Week 5 Lecture 2

Functions and Recursion

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 `int`s.
    - 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

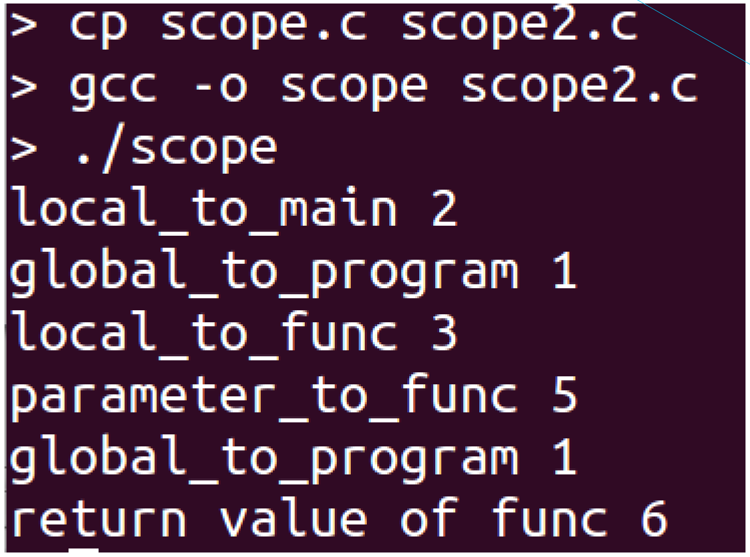
- <name>(<parameters>);
  - E.g., `add(2, 3)`;
  - Executes the body of the function.
  - Compiler recognized a function call because it has parentheses, but no type.
- A function call is an expression whose value is the return value.

Scope and Lifetime

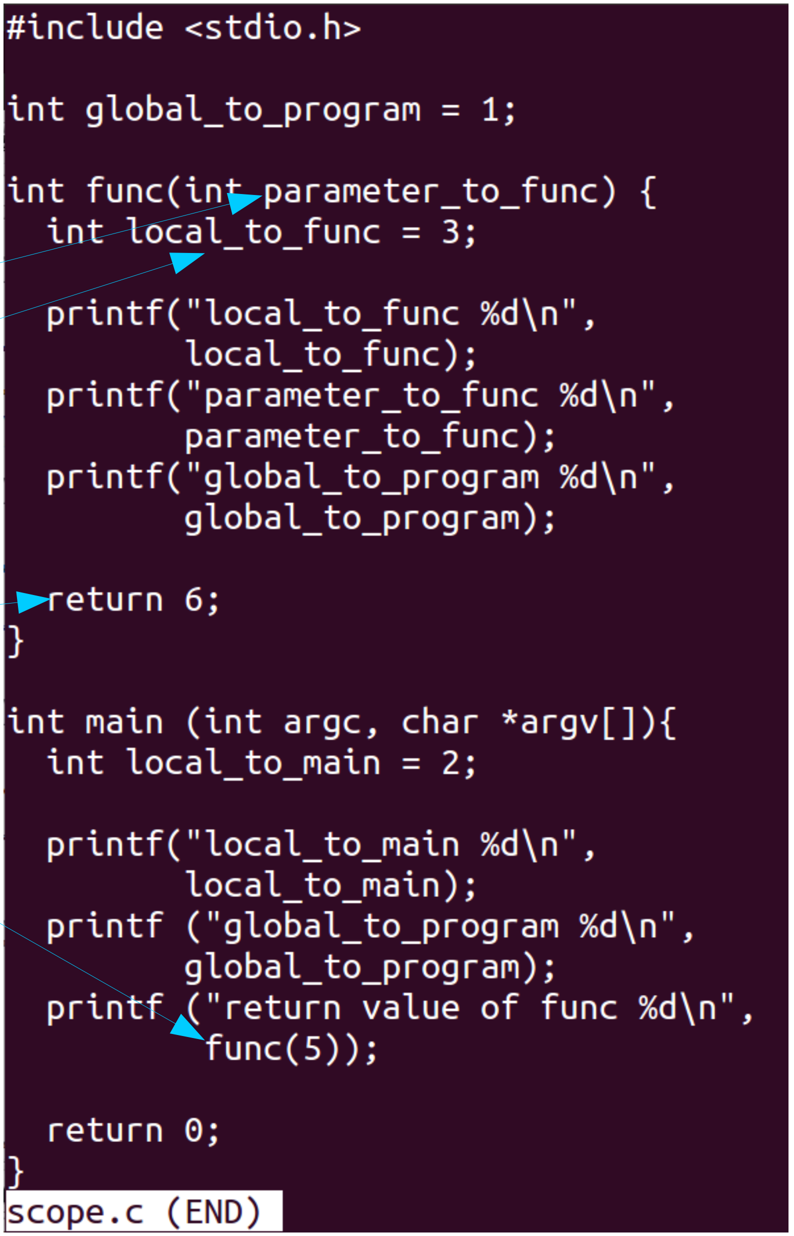
- Scope: defines who can use a variable
  - Local variables can only be used inside a function
  - Global variables can be used anywhere
- Lifetime: defines when a variable exists
  - Local variables exist only while the function is running (by default)
  - Global variables exist as long as the program runs

Scope example

- 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

- Functions 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

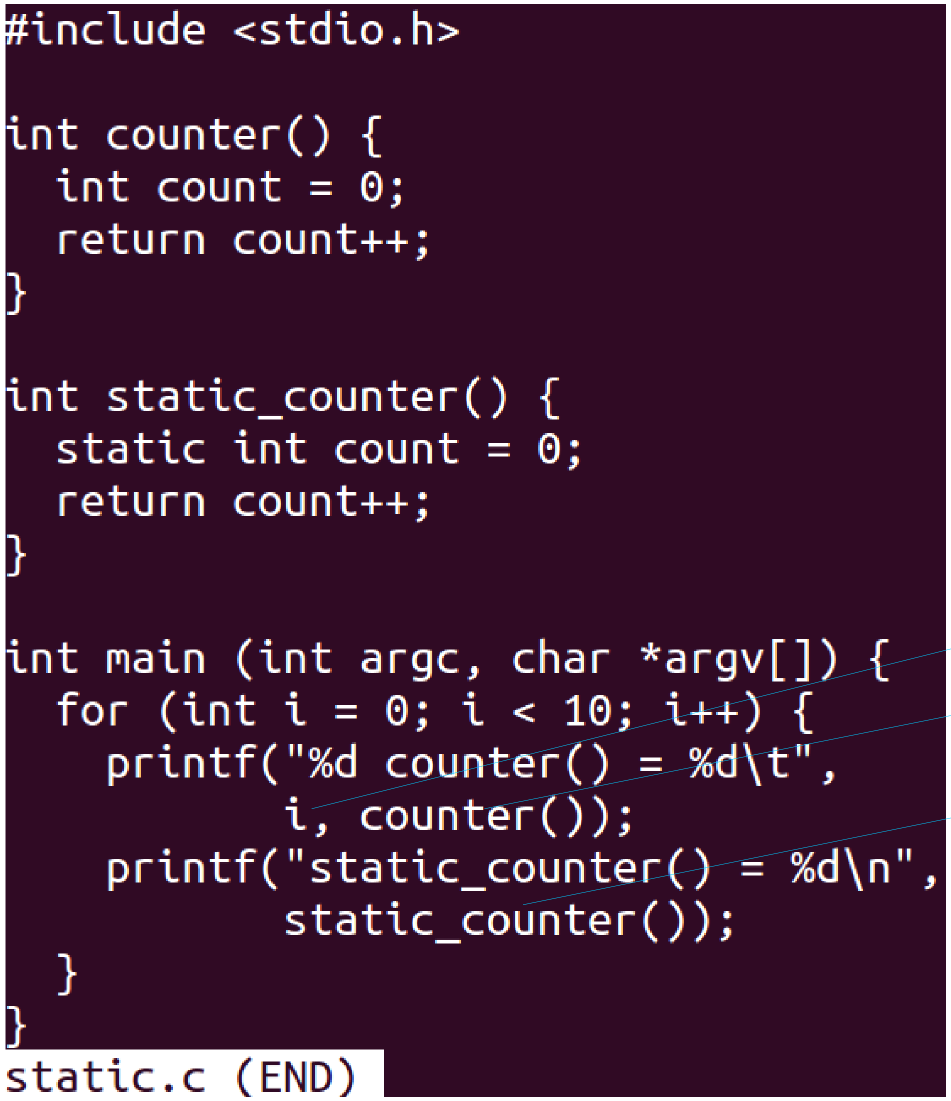
Expressions set function parameters

- 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))

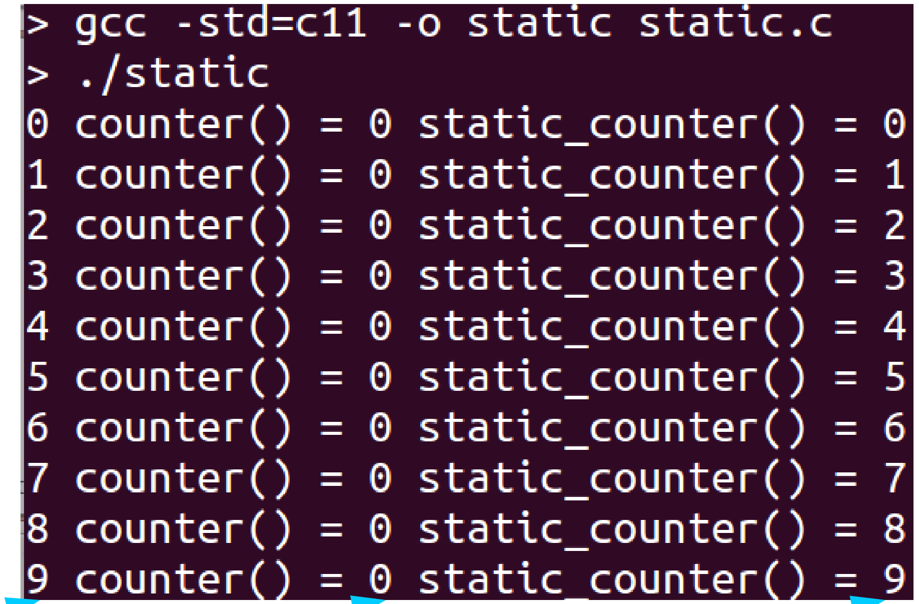
Variable lifetime

- 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)
```



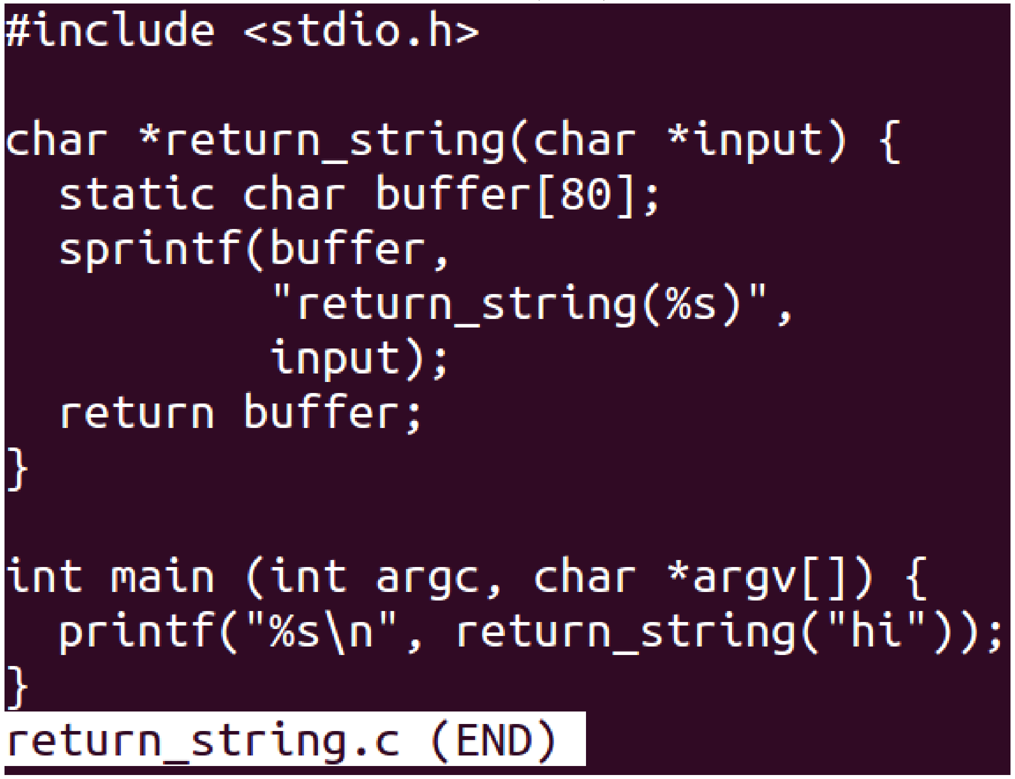
```
> gcc -std=c11 -o static static.c
> ./static
0 counter() = 0 static_counter() = 0
1 counter() = 0 static_counter() = 1
2 counter() = 0 static_counter() = 2
3 counter() = 0 static_counter() = 3
4 counter() = 0 static_counter() = 4
5 counter() = 0 static_counter() = 5
6 counter() = 0 static_counter()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

Static local variables (2)

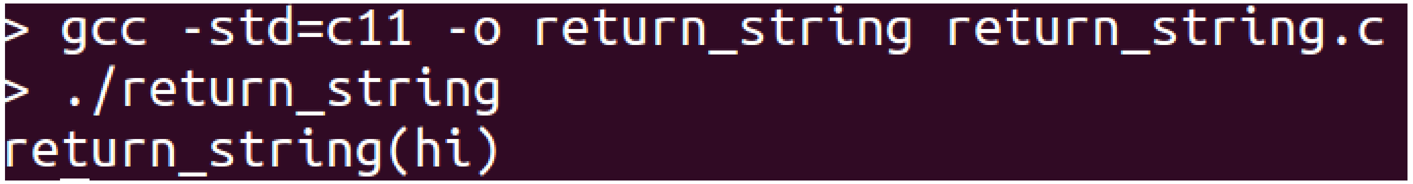
We need to have the memory allocated before we can give it back to a calling function.

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

The variable is inaccessible from main.

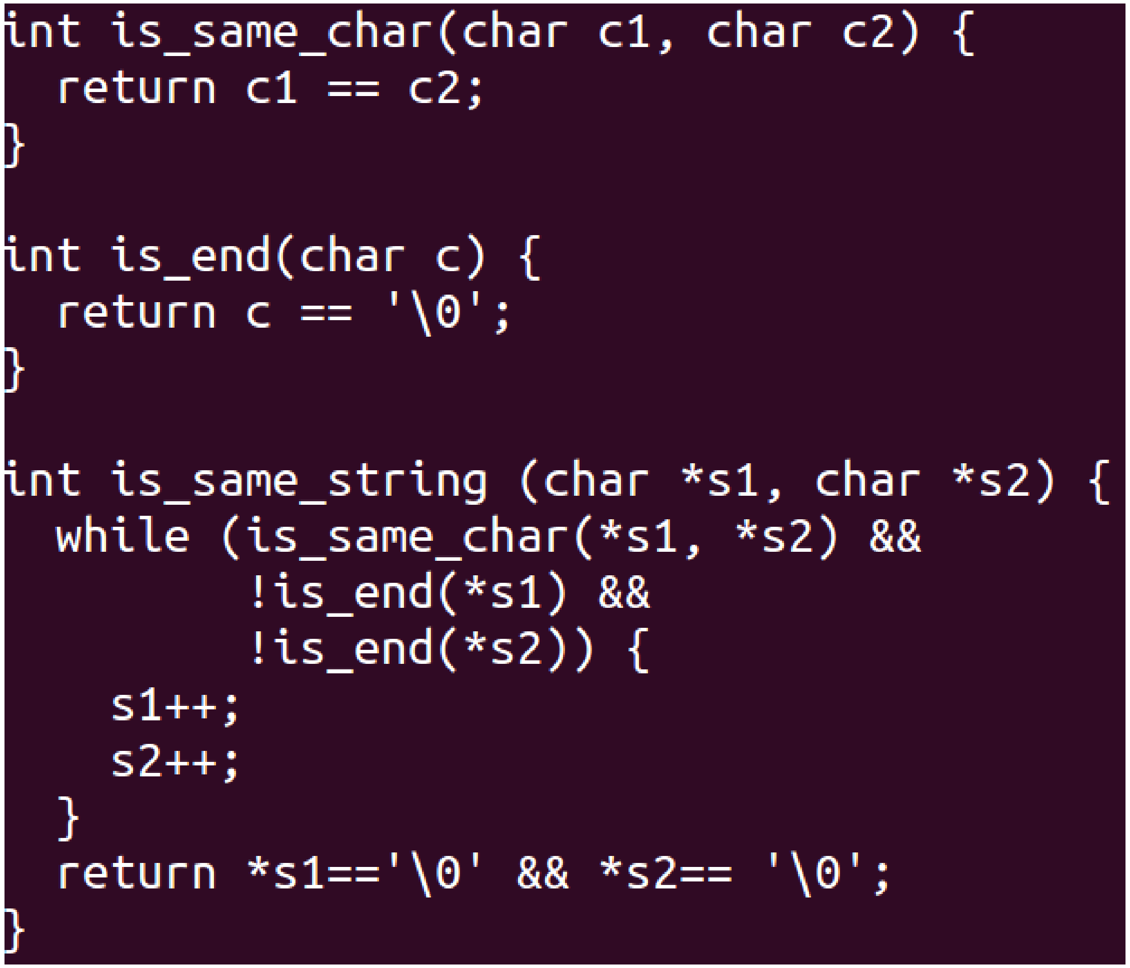


```
#include <stdio.h>

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)
```

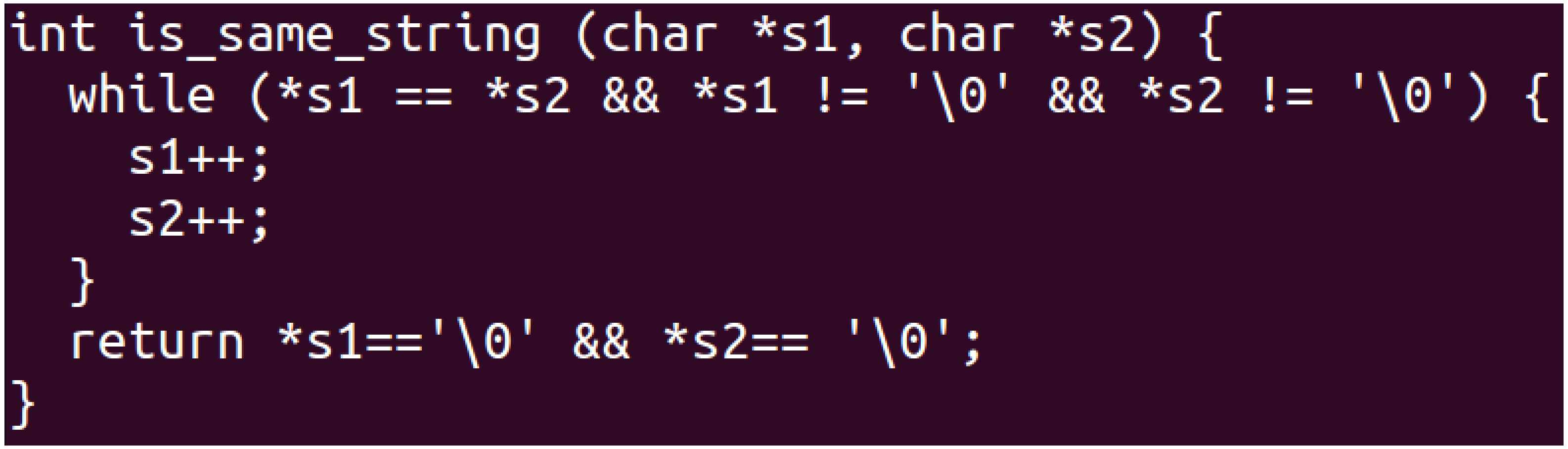
Important: good names == clarity



```
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';
}
```

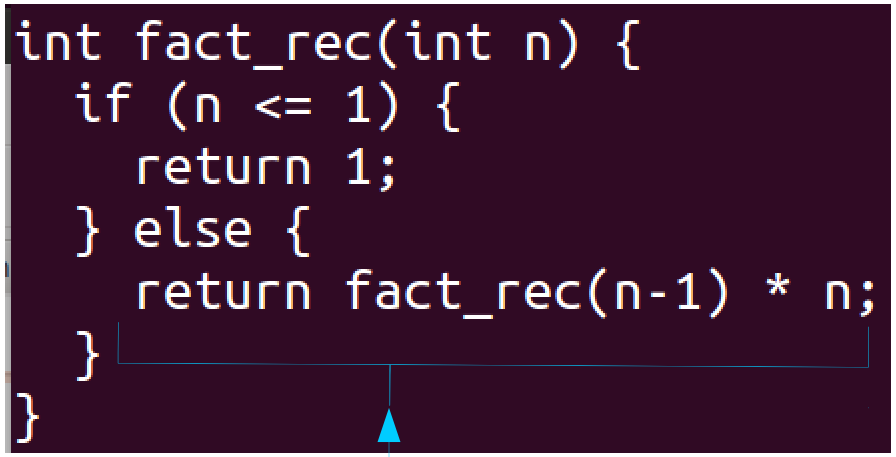
Read Programs

- 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

Recursion

- A function can use itself.
- Mathematical expression
  - n! = 1 * 2 * 3 … * n = (n – 1)! * n
    - By associativity
- Same fact in C
  - factorial(n) = n * factorial(n – 1);

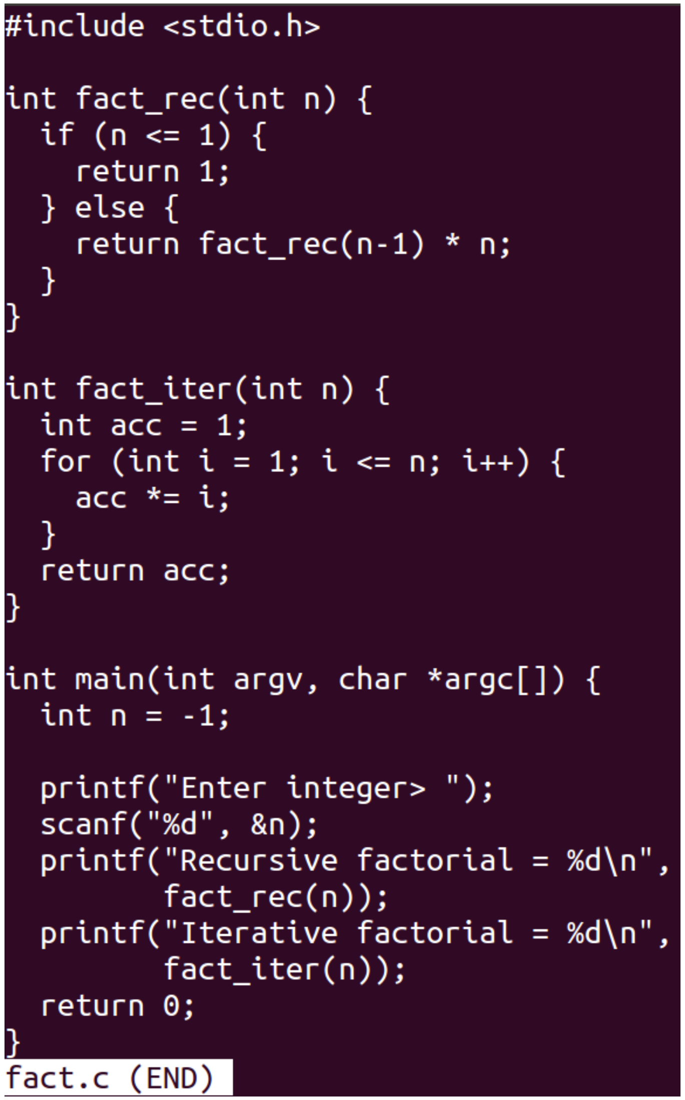
Using the fact in c



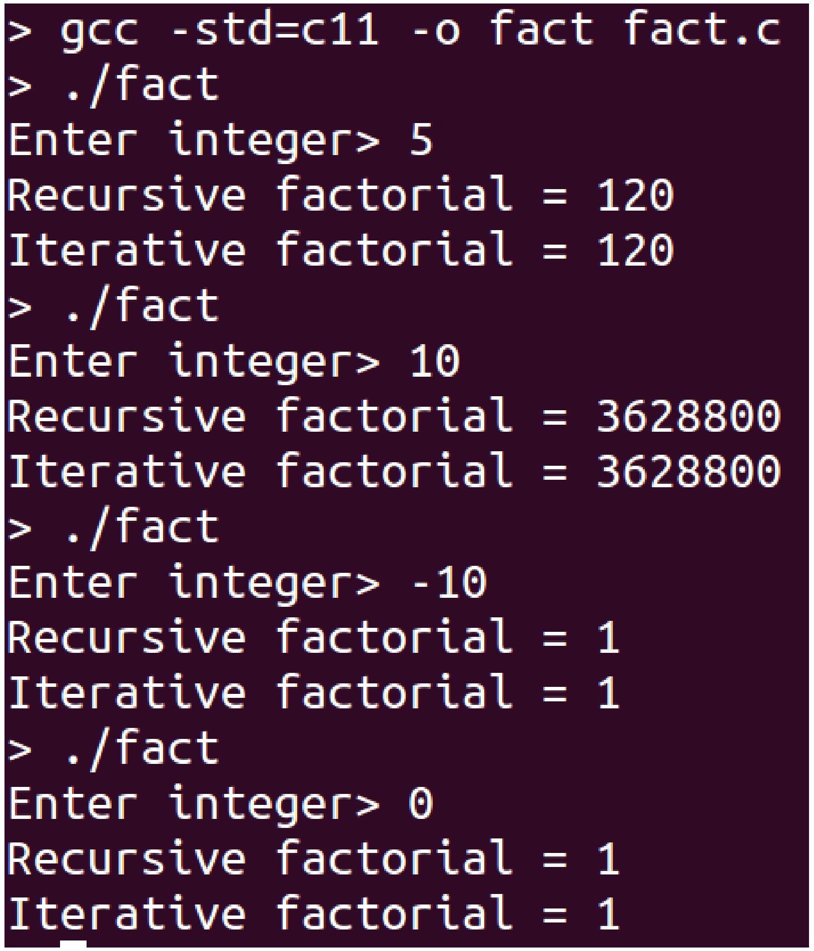
```
int fact_rec(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return fact_rec(n-1) * n;
  }
}
```

- factorial(n) == factorial(n-1) * n

Recursion v Iteration



```
#include <stdio.h>

int fact_rec(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return fact_rec(n-1) * n;
  }
}

int fact_iter(int n) {
  int acc = 1;
  for (int i = 1; i <= n; i++) {
    acc *= i;
  }
  return acc;
}

int main(int argv, char *argc[]) {
  int n = -1;

  printf("Enter integer> ");
  scanf("%d", &n);
  printf("Recursive factorial = %d\n",
         fact_rec(n));
  printf("Iterative factorial = %d\n",
         fact_iter(n));
  return 0;
}
fact.c (END)
```



```
> gcc -std=c11 -o fact fact.c
> ./fact
Enter integer> 5
Recursive factorial = 120
Iterative factorial = 120
> ./fact
Enter integer> 10
Recursive factorial = 3628800
Iterative factorial = 3628800
> ./fact
Enter integer> -10
Recursive factorial = 1
Iterative factorial = 1
> ./fact
Enter integer> 0
Recursive factorial = 1
Iterative factorial = 1
```

Why do we care

- It gives us a different way to reason about programs
  - What is the base case: fact(1) = 1
  - How do we reduce the size of the problem: fact(n) = fact(n-1) * n
- Here similar to iteration
  - Reducing from end instead of beginning.
- Efficient algorithms often result from reducing the size from the middle. i.e., Divide and reconquer.
  - This is much harder iteratively

Divide and Conquer

- Factorial requires a step for each number from one to n
  - It takes n steps
- If we can divide it in half, each step covers half the distance
  - Takes log n steps

Aside: Comments

- Comments do nothing active
- They are important because they allow you to keep notes in a program.
  - They can be invaluable in clarifying a program
- Two types of comments in C
  - /* comment */
    - May be multi-line
  - // comment
    - Extends only to end of line

Value of comments

- Because high level languages are written for humans, text that does nothing is very helpful
- Care must be taken with comments.
  - If they provide no useful information, the clutter the code.
  - Because they do nothing, then can mislead
  - They must be updated whenever the code is updated
- Keeping active elements, such as variables and functions informative can be more useful than comments

Value of comments

- Comments are useful when we have a particularly tricky piece of code that cannot be clarified by the function calls.
- They can also be useful to capture assumptions made when doing a function.