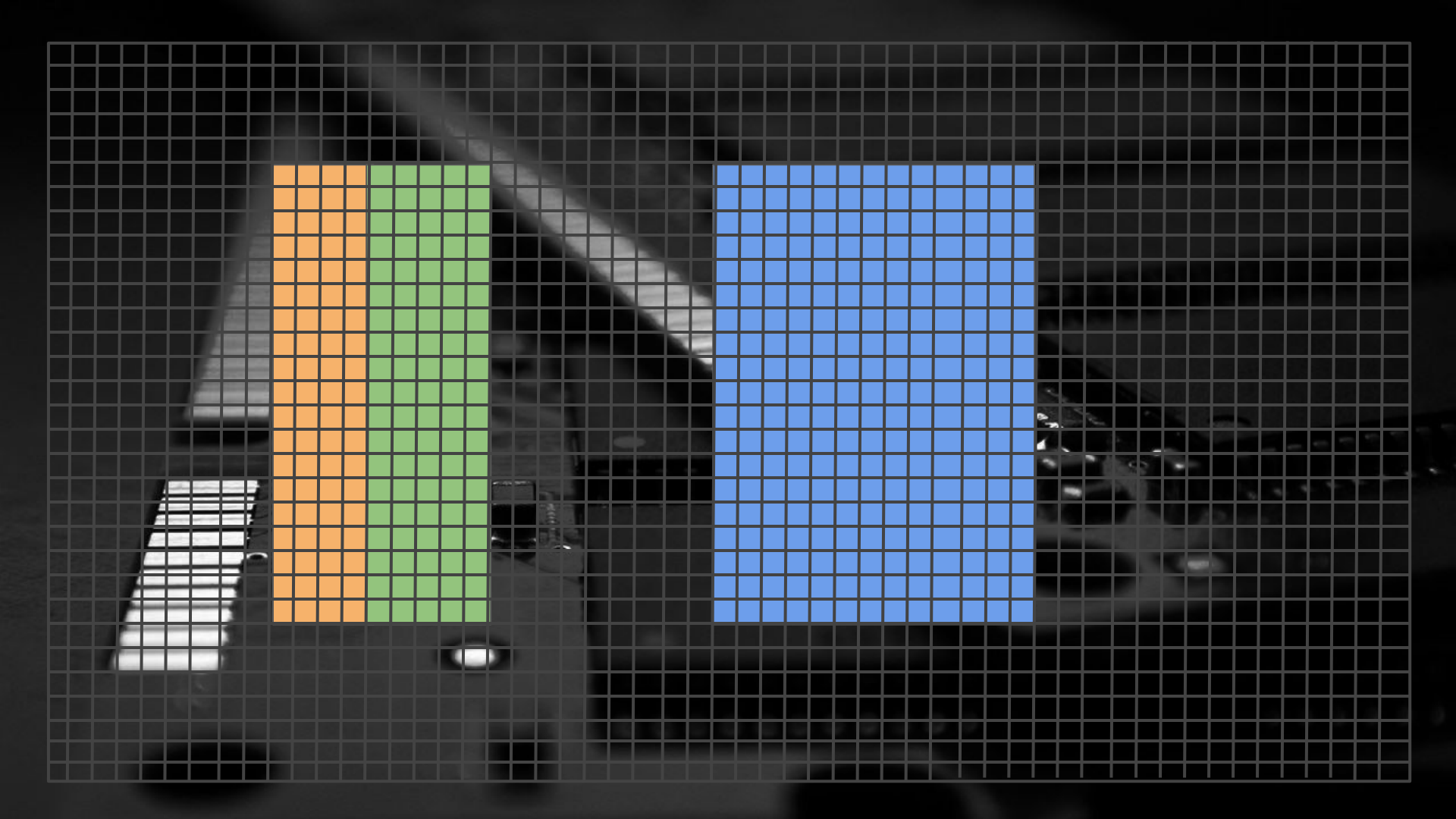
# Practical CS

## Memory Allocation and Garbage Collection in PHP

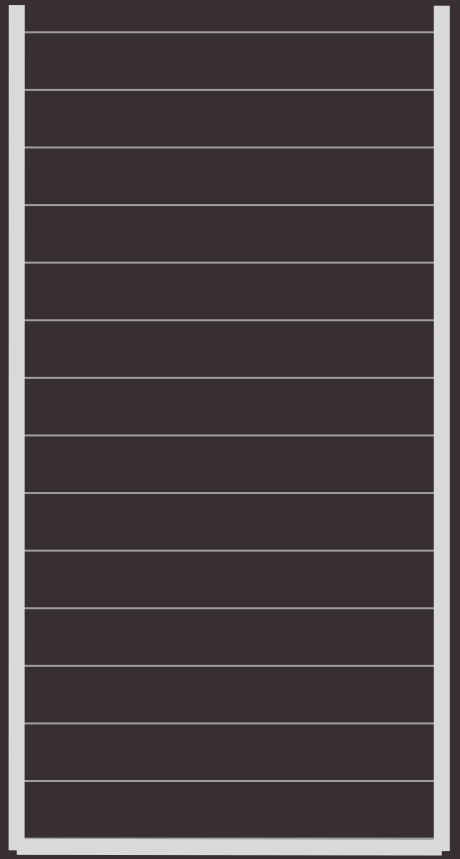
Drupal Camp Asheville 2021


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.

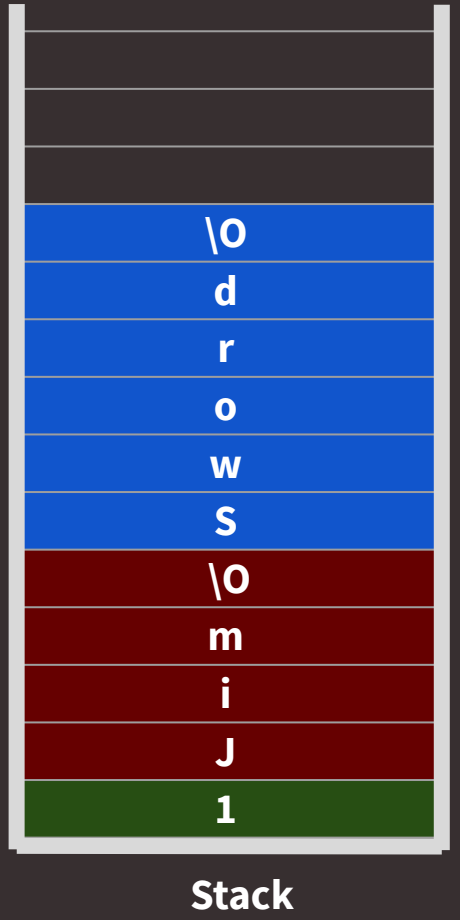


Stack

# 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 …
}
```

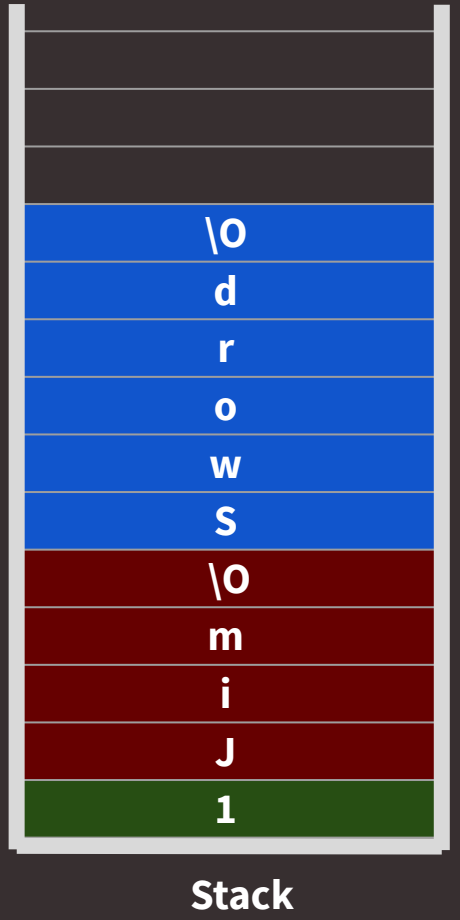


# Stack Memory

- 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";
}
```

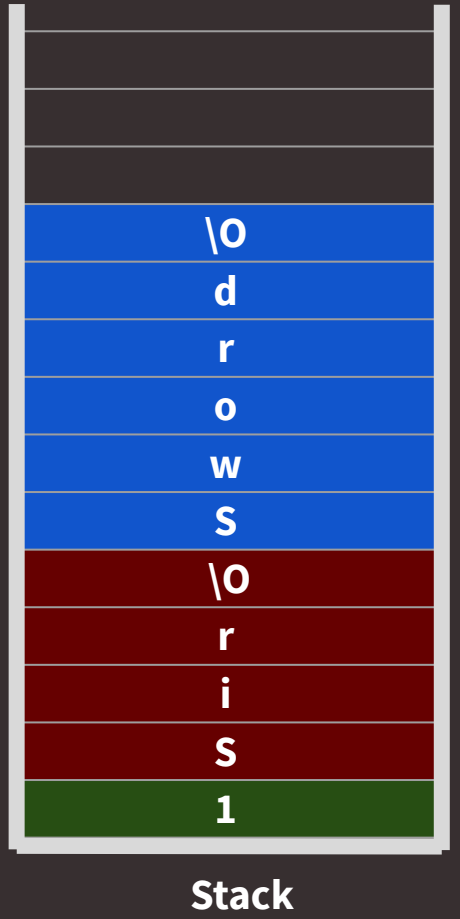


# Stack Memory

- 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";
}
```

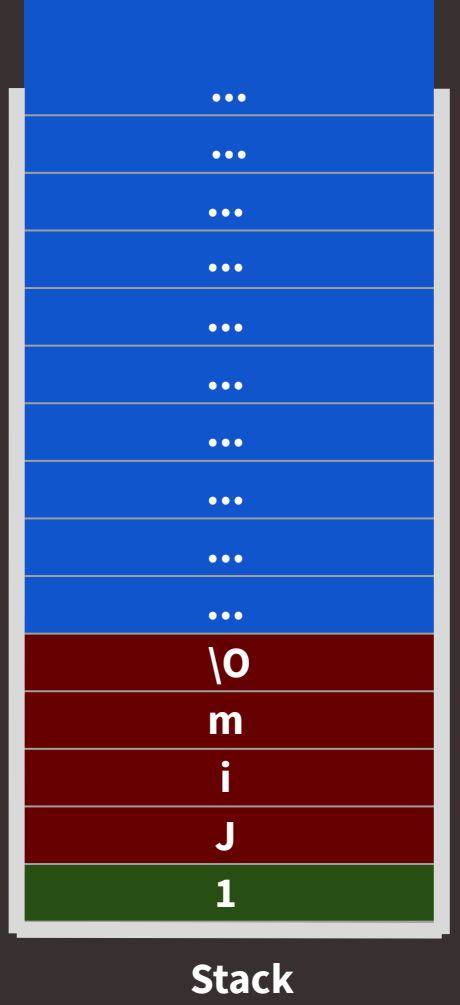


# Stack Memory

- 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 = {...}
}
```

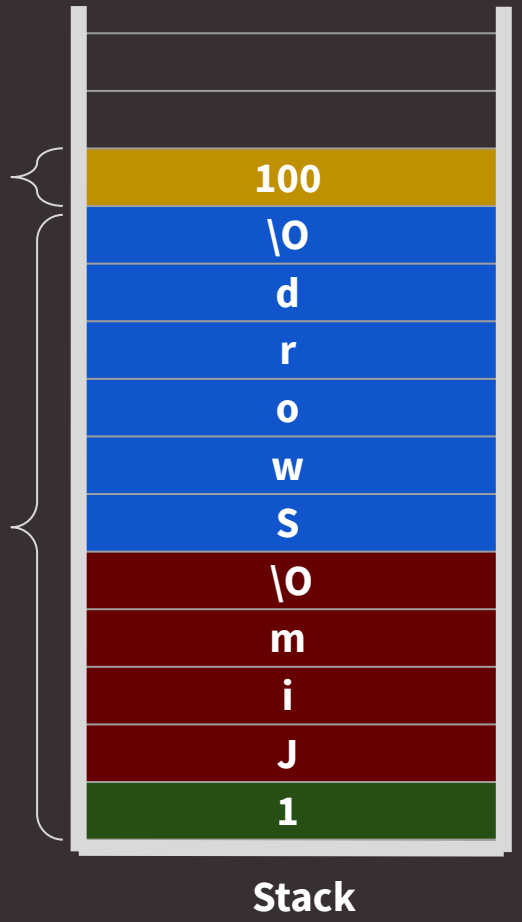


# Stack Memory

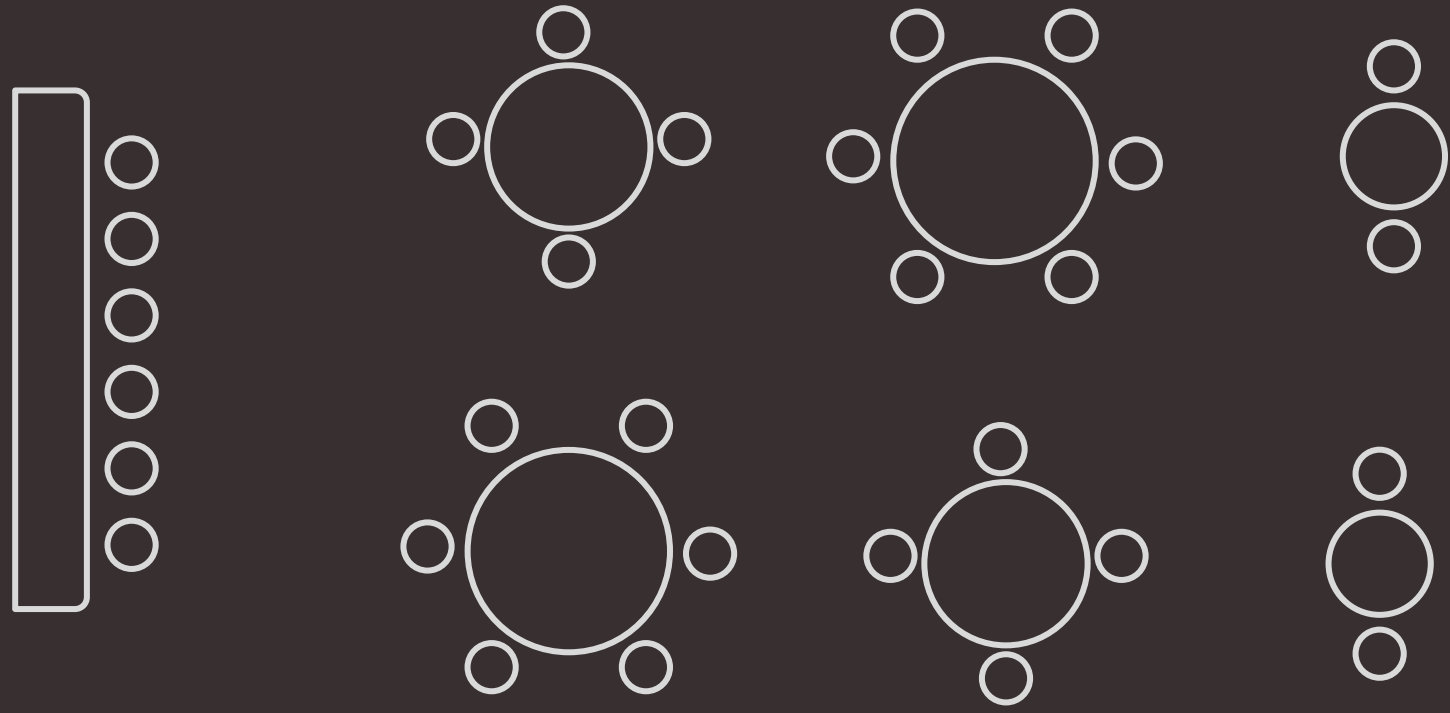
- Memory is allocated in continuous blocks.
- Type are declared when initialized and can not change.
- Can not grow beyond original allocation on the stack.
- Small static variables can be stored on the stack.
- Memory on stack is freed once stackframe is popped off.

```
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
}
```



# Heap Memory

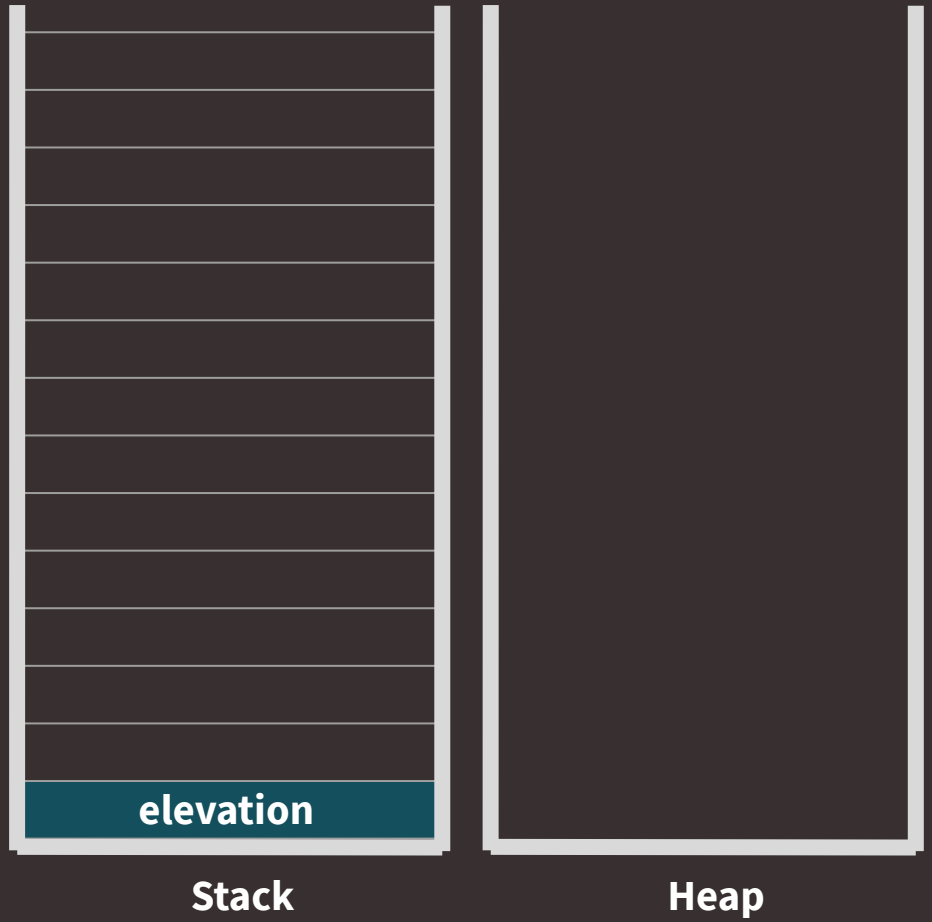


# Heap Memory

- 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];
}
```

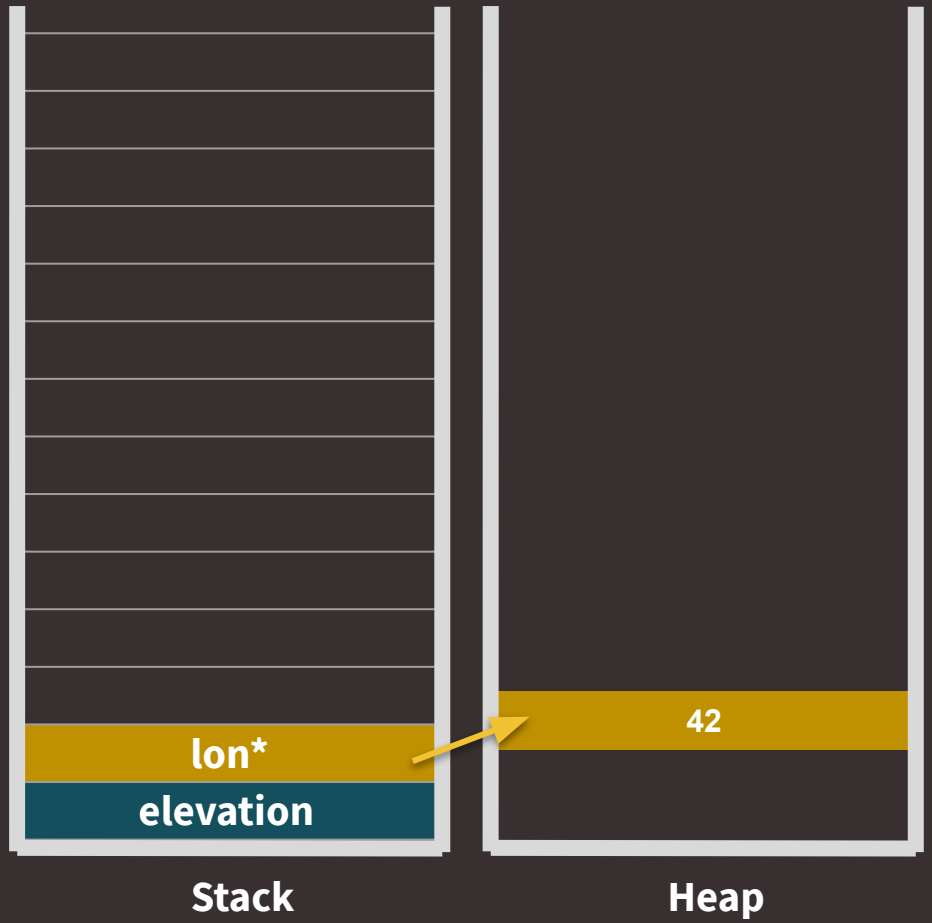


# Heap Memory

- 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];
}
```

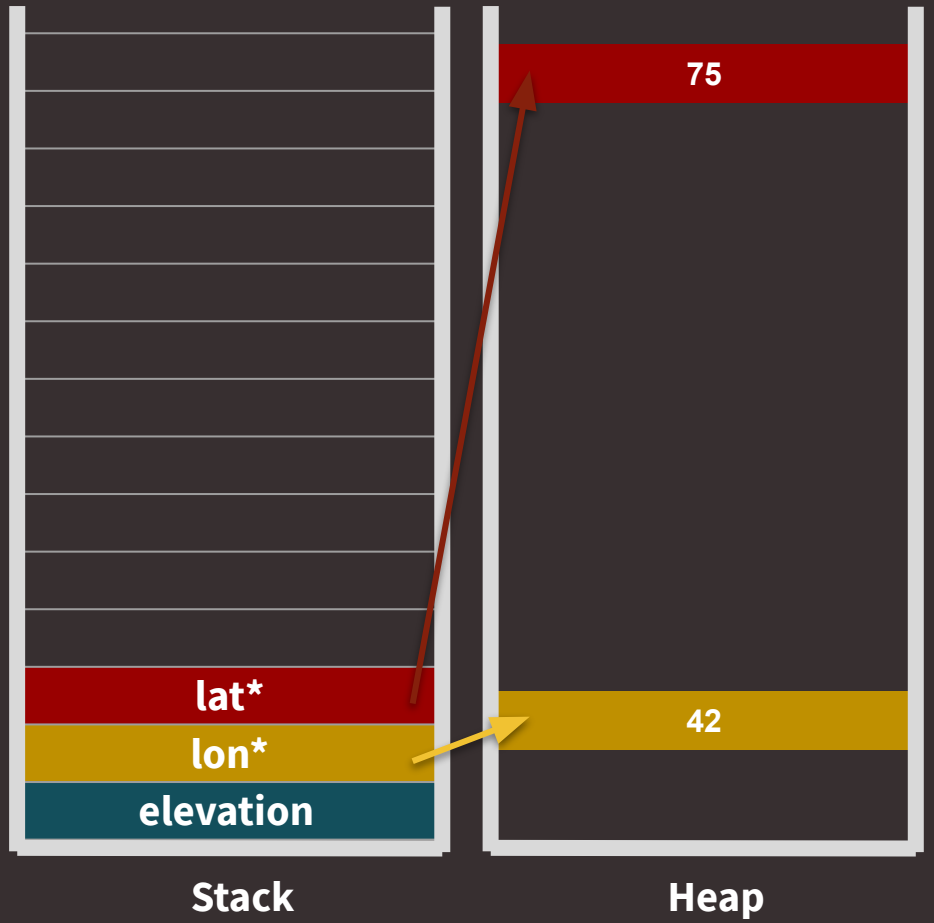


# Heap Memory

- 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];
}
```

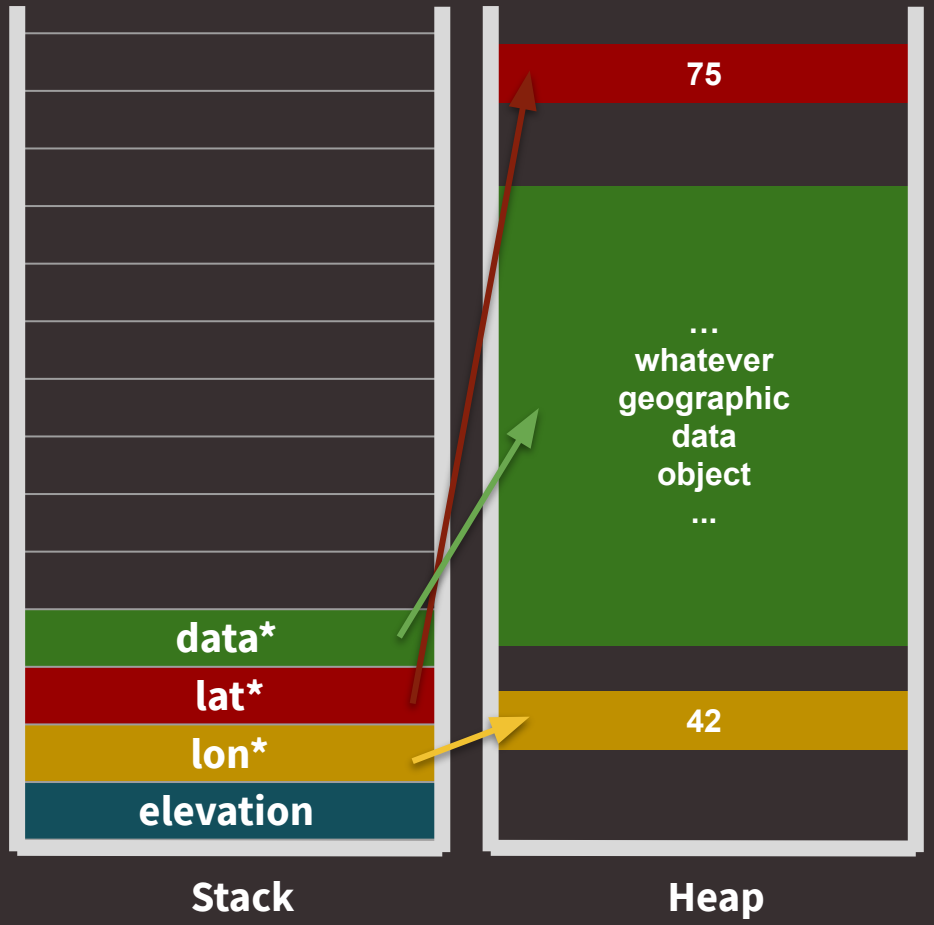


# Heap Memory

- 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];
}
```

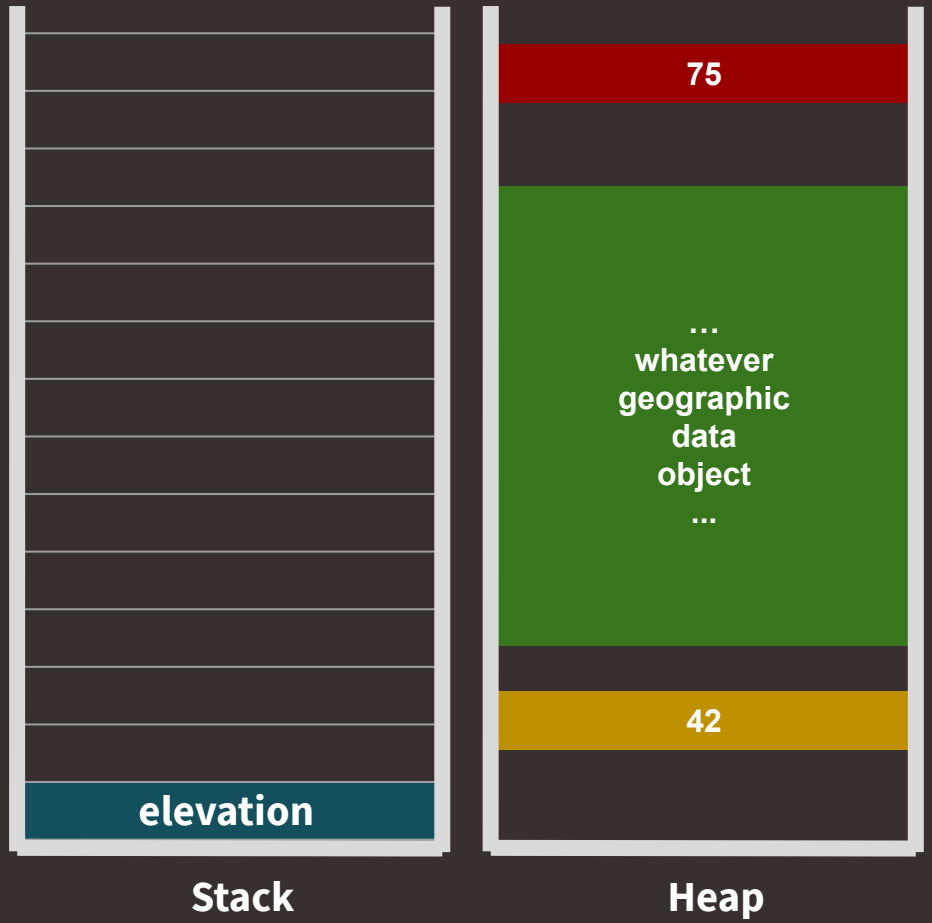


# Heap Memory

- 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 do we care in PHP land?

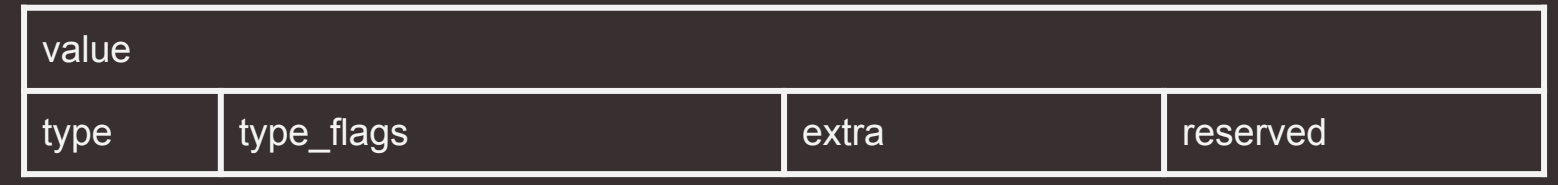
- The PHP interpreter is written in C
- All of the aforementioned rules apply (even if it doesn't feel like it.)

```
// But in PHP elements can resize after they are declared
$ducks = [“Huey”, “Dewey”, “Louie”];
$ducks[] = “Daffy”;
array_push($ducks, ”rubber”);

// And values in PHP can easily change types
$count = FALSE;
$count = 3;
$count = NULL;
$count = [“one”, “two”, “three”];
```

# 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.

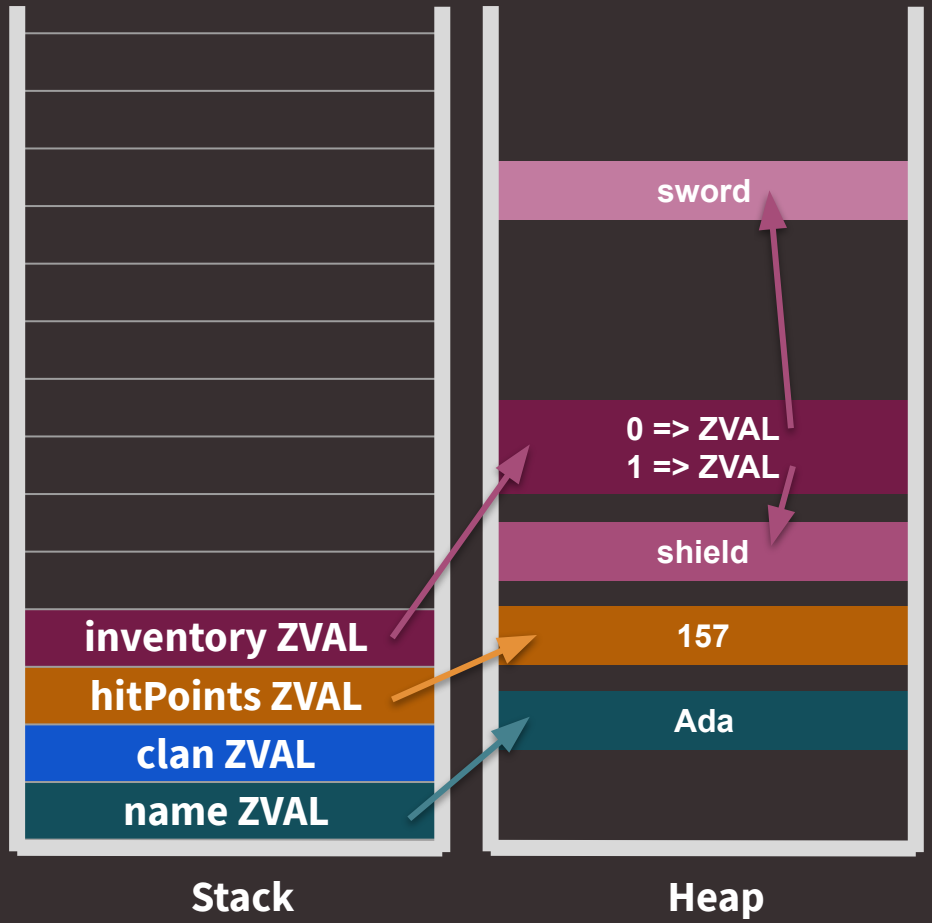



| Types (incomplete) | Storage location |
|---|---|
| IS_NULL | none |
| IS_TRUE or IS_FALSE | none |
| IS_LONG | zend_long lval |
| IS_DOUBLE | double dval |
| IS_STRING | zend_string *str |
| IS_ARRAY | zend_array *arr |
| IS_OBJECT | zend_object *obj |

IS_TYPE_REFCOUNTED

# ZVAL

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

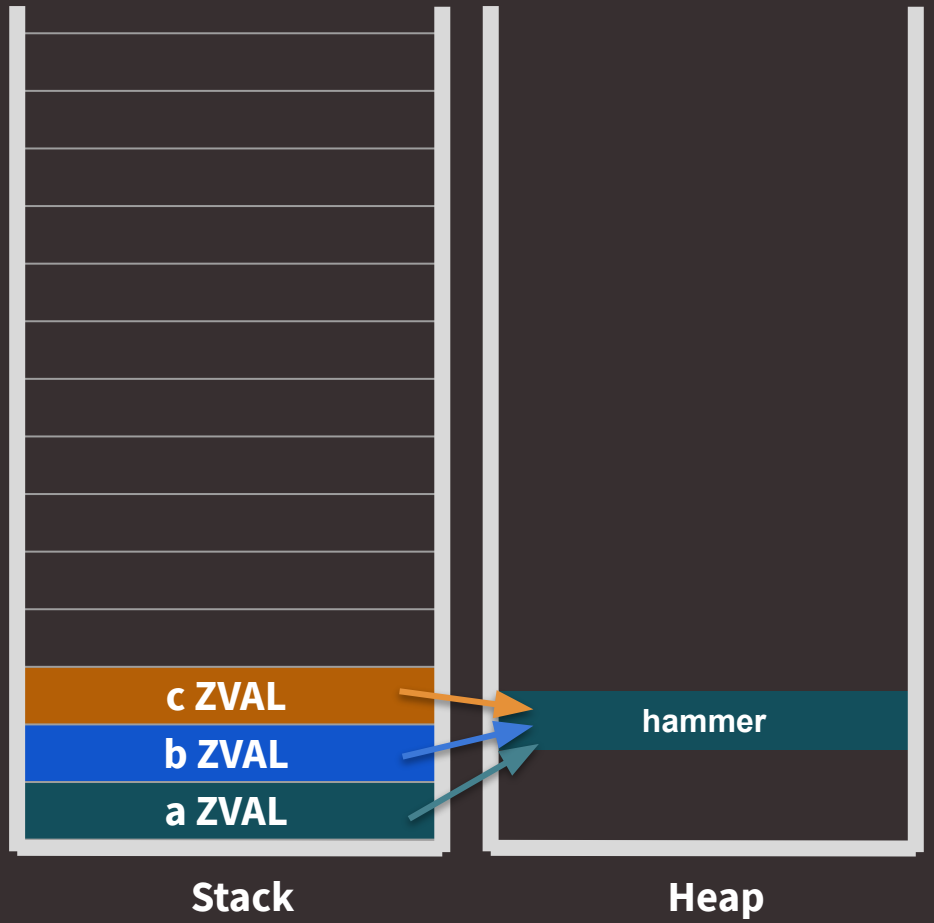


# References

```
// 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'
```

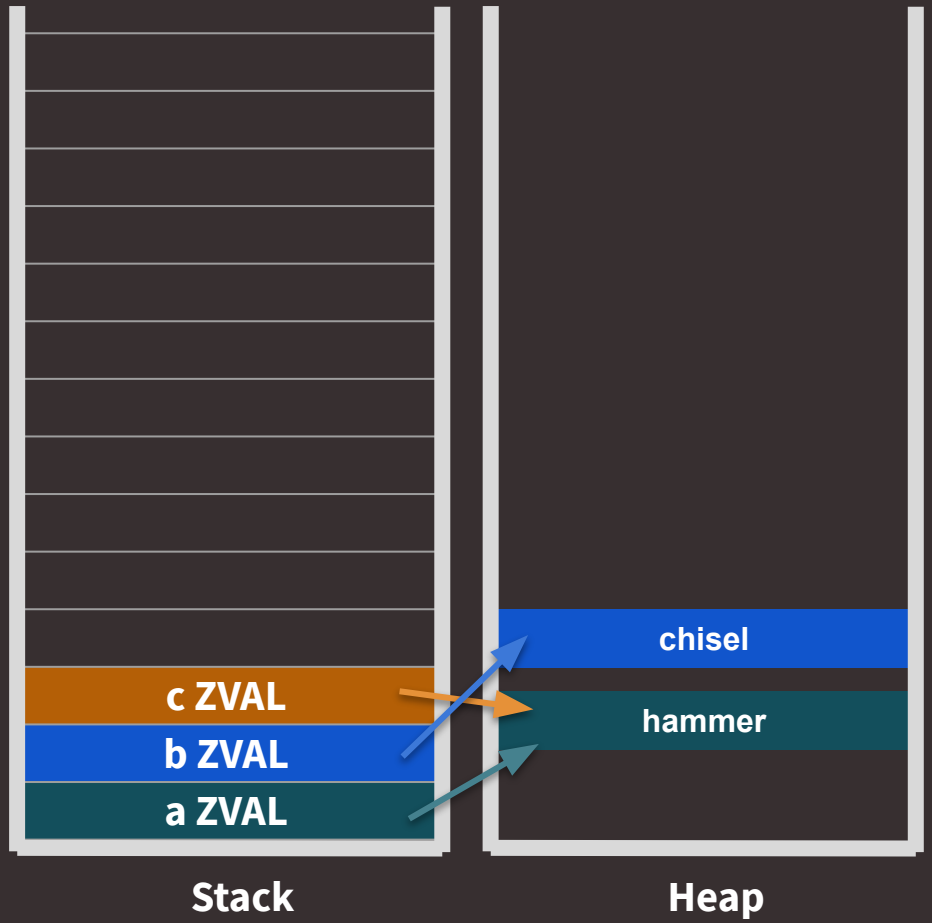


# References

```
// 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'
```

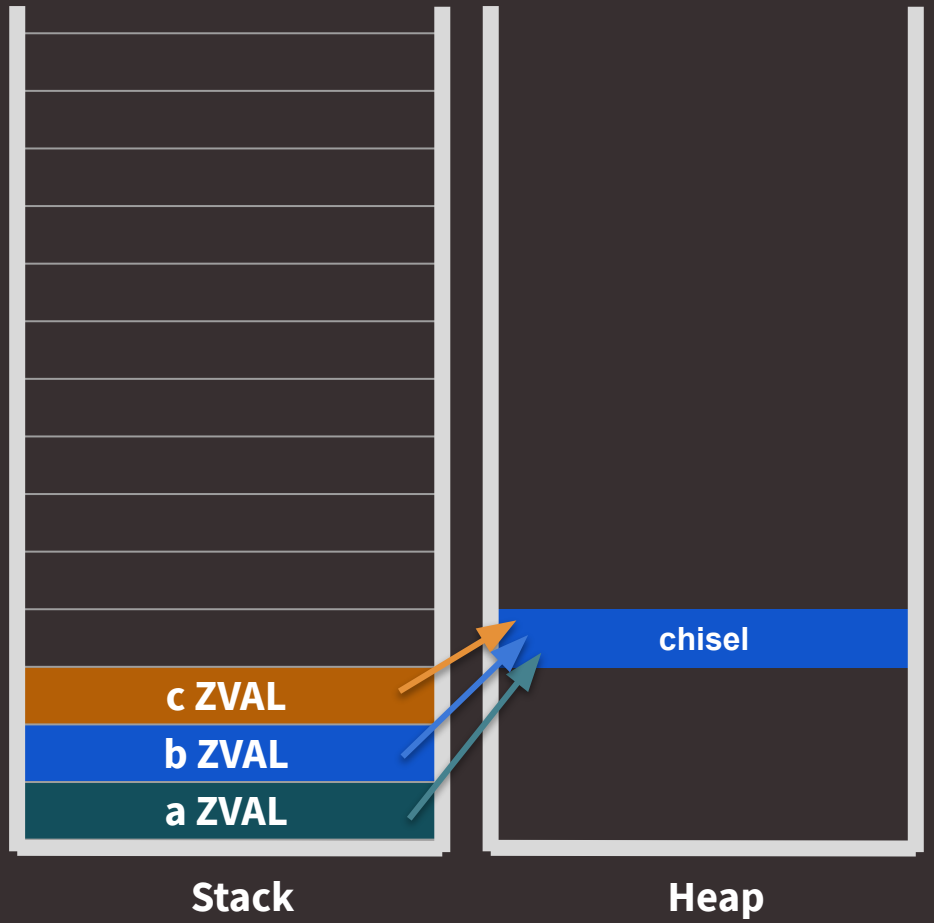


# References

```
// 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'
```

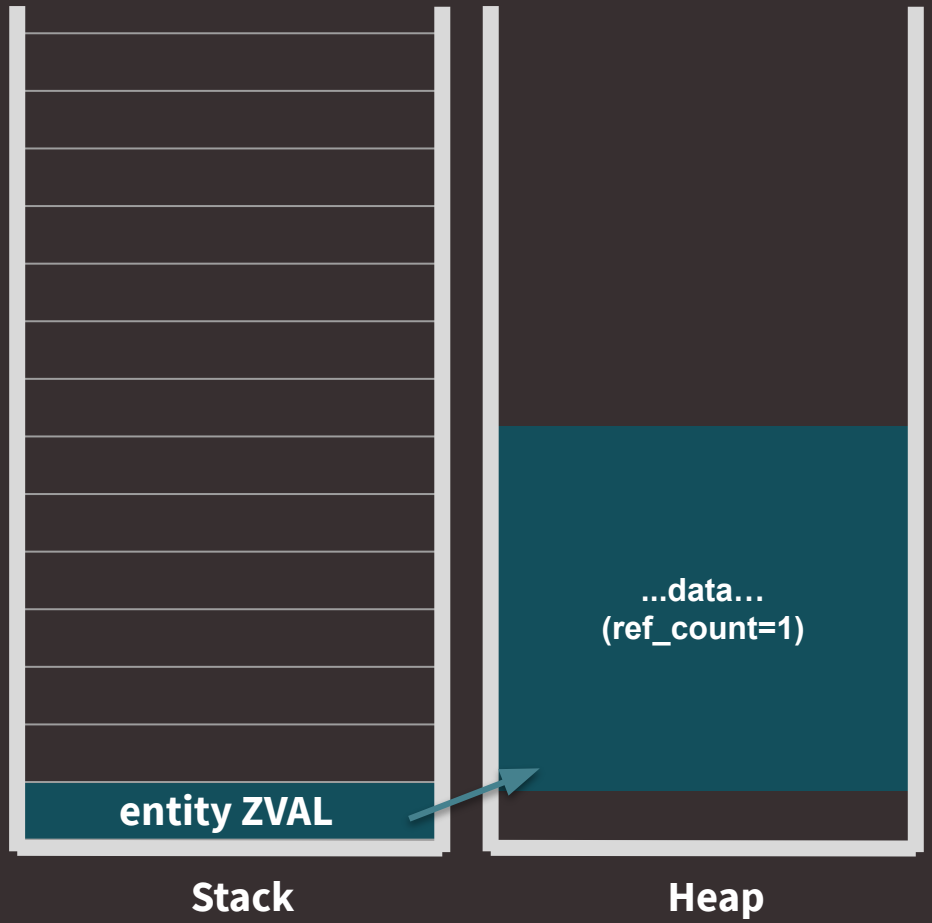


# Garbage Collection

- Memory is allocated for $entity.

```
$entity = Entity:load('123');
updateOwner($entity);
// program continues...

function updateOwner($node) {
  $node->setOwner('1');
  $node->save();
}
```

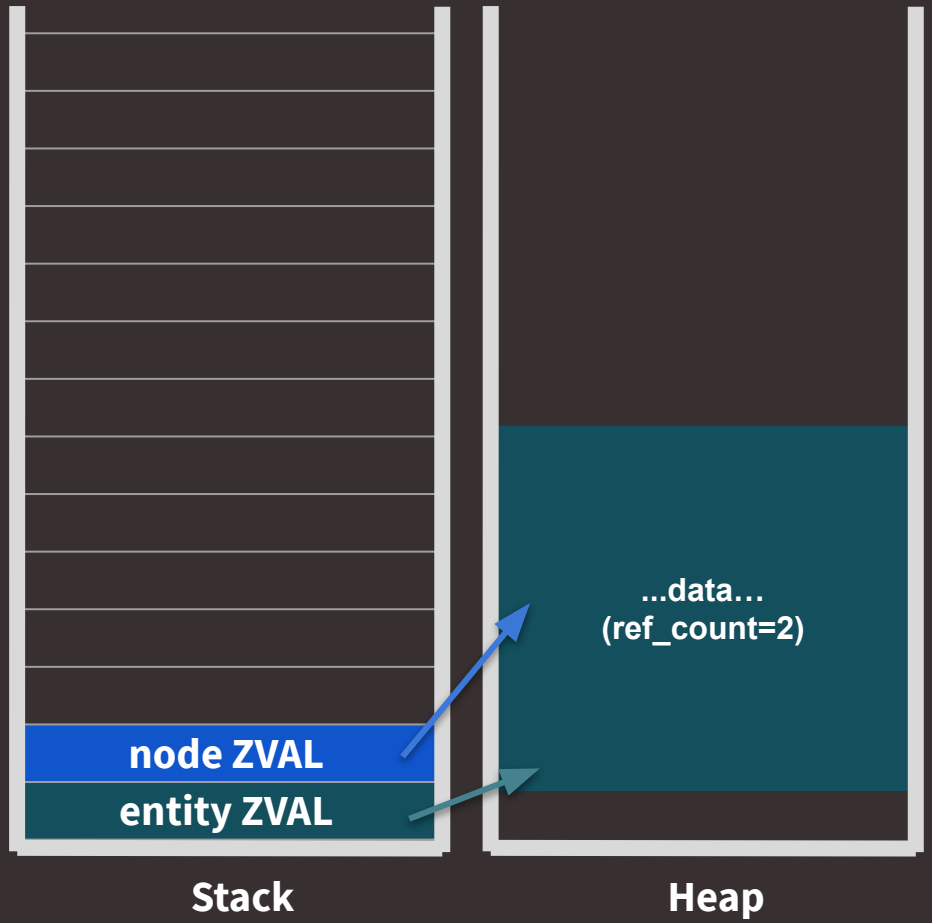


# Garbage Collection

- Step into the updateOwner function where node is created as a soft reference.

```
$entity = Entity:load('123');
updateOwner($entity);
// program continues...

function updateOwner($node) {
  $node->setOwner('1');
  $node->save();
}
```

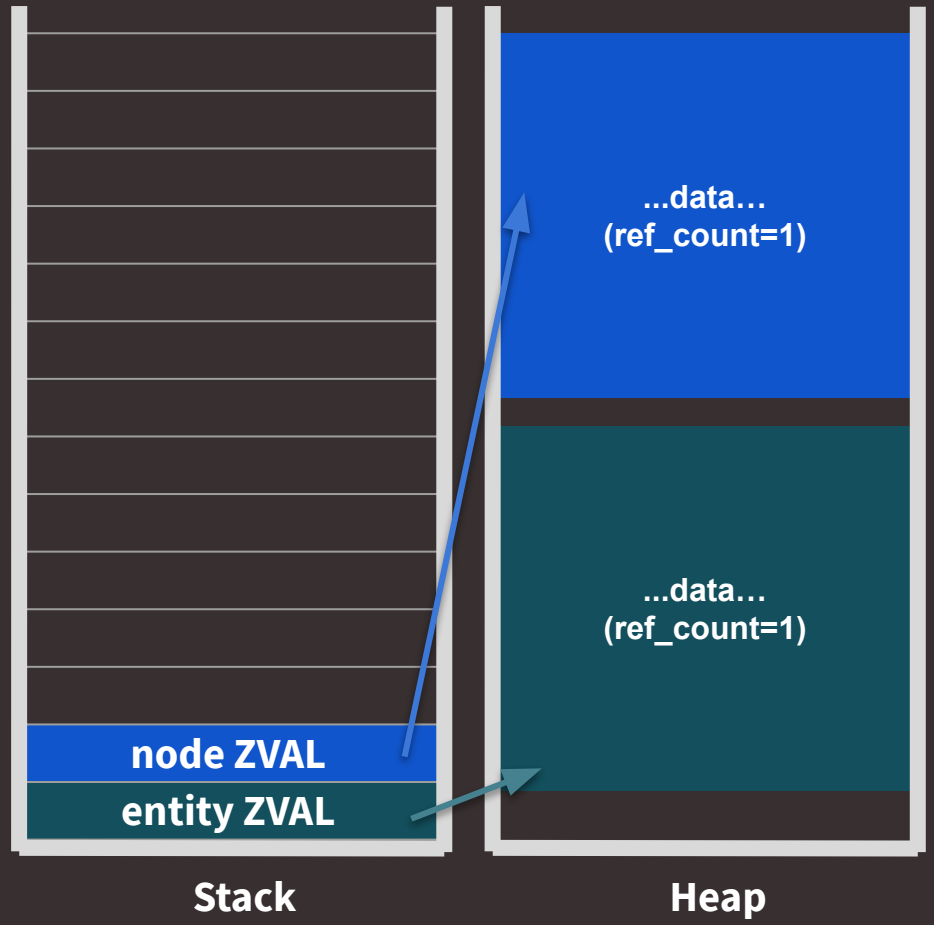


# Garbage Collection

- PHP performs as copy-on-write, allocating memory for node.

```
$entity = Entity:load('123');
updateOwner($entity);
// program continues...

function updateOwner($node) {
  $node->setOwner('1');
  $node->save();
}
```

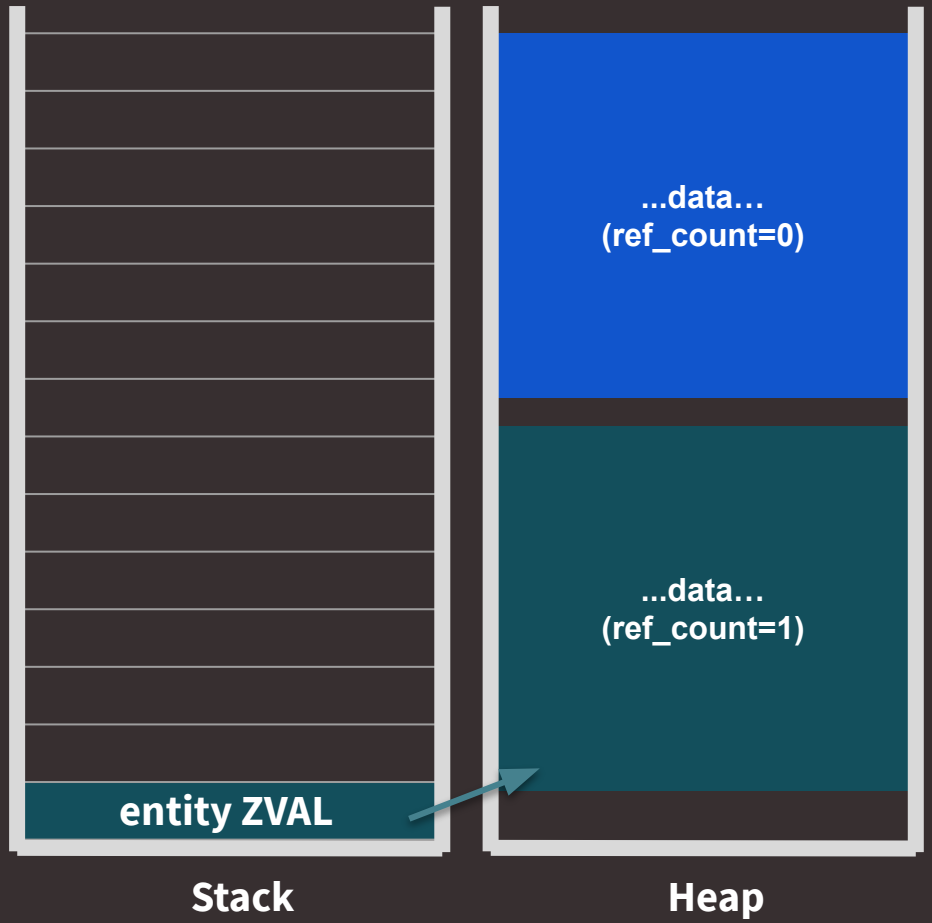


# Garbage Collection

- Once node goes out of scopem the reference count decreases by 1.
- But heap memory is persistent by default.

```
$entity = Entity:load('123');
updateOwner($entity);
// program continues...

function updateOwner($node) {
  $node->setOwner('1');
  $node->save();
}
```

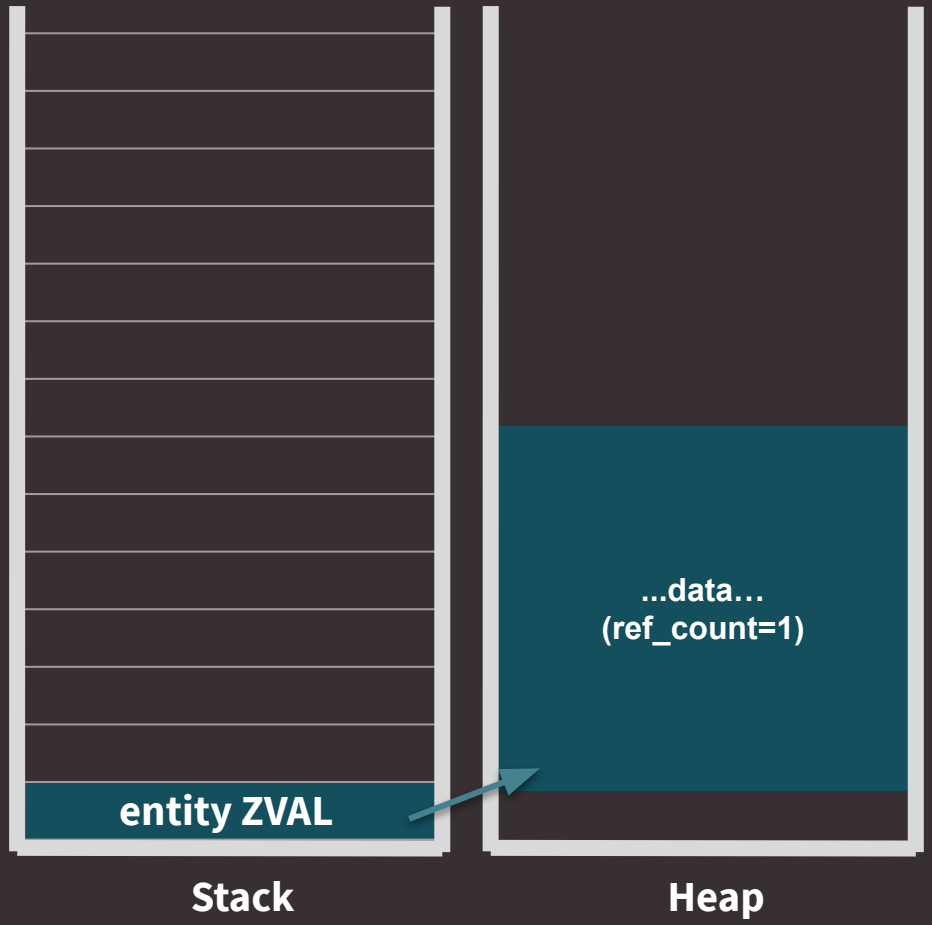


# Garbage Collection

- But no worry of a leak, as the PHP garbage collector frees memory once the ref_count is 0.

```
$entity = Entity:load('123');
updateOwner($entity);
// program continues...

function updateOwner($node) {
  $node->setOwner('1');
  $node->save();
}
```

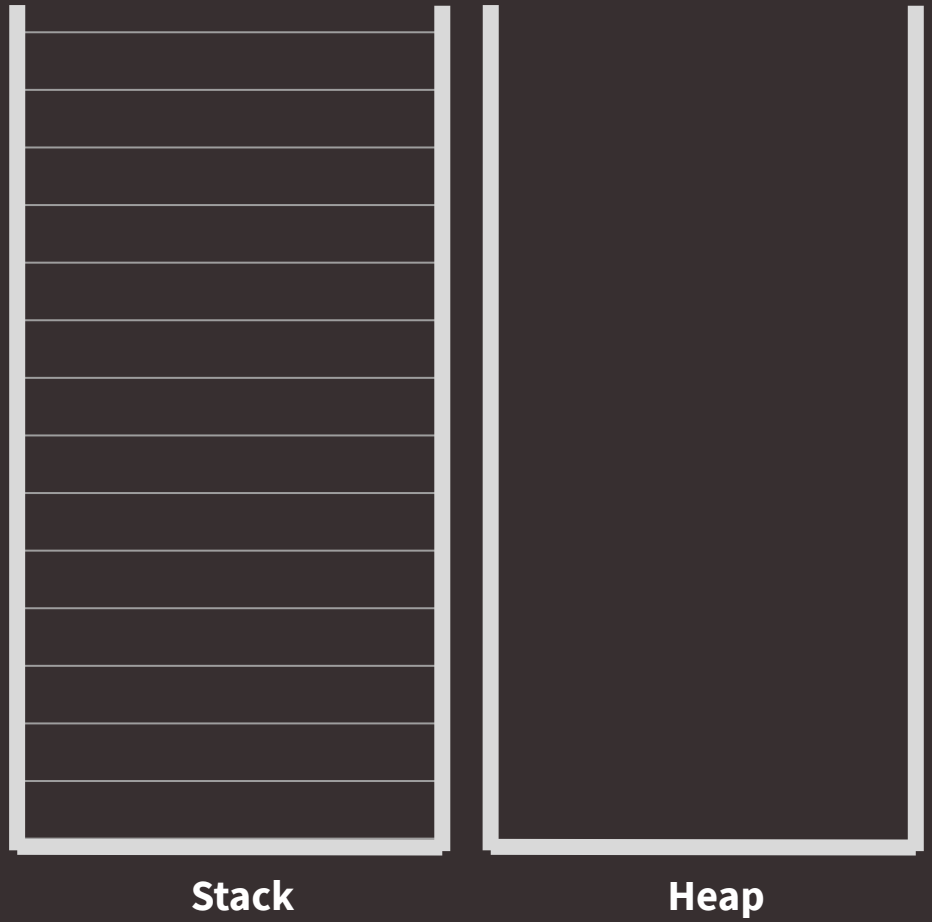


# Garbage Collection

- Unsetting a variable will also decrement the reference counter, letting us free unused memory.

```
$entity = Entity:load('123');
updateOwner($entity);
unset($entity);

function updateOwner($node) {
  $node->setOwner('1');
  $node->save();
}
```



Nuff talk.
Let’s code.

# Takeaways

- Don’t overuse arrays
- Leverage Copy on Write to reduce memory footprint
- Unset variables no longer needed
- Use small functions to allow GC to free unused memory
- Monitor memory usage

# Thank You

https://github.com/nJim/php-memory

Jim.Vomero@FourKitchens.com

 @nJim


# COMING UP NEXT!

- **1:45pm–2pm EDT: Expo Hall & Networking**
- **2pm–2:45pm EDT: Sessions**
  - How Project Management Empowers Accessibility
  - Words Matter: The Language of Accessibility
  - Elevating your skills: Clear intro of tools & tech to learn next!
  - Getting Started With Layout Builder for Drupal 8 & 9