

Dynamic Memory Management

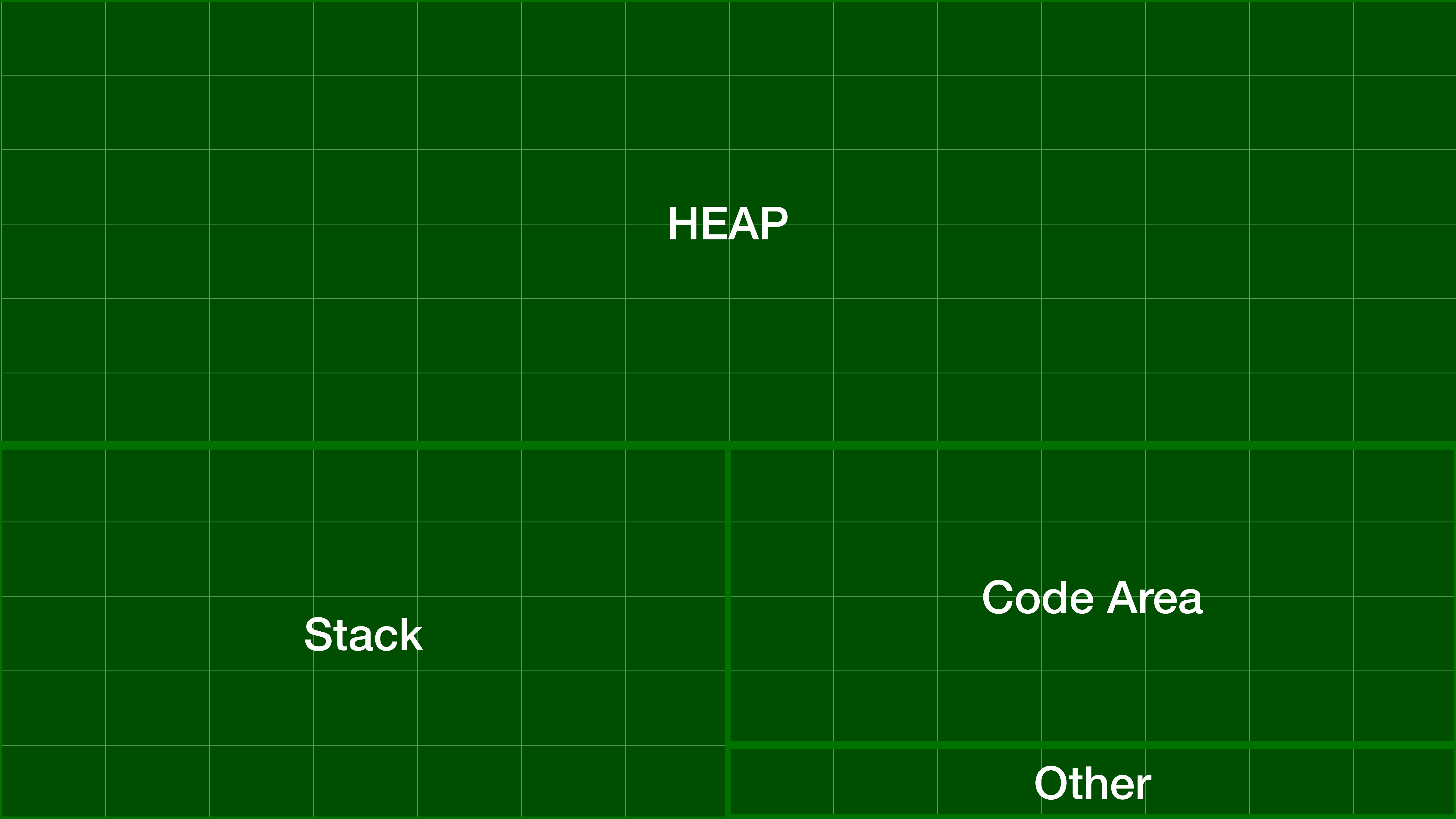
Dynamic Arrays

Dr Jake Renzella

So far, we have been requesting
memory to store variables

**This memory lives in the RAM of
the computer**

Let's take a look at how this
memory is organised



Stores the machine code
instructions in memory

Code Area



Stores the stack frames
from function and
procedure calls

A green grid representing memory allocation, consisting of 14 columns and 6 rows of squares.

HEAP

Stores dynamically
allocated memory

The Stack

```
int my_function(int y) {  
    ...  
}  
  
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}  
  
int main() {  
    my_procedure();  
}
```

The Stack

```
int my_function(int y) {  
    ...  
}  
  
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}
```

```
int main() {  
    my_procedure();  
}
```



main

The Stack

```
int my_function(int y) {  
    ...  
}  
  
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}  
  
int main() {  
    my_procedure();  
}
```



main

The Stack

```
int my_function(int y) {  
    ...  
}  
  
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}  
  
int main() {  
    my_procedure();  
}
```



my_procedure
32 bits <int>

The Stack

```
int my_function(int y) {  
    ...  
}
```

```
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}
```

```
int main() {  
    my_procedure();  
}
```

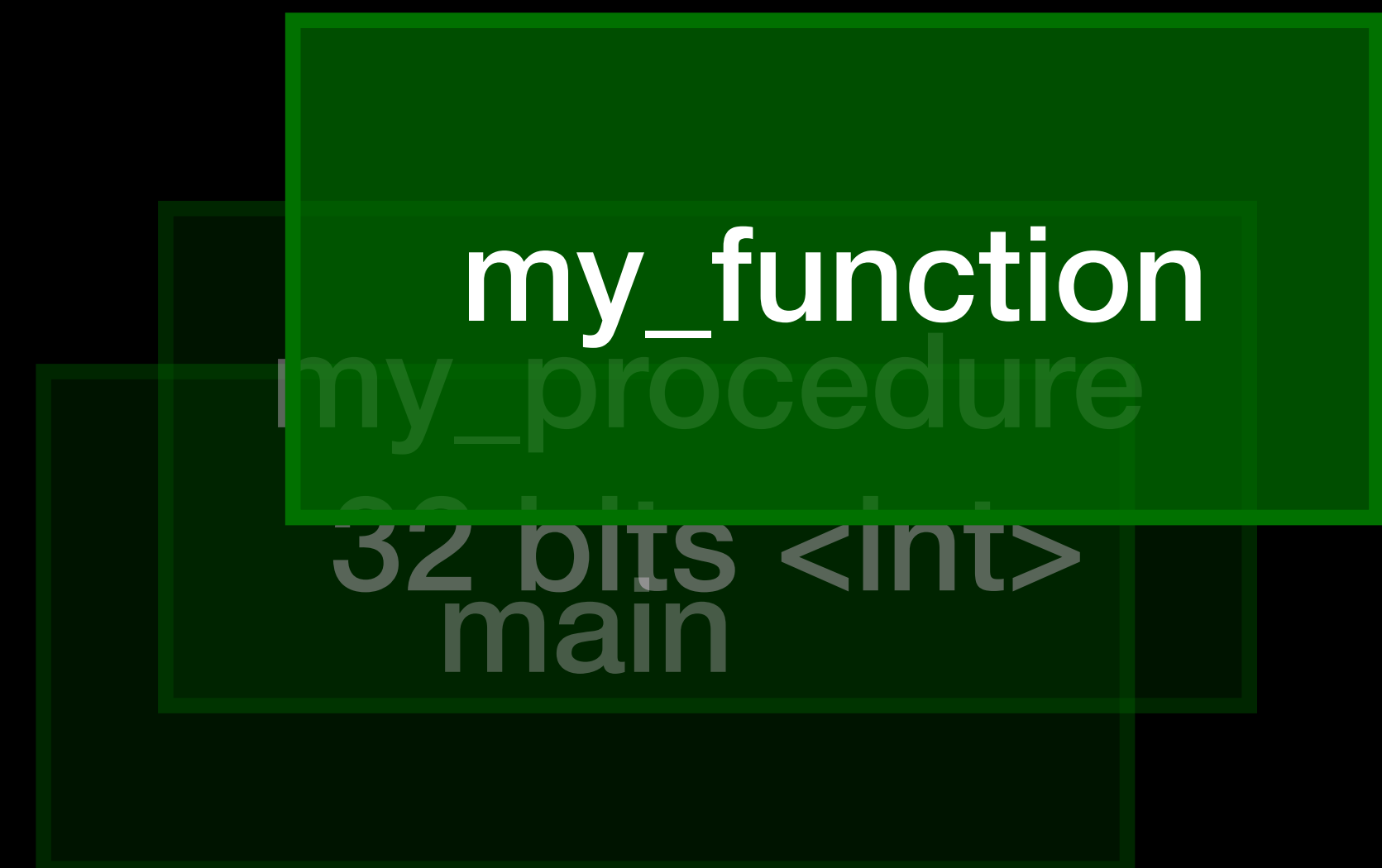


my_procedure
32 bits <int>

```
int my_function(int y) {  
    ...  
}
```

```
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}
```

```
int main() {  
    my_procedure();  
}
```



```
int my_function(int y) {  
    ...  
}
```

```
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}
```

```
int main() {  
    my_procedure();  
}
```



my_procedure
32 bits <int>
main

```
int my_function(int y) {  
    ...  
}  
  
void my_procedure() {  
    int my_value;  
    my_value = my_function();  
}  
  
int main() {  
    my_procedure();  
}
```



main


```
int my_function(int y) {
```

```
    ...
```

```
}
```

```
void my_procedure() {
```

```
    int my_value;
```

```
    my_value = my_function();
```

```
}
```

```
int main() {
```

```
    my_procedure();
```

```
}
```

**We might have
a problem**

If variables are stored on the stack, how does the program know how to fit a large Variable-Length Array?

```
int my_array[10000]
```

It might not.

If only we had a way to manage
memory ourself, outside stack frames

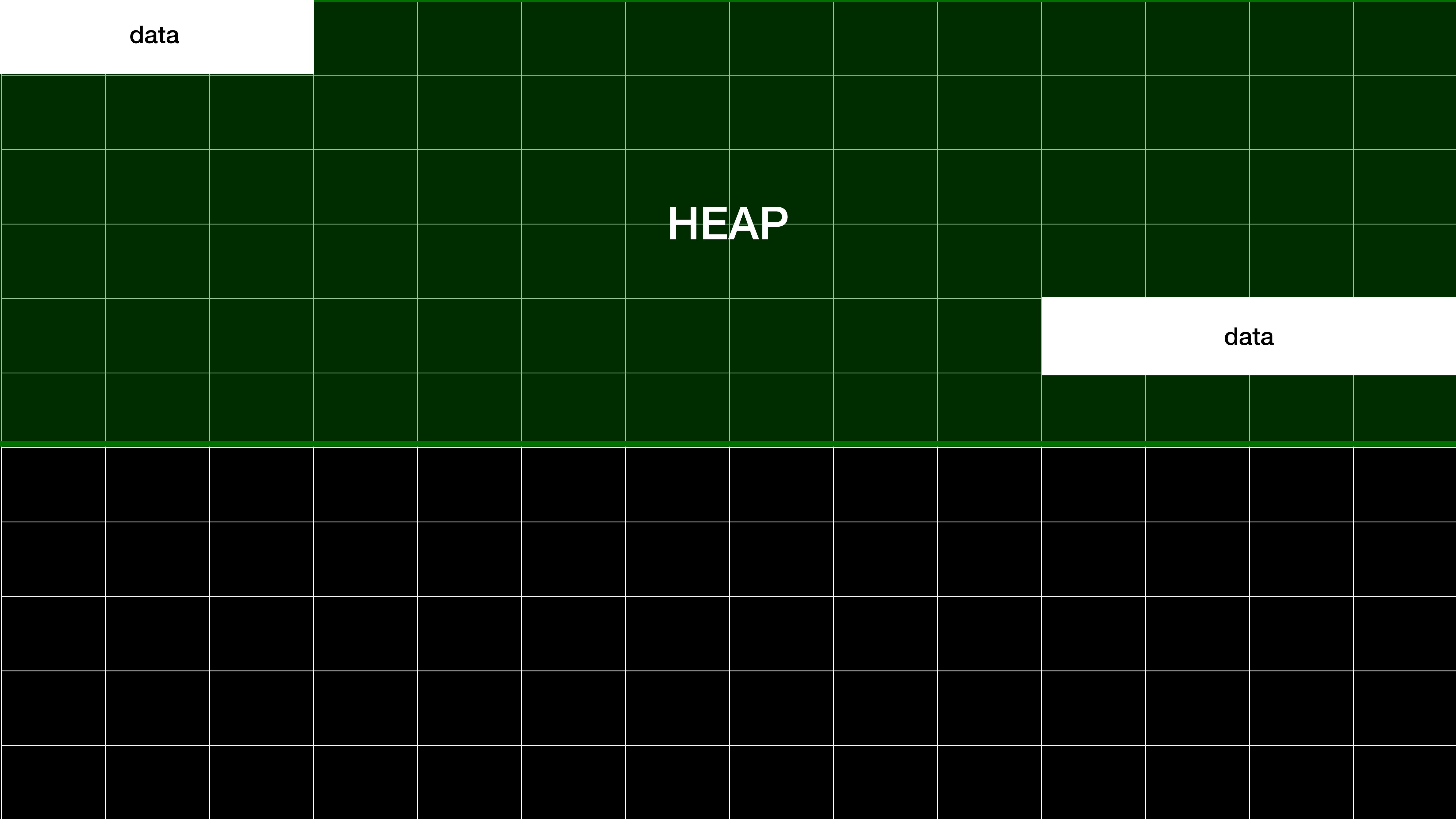
HEAP

The diagram illustrates a memory layout divided into two main sections: **data** and **HEAP**.

The **data** section is located at the top and consists of a grid of 14 columns and 5 rows of dark green blocks. The word "data" is written in the top-left corner of this section.

The **HEAP** section is located below the data section and consists of a grid of 14 columns and 5 rows of dark purple blocks. The word "HEAP" is written in the center of this section.

A horizontal line separates the data section from the HEAP section.



data

HEAP

data

function
main

The diagram illustrates a memory layout divided into three main horizontal sections:

- Top Section (Dark Green):** Labeled "data" in the top-left corner. It contains a large area labeled "HEAP" in the center. To the right of the "HEAP" area, there is a smaller section labeled "data".
- Middle Section (Dark Purple):** This section is empty and serves as a separator between the top and bottom sections.
- Bottom Section (Dark Purple):** Labeled "main" in the bottom-left corner. This section is highlighted with a thick green border.

The diagram illustrates a memory layout divided into three main horizontal sections:

- Top Section (Dark Green):** Labeled "data" in the top-left corner. It contains a large area labeled "HEAP" in the center. To the right of the "HEAP" area, there is a smaller section labeled "data".
- Middle Section (Dark Purple):** This section is empty and serves as a separator between the top and bottom sections.
- Bottom Section (Dark Purple):** Labeled "main" in the bottom-left corner. This section is highlighted with a thick green border.

The diagram illustrates a memory layout divided into three main horizontal sections:

- Top Section (Dark Green):** Labeled "data" in the top-left corner. It contains a large area labeled "HEAP" in the center. A smaller "data" label is located on the right side of this section.
- Middle Section (Dark Purple):** An unlabeled section consisting of a grid of cells.
- Bottom Section (Dark Green):** Labeled "main" in the bottom-left corner. It is highlighted with a thick green border.

The diagram illustrates a memory layout divided into three main horizontal sections:

- Top Section (Dark Green):** Labeled "data" in the top-left corner. It contains a large area labeled "HEAP" in the center. A smaller "data" label is located on the right side of this section.
- Middle Section (Dark Purple):** An unlabeled section consisting of a grid of cells.
- Bottom Section (Dark Green):** Labeled "main" in the bottom-left corner. It is highlighted with a thick green border.

How do we access heap
memory?

Pointers

**Pointers are a variable, which
store a memory address**

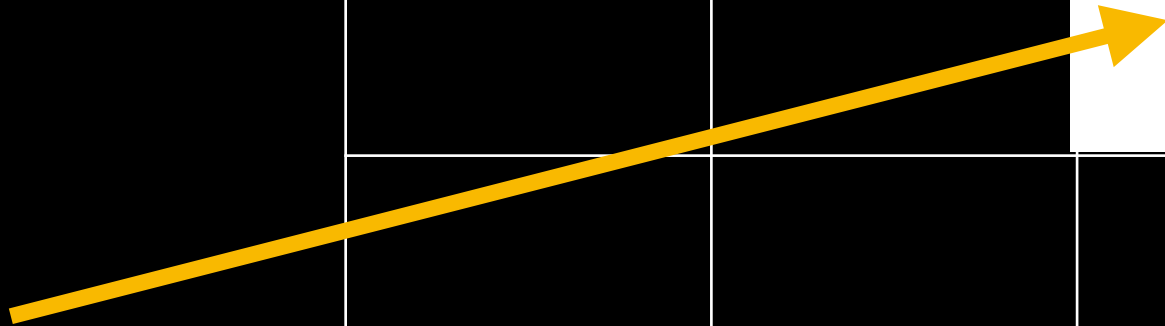
“Address of”
operator looks up
the address of a
value

```
my_pointer = &age;
```

“dereference”
operator looks up
the address of a
value

```
my_age = *age;
```

```
int age = 25;
```



0x7ff17... | 25


```
int age = 25;
```

```
int *age_pointer = &age;
```

0x7ff17... | 25

0x... | 0x7ff17...

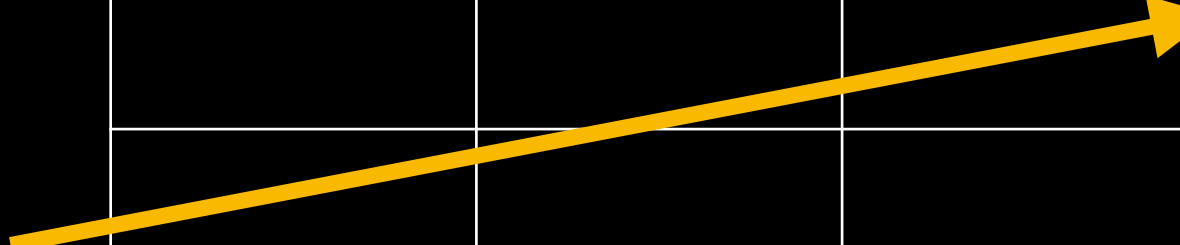
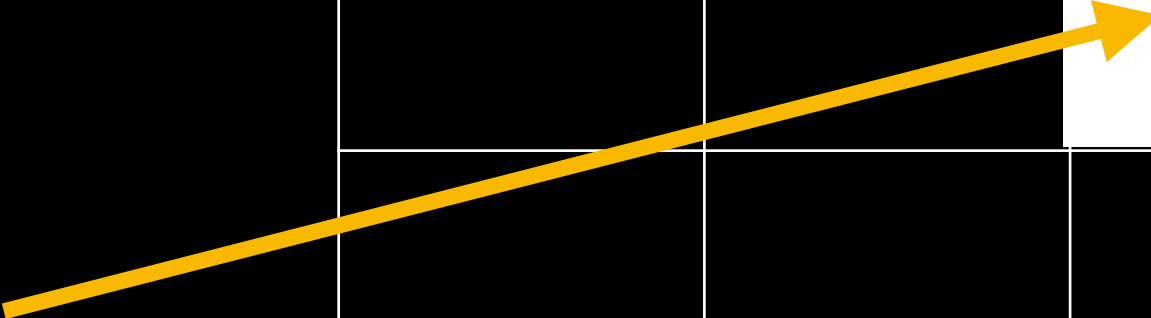
```
int age = 25;
```

```
int *age_pointer = &age;
```

```
printf("%p", age_pointer);
```

0x7ff17... | 25

0x... | 0x7ff17...



int age = 25;

0x7ff17... | 25

int *age_pointer;

0x... | 0x7ff17...

Demo

Pointers to use the Heap

```
int *integer_on_heap = malloc(sizeof(*integer_on_heap));
```

```
int *integer_on_heap = malloc(sizeof(*integer_on_heap));
```



```
int *integer_on_heap = malloc(sizeof(*integer_on_heap));
```




```
int *integer_on_heap = malloc(sizeof(*integer_on_heap));
```

A solid yellow horizontal bar is positioned below the closing parenthesis of the malloc function call in the code snippet.

```
int *integer_on_heap = malloc(sizeof(*integer_on_heap));
```



Declare an array on the heap:

```
int *int_array_on_heap = malloc(sizeof(*integer_on_heap) * size);
```

Increase an existing allocation (and copy over the data)

```
new_p = realloc(my_integer_array, sizeof(*my_integer_array) * new_size);
```