# Call Stack and Scope

# Scope

- The name that identifies a variable has certain visibility throughout the program
- Three fundamental levels of scope
    - Global
    - Local
    - Block

# Global Scope

- Also known as file scope
- Variables declared outside of the functions have global scope
- These variables and their identifiers are accessible in all the functions
- The functions can both read and modify the value of a global variable unless they are declared as constants

# Global Scope

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

double acceleration = 9.8; // m/s^2
int start_time = 20; // seconds


double free_fall_velocity(int time);
void display_velocities(const int *time_array, int size);
void test_velocity_function();

int main() {
```

```c
int main() {

    size_t array_size = 5;
    int array[array_size];

    for (int i = 0; i < array_size; ++i) {
        array[i] = 0;
    }

    int i = 0;

    while (i < array_size) {
        array[i] = start_time;
        start_time = start_time + (4 * i);
        ++i;
    }

    printf("The value of \"i\" is %i\n", i);
    printf("The value of \"start_time\" is %i\n", start_time);
    display_velocities(array, array_size);
    printf("Running Tests!\n");
    test_velocity_function();
    return 0;
}
```

# Global Scope

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

double acceleration = 9.8; // m/s^2
int start_time = 20; // seconds


double free_fall_velocity(int time);
void display_velocities(const int *time_array, int size);
void test_velocity_function();

int main() {
```

```c
double free_fall_velocity(int time) {

    return acceleration * time;
}
```

# Possible Advantages of Global Variables

- Can be used instead of macros in cases when values need to be of specific type
    - No type checking is done in macros
    - macro expansion is done in preprocessing stage
    - Not checked for compilation error
    - Use of macros can lead to errors

# Pitfalls

- Global variables are modifiable unless they are *const*
- If values are not supposed to change, use of global variables may lead to incorrect results
- The identifiers also have global scope
  - We lose out on a few variable names

# Local Scope

- Variables declared within functions have scope local to that function
- Variables array_size and array have local scope within the main() function

```c
int main() {

    size_t array_size = 5;
    int array[array_size];

    for (int i = 0; i < array_size; ++i) {
        array[i] = 0;
    }

    int i = 0;

    while (i < array_size) {
        array[i] = start_time;
        start_time = start_time + (4 * i);
        ++i;
    }

    printf("The value of \"i\" is %i\n", i);
    printf("The value of \"start_time\" is %i\n", start_time);
    display_velocities(array, array_size);
    printf("Running Tests!\n");
    test_velocity_function();
    return 0;
}
```

# Local Scope

- time_array provides access to the data in variable array in main by reference
- the variable named array is NOT visible to this function.

```
void display_velocities(const int *time_array, int size) {

    for (int i = 0; i < size; ++i) {
        printf("The free fall speed after %i seconds is: %lf m/s^2\n",
                time_array[i], free_fall_velocity(time_array[i]));
    }
}
```

# Block Scope

- The varible i has block scope within this for loop
- It is also accessible to any nested loops, conditions, or blocks within this for loop.

```c
int main() {

    size_t array_size = 5;
    int array[array_size];

    for (int i = 0; i < array_size; ++i) {
        array[i] = 0;
    }

    int i = 0;

    while (i < array_size) {
        array[i] = start_time;
        start_time = start_time + (4 * i);
        ++i;
    }

    printf("The value of \"i\" is %i\n", i);
    printf("The value of \"start_time\" is %i\n", start_time);
    display_velocities(array, array_size);
    printf("Running Tests!\n");
    test_velocity_function();
    return 0;
}
```

# Block Scope

- Variable i was declared outside of the while loop
  - still visible after the while loop and it maintains its value.

```
int main() {

    size_t array_size = 5;
    int array[array_size];

    for (int i = 0; i < array_size; ++i) {
        array[i] = 0;
    }

    int i = 0;

    while (i < array_size) {
        array[i] = start_time;
        start_time = start_time + (4 * i);
        ++i;
    }

    printf("The value of \"i\" is %i\n", i);
    printf("The value of \"start_time\" is %i\n", start_time);
    display_velocities(array, array_size);
    printf("Running Tests!\n");
    test_velocity_function();
    return 0;
}
```

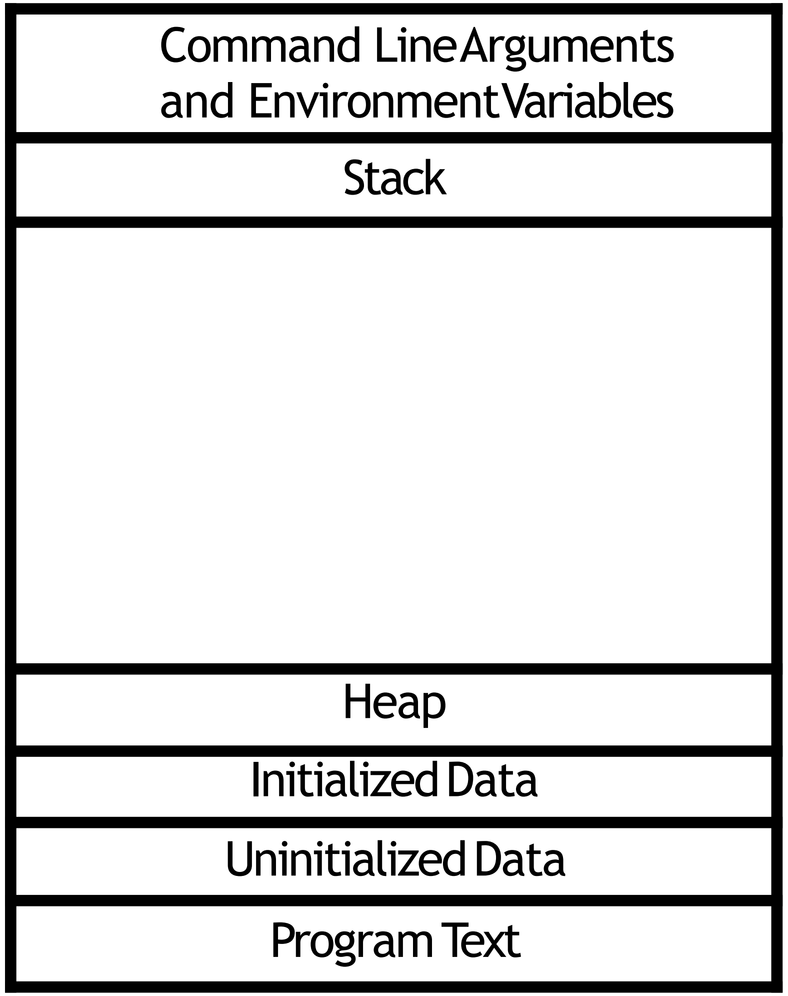
- You can create your own blocks without a conditional or looping structure.
- Any variables declared within curly braces have their scope
  - limited to code within the same (or a nested) block

```
void test_velocity_function() {

    {
        int input_test = 5;
        double expected = 49;
        assert(free_fall_velocity(input_test) == expected);
    }

    {
        int input_test = 2;
        double expected = 19.6;
        assert(free_fall_velocity(input_test) == expected);
    }

    {
        int input_test = 9;
        double expected = 88.2;
        assert(free_fall_velocity(input_test) == expected);
    }
}
```

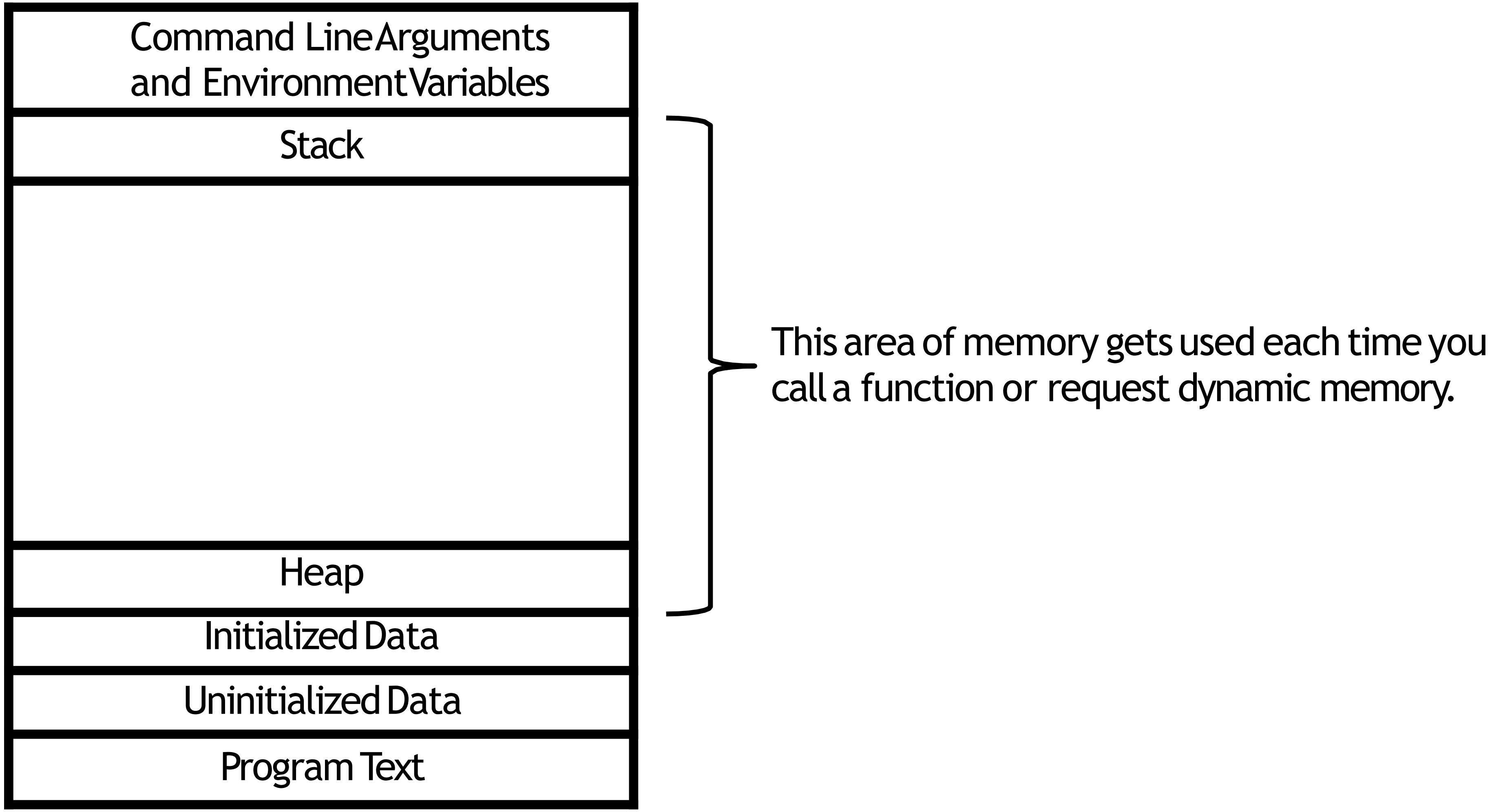
- Can be used for testing
  - You can copy and paste the block, change the values you are testing
  - variable names can be the same since the scope is limited to the block where they are declared.

```
void test_velocity_function() {

    {
        int input_test = 5;
        double expected = 49;
        assert(free_fall_velocity(input_test) == expected);
    }

    {
        int input_test = 2;
        double expected = 19.6;
        assert(free_fall_velocity(input_test) == expected);
    }

    {
        int input_test = 9;
        double expected = 88.2;
        assert(free_fall_velocity(input_test) == expected);
    }
}
```

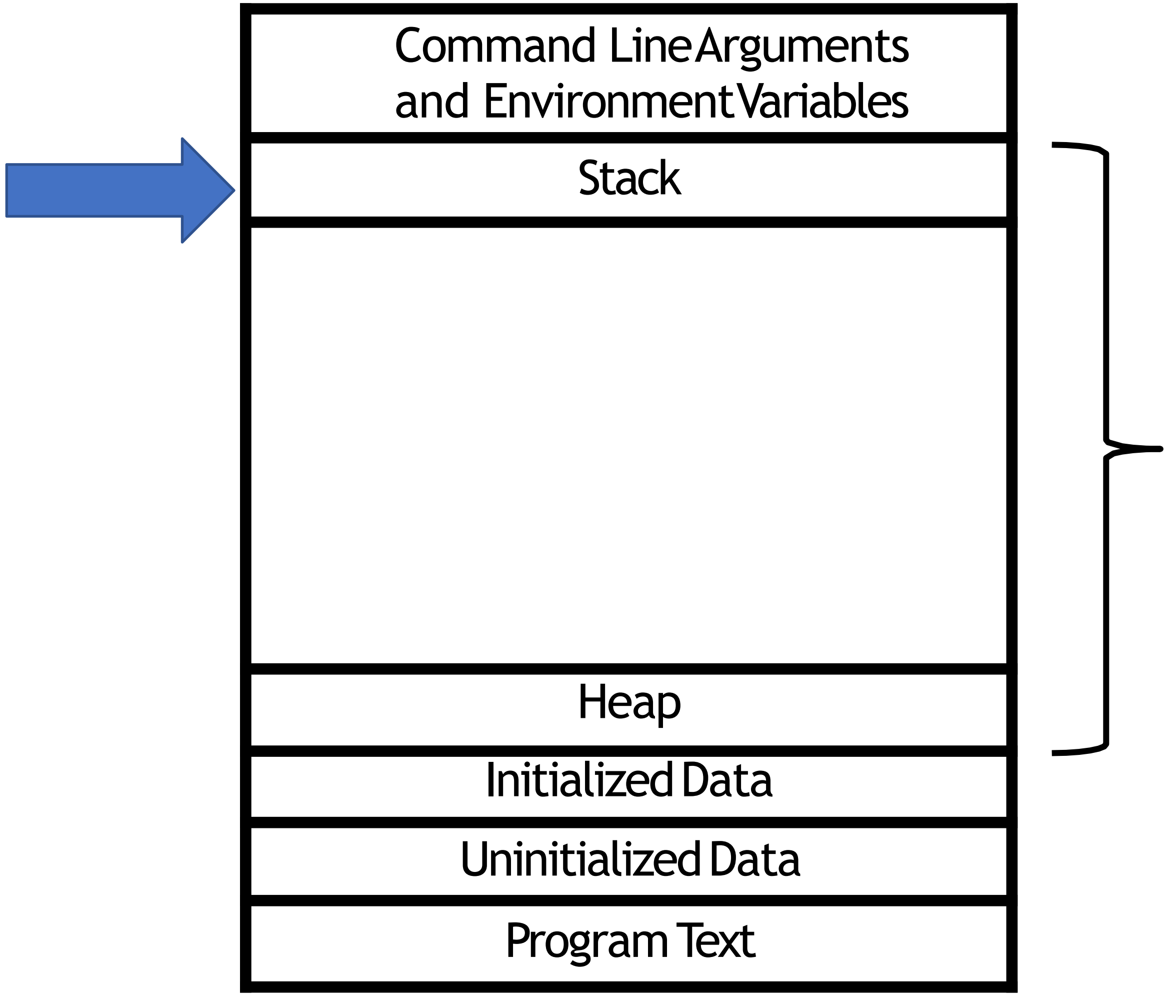
# Programs in Memory



# Programs in Memory



# Programs in Memory




This area of memory gets used each time you call a function or request dynamic memory.

# What is a Stack?

- A stack is a type of data structure
- Think of it like a stack of papers or dishes
- We can:
  - add items to the top with a **PUSH**
  - remove items from the top with a **POP**

# The Call Stack

- Often just called “the stack”.
- Every running program has its own stack
- Each time a function is called a stack frame is pushed onto the stack
  - The function at the top of the stack is the active function
- A stack frame consists of:
  - Return address (where in the code the function was called)
  - Automatic variables used by the function

# Automatic Variables

- All the variables we have been using so far have been automatic variables.
- When a function is called, memory (RAM) is allocated for local variables and function parameters.
- The memory is automatically released when the function returns (or reaches the end in the case of a void function).

# How does this work?

```
int sum(int a, int b) {
  return a + b;
}

int main() {
  int value = sum(1, 2);
  printf("%i\n", value);
  return 0;
}
```

The Stack

# How does this work?

```
int sum(int a, int b) {
  return a + b;
}
```

The main function is pushed to the stack with its variables when the program starts.

```
int main() {
  int value = sum(1, 2);
  printf("%i\n", value);
  return 0;
}
```

## The Stack

Active Function

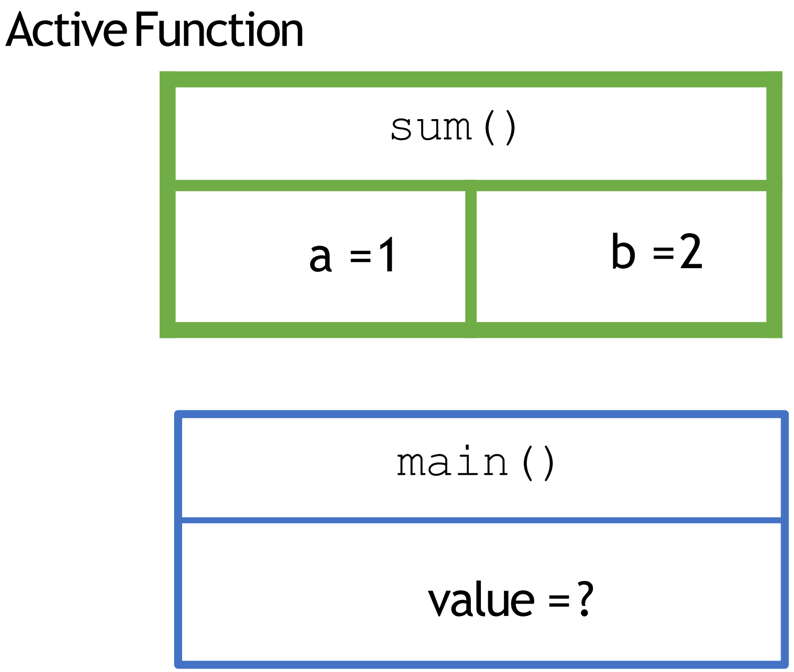
| main() |
|---|
| value = ? |

# How does this work?

```
int sum(int a, int b) {
    return a + b;
}

int main() {
    int value = sum(1, 2);
    printf("%i\n", value);
    return 0;
}
```

Once we reach the call to sum, we need to push that to the stack.

## The Stack



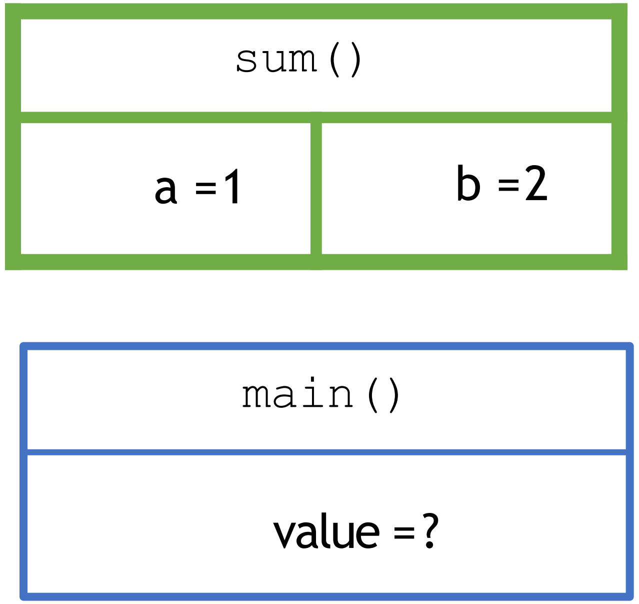
# How does this work?

```
int sum(int a, int b) {
  return a + b;
}


int main() {
  int value = sum(1, 2);
  printf(“%i\n”, value);
  return 0;
}
```

When we return from sum we pop sum off the stack and go back to the main function.

## The Stack

Active Function



# How does this work?

```
int sum(int a, int b) {
  return a + b;
}


int main() {
  int value = sum(1, 2);
  printf("%i\n", value);
  return 0;
}
```

Printf is also a function so that will have to be pushed on the stack.

## The Stack

Active Function

| printf() | |
|---|---|
| "%i\n" | 3 |

| main() |
|---|
| value = 3 |

# How does this work?

When printf is complete, we can remove the function from the stack and resume the main.

```c
int sum(int a, int b) {
   return a + b;
}


int main() {
   int value = sum(1, 2);
   printf("%i\n", value);
   return 0;
}
```

## The Stack

Active Function

| main() |
|---|
| value = 3 |

# How does this work?

The Stack

```
int sum(int a, int b) {
  return a + b;
}

int main() {
  int value = sum(1, 2);
  printf("%i\n", value);
  return 0;
}
```

When main returns it is also removed from the the stack and the program quits.

# You can try this out for yourself!

- Python tutor can visualize the running of a single file C program and show the callstack

- http://pythontutor.com/c.html#mode=edit