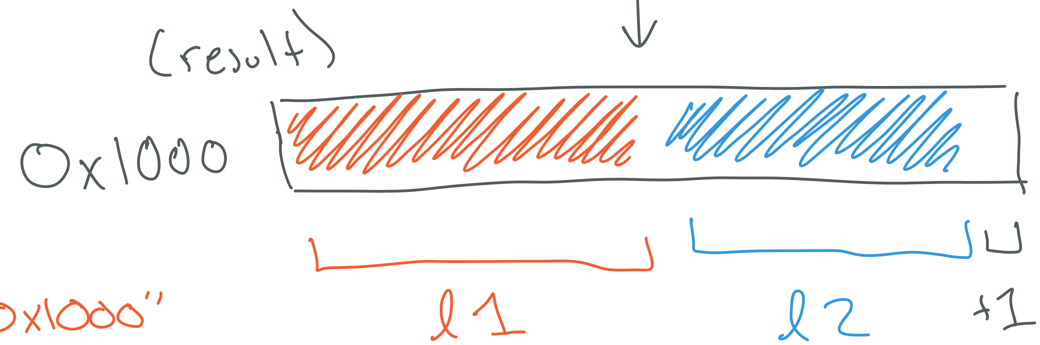
```c
char* concat(char str1[], char str2[]) {
        int l1 = strlen(str1);
        int l2 = strlen(str2);
        char* result = malloc(l1 + l2 + 1);
```

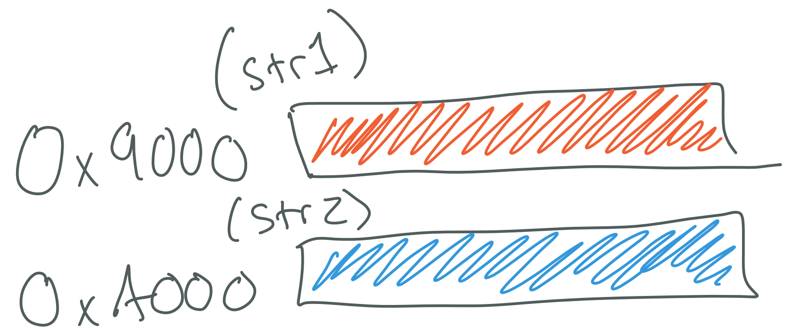
this address to start second strcpy!



strcpy(result, str1) "copy from 0x9000 to 0x1000"

strcpy($\underbrace{\text{\&result[l1]}}$, str2) "copy from 0xA000 to 0x1000 + l1 bytes"

strcpy($\underbrace{\text{result + l1}}$, str2)



both of the 1st args to strcpy compute "0x1000 + l1"

Imagine an example like this for int32_t data.

&result[index] must multiply index by 4 to get to the right address

result + index must also multiply by 4 (gcc does this!)

concat() from Tuesday:

```c
char* concat(char str1[], char str2[]) {
	int l1 = strlen(str1);
	int l2 = strlen(str2);
	char* result = malloc(l1 + l2 + 1);
	for(int i = 0; i < l1; i += 1) {
		result[i] = str1[i];
	}
	for(int j = l1; j < l1 + l2; j += 1) {
		result[j] = str2[j - l1];
	}
	result[l1 + l2] = '\0';
	return result;
}
```

Can we use strcpy to simplify our concat()? How?

→ replace with strcpy(result, str1)

→ replace with $\underbrace{\text{strcpy(result, str2)}}_{\text{would overwrite the beginning of result again}}$ X

strcpy(result[l1], str2) X

what about this option?

expected char*, got char

strcpy

<cstring>

`char * strcpy ( char * destination, const char * source );`

Copy string

Copies the C string pointed by ***source*** into the array pointed by ***destination***, including the terminating null character (and stopping at that point).

To avoid overflows, the size of the array pointed by ***destination*** shall be long enough to contain the same C string as ***source*** (including the terminating null character), and should not overlap in memory with ***source***.

strcpy(&result[l1], str2)

strcpy($\underbrace{\text{result + l1}}_{\text{Pointer arithmetic}}$, str2)

Compute the address l1 bytes after result

How can result + l1 when they are two different type?

Will it give an error?

← shift - 7

& expr

&

"ampersand"

Pointer Arithmetic

Not an error

ptr + n where ptr is a pointer T*
n is an integer

compute the address n * (sizeof(T)) bytes
from ptr

ptr of type T*

ptr[index] access sizeof(T) bytes of memory
at (index * sizeof(T)) bytes after ptr

Structs in C

makes "Point" an abbrev for "struct Point"

"technically" the outline is a struct definition

```c
typedef struct Point {
    int x;
    int y;
} Point;
```

This is a struct declaration, typically at top level of file.

Look up "struct packing" online about how things stored in order in structs

```c
//inside a function
    Point p = {4, 5};
    Point p2 = {22, 777};
```

variable declarations of a struct type allocate stack space for the struct

```c
// field access (member access)
    p.x        p.y
    p2.x       p2.y
```

```c
// field update
    p.x = v        p.y = v
    p2.x = v       p2.y = v
```

```c
Point make_Point(int x, int y) { .... }
```

```c
typedef struct Point {
        int x, y;
} Point;

void example1() {
        Point p1 = { 4, 5 };
        Point p2 = { 200, 900 };
        printf("p1: %d, %d\tp2: %d %d\n", p1.x, p1.y, p2.x, p2.y);

        Point p3 = p2;
        p3.x = 777;
        printf("p2: %d, %d\tp3: %d %d\n", p2.x, p2.y, p3.x, p3.y);
}
```

Struct variable decls. create space on the stack for that struct

```
[jpolitz@ieng6-203]:ss1-25-06-w3r-string-list:372$ ./point
p1: 4, 5        p2: 200 900
p2: 200, 900    p3: 777 900
```

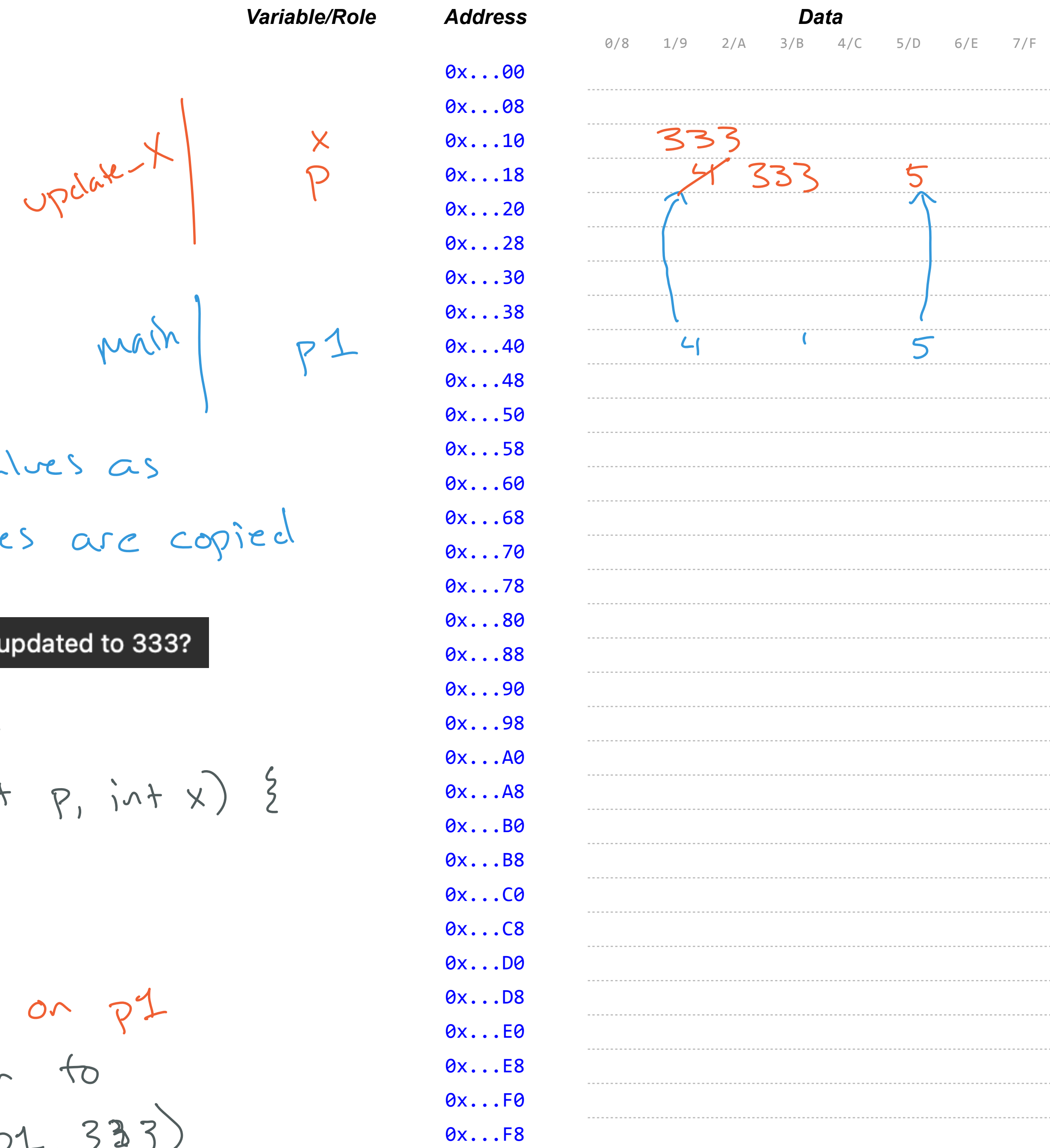
```c
Point make_Point(int x, int y) {
        Point p = { x, y };
        return p;
}
void example2() {
        Point p1 = make_Point(4, 5);
        Point p2 = make_Point(200, 900);
        printf("p1: %d, %d\tp2: %d %d\n", p1.x, p1.y, p2.x, p2.y);
}
```

```
[jpolitz@ieng6-203]:ss1-25-06-w3r-string-list:374$ ./point
p1: 4, 5          p2: 200 900
```

Returning or assigning a struct copies its members' values.

Variable/Role	Address	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...00								
	0x...08								
	0x...10								
	0x...18								
	0x...20								
	0x...28								
	0x...30								
	0x...38								
p3	0x...40		~~200~~		777		900		
p2	0x...48		200				900		
p1	0x...50		4				5		
	0x...58								
	0x...60								
	0x...68								
	0x...70								
	0x...78								
	0x...80								
	0x...88								
	0x...90								
make_Point	0x...98								
y	0x...A0		5		900				
x	0x...A8		4		200				
p	0x...B0		4	200			5		900
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
main	0x...D8								
p2	0x...E0			200					900
p1	0x...E8		4			5			
	0x...F0								
	0x...F8								

```c
void update_X(Point p, int x) {
        p.x = x;
}
void example3() {
        Point p1 = { 4, 5 };
        update_X(p1, 333);
        printf("p1.x: %d\n", p1.x);
}
```



$ ./point
p1.x: 4

When passing struct values as arguments, members' values are copied

if update_x were to return p, would p.x then be updated to 333?

If the only change was

```c
Point update_X(Point p, int x) {
        p.x = x;
        return p;
}
```

This has no effect on p1

If we also changed main to

```c
p1 = update_X(p1, 333)
```

This does change p1 by copying return via assignment

```c
void update_X_ptr(Point* p, int x) {
        p->x = x;
}
void example4() {
        Point p1 = { 4, 5 };
        update_X_ptr(&p1, 444);
        printf("p1.x: %d\n", p1.x);
}
```

$ ./point
p1.x: 444

p -> x

means at offset of x
member from address stored in p

what value?

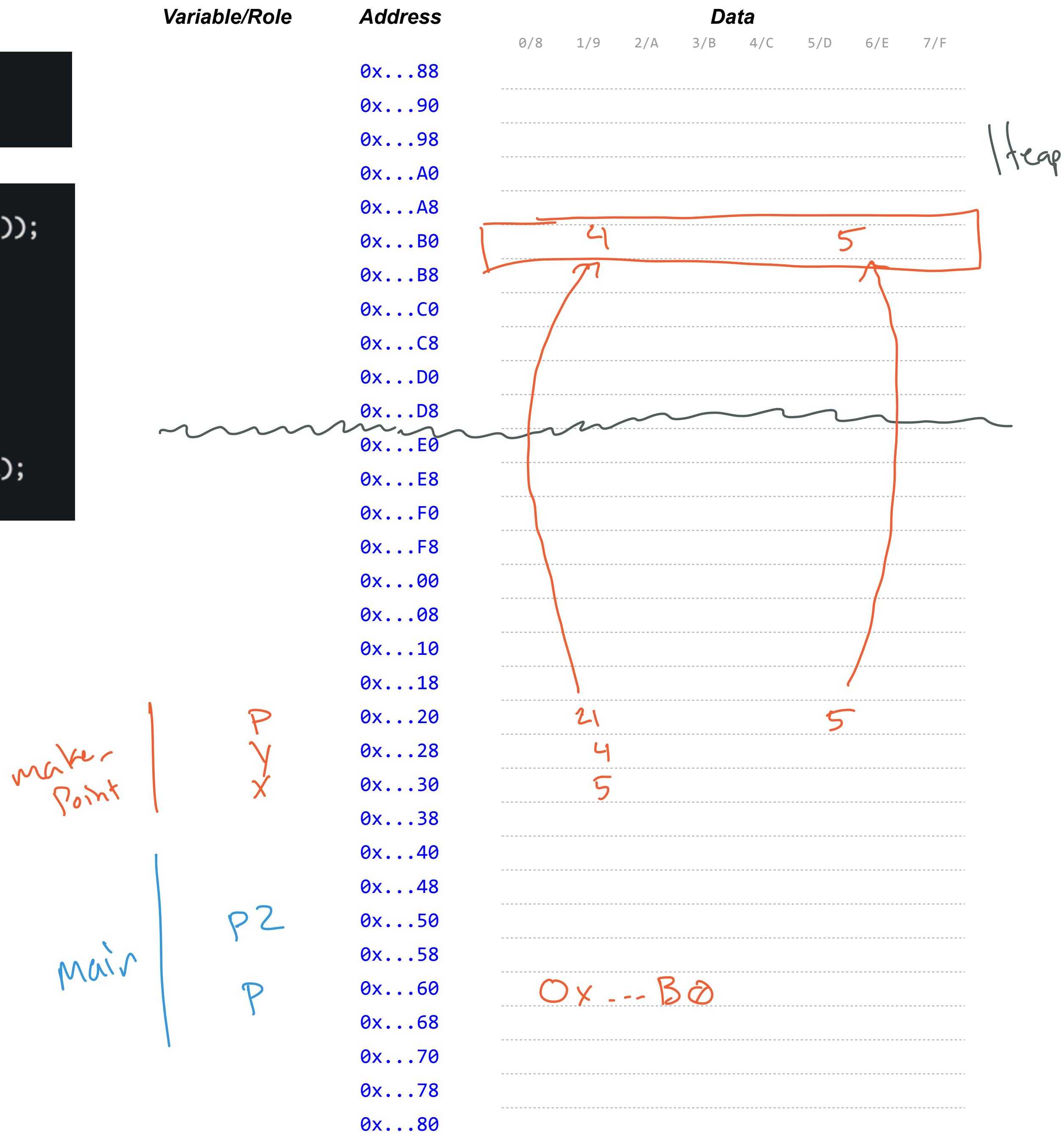
Variable/Role	Address	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...88								
	0x...90								
	0x...98								
u_X_ptr: x	0x...A0	444							
u_X_ptr: p	0x...A8	0x...D0							
	0x...B0								
	0x...B8								
	0x...C0								
	0x...C8								
main: p1	0x...D0	~~4~~ 444					5		
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								
	0x...00								
	0x...08								
	0x...10								
	0x...18								
	0x...20								
	0x...28								
	0x...30								
	0x...38								
	0x...40								
	0x...48								
	0x...50								
	0x...58								
	0x...60								
	0x...68								
	0x...70								
	0x...78								
	0x...80								

```c
void update_X_ptr(Point* p, int x) {
        p->x = x;
}
```

```c
void example5() {
        printf("sizeof(Point): %ld\n", sizeof(Point));
        Point* p = malloc(sizeof(Point));
        *p = make_Point(4, 5);
        update_X_ptr(p, 555);
        printf("p->x: %d\n", p->x);

        Point* p2 = p;
        update_X_ptr(p2, 888);
        printf("p->x: %d, p2->x: %d\n", p->x, p2->x);
}
```

```
sizeof(Point): 8
p->x: 555
p->x: 888, p2->x: 888
```



*p = val

assign into memory at the address stored in p the value val

```c
void update_X_ptr(Point* p, int x) {
        p->x = x;
}
```

```c
void example5() {
        printf("sizeof(Point): %ld\n", sizeof(Point));
        Point* p = malloc(sizeof(Point));
        *p = make_Point(4, 5);
        update_X_ptr(p, 555);
        printf("p->x: %d\n", p->x);

        Point* p2 = p;
        update_X_ptr(p2, 888);
        printf("p->x: %d, p2->x: %d\n", p->x, p2->x);
}
```

```
sizeof(Point): 8
p->x: 555
p->x: 888, p2->x: 888
```

this assignment just copies address in p into p2

Variable/Role	Address	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...88								
	0x...90								
	0x...98								
Heap	0x...A0								
	0x...A8								
	0x...B0	~~4~~ 555				5			
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								
	0x...00								
	0x...08								
	0x...10								
	0x...18								
update_X_ptr: x	0x...20	555							
	0x...28								
update_X_ptr: p	0x...30	0x...B0							
	0x...38								
	0x...40								
	0x...48								
example5: p2	0x...50	0x...B0							
	0x...58								
example5: p	0x...60	0x...B0							
	0x...68								
	0x...70								
	0x...78								
	0x...80								

*p = val

assign into memory at the address stored in p the value val

```c
void update_X_ptr(Point* p, int x) {
        p->x = x;
}
```

```c
void example5() {
        printf("sizeof(Point): %ld\n", sizeof(Point));
        Point* p = malloc(sizeof(Point));
        *p = make_Point(4, 5);
        update_X_ptr(p, 555);
        printf("p->x: %d\n", p->x);

        Point* p2 = p;
        update_X_ptr(p2, 888);
        printf("p->x: %d, p2->x: %d\n", p->x, p2->x);
}
```

```
sizeof(Point): 8
p->x: 555
p->x: 888, p2->x: 888
```

this assignment just copies address in p into p2

p and p2 refer to same heap-allocated struct "aliasing"

$*p = val$

assign into memory at the address stored in p the value val

Variable/Role	Address	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...88								
	0x...90								
Heap	0x...98								
	0x...A0								
	0x...A8								
	0x...B0	~~4~~ ~~555~~ 888				5			
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								
	0x...00								
	0x...08								
	0x...10								
	0x...18								
update_X_ptr: x	0x...20	888							
	0x...28								
update_X_ptr: p	0x...30	0x...B0							
	0x...38								
	0x...40								
	0x...48								
example5: p2	0x...50	0x...B0							
	0x...58								
example5: p	0x...60	0x...B0							
	0x...68								
	0x...70								
	0x...78								
	0x...80								

15 MIN BREAK RESUME 12:45

ptr + n — "pointer arithmetic" — add n * sizeof(T) to T* ptr
Computes new address of type T*

ptr[n] — "indexing" — look up sizeof(T) bytes of memory at offset n * sizeof(T) from ptr
Returns value of type T

& x — "addressof" — computes address of variable x
For x of type T, returns T*

& ptr[n] — "address of" — same meaning as ptr + n

* ptr — "dereference" — Return sizeof(T) bytes of memory at address stored in ptr of type T* (Return type T)

*ptr = val
ptr[n] = val — "assignment" — Compute addresses as above, but store val there rather than look up
ptr : T*   val : T

```c
Str join(Str delim, Str strs[], int size) {
        Str result = str("");
        // high-level strategy: use concat() in a for loop
        for(int i = 0; i < size; i += 1) {
                result = concat(result, strs[i]);
                if(i < size - 1) {
                        result = concat(result, delim);
                }
        }
        return result;
}
      Str abcd[] = { str("a"), str("b"), str("c"), str("d") };
      Str abcd_result = join(str("-"), abcd, 4);
      printf("Expect a-b-c-d: %s %d\n", abcd_result.data, abcd_result.bytes);
```

Heap

0xA0 "a\0"
0xB0 "b\0"
0xC0 "c\0"
0xD0 "d\0"
0xF0 "-\0"

made from concat

0x100 "\0"
0x1A0 "a\0"
0x1B0 "a-\0"
0x1C0 "a-b\0"
0x1D0 "a-b-\0"
⋮
0x200 "a-b-c-d\0"

none of these heap-allocated values are usable anymore!

join

result ~~[0, 0x100]~~ ~~[1, 0x1A0]~~ ~~[2, 0x1B0]~~ ~~[3, 0x1C0]~~ ~~[4, 0x1D0]~~ ... [7, 0x200]
delim [1, 0xF0]
strs 0x1000
size 4

main

abcd 0x1000
[1 0xA0]
[1 0xB0]
[1 0xC0]
[1 0xD0]

"garbage"
"memory leak"

This is why we have free(ptr) takes a malloc'd ptr and tells malloc the space can be re-used

As a programmer, find moments right before a heap-allocated value becomes unreachable, unusable, or otherwise not accessed again, and free at that point.

$\boxed{\text{RUST (proglang)}}$

```
[jpolitz@ieng6-203]:ss1-25-06-w3r-string-list:398$ valgrind ./str
==1193243== Memcheck, a memory error detector
==1193243== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1193243== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1193243== Command: ./str
==1193243==
abcdef 6
Should be hello,world: hello,world 11
Expect a-b-c-d: a-b-c-d 7
==1193243==
==1193243== HEAP SUMMARY:
==1193243==     in use at exit: 101 bytes in 23 blocks
==1193243==   total heap usage: 24 allocs, 1 frees, 1,125 bytes allocated
==1193243==
==1193243== LEAK SUMMARY:
==1193243==    definitely lost: 101 bytes in 23 blocks
==1193243==    indirectly lost: 0 bytes in 0 blocks
==1193243==      possibly lost: 0 bytes in 0 blocks
==1193243==    still reachable: 0 bytes in 0 blocks
==1193243==         suppressed: 0 bytes in 0 blocks
==1193243== Rerun with --leak-check=full to see details of leaked memory
==1193243==
==1193243== For lists of detected and suppressed errors, rerun with: -s
==1193243== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
    // How to write a test for join()?
    // join(str(","), {str("hello"), str("world")]) -> str("hello, world")
    // join(str(","), {str("a"), str("b"), str("c")}) -> str("a,b,c")
    Str strs[] = { str("hello"), str("world") };
    Str result2 = join(str(","), strs, 2);
    printf("Should be hello,world: %s %d\n", result2.data, result2.bytes);

    Str abcd[] = { str("a"), str("b"), str("c"), str("d") };
    Str abcd_result = join(str("-"), abcd, 4);
    printf("Expect a-b-c-d: %s %d\n", abcd_result.data, abcd_result.bytes)
```

"block" = malloc

3 blocks

1
"\0"

2 3 4 5 6 7
"hello\0"

8 9 10 11 12 13 14
"hello,\0"

14 bytes

7 blocks

"\0"

"a\0"

"a-\0"

"a-b\0"

"a-b-\0"

"a-b-c\0"

"a-b-c-\0"

28 bytes

with a infinite loop with malloc and etc., and without freeing any of the heap memory, could we technically find the storage limit of the heap with valgrind?

just count + check when malloc() = NULL

what happens if you run out of space? i.e your memory leaks are larger than the space you have

malloc will return NULL!

does free only deletes the data in malloc?

free tells malloc that space can be re-used

free can only be used on a ptr that was returned from malloc

Variable/Role	Address	Data							
		0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
	0x...88								
	0x...90								
	0x...98								
	0x...A0								
	0x...A8								
	0x...B0								
	0x...B8								
	0x...C0								
	0x...C8								
	0x...D0								
	0x...D8								
	0x...E0								
	0x...E8								
	0x...F0								
	0x...F8								
	0x...00								
	0x...08								
	0x...10								
	0x...18								
	0x...20								
	0x...28								
	0x...30								
	0x...38								
	0x...40								
	0x...48								
	0x...50								
	0x...58								
	0x...60								
	0x...68								
	0x...70								
	0x...78								
	0x...80								