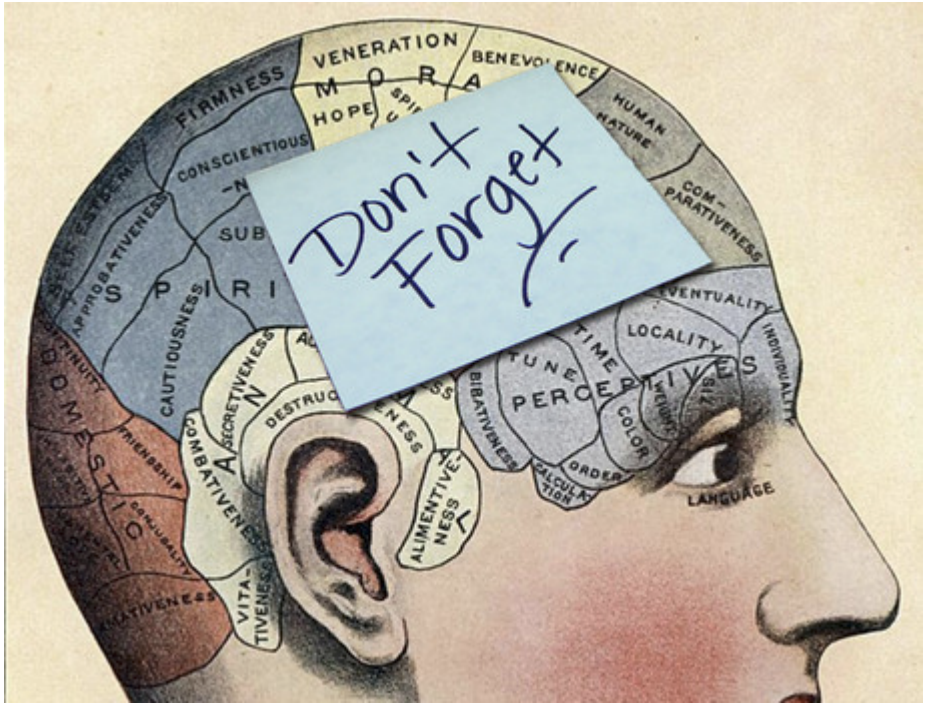
Memory!


VENERATION
FIRMNESS
CONSCIENTIOUS
HUMAN NATURE
COM-PARATIVENESS
LOCALITY
INDIVIDUALITY
TIME
COLOR
ORDER
LANGUAGE
CAUTIOUSNESS
DESTRUC
ALIMENTIVE-NESS
VITA-TIVENESS
FRIENDSHIP
Don't Forget

Java references

```
class Blob {...}

...
Blob b1 = new Blob();
Blob b2 = new Blob();
```

Java references

```
class Blob {...}

...
Blob b1 = new Blob();
Blob b2 = b1;
```

Pointer example

```
int *p;     // &p == 0x0;
int x = 6; // &x == 0xf8;
int y = 3; // &y == 0x4;
```

Pointer example

```
int *p;     // &p == 0x0;
int x = 6; // &x == 0xf8;
int y = 3; // &y == 0x4;
p = &x;
```

Pointer example

```
int *p;     // &p == 0x0;
int x = 6; // &x == 0xf8;
int y = 3; // &y == 0x4;
p = &x;
y = 1 + *p;
```

Pointer example

```
int *p;     // &p == 0x0;
int x = 6; // &x == 0xf8;
int y = 3; // &y == 0x4;
p = &x;
y = 1 + *p;
*p = 100;
```

Another pointer example

```
void swap(int* p1, int* p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
void main() {
    int x = 5;
    int y = 3;
    // x is 5, y is 3
    swap(&x, &y);
    // x is 3, y is 5
}
```

A Pointer on Pointers

- “… there are two things traditionally taught in universities as a part of a computer science curriculum which many people just never really fully comprehend: pointers and recursion.”
  – Joel Spolsky (founder, Stack Overflow)

A Pointer on Pointers

- “Pointers can feel quite intimidating when you first see them… If you’re feeling a bit lost… don’t panic! That’s totally normal. To get comfortable with pointers, you’ll need to spend a lot of time toying with code… and drawing lots of diagrams.” – Stanford lecture notes

A Pointer on Pointers

- “The topic of pointers is no doubt the hardest topic that we will cover/have covered for this class. The basic concepts of pointers take a lot of thinking to understand. This ***will*** be hard stuff…” UC Berkeley lecture notes

- Every variable in C has
  - A **value** and
  - An **address**
- Including pointers
- x → x’s value
- &x → x’s address

```
int main(void) {
```

	Value	Address
`int x = 3;`		`0xc0`

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`		0xc4

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`	4	0xc4
`int *p = &x;`		0xc8

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`	4	0xc4
`int *p = &x;`	0xc0	0xc8
`int *q = &y;`		0xcc

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`	4	0xc4
`int *p = &x;`	0xc0	0xc8
`int *q = &y;`	0xc4	0xcc
`int *r = p;`		0xd0

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`	4	0xc4
`int *p = &x;`	0xc0	0xc8
`int *q = &y;`	0xc4	0xcc
`int *r = p;`	0xc0	0xd0
`int *s = q;`		0xd4

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`	4	0xc4
`int *p = &x;`	0xc0	0xc8
`int *q = &y;`	0xc4	0xcc
`int *r = p;`	0xc0	0xd0
`int *s = q;`	0xc4	0xd4
`int a = *p;`		0xd8

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`	4	0xc4
`int *p = &x;`	0xc0	0xc8
`int *q = &y;`	0xc4	0xcc
`int *r = p;`	0xc0	0xd0
`int *s = q;`	0xc4	0xd4
`int a = *p;`	3	0xd8
`int b = *s;`		0xdc

```
}
```

Automatically allocated

```
int main(void) {
```

	Value	Address
`int x = 3;`	3	0xc0
`int y = 4;`	4	0xc4
`int *p = &x;`	0xc0	0xc8
`int *q = &y;`	0xc4	0xcc
`int *r = p;`	0xc0	0xd0
`int *s = q;`	0xc4	0xd4
`int a = *p;`	3	0xd8
`int b = *s;`	4	0xdc

```
}
```

Automatically allocated

Arrays in C

- `T x[y];  // declare an array of type T`
  `// named x of length y`
- `int x[10]; // array of 10 ints`
- `x[i]; // the ith index`
- `int x[3] = {1, 2, 3}; // initialize`

Arrays in C

```
int a[3]; // &a == 0x10


a[0] = 0xff;
a[2] = 0xcc;
a[3] = 0xbad; // !?
a[-1] = 0xbad; // !?
```

Arrays in C → Pointers

```
int a[3]; // &a == 0x10
int *b = a;
b[0] = 0xff;
b[2] = 0xcc;
b[3] = 0xbad; // !?
b[-1] = 0xbad; // !?
```

Pointer arithmetic

```
int a[3]; // &a == 0x10
int *b = &a[1];
b[-1]= 0xff;
b[1] = 0xcc;
b[2] = 0xbad; // !?
b[-2] = 0xbad; // !?
```

Pointer arithmetic

```
int a[3]; // &a == 0x10
int *b = a;
b[0] = 0xff; // *(b + 0)
b[2] = 0xcc; // *(b + 2)
```

Pointer arithmetic

```
int a[3]; // &a == 0x10
int *b = a;
*(b + 0) = 0xff; // b[0]
*(b + 2) = 0xcc; // b[2]
```

Pointer arithmetic

```
int a[3]; // &a == 0x10
int *b = a;
*(0 + b) = 0xff; // b[0]
*(2 + b) = 0xcc; // b[2]
```

Pointer arithmetic

```
int a[3]; // &a == 0x10
int *b = a;
0[b] = 0xff; // b[0]
2[b] = 0xcc; // b[2]
```

Resulting type

Operation Type	`&x`	`++x`	`*x`
`int x;`			
`int *x;`			
`int **x;`			

Resulting type

Operation Type	&x	++x	*x
int x;	int *	int	Error
int *x;	int **	int *	int
int **x;	int ***	int **	int *

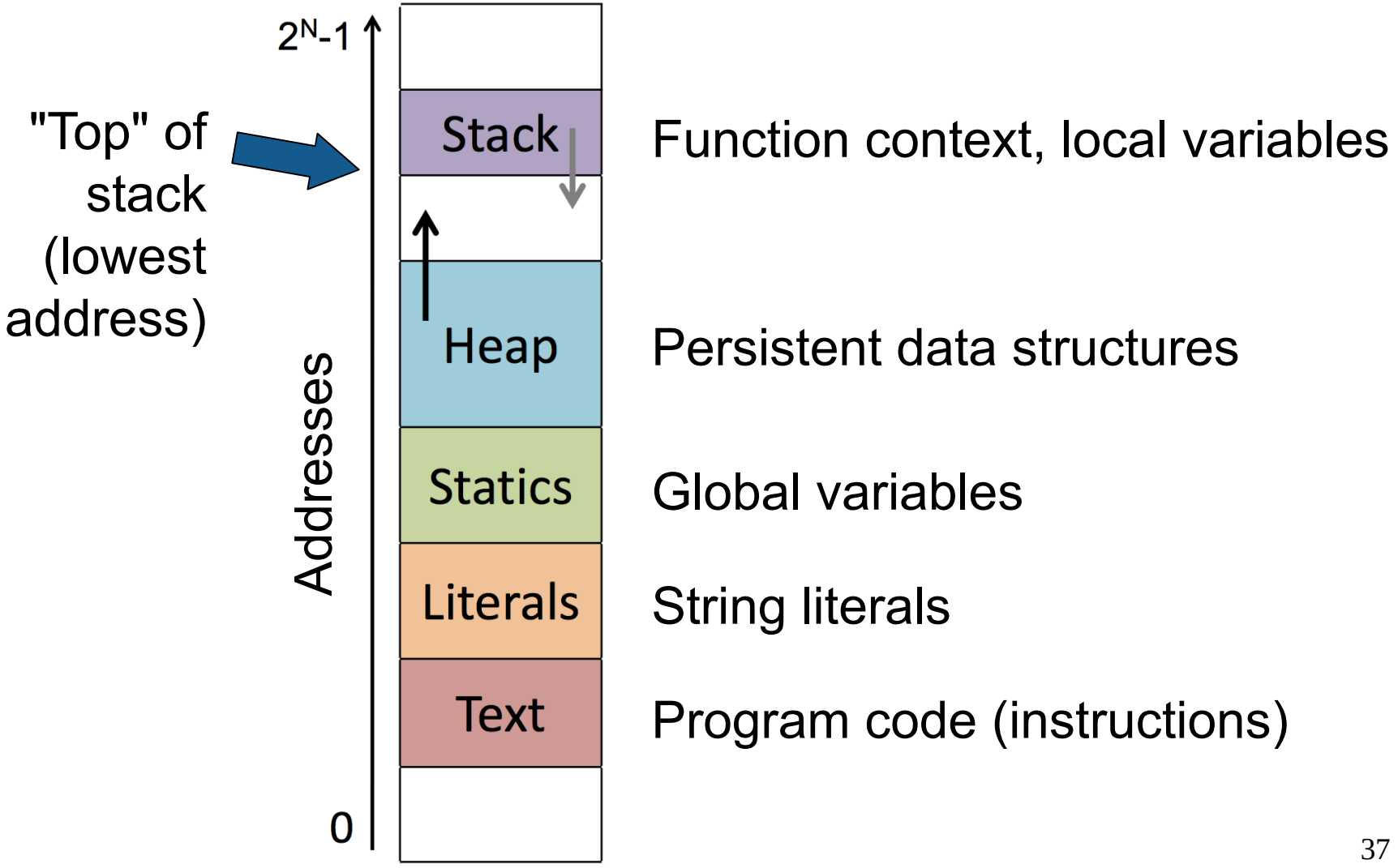
```c
#include <stdio.h>
#include <string.h>

int main() {

  char str[] = "Hello";
  size_t len = strlen(str);

  printf("String: %s\n", str);
  printf("Length: %zu\n", len);
  printf("Values: %d %d %d %d %d %d\n",
      str[0], str[1], str[2], str[3], str[4], str[5]);

  return 0;
}
```

```
// version 1: array notation
int string_length_v1(char str[]) {
  int len = 0;
  while (str[len]) { // same as checking != '\0' or != 0
    len++;
  }
  return len;
}
```

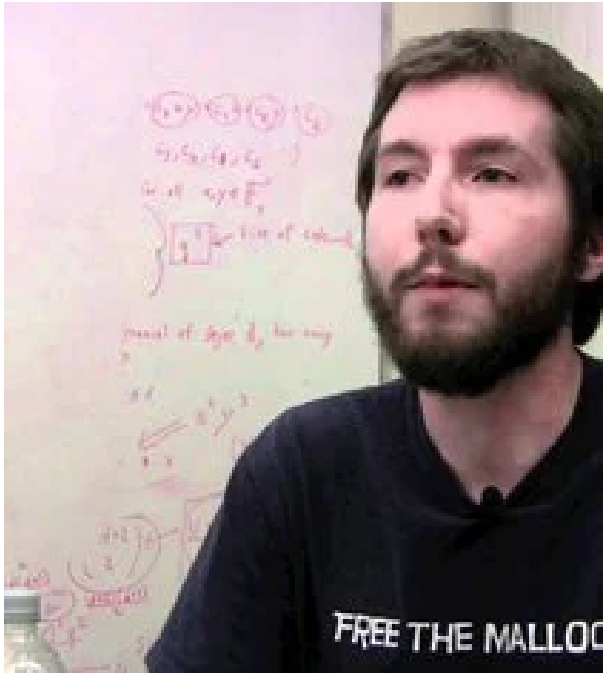
```
// version 2: pointers with quasi-array notation
int string_length_v2(char *str) {
  int len = 0;
  while (*(str + len)) { // same as checking != '\0' or != 0
    len++;
  }
  return len;
}
```

```
// version 3: pointers with pointer arithmetic
int string_length_v3(char *str) {
  int len = 0;
  while (*str) { // same as checking != '\0' or != 0
    str++; // pointer arithmetic
    len++;
  }
  return len;
}
```

```
// version 4: pointers with no counter
int string_length_v4(char *str) {
  char *p = str;
  while (*p) {  // same as checking != '\0' or != 0
    p++;
  }
  return p - str; // pointer subtraction!
}
```


"Top" of stack (lowest address)
2N-1
Addresses
0
Stack
Heap
Statics
Literals
Text
Function context, local variables
Persistent data structures
Global variables
String literals
Program code (instructions)

Free the Mallocs!



heisenbug: n.

A bug that disappears or alters its behavior when one attempts to probe or isolate it. (This usage is not even particularly fanciful; the use of a debugger sometimes alters a program's operating environment significantly enough that buggy code, such as that which relies on the values of uninitialized memory, behaves quite differently.) Antonym of Bohr bug; see also mandelbug, schroedinbug. In C, nine out of ten heisenbugs result from uninitialized auto variables, fandango on core phenomena (esp. lossage related to corruption of the malloc arena) or errors that smash the stack.