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
Stores the stack frames from function and procedure calls
HEAP

Stores dynamically allocated memory
```c
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();
}

The Stack
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();
}
```c
int my_function(int y) {
    ...
}

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

int main() {
    my_procedure();
}
```
```c
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();
}
int my_function(int y) {
    ...
}

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

int main() {
    my_procedure();
}

```c
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?

```c
int my_array[10000]
```
It might not.
If only we had a way to manage memory ourself, outside stack frames
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

HEAP
| data | HEAP |
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

```c
my_age = *age;
```
int age = 25;
int age = 25;
int *age_pointer = &age;
int age = 25;

int *age_pointer = &age;

printf("%p", age_pointer);
int age = 25;

int *age_pointer;
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));
int *integer_on_heap = malloc(sizeof(*integer_on_heap));
Declare an array on the heap:

```c
int *int_array_on_heap = malloc(sizeof(*integer_on_heap) * size);
```
Increase an existing allocation (and copy over the data)

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