in  
    let d w = c (w+1) in  
    d (y+1) in (* b *)  
  let e q = b (q+1) in  
e (x+1) (* a *)
```

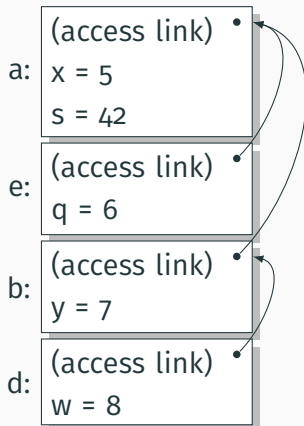


What does “a 5 42” give?



## Implementing Nested Functions with Access Links

```
let a x s =  
  let b y =  
    let c z = z + s in  
    let d w = c (w+1) in  
    d (y+1) in (* b *)  
  let e q = b (q+1) in  
e (x+1) (* a *)
```

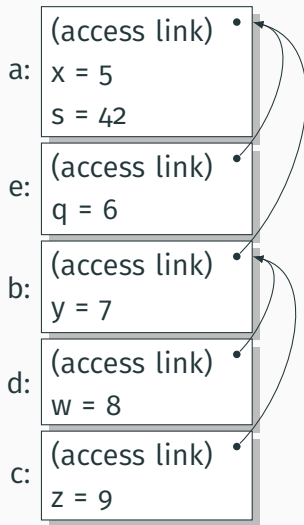


What does “a 5 42” give?

## Implementing Nested Functions with Access Links

```
let a x s =  
  let b y =  
    let c z = z + s in  
    let d w = c (w+1) in  
    d (y+1) in (* b *)  
  let e q = b (q+1) in  
e (x+1) (* a *)
```

What does “a 5 42” give?



# **In-Memory Layout Issues**

---

# Layout of Records and Unions

Modern processors have byte-addressable memory.

0  
1  
2  
3



The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

16-bit integer:

1 0

32-bit integer:

3 2 1 0

# Layout of Records and Unions

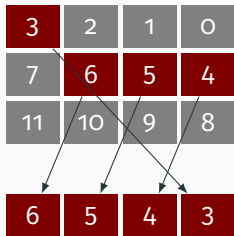
Modern memory systems read data in 32-, 64-, or 128-bit chunks:

3	2	1	0
7	6	5	4
11	10	9	8

Reading an aligned 32-bit value is fast: a single operation.

3	2	1	0
7	6	5	4
11	10	9	8

How about reading an **unaligned** value?

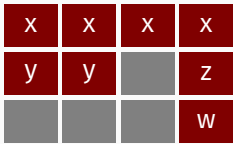


# Padding

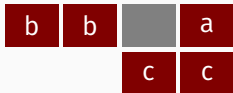
To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each  $n$ -byte object must start on a multiple of  $n$  bytes (no unaligned accesses).
- Any object containing an  $n$ -byte object must be of size  $mn$  for some integer  $m$  (aligned even when arrayed).

```
struct padded {  
    int x;    /* 4 bytes */  
    char z;   /* 1 byte  */  
    short y;  /* 2 bytes */  
    char w;   /* 1 byte  */  
};
```



```
struct padded {  
    char a;   /* 1 byte  */  
    short b;  /* 2 bytes */  
    short c;  /* 2 bytes */  
};
```



# Padding

To avoid unaligned accesses, the C compiler pads the layout of unions and records. Rules:

- Each  $n$ -byte object must start on a multiple of  $n$  bytes (no unaligned accesses).
- Any object containing an  $n$ -byte object must be of size  $mn$  for some integer  $m$  (aligned even when arrayed).

```
struct padded {  
    int x;    /* 4 bytes */  
    char z;  /* 1 byte  */  
    char w;  /* 1 byte  */  
    short y; /* 2 bytes */  
};
```

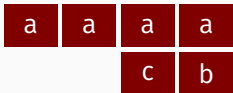


```
struct padded {  
    char a; /* 1 byte */  
    short b; /* 2 bytes */  
    short c; /* 2 bytes */  
};
```



## Padding: (1) or (2)?

```
struct padded {  
    int a; /* 4 bytes */  
    char b; /* 1 byte */  
    char c; /* 1 byte */  
};
```



(1)



(2)



# Unions

A C *union* shares one space among all fields

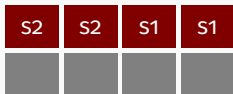
```
union intchar {  
    int i;    /* 4 bytes */  
    char c;  /* 1 byte  */  
};
```



```
union twostructs {  
    struct {  
        char c;    /* 1 byte */  
        int i;     /* 4 bytes */  
    } a;  
    struct {  
        short s1; /* 2 bytes */  
        short s2; /* 2 bytes */  
    } b;  
};
```



or



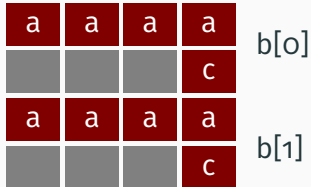
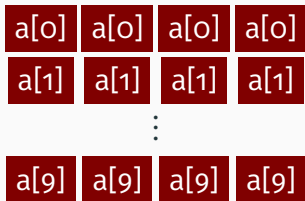
# Arrays

Basic policy in C: an array is just one object after another in memory.

```
int a[10];
```

What if we remove rule 2 of padding?

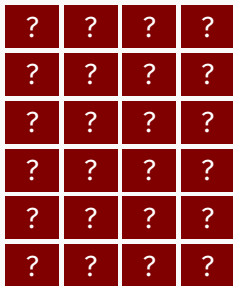
```
struct {  
    int a;  
    char c;  
} b[2];
```



# Arrays and Aggregate types

The largest primitive type  
dictates the alignment

```
struct {  
    short a;  
    short b;  
    char c;  
} d[4];
```



# Arrays and Aggregate types

The largest primitive type  
dictates the alignment

```
struct {  
    short a;  
    short b;  
    char c;  
} d[4];
```

b	b	a	a	d[0]
a	a		c	d[1]
	c	b	b	
b	b	a	a	d[2]
a	a		c	d[3]
	c	b	b	

# Arrays of Arrays

```
char a[4];
```

a[3]	a[2]	a[1]	a[0]
------	------	------	------

```
char a[3][4];
```

a[0][3]	a[0][2]	a[0][1]	a[0][0]	a[0]
a[1][3]	a[1][2]	a[1][1]	a[1][0]	a[1]
a[2][3]	a[2][2]	a[2][1]	a[2][0]	a[2]

# The Heap

---

## Heap-Allocated Storage

A *heap* is a region of memory where blocks can be **dynamically** allocated and deallocated in any order.

## Dynamic Storage Allocation in C

```
struct point {
    int x, y;
};

int play_with_points(int n)
{
    int i;
    struct point *points;

    points = malloc(n * sizeof(struct point));

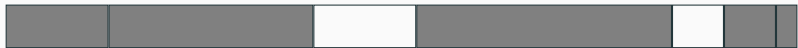
    for ( i = 0 ; i < n ; i++ ) {
        points[i].x = random();
        points[i].y = random();
    }

    /* do something with the array */

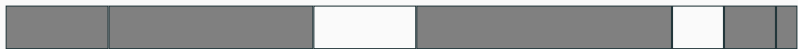
    free(points);
}
```



# Dynamic Storage Allocation

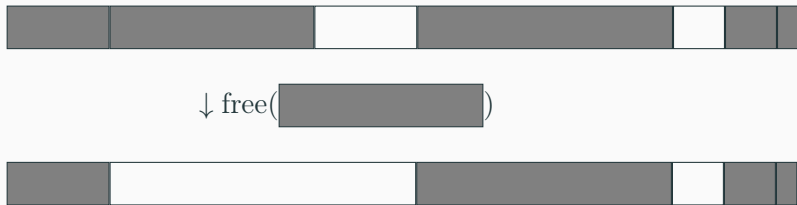


# Dynamic Storage Allocation

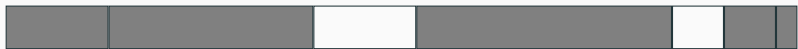


↓ free(  )

# Dynamic Storage Allocation



# Dynamic Storage Allocation

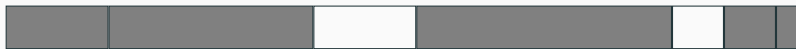


↓ free(  )



↓ malloc(  )

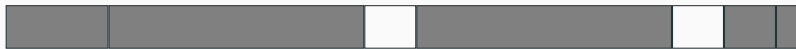
# Dynamic Storage Allocation



↓ free(  )



↓ malloc(  )



# Dynamic Storage Allocation

Rules:

- Each allocated block contiguous

- Blocks stay fixed once allocated

`malloc()`

`free()`

# Simple Dynamic Storage Allocation

Maintaining information about free memory

Simplest: Linked list

The algorithm for locating a suitable block

Simplest: First-fit

The algorithm for freeing an allocated block

Simplest: Coalesce adjacent free blocks

# Simple Dynamic Storage Allocation





# Simple Dynamic Storage Allocation



malloc(  )

# Simple Dynamic Storage Allocation



malloc(  )



# Simple Dynamic Storage Allocation

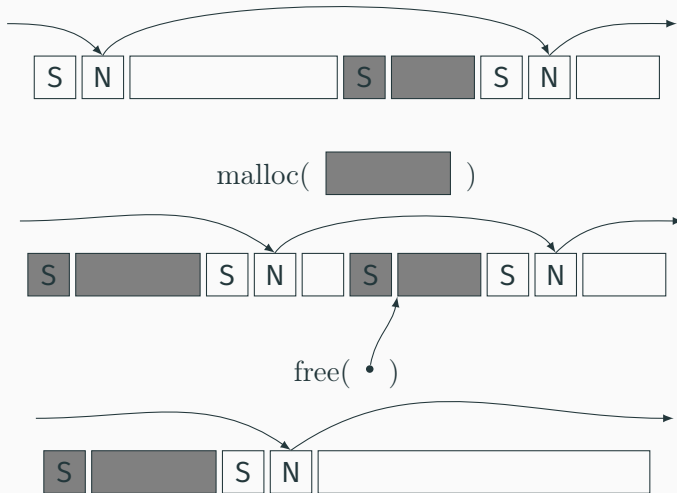


malloc(  )



free( • )

# Simple Dynamic Storage Allocation



# Fragmentation

malloc(  ) seven times give



free() four times gives



malloc(  ) ?

Need more memory; can't use fragmented memory.



Zebra

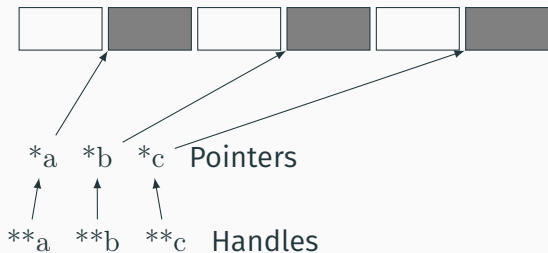


Tapir

# Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”

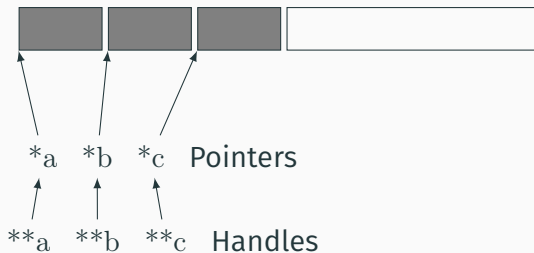


The original Macintosh did this to save memory.

# Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”



The original Macintosh did this to save memory.

# **Automatic Garbage Collection**

---



# Automatic Garbage Collection

Entrust the runtime system with freeing heap objects

Now common: Java, C#, Javascript, Python, Ruby, OCaml and most functional languages

**Advantages?**

**Disadvantages?**

# Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b

0	42, 17
---	--------

# Reference Counting

What and when to free?

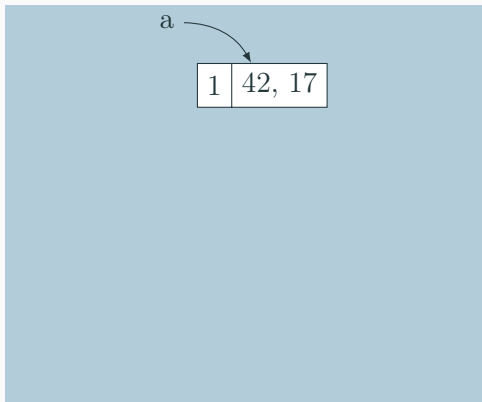
- Maintain count of references to each object
- Free when count reaches zero

let `a = (42, 17)` in

let `b = [a;a]` in

let `c = (1,2)::b` in

`b`

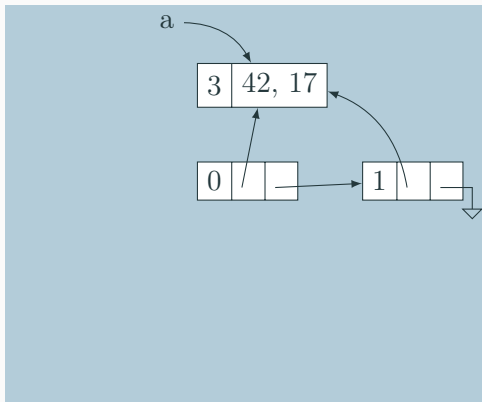


# Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in  
let b = [a;a] in  
let c = (1,2)::b in  
b
```



# Reference Counting

What and when to free?

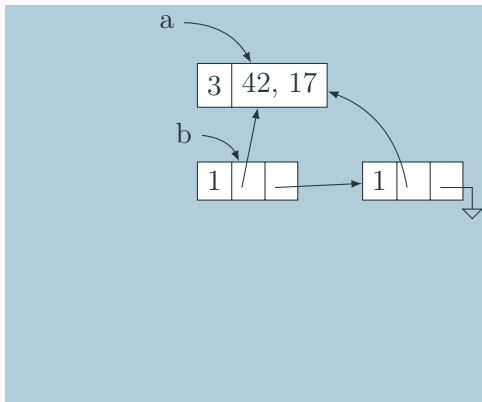
- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b

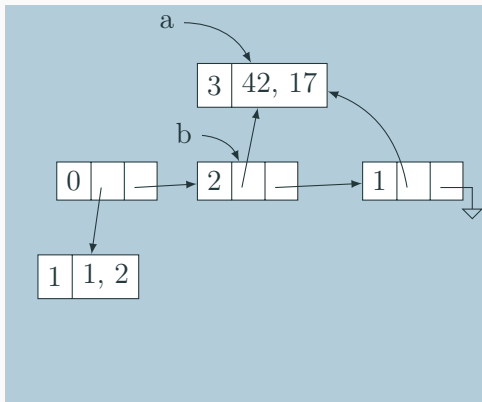


# Reference Counting

What and when to free?

- Maintain count of references to each object
- Free when count reaches zero

```
let a = (42, 17) in
let b = [a;a] in
let c = (1,2)::b in
b
```





# Reference Counting

What and when to free?

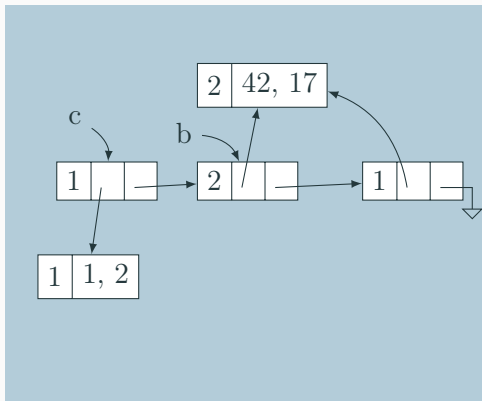
- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b





# Reference Counting

What and when to free?

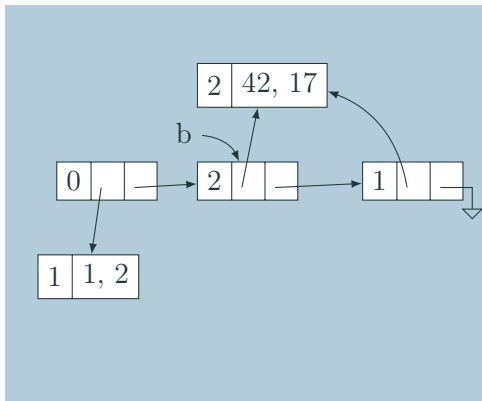
- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b



# Reference Counting

What and when to free?

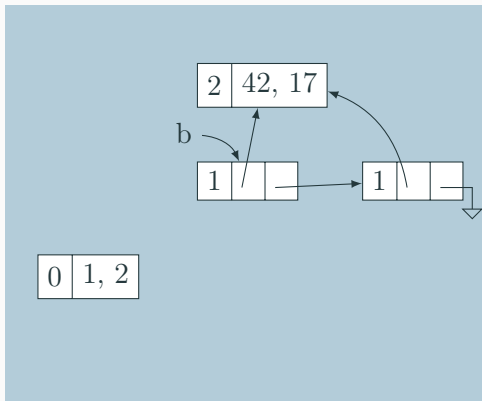
- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b



# Reference Counting

What and when to free?

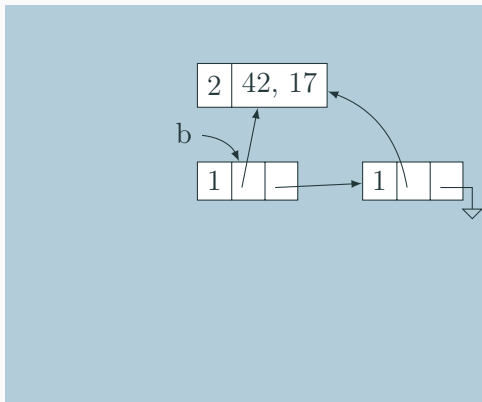
- Maintain count of references to each object
- Free when count reaches zero

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b



# Issues with Reference Counting

Circular structures defy reference counting?



# Mark-and-Sweep

What and when to free?

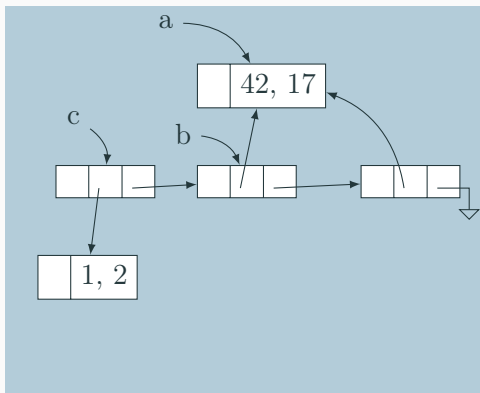
- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b



# Mark-and-Sweep

What and when to free?

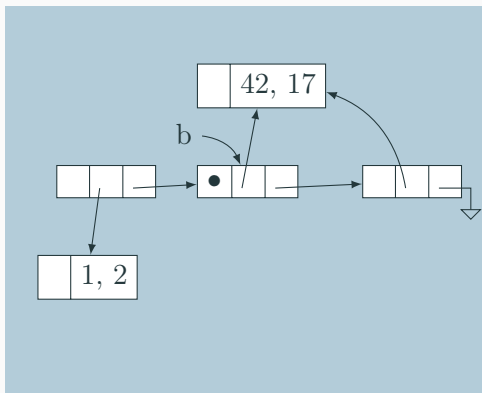
- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b



# Mark-and-Sweep

What and when to free?

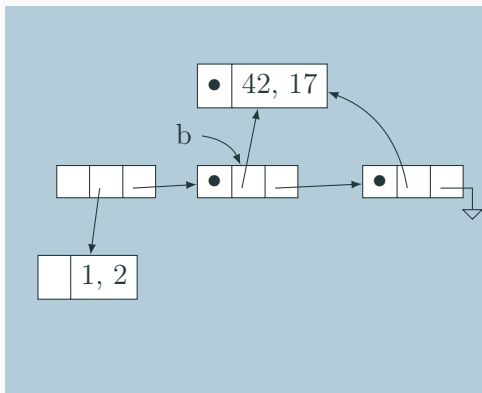
- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b



# Mark-and-Sweep

What and when to free?

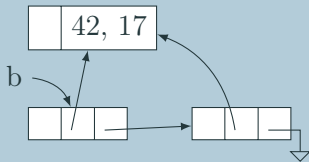
- Stop-the-world algorithm invoked when memory full
- Breadth-first-search marks all reachable memory
- All unmarked items freed

let a = (42, 17) in

let b = [a;a] in

let c = (1,2)::b in

b





# Mark-and-Sweep

Mark-and-sweep is faster overall; may induce big pauses

Mark-and-compact variant also moves or copies reachable objects to eliminate fragmentation

Incremental garbage collectors try to avoid doing everything at once

Most objects die young; generational garbage collectors segregate heap objects by age

Parallel garbage collection tricky

Real-time garbage collection tricky