Week 4 Lecture 1

Expressions and Functions

Expressions

- A representation of a value
  - Expressions have a type
  - Expressions have a value
- Examples
  - 1 + 2: type int; value 3
  - 1.2 + 3: type float; value 4.2

More expression examples

- If you declare two int variables:

  ```
  int a = 1;
  int b = 2;
  ```

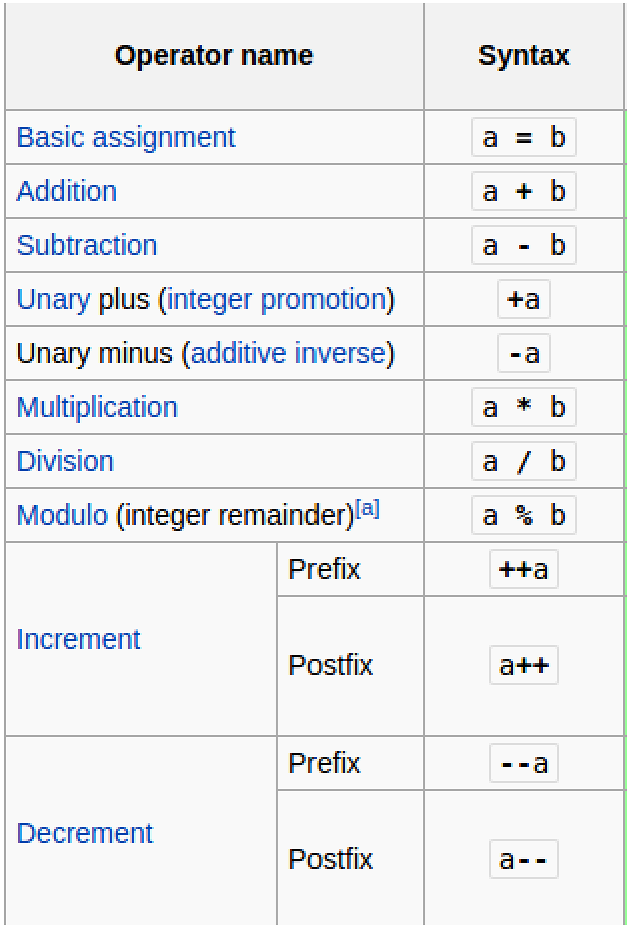
  - Expression with value 1: `a`
  - Expression with value 2: `a + 1`
  - Expression with value 6: `2 * (a + b)`
  - Expression with value 2: `++a`
  - Expression with value 1: `a++`
    - Huh?

Operators

- Expressions comprise operations with variable or constants
  - Examples
    - 1 + 2
    - 2 * (a + b)
    - -1 + b

Arithmetic Operators

- Assignment
  - In C = does not mean “equals”
  - It means put the value on the right in the location on the left.
- Modulo (%)
  - Int only
  - %: remainder after int division
- Division (/)
  - Int: the integer part of division
    - e.g., 3/2 == 1
  - Float: the closes float to the result of the division (3.0/2 == 1.4)

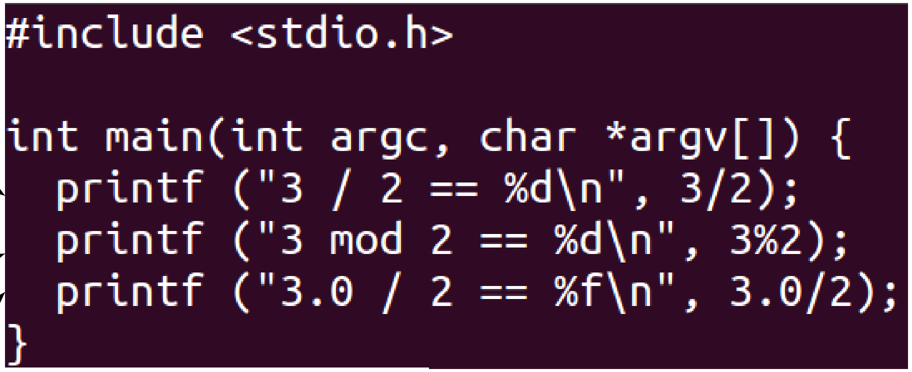


Operator name		Syntax
Basic assignment		`a = b`
Addition		`a + b`
Subtraction		`a - b`
Unary plus (integer promotion)		`+a`
Unary minus (additive inverse)		`-a`
Multiplication		`a * b`
Division		`a / b`
Modulo (integer remainder)[a]		`a % b`
Increment	Prefix	`++a`
	Postfix	`a++`
Decrement	Prefix	`--a`
	Postfix	`a--`

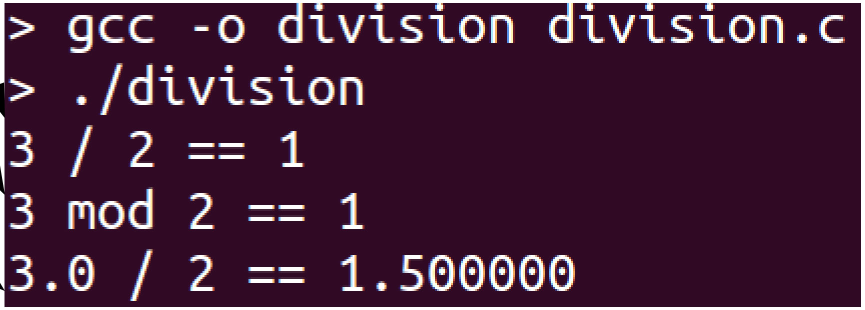
From Wikipedia: Operators in C and C++

Division and modulus

- Integer division
- Integer modulus
- Float division

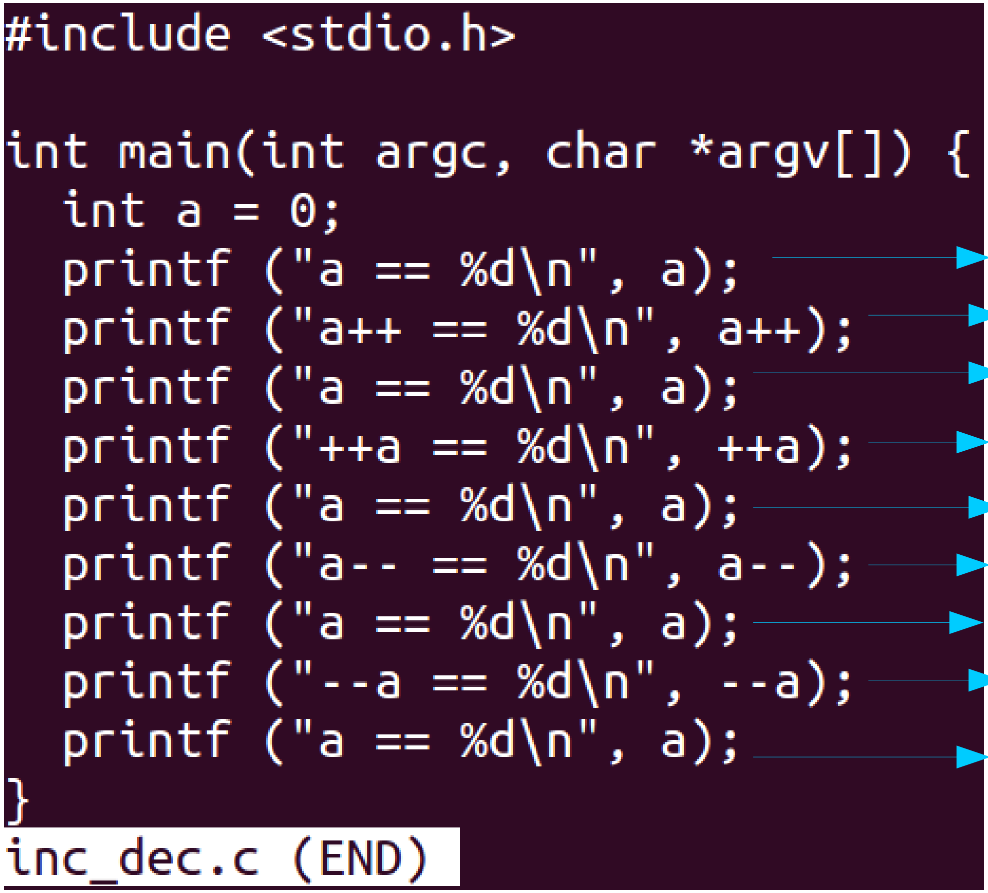


```
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf ("3 / 2 == %d\n", 3/2);
  printf ("3 mod 2 == %d\n", 3%2);
  printf ("3.0 / 2 == %f\n", 3.0/2);
}
```

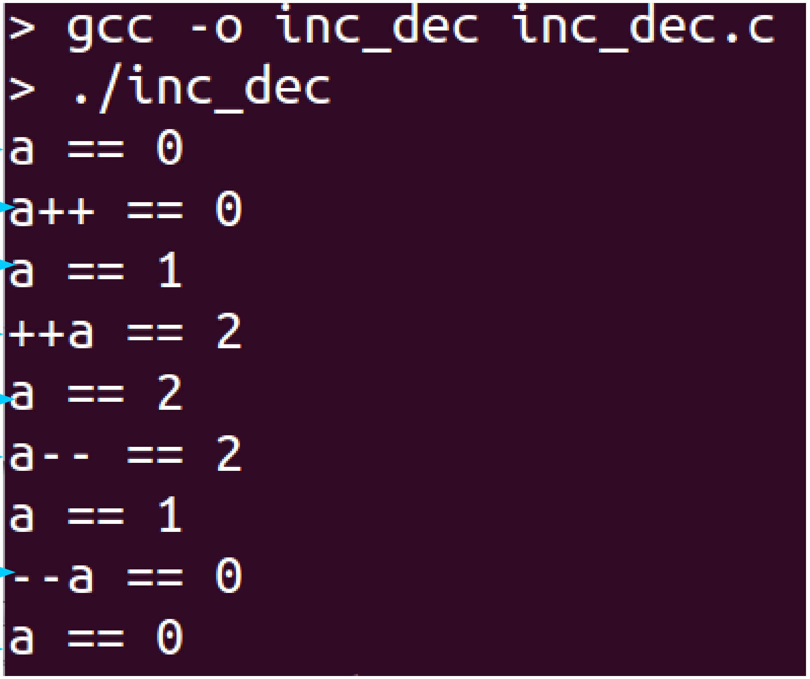


```
> gcc -o division division.c
> ./division
3 / 2 == 1
3 mod 2 == 1
3.0 / 2 == 1.500000
```

Increment and Decrement



```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int a = 0;
  printf ("a == %d\n", a);
  printf ("a++ == %d\n", a++);
  printf ("a == %d\n", a);
  printf ("++a == %d\n", ++a);
  printf ("a == %d\n", a);
  printf ("a-- == %d\n", a--);
  printf ("a == %d\n", a);
  printf ("--a == %d\n", --a);
  printf ("a == %d\n", a);
}
inc_dec.c (END)
```



```
> gcc -o inc_dec inc_dec.c
> ./inc_dec
a == 0
a++ == 0
a == 1
++a == 2
a == 2
a-- == 2
a == 1
--a == 0
a == 0
```

Assignment Operators

- The value on the left of the '=' is treated as an location
- The value on the right is put in that location

Operator	Meaning
a = b	Simple assignment
a += b	a = a + b
a -= b	a = a – b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

Precedence

Precedence	Operator	Associativity
1 (highest)	++ (suffix) – (suffix) () []	Left-to-right
2	++ (prefix) – (prefix) + (unary) - (unary) ! (<type>) * (value at) & (address of)	Right-to-left
3	* / %	Left-to-right
4	+ -	Left-to-right

Precedence

Precedence	Operator	Associativity
5	< <= > >=	Left-to-right
6	== !=	Left-to-right
7	&&	Left-to-right
8	||	Left-to-right
9	?:	Right-to-left
10	= += -= *= /= %=	Right-to-left

Precedence Moral

- Always parenthesize your expressions.

Function Declaration

- <type> <name>(<parameter_list>);
  - E.g., int add(int a, int b);
    - Declares a function called add that returns an int when passed two ints.
    - The first int passed will be called a inside the function; the second will be called b.
  - The compiler knows it is a function declaration by the type, parentheses and semi-colon.
- Function declarations indicate the syntax of the function
- Functions must be declared before they are called.

Function Definition

- <type> <name>(<parameters>) { }
  - E.g., int add (int a, int b)

    ```
    {
        return a + b;
    }
    ```

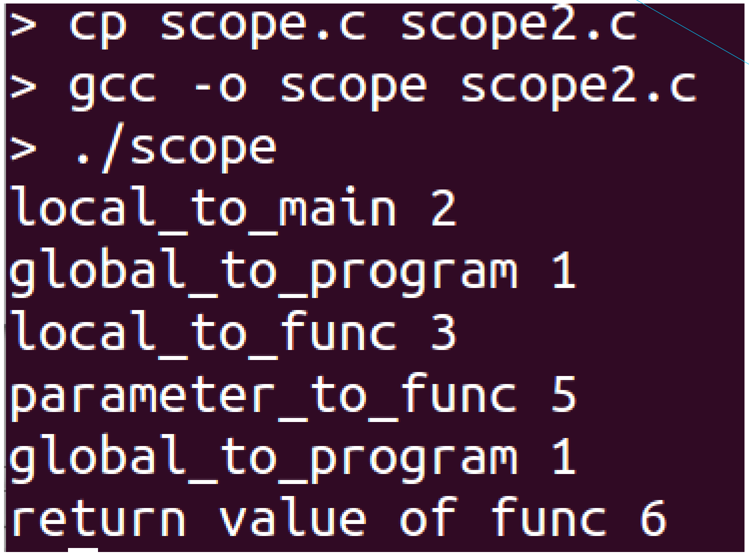
    - Defines the function to return the sum of its two parameters.
    - Body: { return a + b }
  - The compiler can tell it is a function definition by the type, parentheses and curly brackets.
- Function definitions indicate what the function does.

Function Call

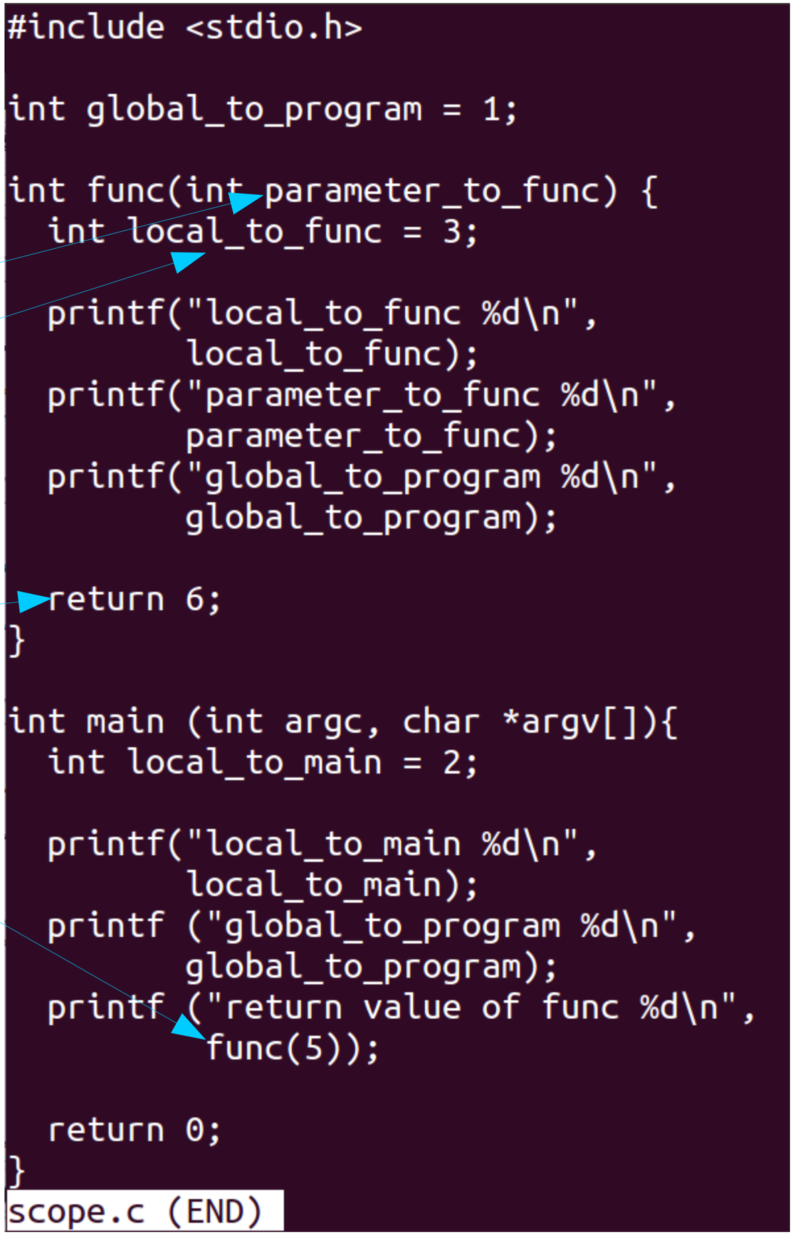
- <name>(<parameters>);
  - E.g., add(2, 3);
  - Executes the body of the function.
  - Compiler recognized a function call because it has parentheses, but no type.
- A function call is an expression whose value is the return value.

Scope example 2

- Parameter
- Local
- Return value



```
> cp scope.c scope2.c
> gcc -o scope scope2.c
> ./scope
local_to_main 2
global_to_program 1
local_to_func 3
parameter_to_func 5
global_to_program 1
return value of func 6
```



```
#include <stdio.h>

int global_to_program = 1;

int func(int parameter_to_func) {
  int local_to_func = 3;

  printf("local_to_func %d\n",
         local_to_func);
  printf("parameter_to_func %d\n",
         parameter_to_func);
  printf("global_to_program %d\n",
         global_to_program);

  return 6;
}

int main (int argc, char *argv[]){
  int local_to_main = 2;

  printf("local_to_main %d\n",
         local_to_main);
  printf ("global_to_program %d\n",
         global_to_program);
  printf ("return value of func %d\n",
          func(5));

  return 0;
}
scope.c (END)
```

() Operator

- () is an operator.
  - It is applied to an pointer.
  - When you define a function, you define a name that points to a location in memory that contains executable code.
  - When you call a function, you execute that code
    - The value of the expression is the value the function returns.
  - Style: the () operator goes immediately after the function name [i.e., no space; e.g., func()]

Functions are expressions

- This is important: function are like variables
  - They can be used wherever variables are used
  - Well almost: you cannot assign values to a function
    - But you can assign functions to functions.
- E.g., Expression using (badly named) function f and g.
  - (f() + 1) * 3) || g() == f()
  - (f() * g()) + 3

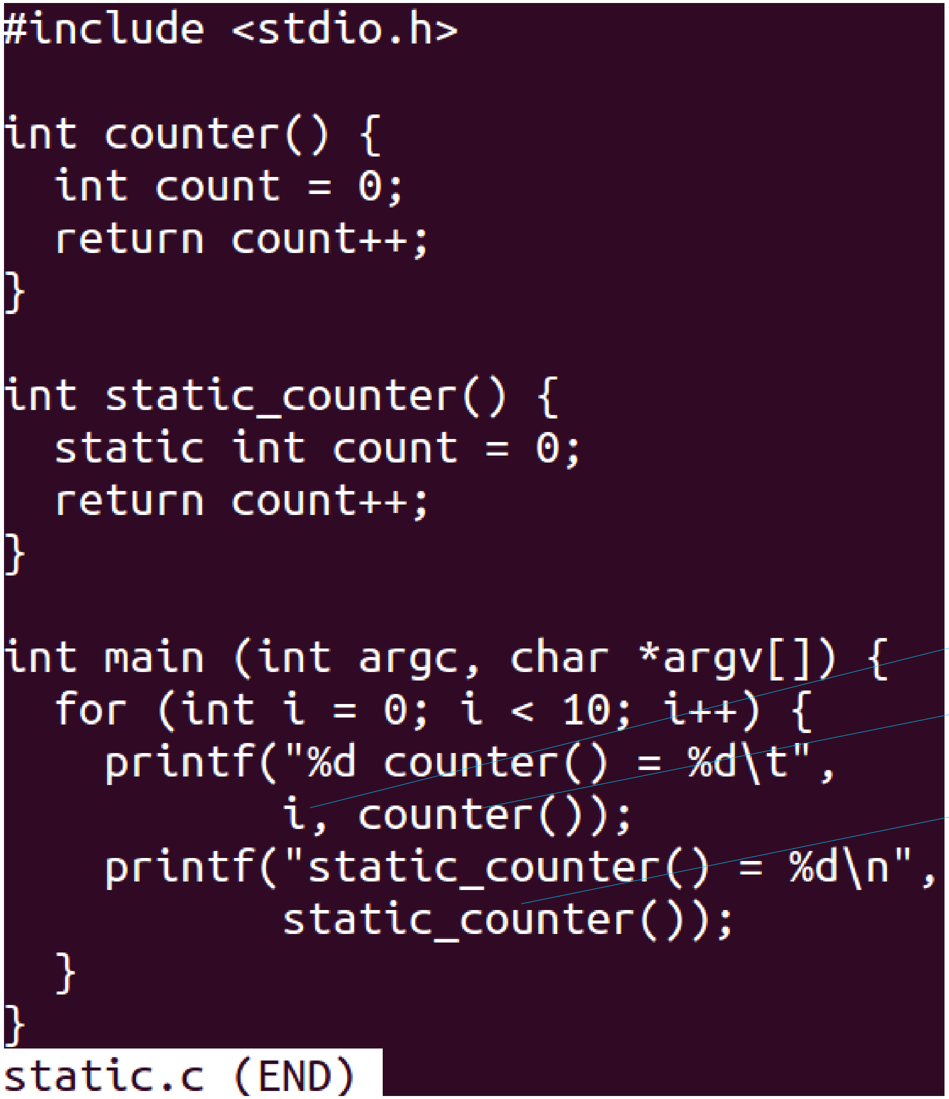
Expressions set function parameters

- E.g., int f(int a, int b); int g(int a);
  - f(1+2, 3*4)
  - f(g(1+2), 5)
  - f(g(1+2), g(3*4)
  - g(f(1,2))

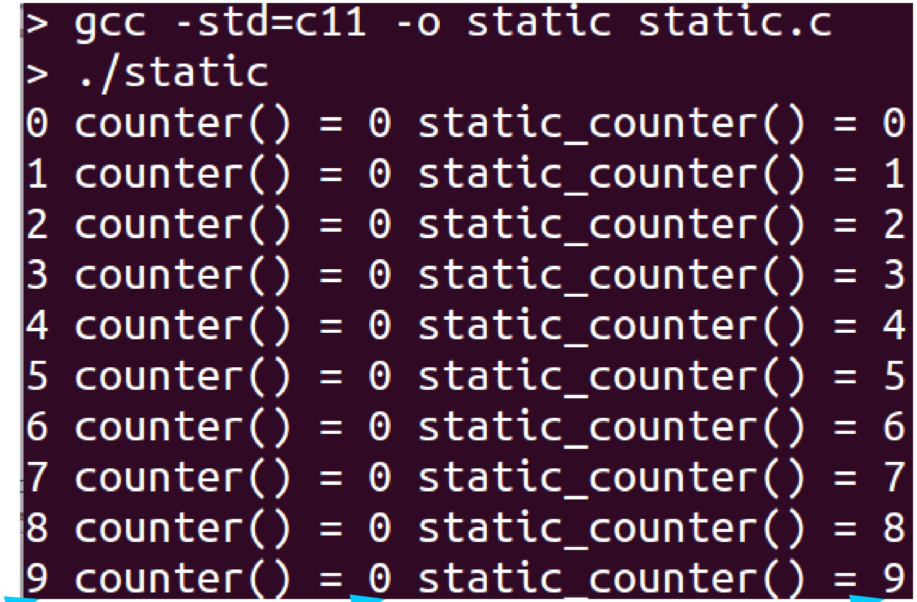
Variable lifetime

- Local variables disappear when the function returns.
- The keyword static gives the variable the same lifetime as a global variable
  - Can return strings.
  - Can share information between functions calls.

Static local variables (1)



```
#include <stdio.h>

int counter() {
  int count = 0;
  return count++;
}

int static_counter() {
  static int count = 0;
  return count++;
}

int main (int argc, char *argv[]) {
  for (int i = 0; i < 10; i++) {
    printf("%d counter() = %d\t",
           i, counter());
    printf("static_counter() = %d\n",
           static_counter());
  }
}
static.c (END)
```



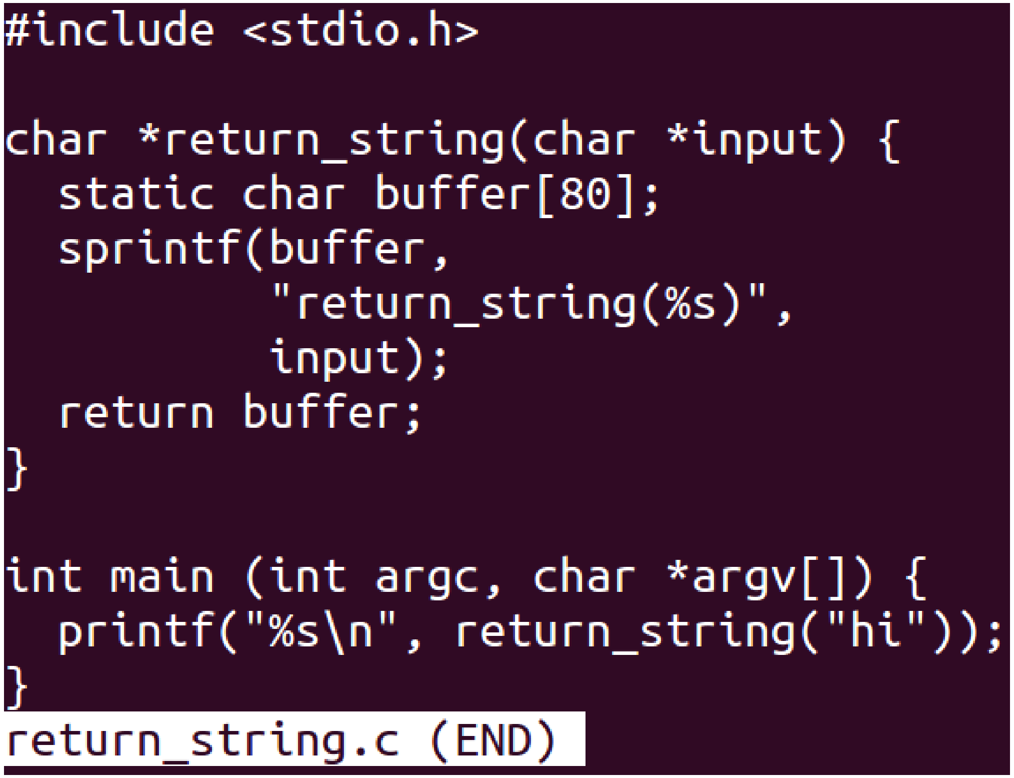
```
> gcc -std=c11 -o static static.c
> ./static
0 counter() = 0 static_counter() = 0
1 counter() = 0 static_counter() = 1
2 counter() = 0 static_counter() = 2
3 counter() = 0 static_counter() = 3
4 counter() = 0 static_counter() = 4
5 counter() = 0 static_counter() = 5
6 counter() = 0 static_counter() = 6
7 counter() = 0 static_counter() = 7
8 counter() = 0 static_counter() = 8
9 counter() = 0 static_counter() = 9
```

Static local variables (2)

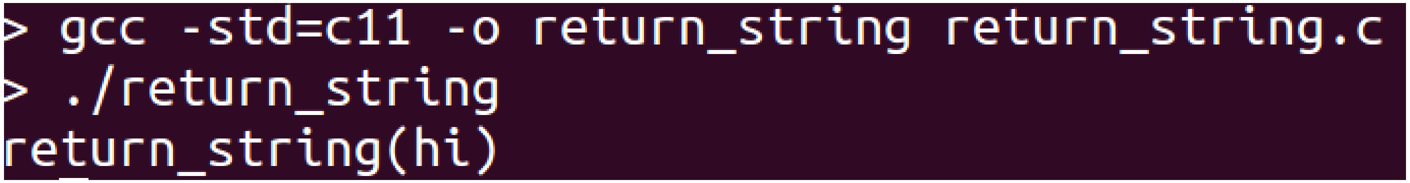
We need to have the memory allocated before we can give it back to a calling function.

By declaring the local variable static, the variable continues to live after the function is gone.

The variable is inaccessible from main.

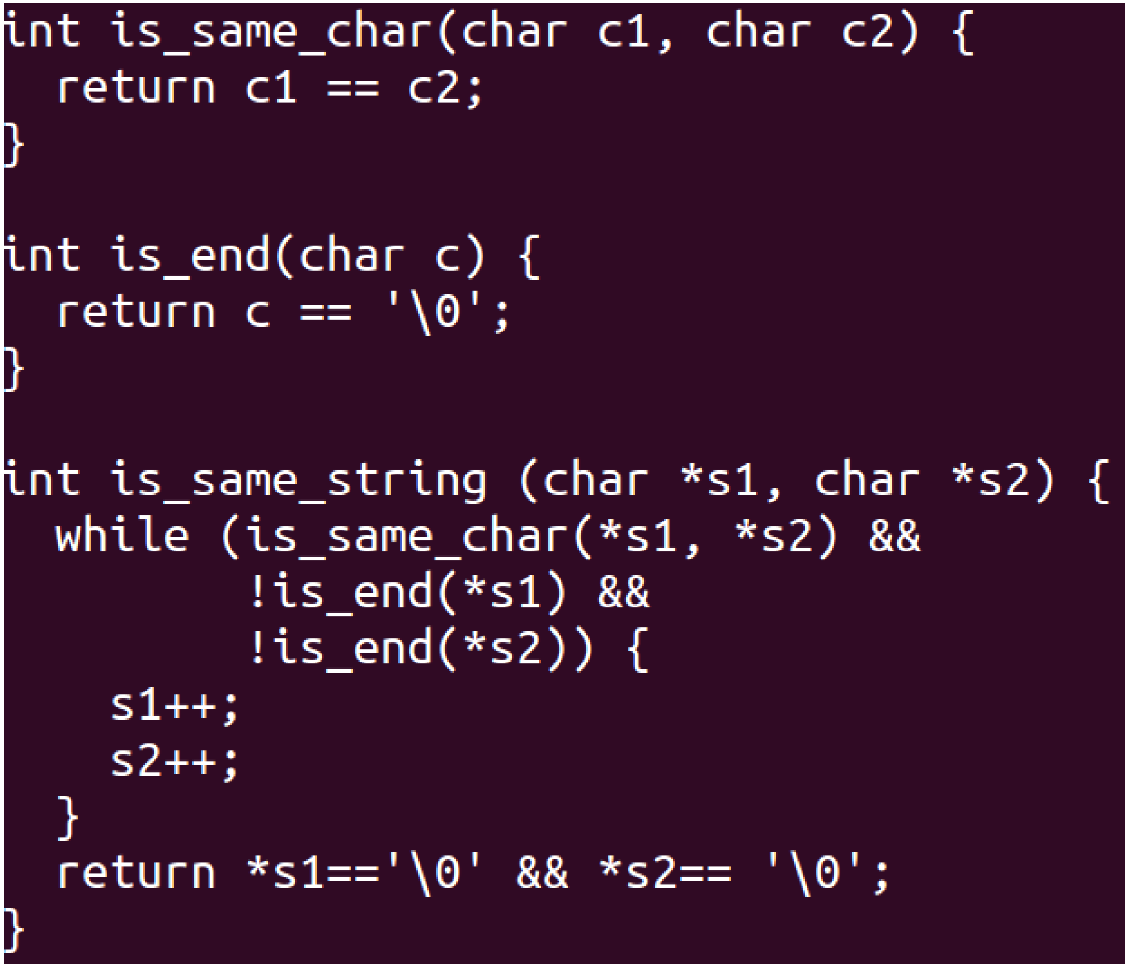


```
#include <stdio.h>

char *return_string(char *input) {
  static char buffer[80];
  sprintf(buffer,
          "return_string(%s)",
          input);
  return buffer;
}

int main (int argc, char *argv[]) {
  printf("%s\n", return_string("hi"));
}
return_string.c (END)
```



```
> gcc -std=c11 -o return_string return_string.c
> ./return_string
return_string(hi)
```

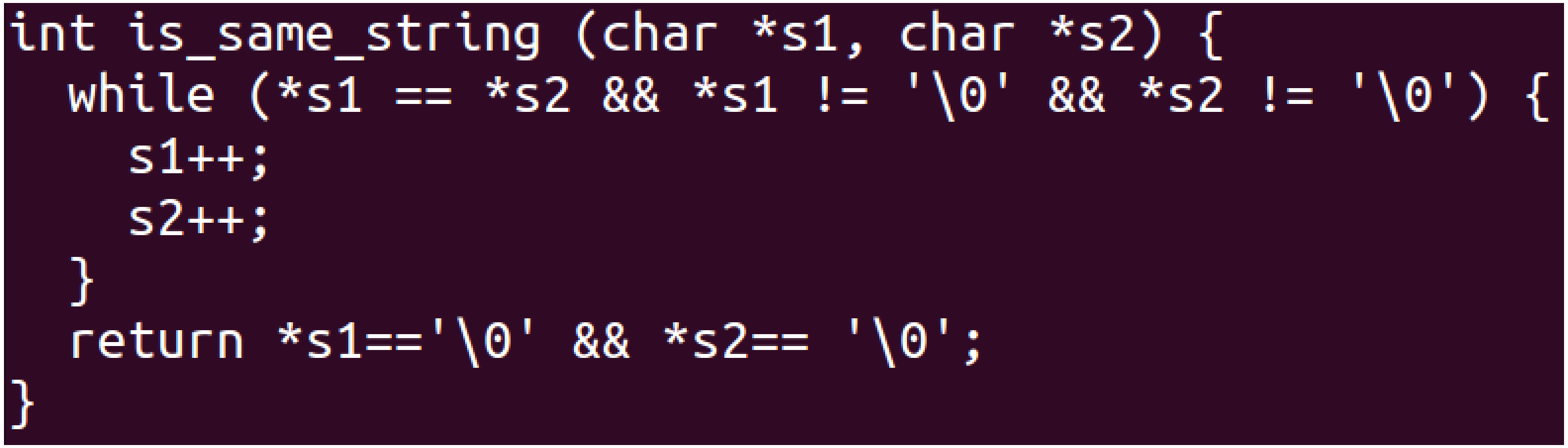
Important: good names == clarity



```
int is_same_char(char c1, char c2) {
  return c1 == c2;
}

int is_end(char c) {
  return c == '\0';
}

int is_same_string (char *s1, char *s2) {
  while (is_same_char(*s1, *s2) &&
         !is_end(*s1) &&
         !is_end(*s2)) {
    s1++;
    s2++;
  }
  return *s1=='\0' && *s2== '\0';
}
```

Clarity changes

- This program is clearer to an experienced programmer.
  - It has less code and it easy to see that it does what it says.



```
int is_same_string (char *s1, char *s2) {
  while (*s1 == *s2 && *s1 != '\0' && *s2 != '\0') {
    s1++;
    s2++;
  }
  return *s1=='\0' && *s2== '\0';
}
```

Read Programs

- You learn to write English by reading English
- You learn to write Hindi by reading Hindi
- You learn to write C by reading C.
  - You learn to write good C by reading good C.
  - Look at Kernigan and Richie
  - Look at the Linux kernel code