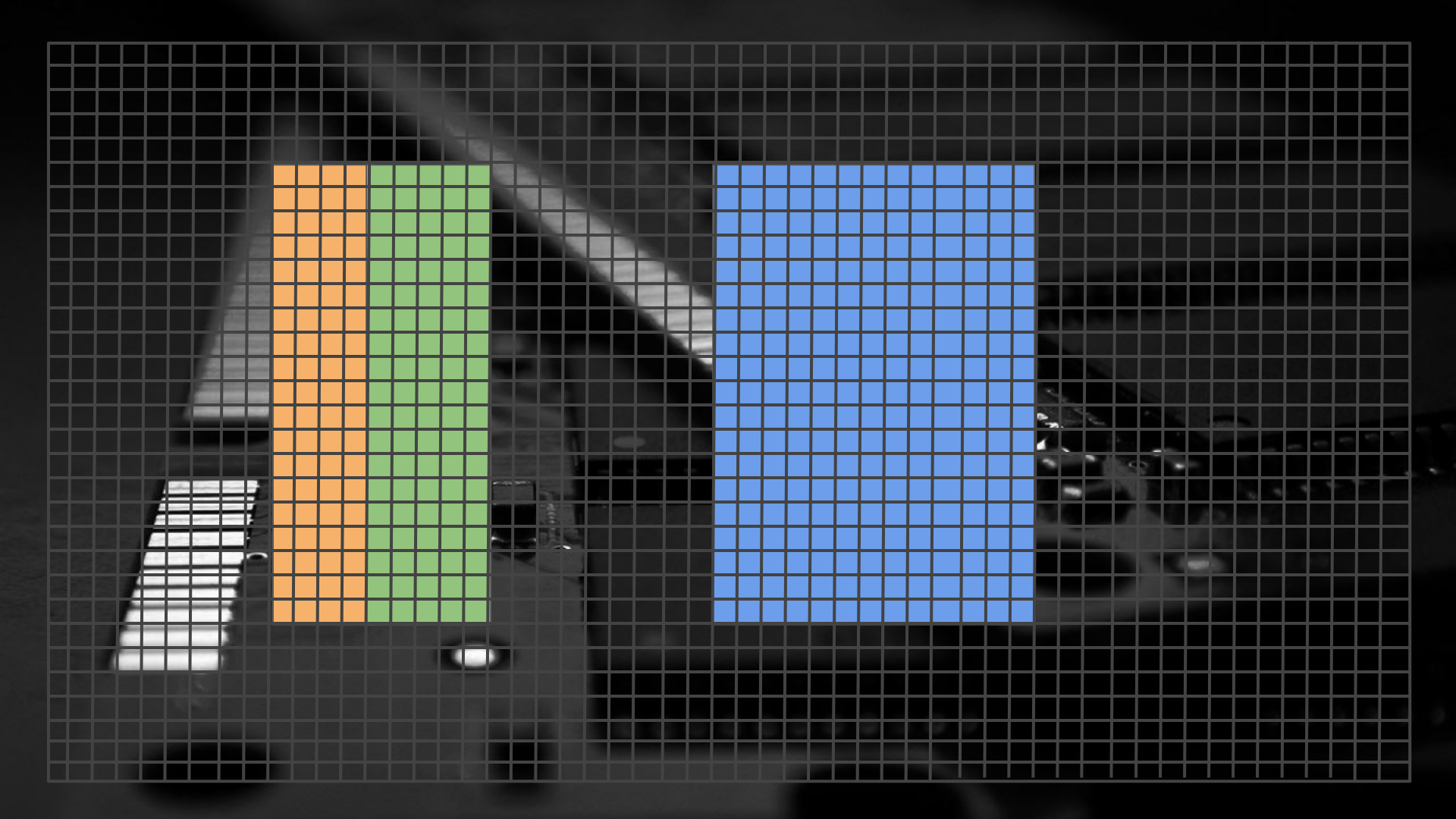
Practical CS

Memory Allocation and Garbage Collection

TechBash 2024

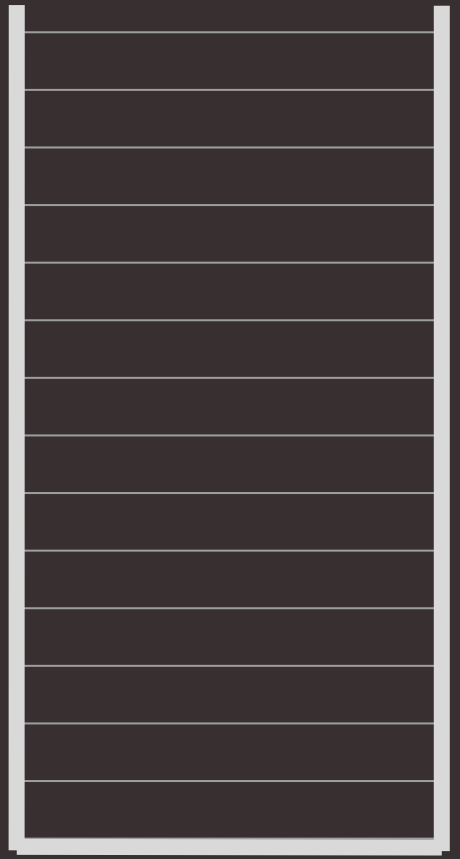
Four Kitchens


Jim.Vomero@FourKitchens.com
@nJim



Stack Memory

- Memory is allocated in continuous blocks.
- Type are declared when initialized and can not change.
- Can not grow beyond original allocation on the stack.
- Small static variables can be stored on the stack.
- Memory on stack is freed once stackframe is popped off.

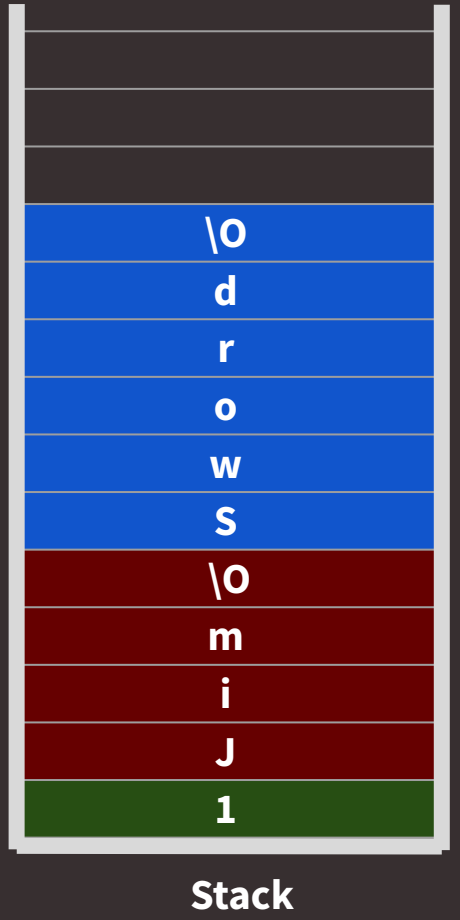


Stack

Stack Memory

- Memory is allocated in continuous blocks.
- Type are declared when initialized and can not change.
- Can not grow beyond original allocation on the stack.
- Small static variables can be stored on the stack.
- Memory on stack is freed once stackframe is popped off.

```
int main() {
  int level = 1;
  string name = "Jim";
  string inventory = "Sword";
  // Program continues …
}
```

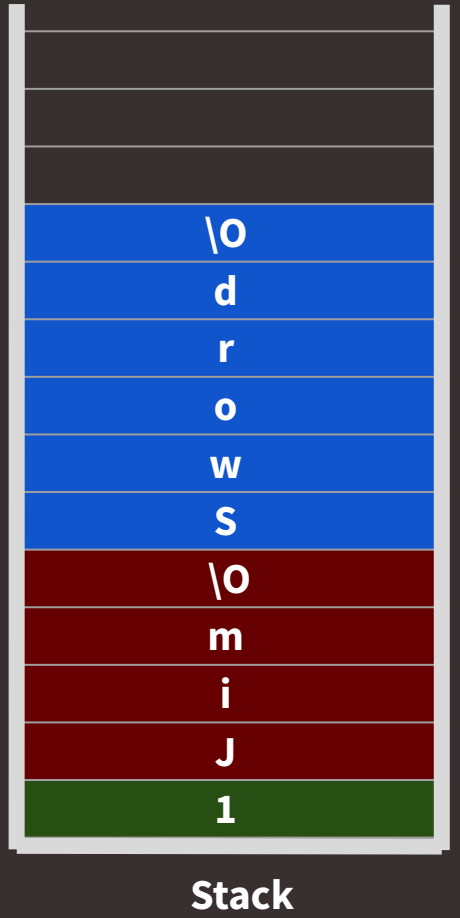


Stack Memory

- Memory is allocated in continuous blocks.
- Type are declared when initialized and can not change.
- Can not grow beyond original allocation on the stack.
- Small static variables can be stored on the stack.
- Memory on stack is freed once stackframe is popped off.

```
int main() {
  int level = 1;
  string name = "Jim";
  string inventory = "Sword";
  // Program continues …

  // error: invalid conversion of type
  level = "1-2";
}
```

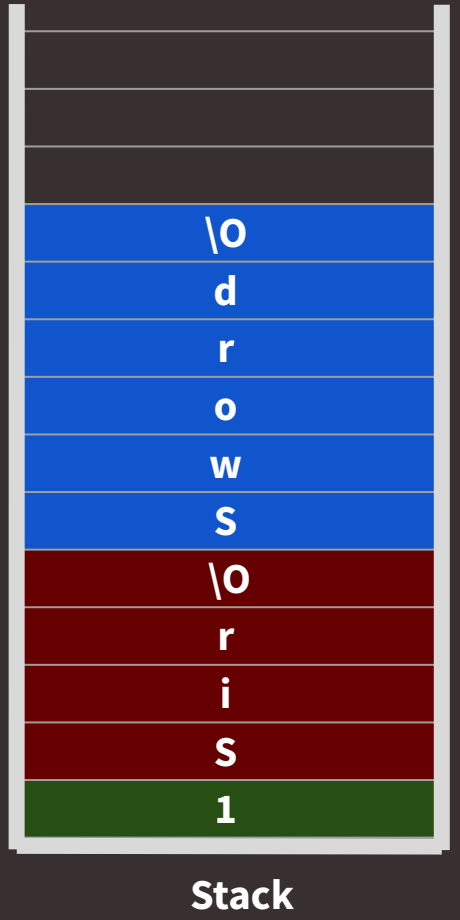


Stack Memory

- Memory is allocated in continuous blocks.
- Type are declared when initialized and can not change.
- Can not grow beyond original allocation on the stack.
- Small static variables can be stored on the stack.
- Memory on stack is freed once stackframe is popped off.

```
int main() {
  int level = 1;
  string name = "Jim";
  string inventory = "Sword";
  // Program continues …

  // warning: character constant too long for type
  name = "Sir Jim";
}
```

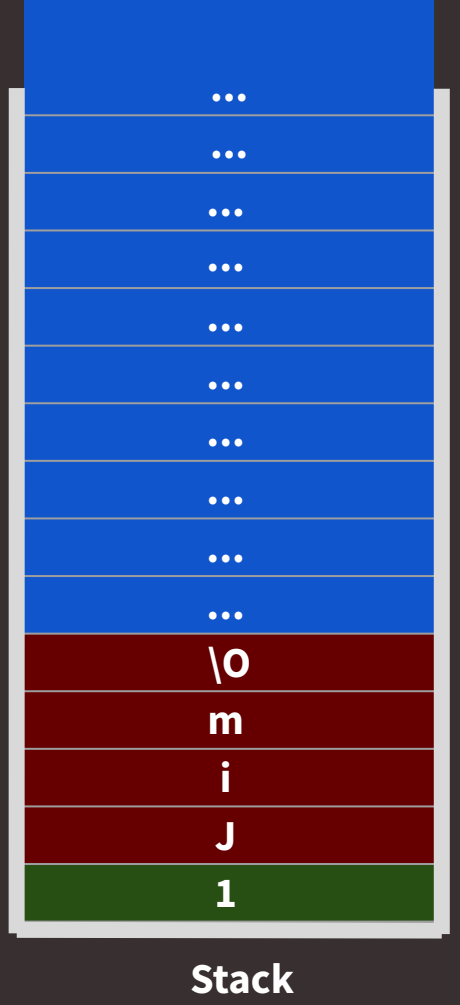


Stack Memory

- Memory is allocated in continuous blocks.
- Type are declared when initialized and can not change.
- Can not grow beyond original allocation on the stack.
- Small static variables can be stored on the stack.
- Memory on stack is freed once stackframe is popped off.

```
int main() {
  int level = 1;
  string name = "Jim";

  // Pretend inventory is a large object.
  // Error: stack overflow.
  char[] inventory = {...}
}
```

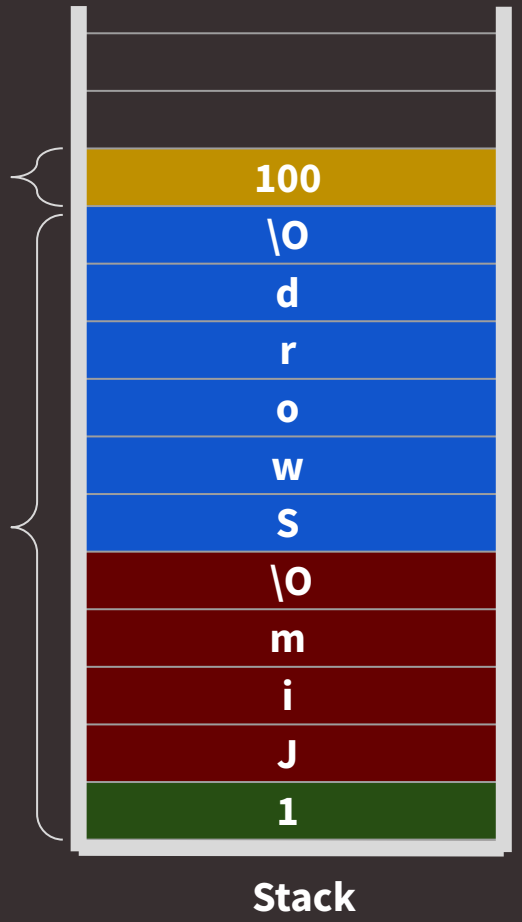


Stack Memory

- Memory is allocated in continuous blocks.
- Type are declared when initialized and can not change.
- Can not grow beyond original allocation on the stack.
- Small static variables can be stored on the stack.
- Memory on stack is freed once stackframe is popped off.

```
int main() {
  int level = 1;
  string name = "Jim";
  string inventory = "Sword";
  if (canAccessCave()) { ... }
}

void canAccessCave() {
  int minPower = 100;
  // some check to see if user can access cave
}
```



Nuff talk.
Let’s code.

Stack Memory: Takeaways

- Size variables appropriately (minimize memory usage)
- Leverage immutability
- Store small, short lived variables on the stack (performant)

C#	int	4 bytes		char	2 bytes
	float	4 bytes		long	8 bytes
	double	8 bytes		short	2 bytes
	bool	1 byte*		decimal	18 bytes

Stack Memory: Takeaways

- Size variables appropriately (minimize memory usage)
- Leverage immutability
- Store small, short lived variables on the stack (performant)

JS	Number	8 bytes		String	2 bytes/char
	BigInt	varies		null	4 bytes
	Symbol	varies		undefined	4 bytes
	Boolean	4 bytes			

Stack Memory: Takeaways

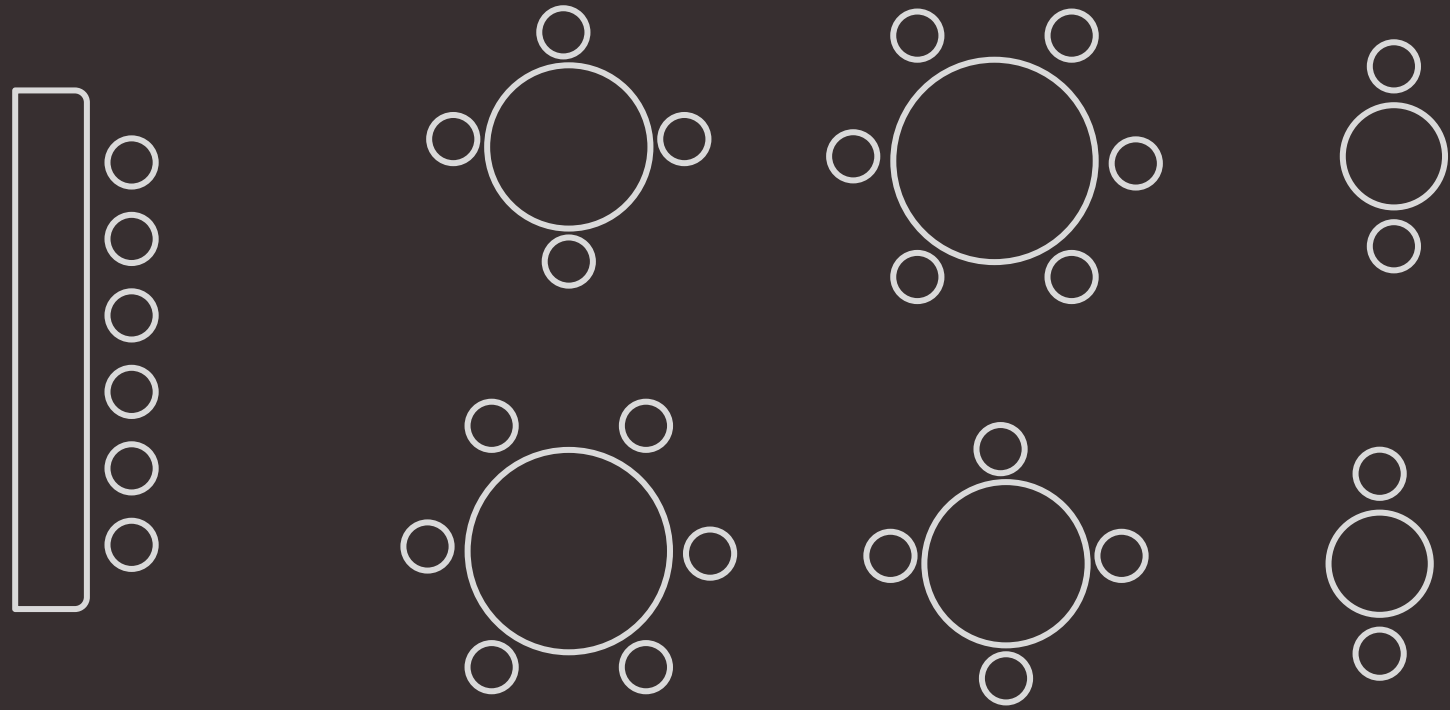
- Size variables appropriately (minimize memory usage)
- Leverage immutability
- Store small, short lived variables on the stack (performant)

```
function longerMethod() {
    data = loadLargeData()
    result = processData(data)
    saveResult(result)
}
```

```
function shortMethod() {
    result = loadAndProcessData()
    saveResult(result)
}

function loadAndProcessData() {
    data = loadLargeData()
    result = processData(data)
    return result
}
```

Heap Memory

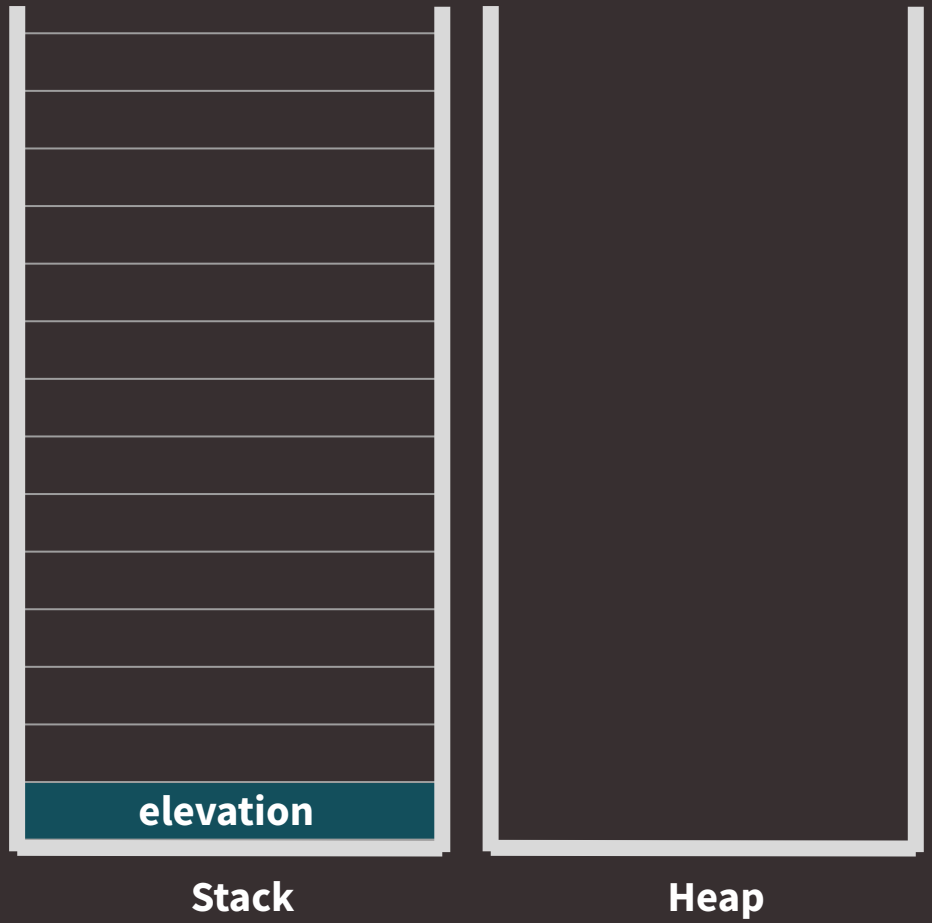


Heap Memory

- Uses a pointer on the stack.
- For large or complex data types.
- Can be resized as needed.
- Must be manually deallocated.

```
int main() {
  int elevation = get_elevation();
  // Program continues...
}

int get_elevation () {
  int *lon = new int (42);
  int *lat = new int (75);
  // do some maths or call apis.
  char data[] = get_geo(lon,lat)
  return data[0];
}
```

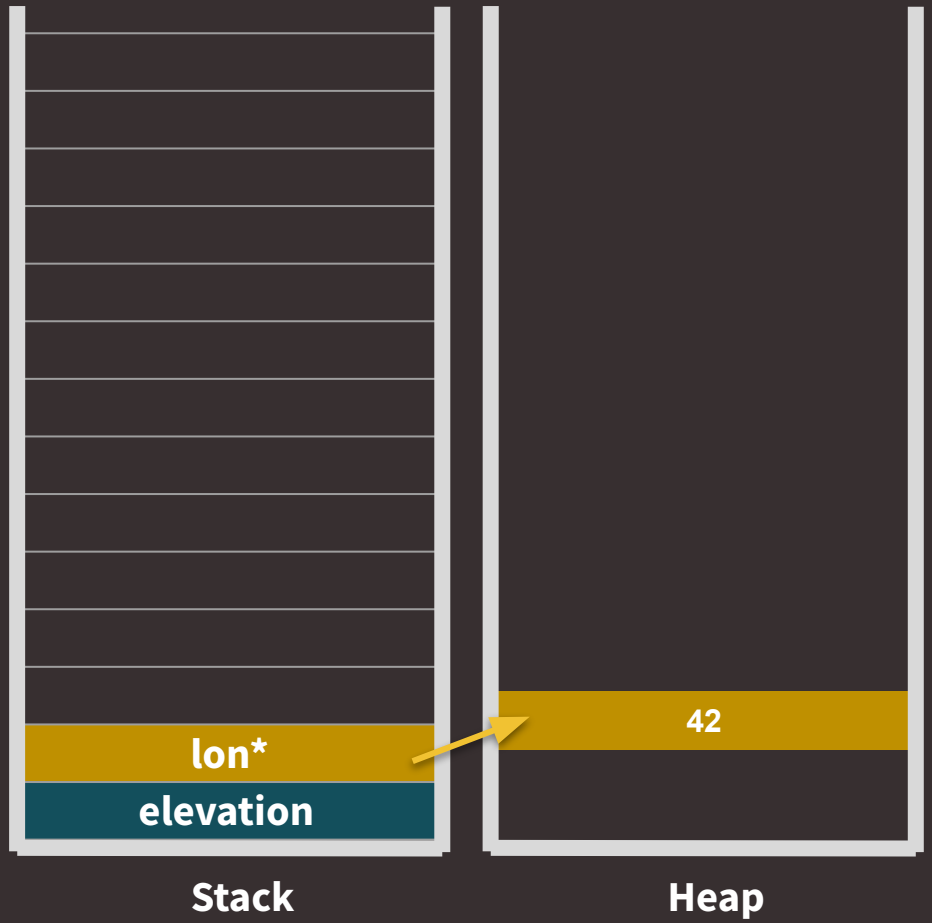


Heap Memory

- Uses a pointer on the stack.
- For large or complex data types.
- Can be resized as needed.
- Must be manually deallocated.

```
int main() {
  int elevation = get_elevation();
  // Program continues...
}

int get_elevation () {
  int *lon = new int (42);
  int *lat = new int (75);
  // do some maths or call apis.
  char data[] = get_geo(lon,lat)
  return data[0];
}
```

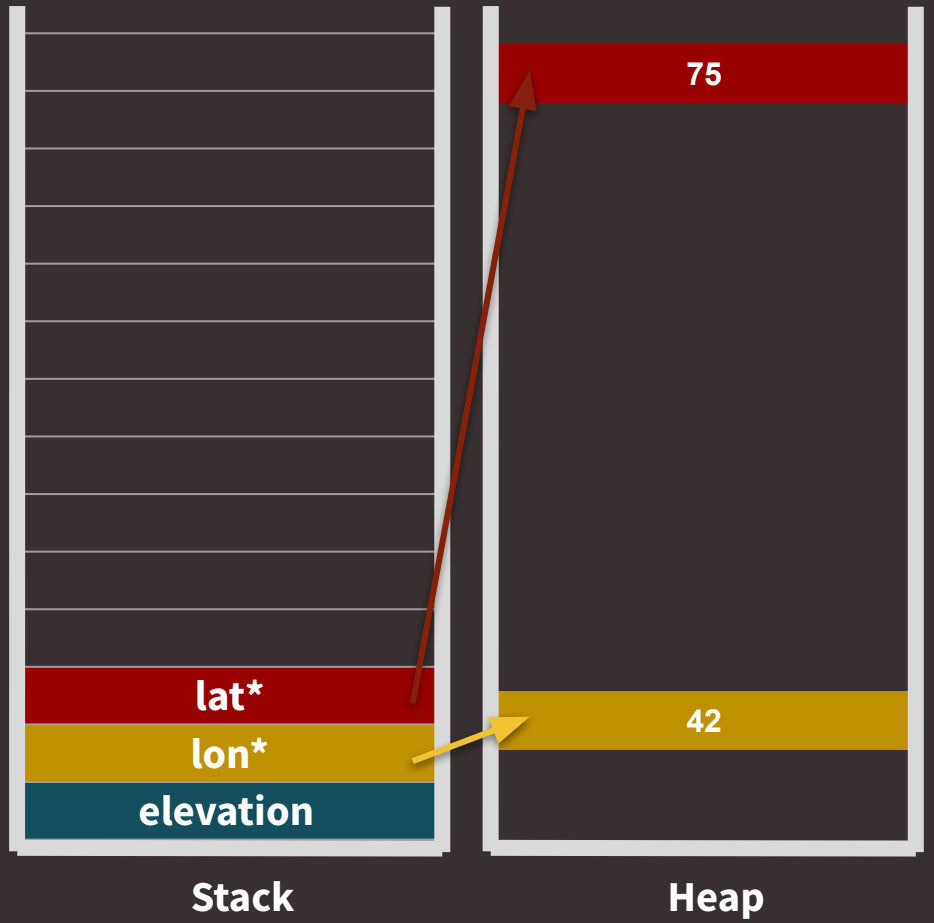


Heap Memory

- Uses a pointer on the stack.
- For large or complex data types.
- Can be resized as needed.
- Must be manually deallocated.

```
int main() {
  int elevation = get_elevation();
  // Program continues...
}

int get_elevation () {
  int *lon = new int (42);
  int *lat = new int (75);
  // do some maths or call apis.
  char data[] = get_geo(lon,lat)
  return data[0];
}
```

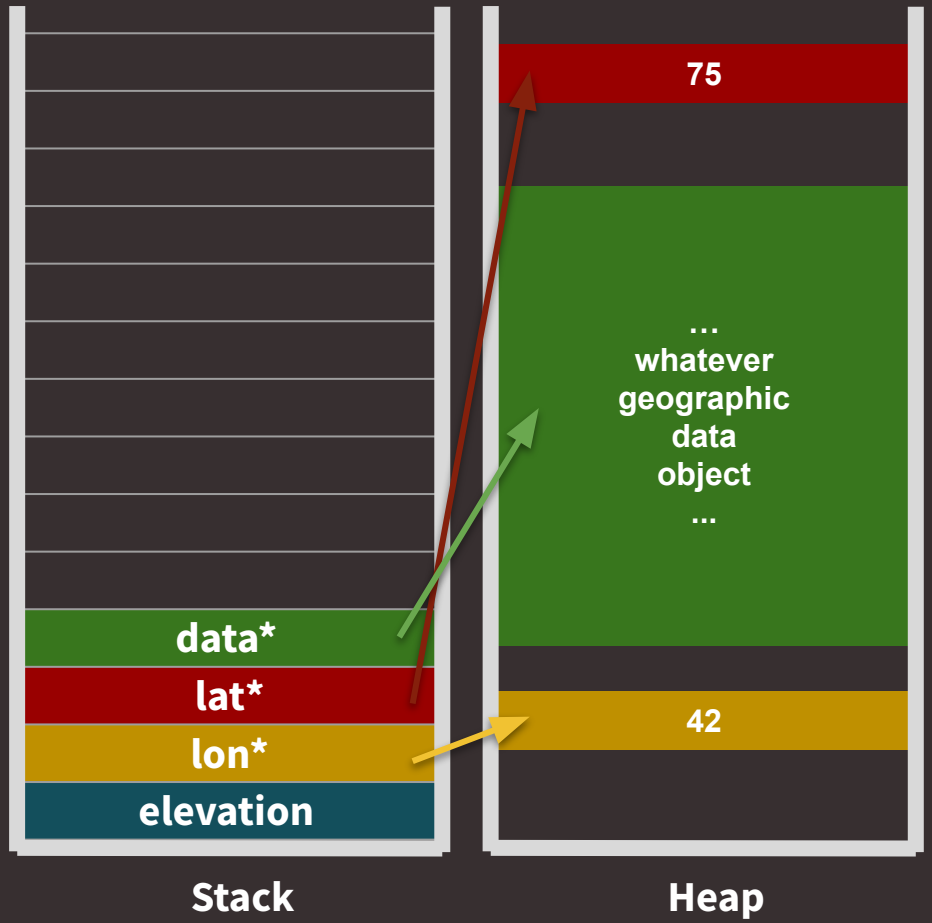


Heap Memory

- Uses a pointer on the stack.
- For large or complex data types.
- Can be resized as needed.
- Must be manually deallocated.

```
int main() {
  int elevation = get_elevation();
  // Program continues...
}

int get_elevation () {
  int *lon = new int (42);
  int *lat = new int (75);
  // do some maths or call apis.
  char data[] = get_geo(lon,lat)
  return data[0];
}
```

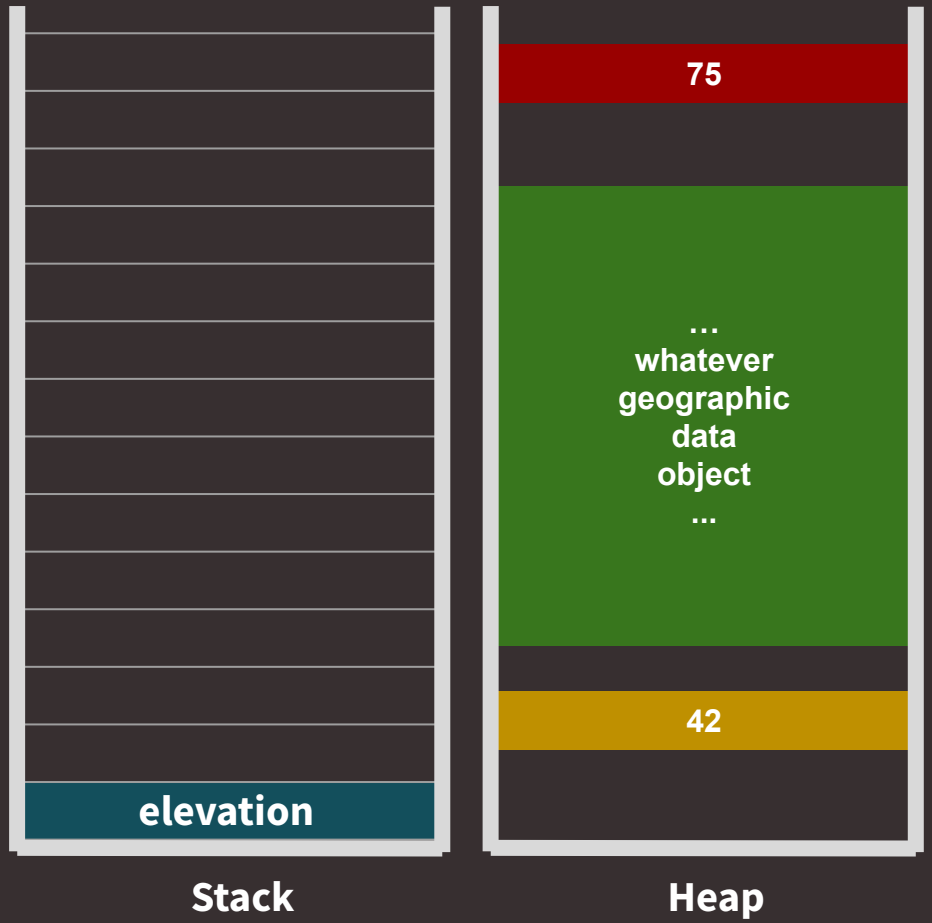


Heap Memory

- Uses a pointer on the stack.
- For large or complex data types.
- Can be resized as needed.
- Must be manually deallocated. 😬

```
int main() {
  int elevation = get_elevation();
  // Program continues...
}

int get_elevation () {
  int *lon = new int (42);
  int *lat = new int (75);
  // do some maths or call apis.
  char data[] = get_geo(lon,lat)
  return data[0];
}
```



Why care in dynamic typing?

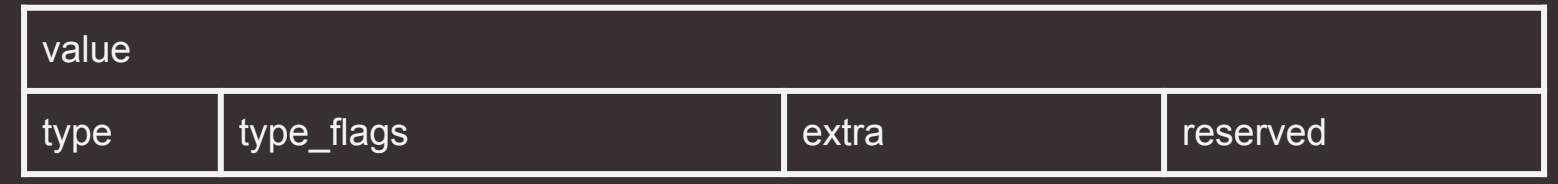
- Many interpreters are written in C or derived originally from C.
- All of the aforementioned rules apply (even if it doesn't feel like it.)

```
// In JS, arrays are dynamic and can be resized after declared
let ducks = ["Huey", "Dewey", "Louie"];
ducks.push("Daffy");
ducks = [...ducks, "rubber"];

// And in JavaScript, variables can change types as well
let count = false;
count = 3;
count = null;
count = ["one", "two", "three"];
```

PHP ZVAL

To allow dynamic variables PHP values are represented as two 64-bit words. The first word keeps the value and the second stores metadata.

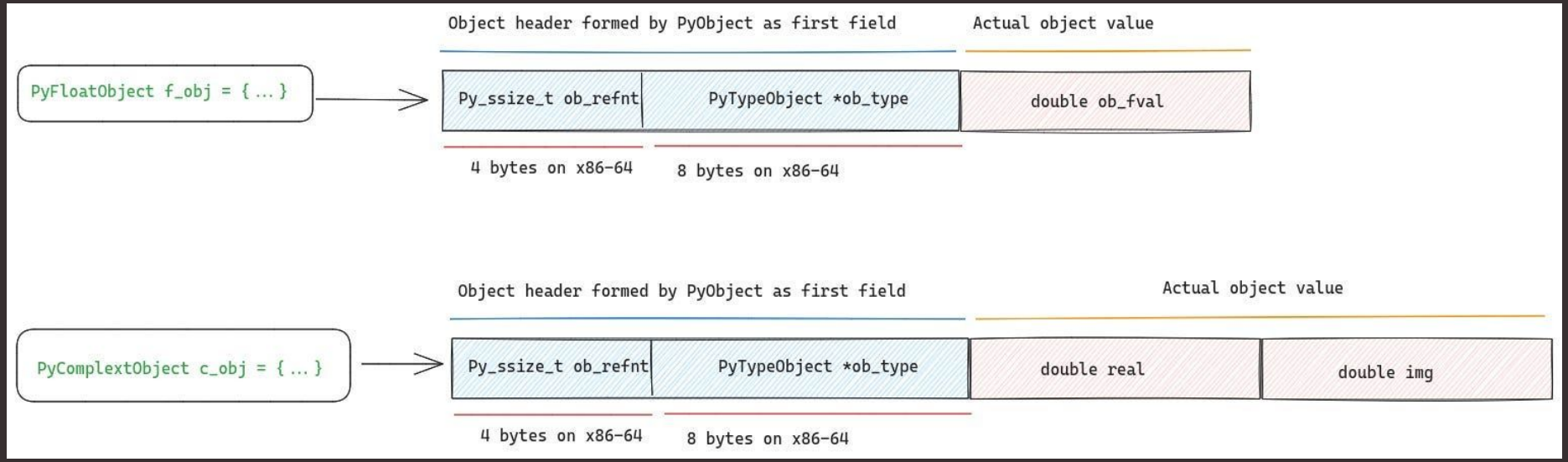


Types (incomplete)	Storage location
IS_NULL	none
IS_TRUE or IS_FALSE	none
IS_LONG	zend_long lval
IS_DOUBLE	double dval
IS_STRING	zend_string *str
IS_ARRAY	zend_array *arr
IS_OBJECT	zend_object *obj

IS_TYPE_REFCOUNTED

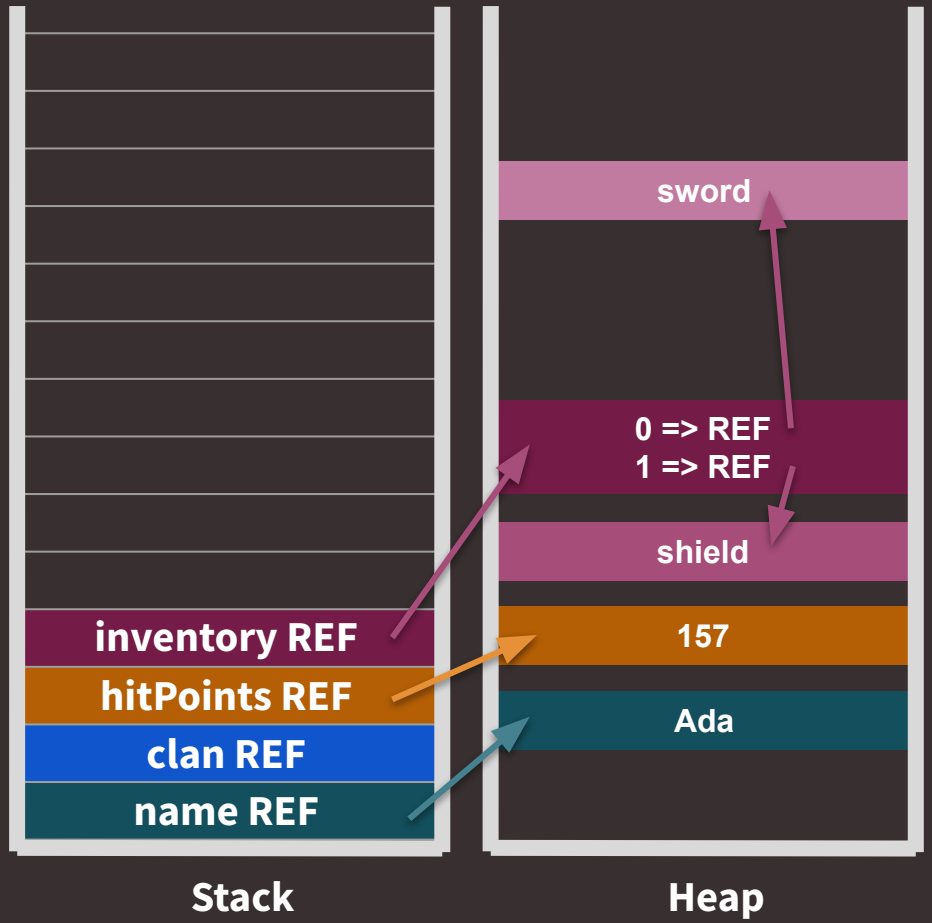
Python PyObject Structure

Python has a similar structure called PyObject for storing values and reference counting.



Reference

```
$name = "Ada";
$clan = NULL;
$hitPoints = 157;
$inventory = ["sword", "shield"];
```

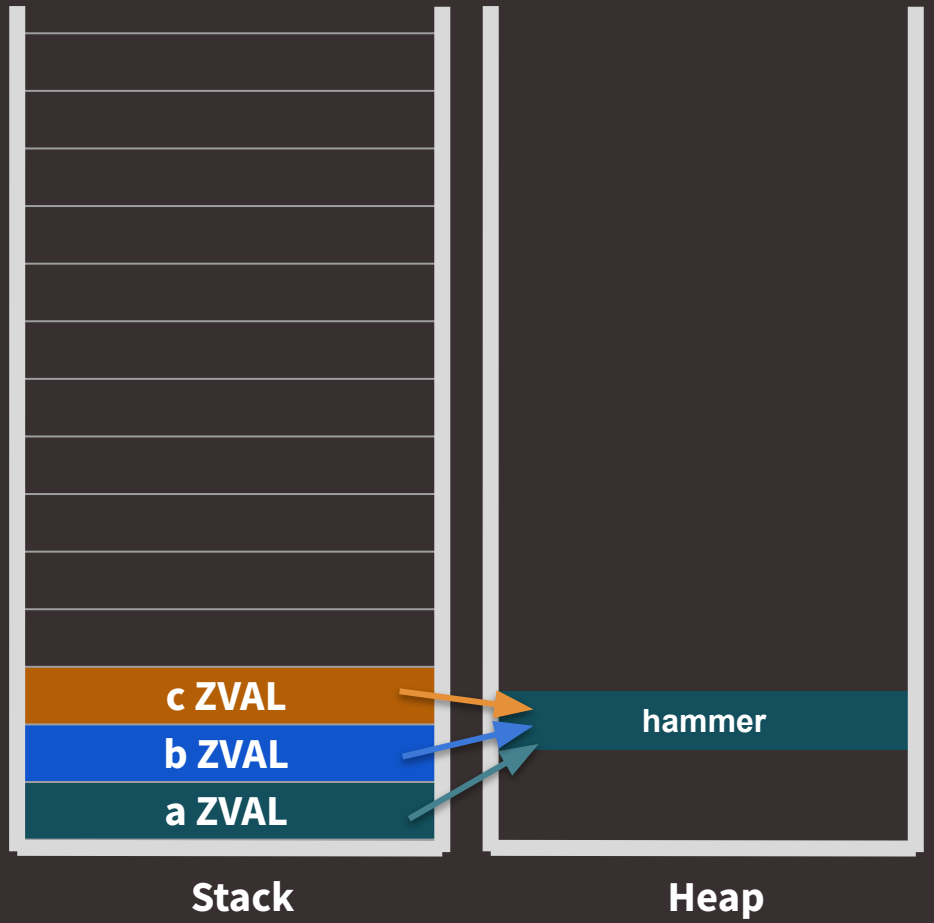


Copy-on-Write

```
// Simple assignment
$a = “hammer”;
$b = $a;
$c = $b;
```

```
var_dump($a, $b, $c);
string(6) "hammer"
string(6) "hammer"
string(6) "hammer"
```

```
xdebug_debug_zval('a', 'b', 'c');
a: (refcount=3, is_ref=0)='hammer'
b: (refcount=3, is_ref=0)='hammer'
c: (refcount=3, is_ref=0)='hammer'
```

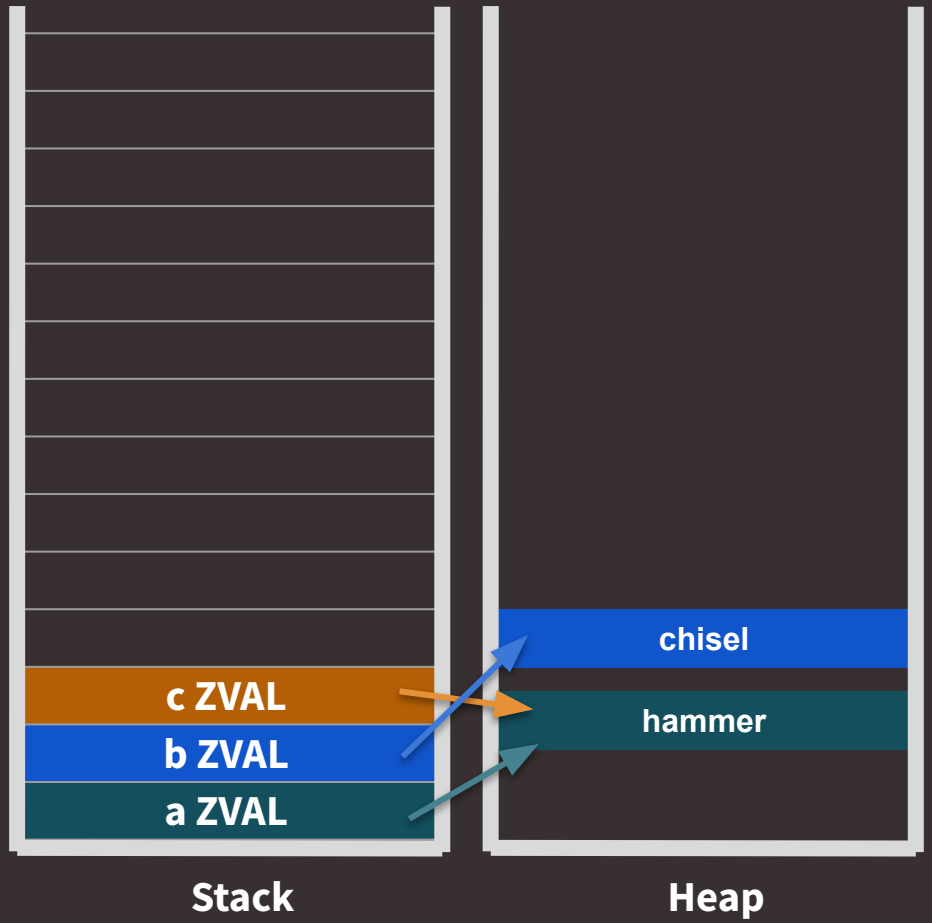


Copy-on-Write

```
// Simple assignment
$a = “hammer”;
$b = $a;
$c = $b;
$b = “chisel”;

var_dump($a, $b, $c);
string(6) "hammer"
string(6) "chisel"
string(6) "hammer"

xdebug_debug_zval('a', 'b', 'c');
a: (refcount=2, is_ref=0)='hammer'
b: (refcount=1, is_ref=0)='chisel'
c: (refcount=2, is_ref=0)='hammer'
```

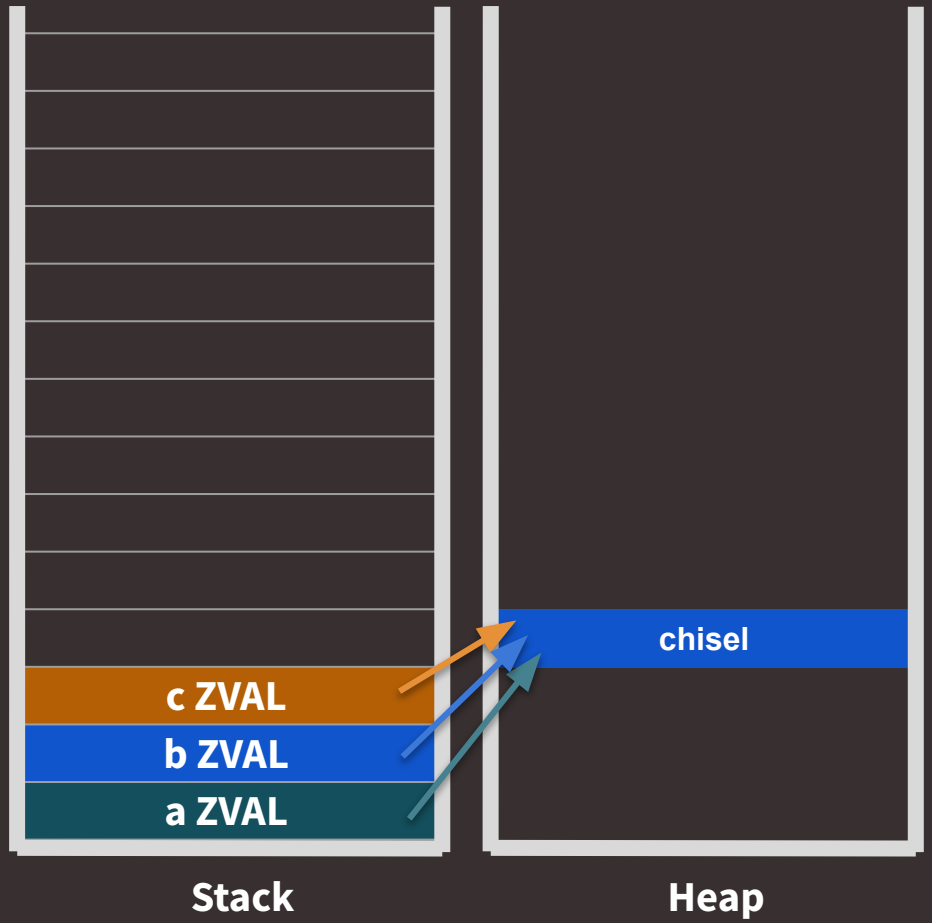


References

```
// Assign by reference
$a = “hammer”;
$b = &$a;
$c = &$b;
$b = “chisel”;

var_dump($a, $b, $c);
string(6) "chisel"
string(6) "chisel"
string(6) "chisel"

xdebug_debug_zval('a', 'b', 'c');
a: (refcount=3, is_ref=1)='chisel'
b: (refcount=3, is_ref=1)='chisel'
c: (refcount=3, is_ref=1)='chisel'
```



References

```
// In JS objects and arrays are refs
let pets = ['dog', 'cat']
let animals = pets
animals.push('bird')

pets: (refcount=2, is_ref=1)
animals: (refcount=2, is_ref=1)
```

Nuff talk.
Let’s code.

Garbage Collection

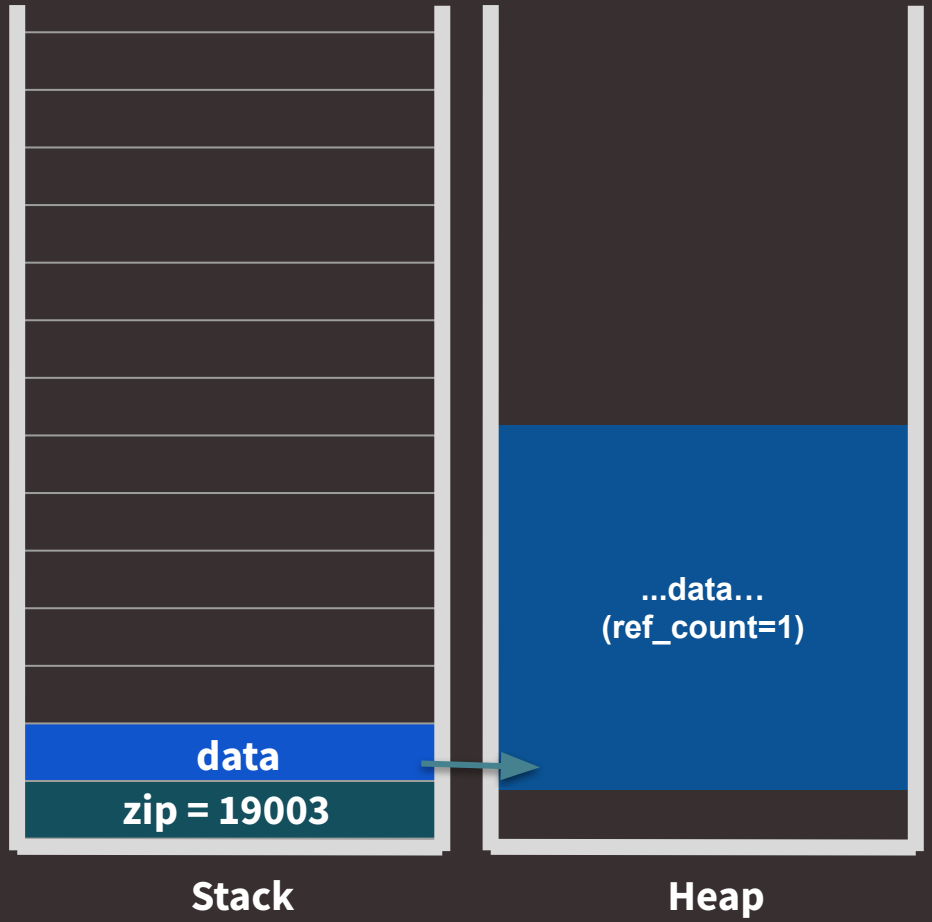
- A feature that automatically manages memory by freeing up space that's no longer in use.

```
function main() {
  const zip = 19003
  const temp = getTemp(zip)
}
function getTemp(zip){
  const data = getWeather(zip)
  return data.temp
}
```

Garbage Collection

- A feature that automatically manages memory by freeing up space that's no longer in use.

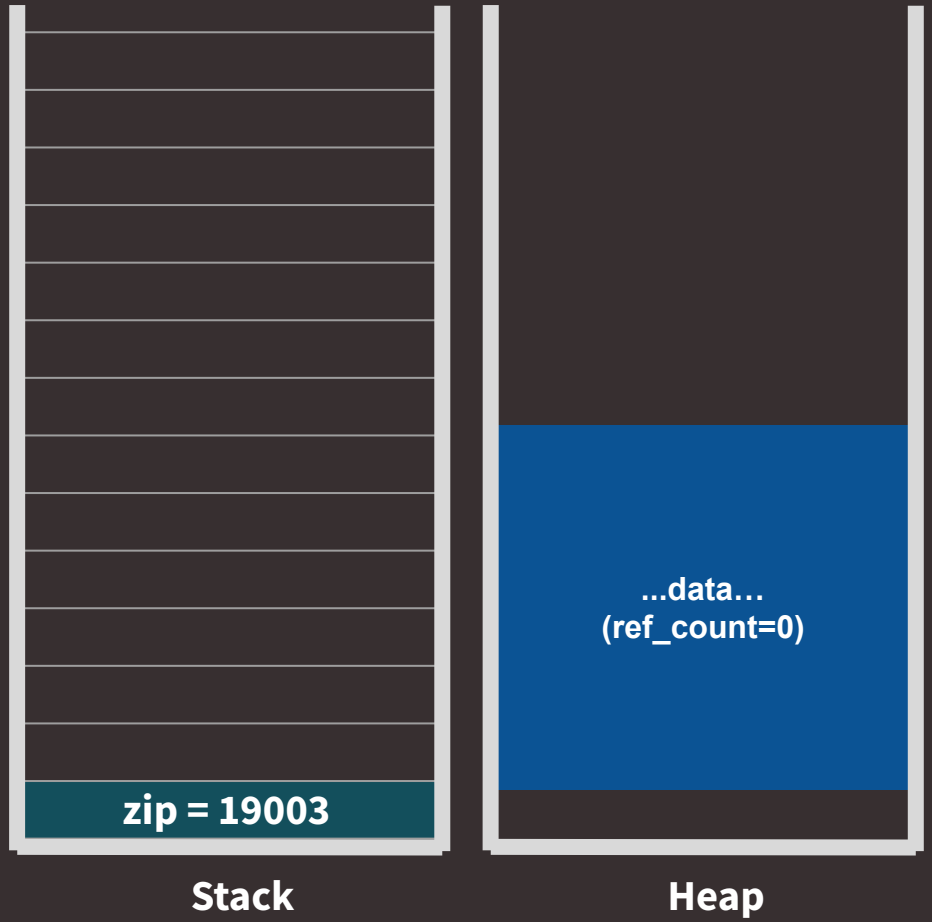
```
function main() {
  const zip = 19003
  const temp = getTemp(zip)
}
function getTemp(zip){
  const data = getWeather(zip)
  return data.temp
}
```



Garbage Collection

- A feature that automatically manages memory by freeing up space that's no longer in use.

```
function main() {
  const zip = 19003
  const temp = getTemp(zip)
}
function getTemp(zip){
  const data = getWeather(zip)
  return data.temp
}
```



Garbage Collection

- References can also be manually removed
- Memory will be freed via garbage collection is it is no longer needed.

```
// JavaScript
unset(data)
data = null
data = undefined

# Python
del results
del results['apples']
results = None
```

```
// C++
delete record

// C#
person = null;
// or using Dispose() on
// unmanaged resource
```

Garbage Collection: Leaks

- There are many different example of memory leaks (when Garbage Collection can not clean up memory). You will need to test you code to find them.
- Example: Circular reference

```
function Person(name) {
    this.name = name;
    this.friend = null;
}

function meetPeople() {
  let person1 = new Person("Alice");
  let person2 = new Person("Bob");
  // Creating a circular reference
  person1.friend = person2;  // Alice's friend is Bob
  person2.friend = person1;  // Bob's friend is Alice
}
```

Garbage Collection: Types

- Many different types of garbage collection exist and some languages employ multiple strategies as a way of combating memory leaks.

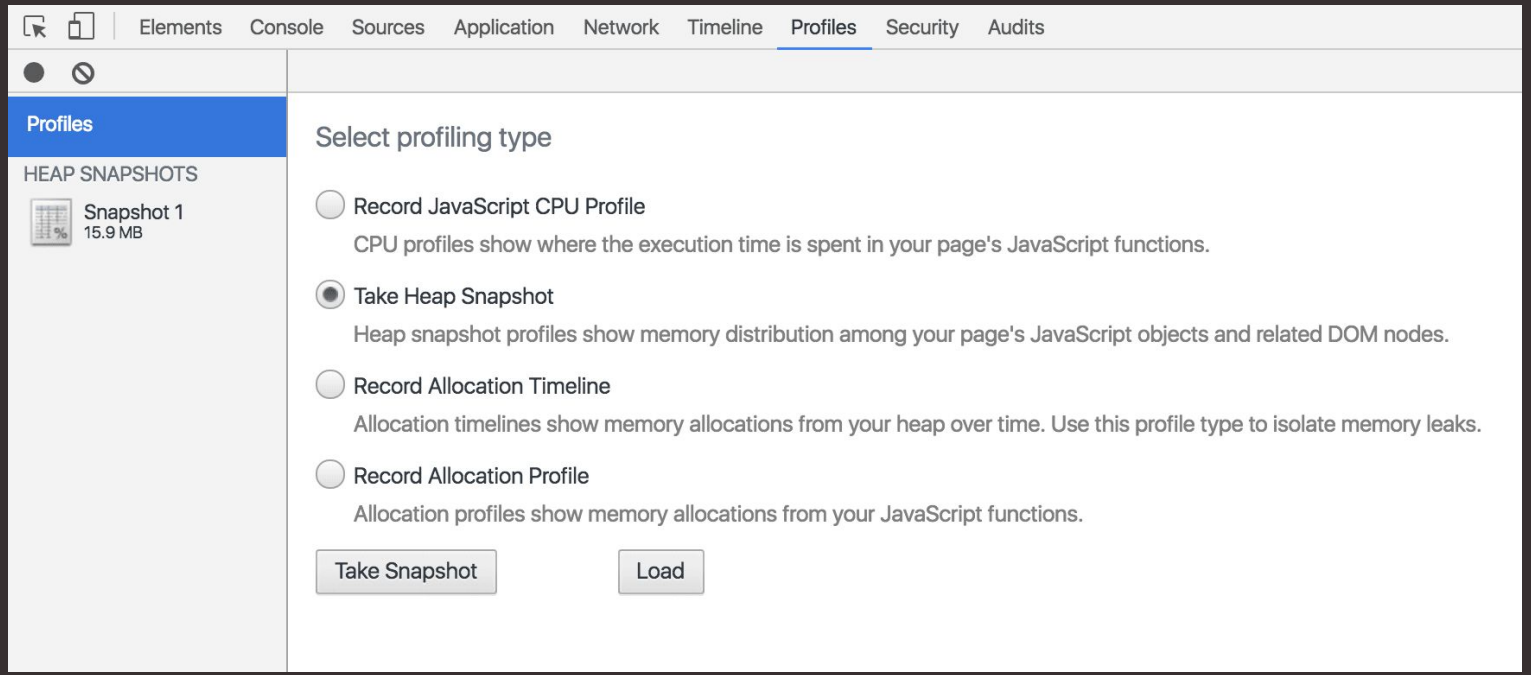
1. Reference Counting Garbage Collection
   Python, PHP, Objective-C, Swift
2. Mark-and-Sweep Garbage Collection
   JavaScript (V8), Python, Ruby, PHP
3. Generational Garbage Collection
   Java (HotSpot JVM), C# (.NET), Python

Garbage Collection: Takeaways

- Minimize Allocations
- Avoid Unnecessary Object References
- Use small functions to free memory
- Unset/delete/free variables no longer needed
- Avoid globals
- Avoid running garbage collection manually
- Use Pools for reusing objects

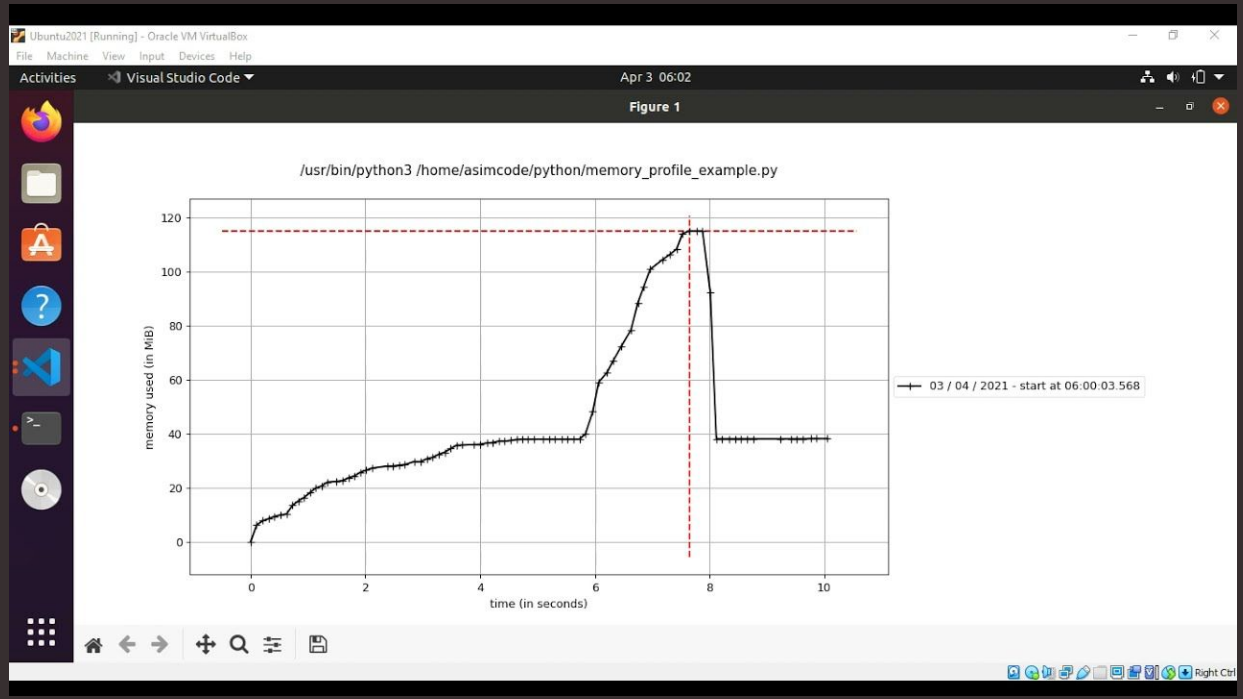
Profiling Tools

JavaScript: Chrome DevTools (Built-in Browser Tool)



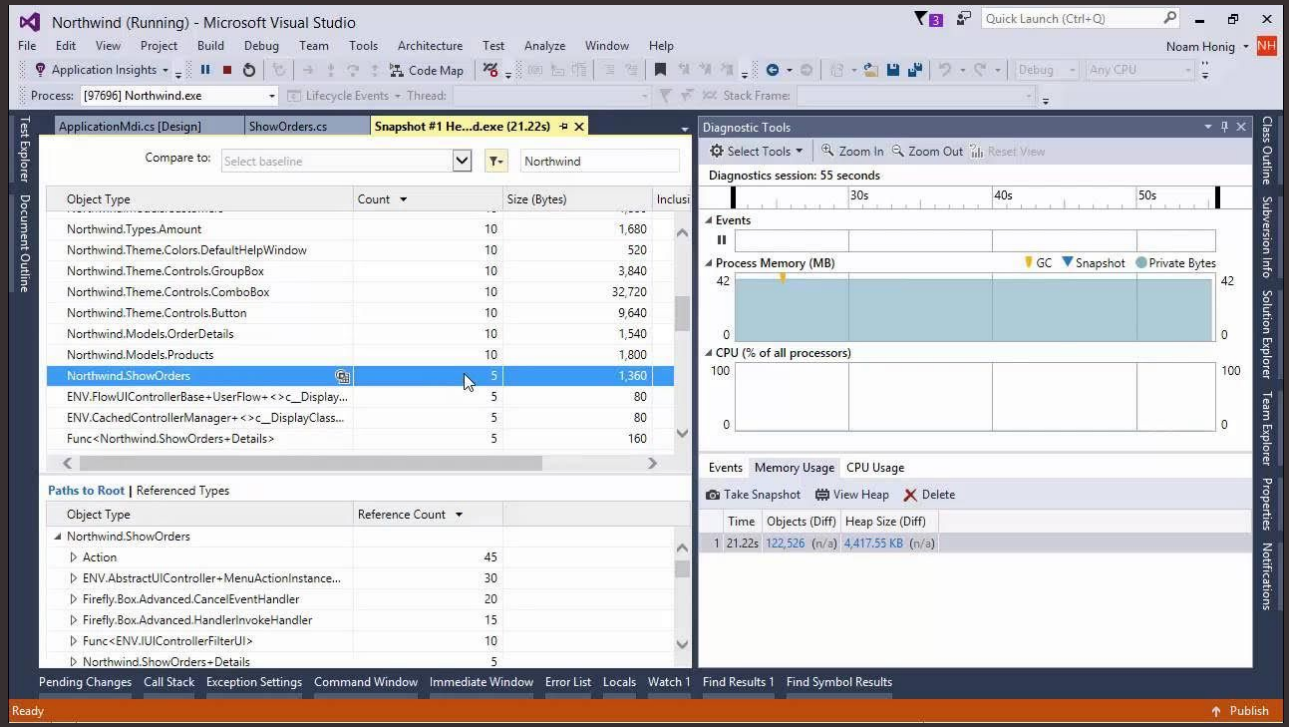
Profiling Tools

Python: memory_profiler, tracemalloc, guppy3 (Heapy)



Profiling Tools

C#: Visual Studio Diagnostic Tools, dotMemory (JetBrains), PerfView



Thank You

I hope you leave this session with a better understanding of how memory works and feel empowered to monitor and reduce your memory footprint in producing more performant and healthy code.

Jim.Vomero@FourKitchens.com

 **@nJim**

Four Kitchens