UES103

Programming for Problem Solving

Saif Nalband, PhD

1A27-31/1A42-46

Lab Password: cslab768

Functions: Declaration, Definition, Call and return, Scope of variables, Storage classes, Recursive functions, Recursion vs Iteration.

Functions

- Function
  - A program segment that carries out a specific, well-defined task.
  - Examples
    - A function to find the gcd of two numbers
    - A function to find the largest of n numbers

- A function will carry out its intended task whenever it is called
  - Functions may call other functions (or itself)
  - A function may be called multiple times (with different arguments)

- Every C program consists of one or more functions.
    - One of these functions must be called “main”.
    - Execution of the program always begins by carrying out the instructions in “main”.

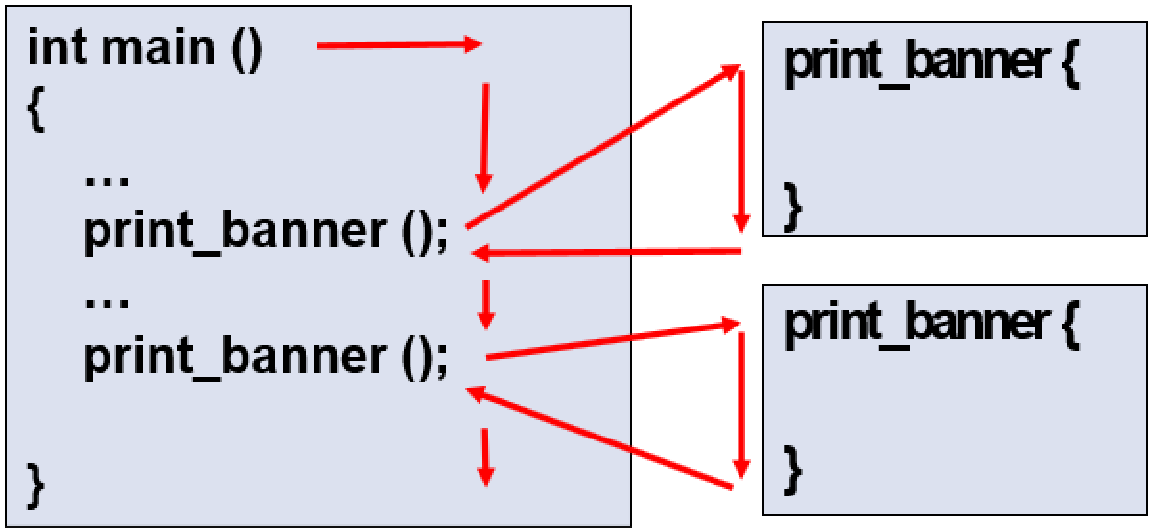
Code

```
void print_banner ( )
{
    printf("********\n");
}
```

```
int main ( )
{
      . . .
      print_banner ( ) ;
      . . .
      print_banner ( ) ;

}
```

Execution



If function A calls function B:

A : calling function / caller function

B : called function

Why Functions?

- Functions allow one to develop a program in a modular fashion.
  - Codes become readable
  - Codes become manageable to debug and maintain
- Write your own functions to avoid writing the same code segments multiple times
  - If you check several integers for primality in various places of your code, just write a single primality-testing function, and call it on all occasions
- Use existing functions as building blocks for new programs
  - Use functions without rewriting them yourself every time it is needed
  - These functions may be written by you or by others (like sqrt(), printf())
- Abstraction: Hide internal details (library functions)

Use of functions: *Area of a circle*

```c
#include <stdio.h>

/* Function to compute the area of a circle */
float   myfunc (float r)
{    float   a;
     a = 3.14159 * r * r;
     return a;          /* return result */
}

main()
{
    float   radius, area;

    scanf ("%f", &radius);
    area = myfunc (radius);
    printf ("\n Area is %f \n", area);
}
```

Function definition

Function argument

Function call

Use of functions: *Area of a circle*

```
#include <stdio.h>

/* Function to compute the area of a circle */
float   myfunc (float r)
{     float   a;
      a = 3.14159 * r * r;
      return a;          /* retur
}

main()
{
     float   radius, area;

     scanf ("%f", &radius);
     area = myfunc (radius);
     printf ("\n Area is %f \n", area);
}
```

Function definition

- A called function processes information that is passed to it from the calling function, and the called function may return a single value (result) to the calling function.
  - Information passed to the function via special identifiers called *arguments* or *parameters*.
  - The value is returned by the the `return` statement.

Function call

Defining a Function

A function definition has two parts:

- The first line
- The body of the function

General syntax:

return-value-type function-name (*parameter-list*)

{

declarations and statements

}

return value type

```
#include <stdio.h>


/* Function to compute the area of a
circle */
float   myfunc (float r)
{
    float   a;
    a = 3.14159 * r * r;
    return a;
}

main()
{
    float   radius, area;

    scanf ("%f", &radius);
    area = myfunc (radius);
    printf ("\n Area is %f \n", area);
}
```

The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in ( ).

- Each argument has an associated type declaration.
- The arguments are called *formal arguments* or *formal parameters*.

Example:

float myfunc (float r)

int gcd (int A, int B)

Calling a function

- Called by specifying the function name and parameters in an instruction in the calling **function**.
- When a function is called from some other function, the corresponding arguments in the function call are called **actual arguments** or **actual parameters**.
- The function call must include a matching actual parameter for each **formal parameter**.
- Position of an actual parameters in the parameter list in the call must match the position of the corresponding formal parameter in the function definition.
- The formal and actual arguments would match in their data types. Mismatches are auto-typecasted if possible.
- The actual parameters can be expressions possibly involving other function calls (like f(g(x)+y)).

```
#include <stdio.h>


/* Function to compute the area of a
circle */
float  myfunc (float r)
{
     float   a;
     a = 3.14159 * r * r;
     return a;
}

main()
{
     float   radius, area;

     scanf (“%f”, &radius);
     area = myfunc (radius);
     printf (“\n Area is %f \n”, area);
}
```

Function Prototypes: declaring a function

- **Usually, a function is defined before it is called.**
  - **main() is usually the last function in the program written.**
  - **Easy for the compiler to identify function definitions in a single scan through the file.**
- **Some prefer to write the functions after main(). There may be functions that call each other.**
  - **Must be some way to tell the compiler what is a function when compilation reaches a function call.**
  - **Function prototypes are used for this purpose**
    - **Only needed if function definition comes after a call to that function.**
- **Function prototypes are usually written at the beginning of a program, ahead of any functions (including main()).**
- **Prototypes must specify the types. Parameter names are optional (ignored by the compiler).**
- **Examples:**
  - **int gcd (int , int);**
  - **void div7 (int number);**
  - **Note the semicolon at the end of the line.**
  - **The parameter name, if specified, can be anything; but it is a good practice to use the same names as in the function definition.**

Example:

Function prototype / declaration

```c
#include <stdio.h>
int sum( int, int );
int main( )
{
    int x, y;
    scanf("%d%d", &x, &y);
    printf("Sum = %d\n", sum(x, y));
}

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

This program needs a function prototype or function declaration since the function call comes before the function definition.

Function call

Function definition

Return value

- A function can return a single value Using return statement
- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the calling function

```
int main( )
{
    int x, y, s;
    scanf("%d%d", &x, &y);
    s = sum(x, y);
}
int sum (int a, int b)
{
    return a + b;
}
```

- Sometimes a function is not meant for returning anything
- Such functions are of type void

Example: A function which prints if a number is divisible by 7 or not.

```
void  div7 (int n)
{
  if  ((n % 7) == 0)
     printf ("%d divisible by 7", n);
  else
     printf ("%d not divisible by 7", n);
  return;
}
```

- The return type is void
- The return statement for void functions is optional at the end

The return statement

In a value-returning function, return does two distinct things:

- Specify the value returned by the execution of the function.
- Terminate the execution of the called function and transfer control back to the caller function.

A function can only return one value.

- The value can be any expression matching the return type.
- It might contain more than one return statement.

In a void function:

- "return" is optional at the end of the function body.
- "return" may also be used to terminate execution of the function explicitly before reaching the end.
- No return value should appear following "return".

```
void compute_and_print_itax ()
{
   float income;
   scanf ("%f", &income);
   if (income < 50000)    {
     printf ("Income tax = Nil\n");
     return;  /* Terminates function execution */
   }
   if (income < 60000)    {
     printf ("Income tax = %f\n", 0.1*(income-50000));
     return;  /* Terminates function execution */
   }
   if (income < 150000)   {
     printf ("Income tax = %f\n",0.2*(income-60000)+1000);
     return ; /* Terminates function execution */
   }
   printf ("Income tax = %f\n",0.3*(income-150000)+19000);
}
```

Another Example: What is happening here?

```
int main()
{
  int numb, flag, j=3;
  scanf("%d",&numb);
  while (j <= numb)  {
     flag = prime(j);
     if (flag == 0)
       printf( "%d is prime\n", j );
       j++;
     }
  return 0;
}
```

```
int prime (int x)
{
   int i, test;
   i=2, test =0;
   while ((i <= sqrt(x)) && (test ==0))
   {
       if (x%i==0) test = 1;
       i++;
   }
   return test;
}
```

Tracking the flow of control

```
int main()
{
  int numb, flag, j=3;
  scanf("%d",&numb);
  printf("numb = %d \n",numb);
  while (j <= numb)
  {
    printf("\nMain, j = %d\n",j);
    flag = prime(j);
    printf("Main, flag = %d\n",flag);

    if (flag == 0) printf("%d is prime\n",j);
    j++;
  }
  return 0;
}
```

```
int prime(int x)
{
  int i, test;
  i = 2;  test = 0;

  printf("In function, x = %d \n",x);
  while ((i <= sqrt(x)) && (test == 0))
  {
     if (x%i == 0) test = 1;
     i++;
  }
  printf("Returning, test = %d \n",test);

  return test;
}
```

PROGRAM OUTPUT

```
5
numb = 5

Main, j = 3
In function, x = 3
Returning, test = 0
Main, flag = 0
3 is prime

Main, j = 4
In function, x = 4
Returning, test = 1
Main, flag = 1

Main, j = 5
In function, x = 5
Returning, test = 0
Main, flag = 0
5 is prime
```

Nested Functions

- **A function cannot be defined within another function. It can be called within another function.**
  - **All function definitions must be disjoint.**

- **Nested function calls are allowed.**
  - **A calls B, B calls C, C calls D, etc.**
  - **The function called last will be the first to return.**

- **A function can also call itself, either directly or in a cycle.**
  - **A calls B, B calls C, C calls back A.**
  - **Called recursive call or recursion.**

Example: main( ) calls ncr( ), ncr( ) calls fact( )

```c
#include  <stdio.h>

int ncr (int n, int r);
int fact (int n);

main()
{
   int i, m, n, sum=0;
   scanf ("%d %d", &m, &n);

   for (i=1; i<=m; i+=2)
      sum = sum + ncr(n, i)  ;

   printf ("Result: %d \n", sum);
}
```

```c
int  ncr (int n, int r)
{
   return (fact(n)/fact(r)/fact(n-r));
}


int  fact (int n)
{
      int  i, temp=1;
      for (i=1; i<=n; i++)
         temp *= i;
      return (temp);
}
```

Local variables

- A function can define its own local variables.
- The local variables are known (can be accessed) only within the function in which they are declared.
  - Local variables cease to exist when the function returns.
- Each execution of the function uses a new set of

local variables. Parameters are also local.

```
/* Find the area of a circle with
diameter d */
double circle_area (double d)
{
        double radius, area;
        radius = d/2.0;
        area = 3.14*radius*radius;
        return (area);
}
```

parameter

local variables

Revisiting nCr

```
int  ncr (int n, int r)
{
   return (fact(n)/fact(r)/fact(n-r));
}

int  fact (int n)
{
   int  i, temp=1;
   for (i=1; i<=n; i++)
      temp  *=  i;
      return (temp);
}
```

The n in ncr( )
and the n in
fact( ) are
different

Scope of a variable

- **Part of the program from which the value of the variable can be used (seen).**
- **Scope of a variable - Within the block in which the variable is defined.**
    - **Block = group of statements enclosed within { }**
- **Local variable – scope is usually the function in which it is defined.**
    - **So two local variables of two functions can have the same name, but they are different variables**
- **Global variables – declared outside all functions (even main).**
    - **Scope is entire program by default, but can be hidden in a block if local variable of same name defined**
    - **You are encouraged to avoid global variables.**

What happens here?

Scope of
global A

```
#include  <stdio.h>
int A;    /* This A is a global variable */
void main( )
{
     A = 1;
     myProc( );
    printf ( "A = %d\n", A );
}

void myProc( )
{
     A = 2;

      /* other statements */
     printf ( "A = %d\n", A );
 }
```

A=2

A=2

What Happens?

```
#include  <stdio.h>
int A;    /* This A is a global variable */
void main( )
{
    A = 1;
    myProc( );
    printf ( "A = %d\n", A );
}

void myProc( )
{
    A = 2;

    /* other statements */
    printf ( "A = %d\n", A );
}
```

Scope of global A

A=2

A=2

Local Scope replaces Global Scope

Scope of global A

Scope of local A

```
#include  <stdio.h>
int A;    /* This A is a global variable */
void main( )
{
    A = 1;
   myProc( );
   printf ( "A = %d\n", A );   --> A=1
}

void myProc( )
{
    int A = 2;   /* This A is a local variable */

     /* other statements */
     /* within this function, A refers to the local A */

      printf ( "A = %d\n", A );   --> A=2
 }
```

Parameter Passing

- **When the function is called, the value of the actual parameter is copied to the formal parameter.**

parameter passing

```
int main ()
{       . . .
        double radius,
a;
        . . .
        a =
area(radius);
        . . .
}
```

```
double area (double r)
{
        return (3.14*r*r);
}
```

Parameter Passing by Value in C

- Used when invoking functions

- Call by value / parameter passing by value
  - Called function gets a copy of the value of the actual argument passed to the function.
  - Execution of the function does not change the actual arguments.
    - All changes to a parameter done inside the function are done on the copy.
    - The copy is removed when the control returns to the caller function.
    - The value of the actual parameter in the caller function is not affected.
  - The arguments passed may very well be expressions (example: fact(n-r)).

- Call by reference
  - Passes the address *of* the original argument to a called function.
  - Execution of the function may affect the original argument in the calling function.
  - Not directly supported in C, but supported in some other languages like C++.
  - In C, you can pass *copies* of addresses to get the desired effect.

Parameter passing and return: 1

```
int main()
{
    int  a=10, b;
    printf  ("Initially a = %d\n", a);
    b = change (a);
    printf  ("a = %d, b = %d\n", a, b);
    return 0;
}

int  change  (int x)
{
   printf  ("Before x = %d\n",x);
   x = x / 2;
   printf  ("After x = %d\n", x);
   return (x);
}
```

Output

```
Initially a = 10
Before x = 10
After x = 5
a = 10, b = 5
```

Parameter passing and return: 2

```
int main()
{
    int  x=10, b;
    printf  ("M: Initially x = %d\n", x);
    b = change (x);
    printf  ("M: x = %d, b = %d\n", x, b);
    return 0;
}

int  change  (int x)
{
   printf  ("F: Before x = %d\n",x);
   x = x / 2;
   printf  ("F: After x = %d\n", x);
   return (x);
}
```

Output

M: Initially x = 10

F: Before x = 10

F: After x = 5

M: x = 10, b = 5

Parameter passing and return: 3

```
int main()
{
    int  x=10, y=5;
    printf  ("M1:  x = %d, y = %d\n", x, y);
    interchange (x, y);
    printf  ("M2:  x = %d, y = %d\n", x, y);
    return 0;
}

void interchange  (int x, int y)
{
    int temp;
    printf  ("F1:  x = %d, y = %d\n", x, y);
    temp= x; x = y; y = temp;
    printf  ("F2:  x = %d, y = %d\n", x, y);
}
```

Output

M1: x = 10, y = 5

F1: x = 10, y = 5

F2: x = 5, y = 10

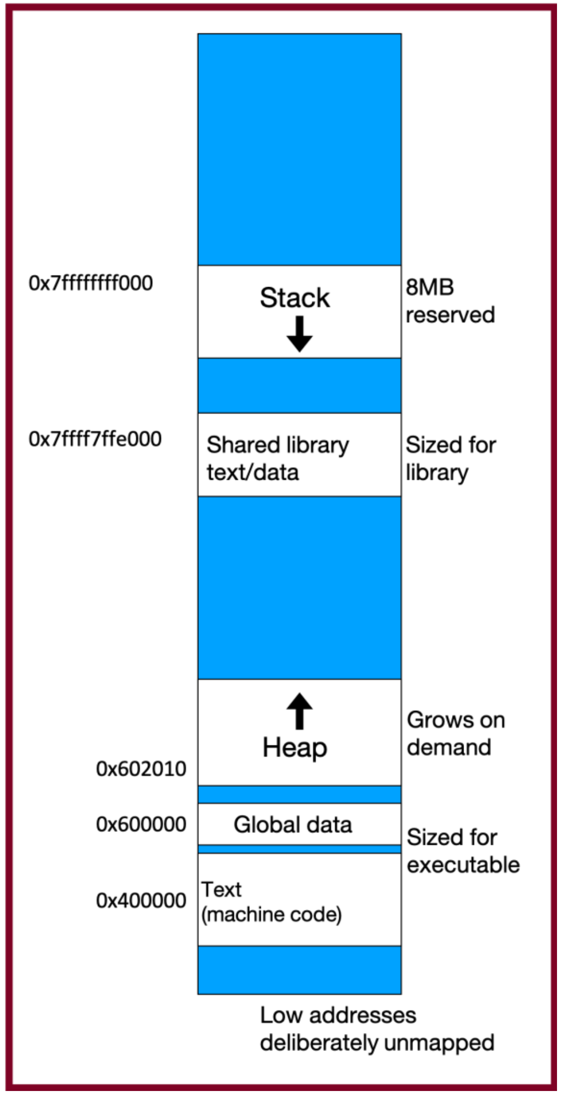
M2: x = 10, y = 5

How do we write an interchange function?
(will see later)

Recursion

Memory Layout

- We are going to dive deeper into different areas of memory used by our programs.
- The **stack** is the place where all local variables and parameters live for each function. A function's stack "frame" goes away when the function returns.
- The stack grows **downwards** when a new function is called and shrinks **upwards** when the function is finished.

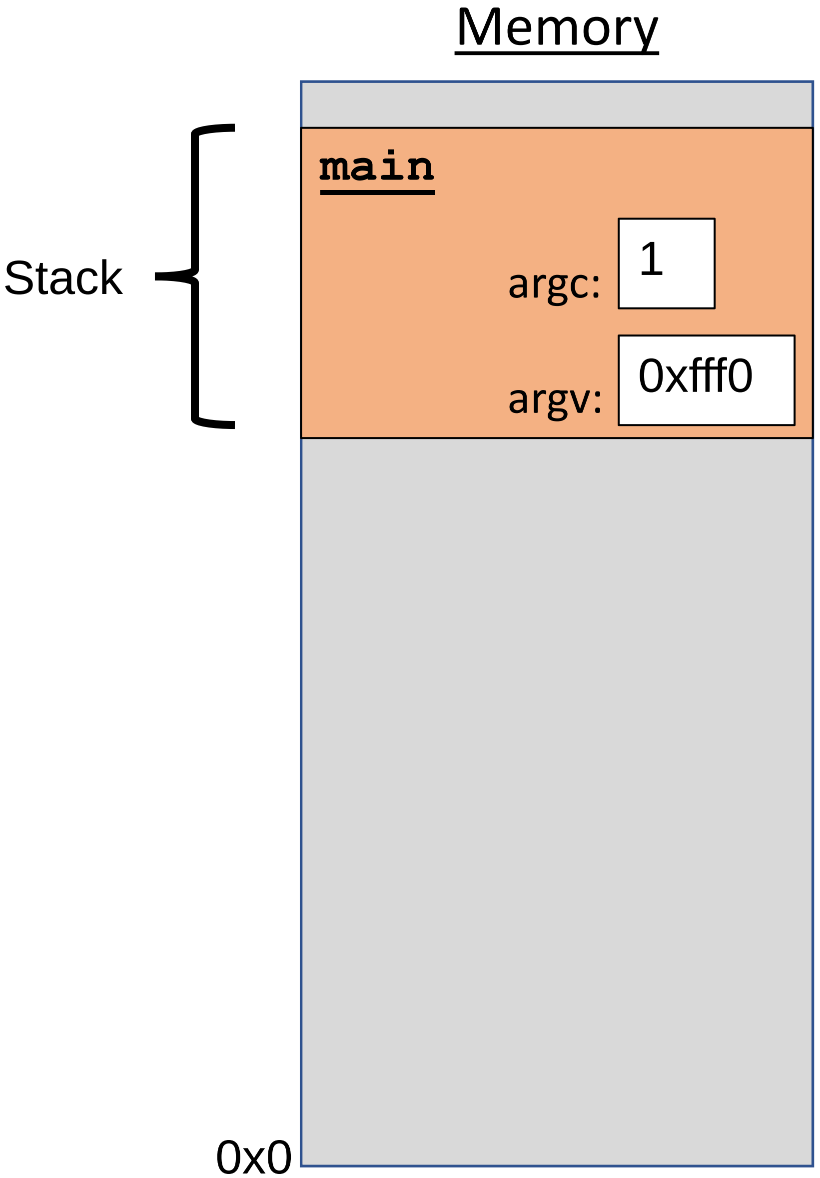


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

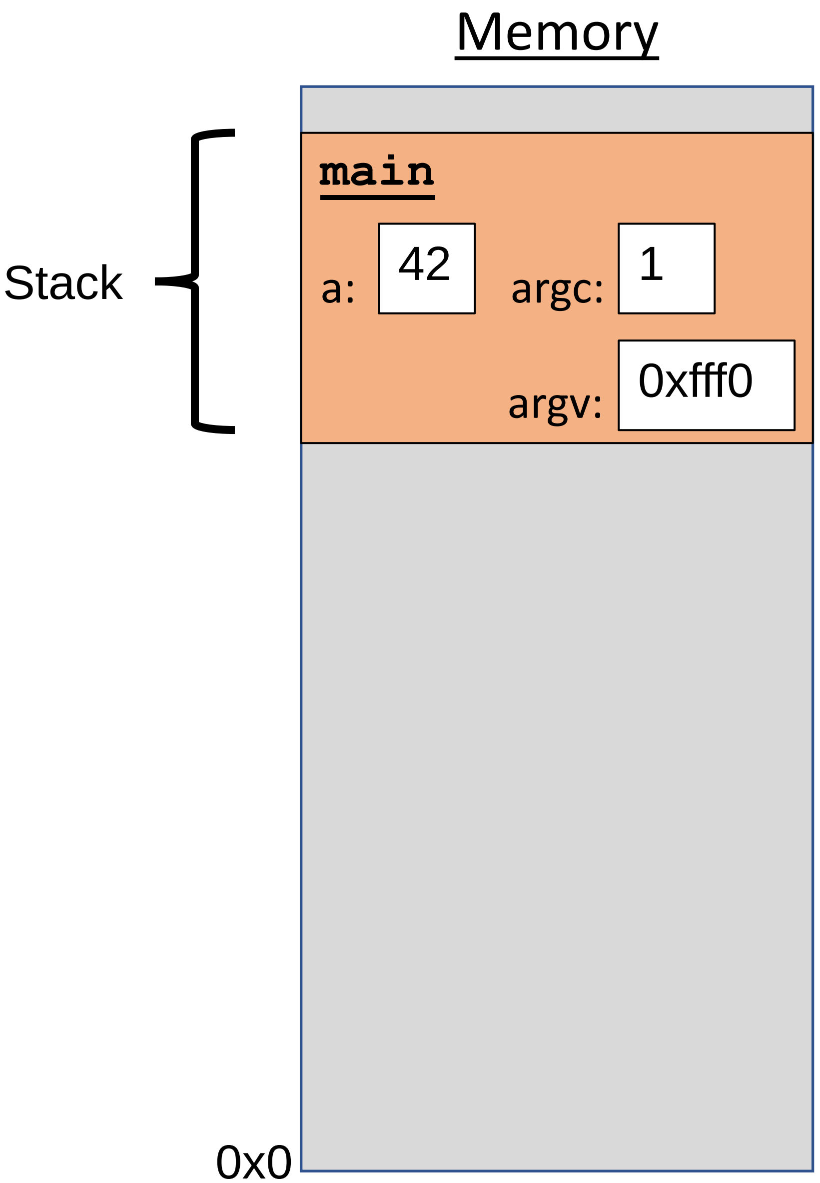


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

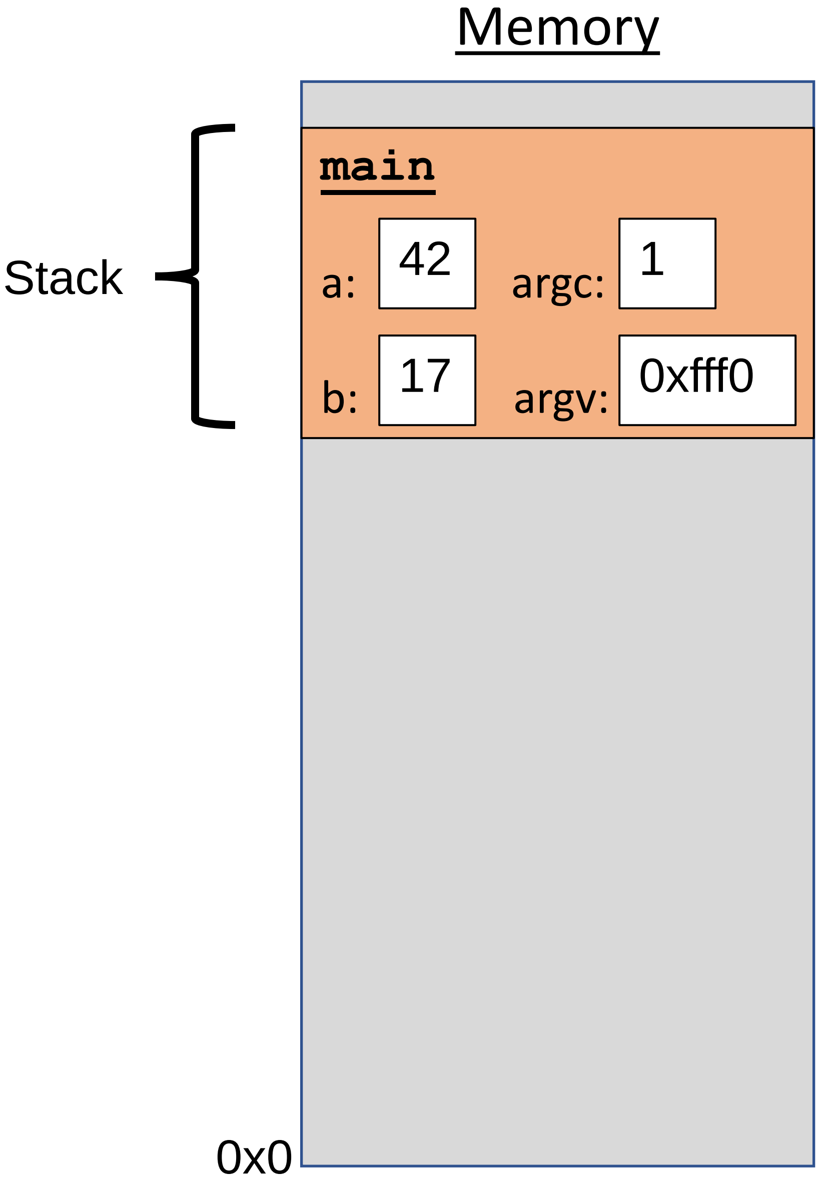


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

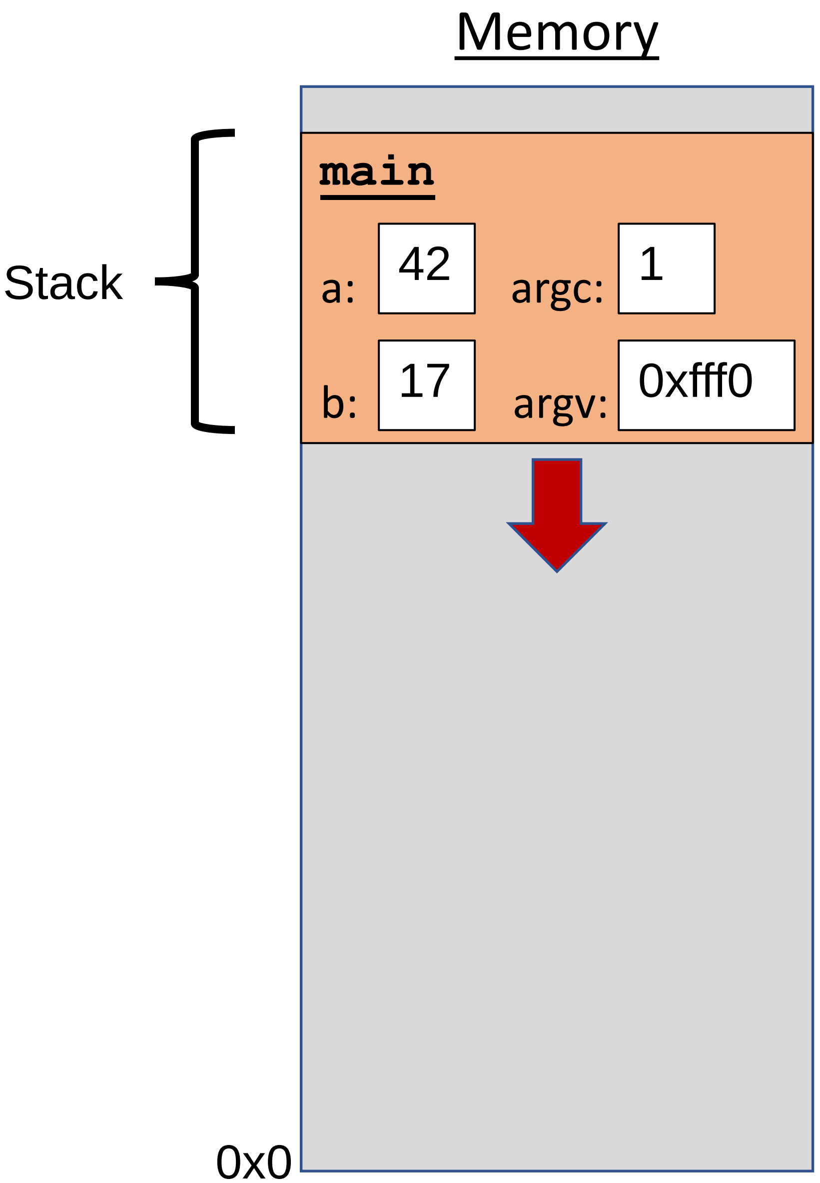


The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

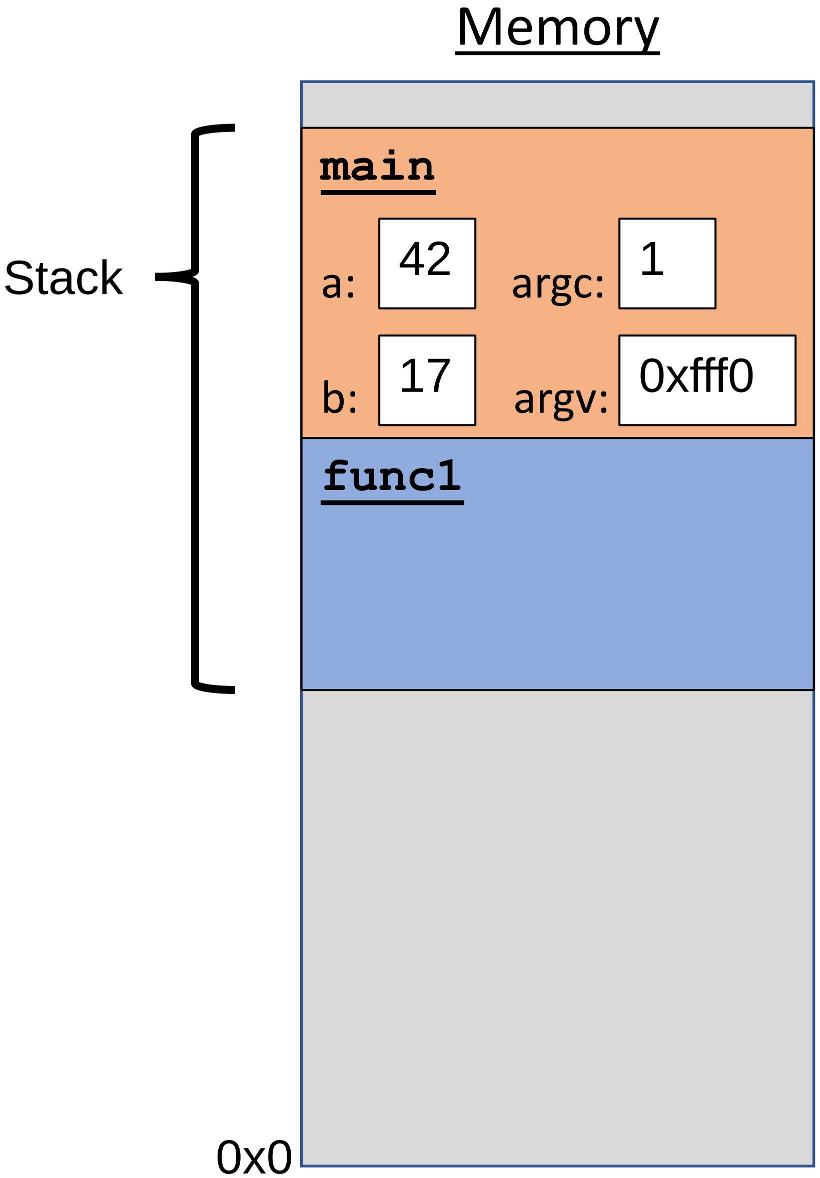


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

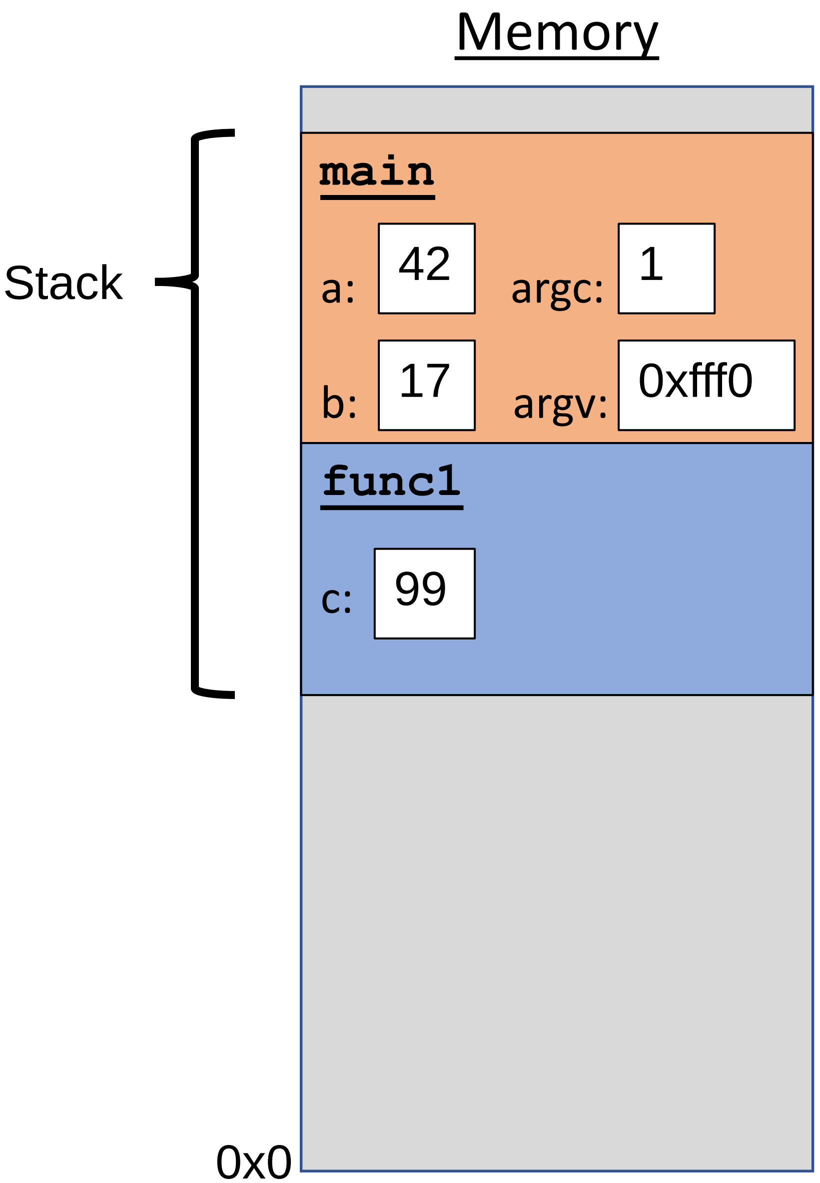


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

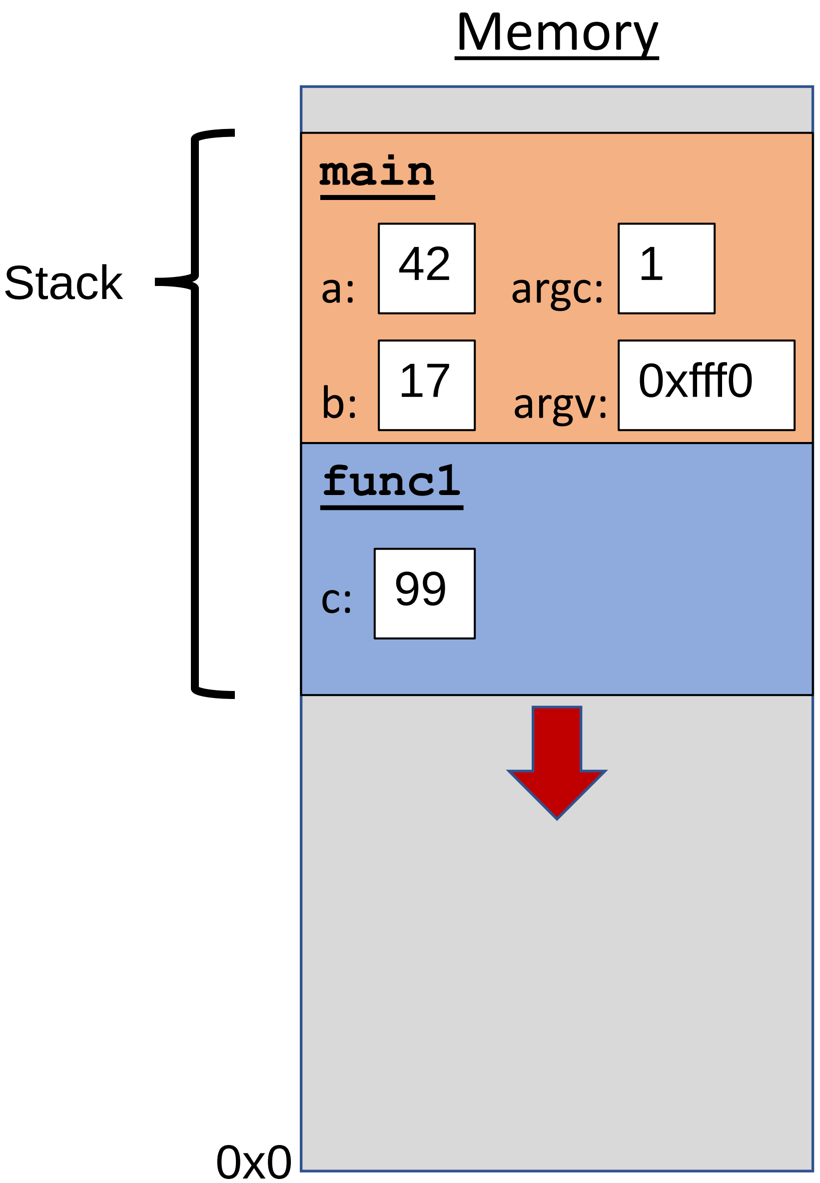


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

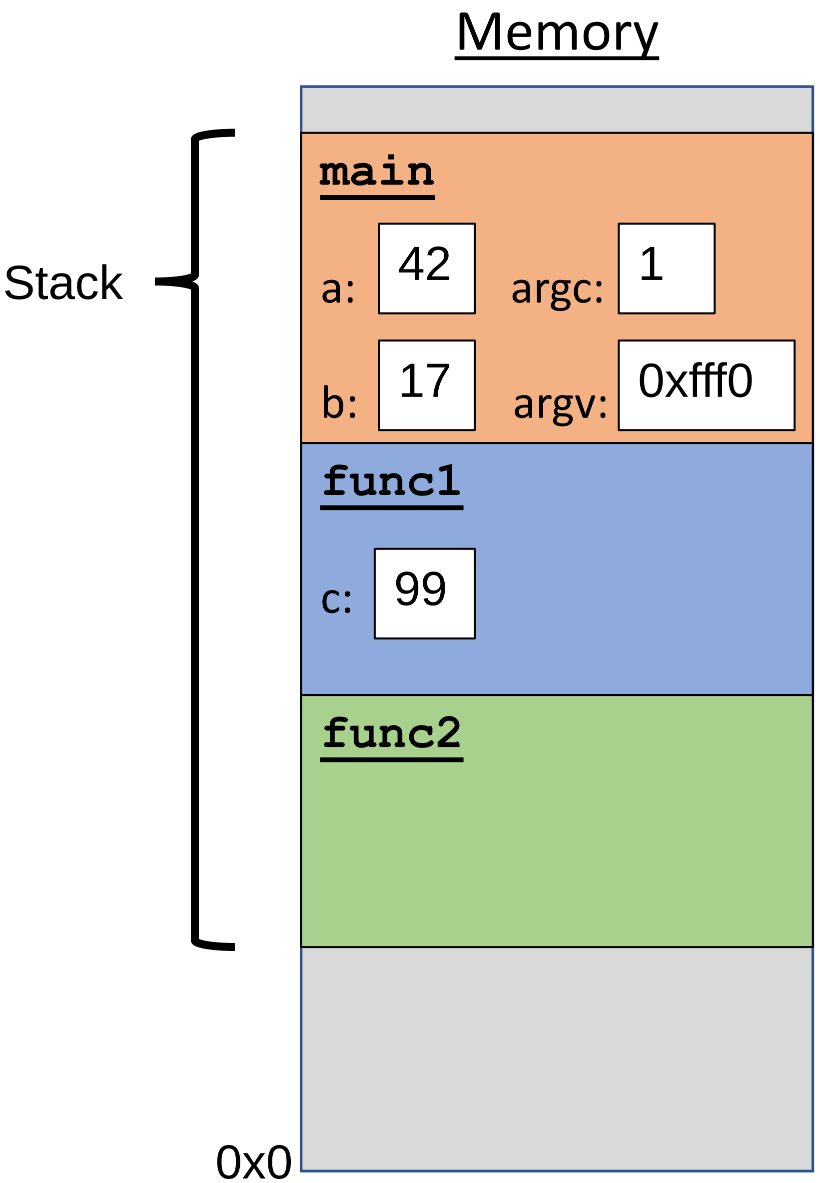


The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

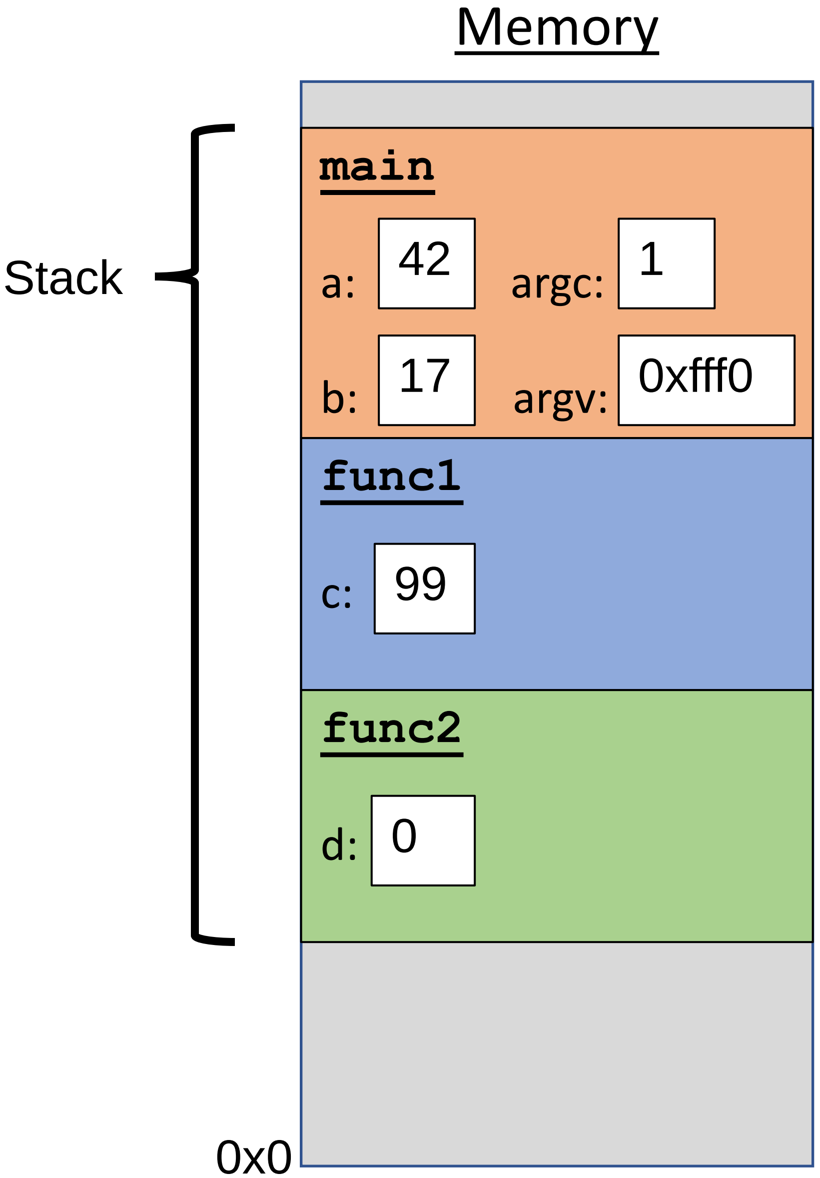


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

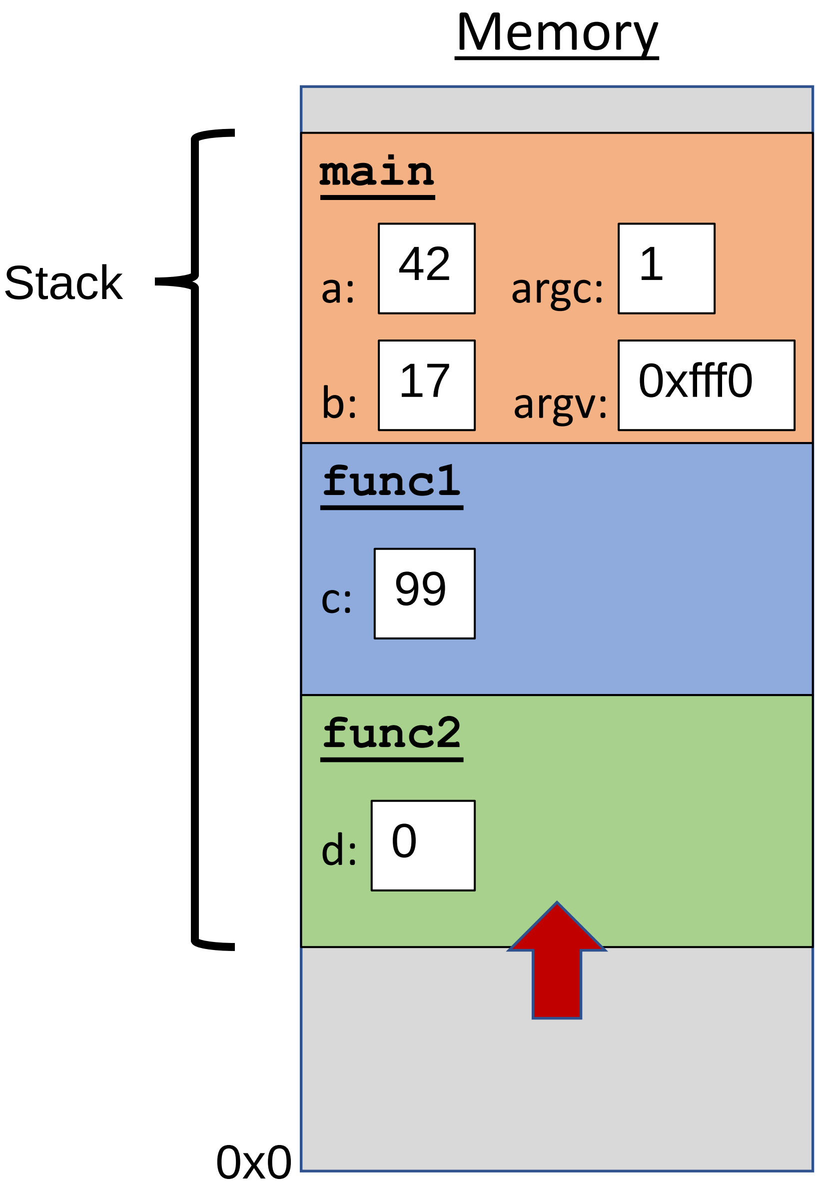


The Stack

```c
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

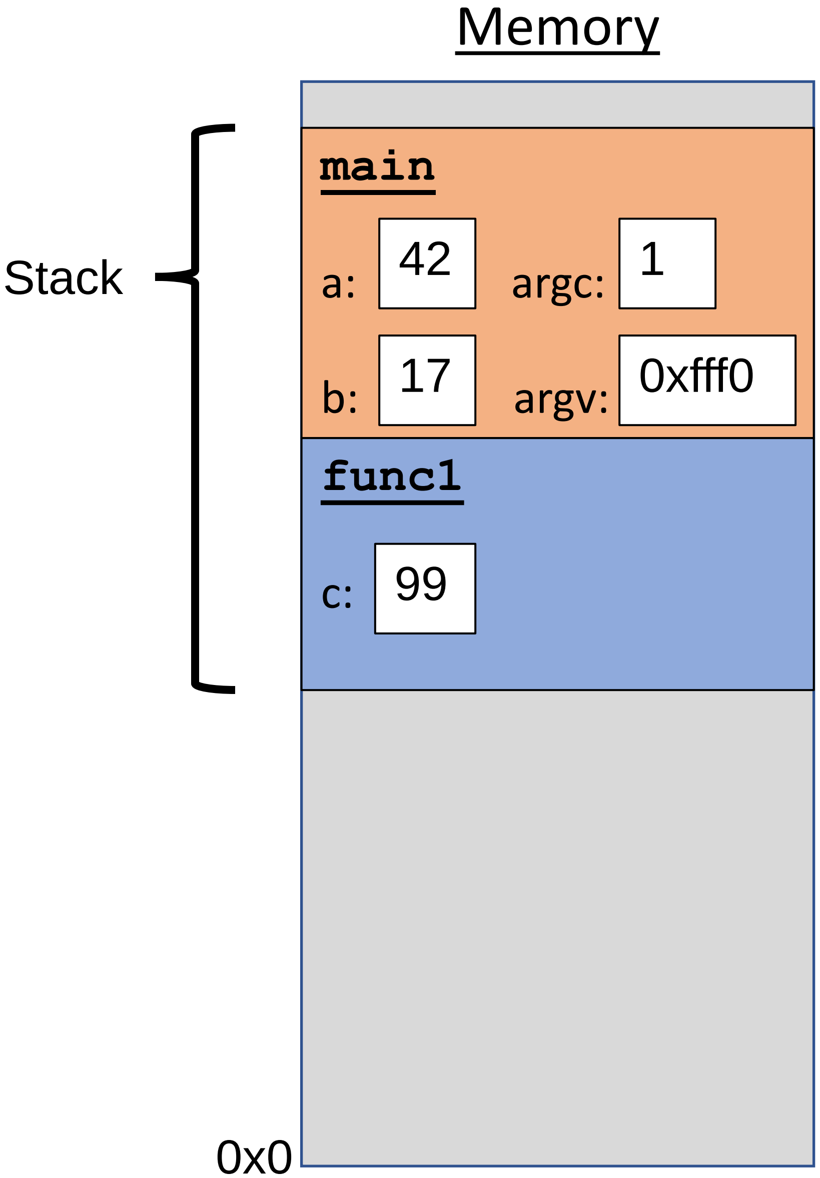


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

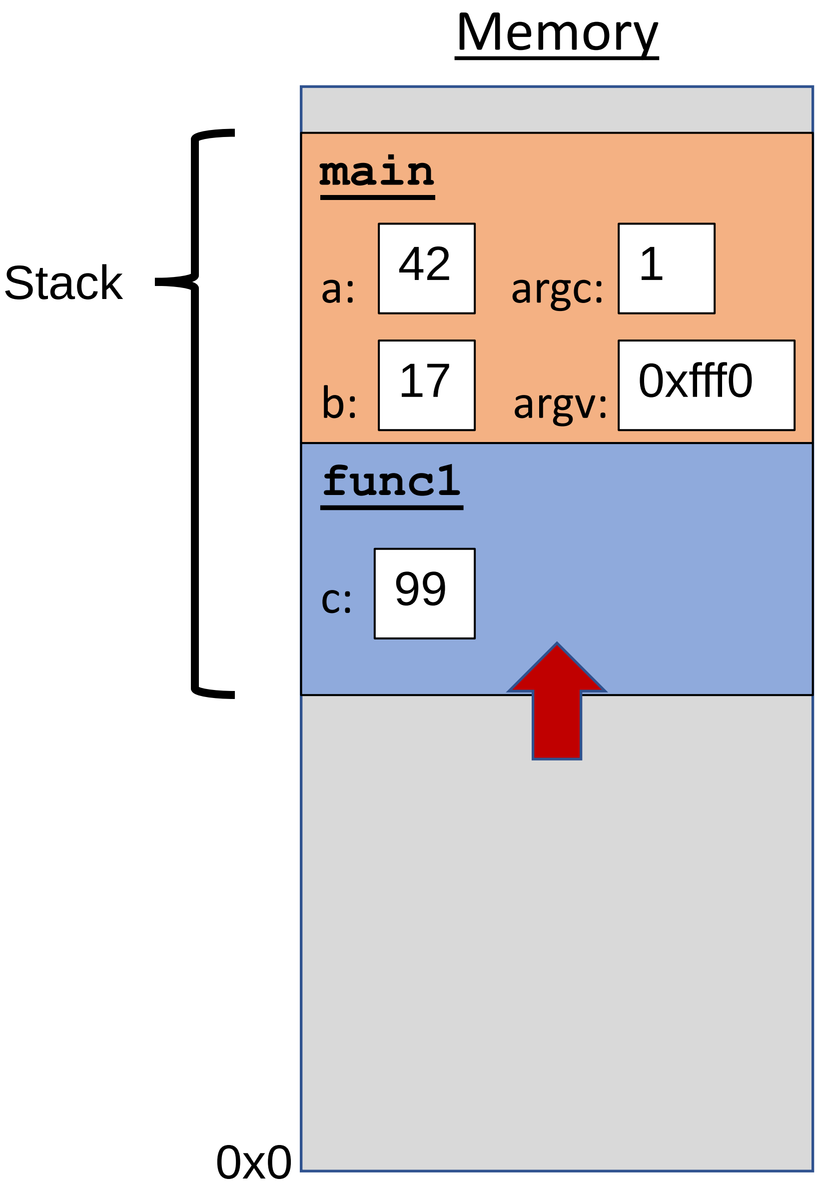


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

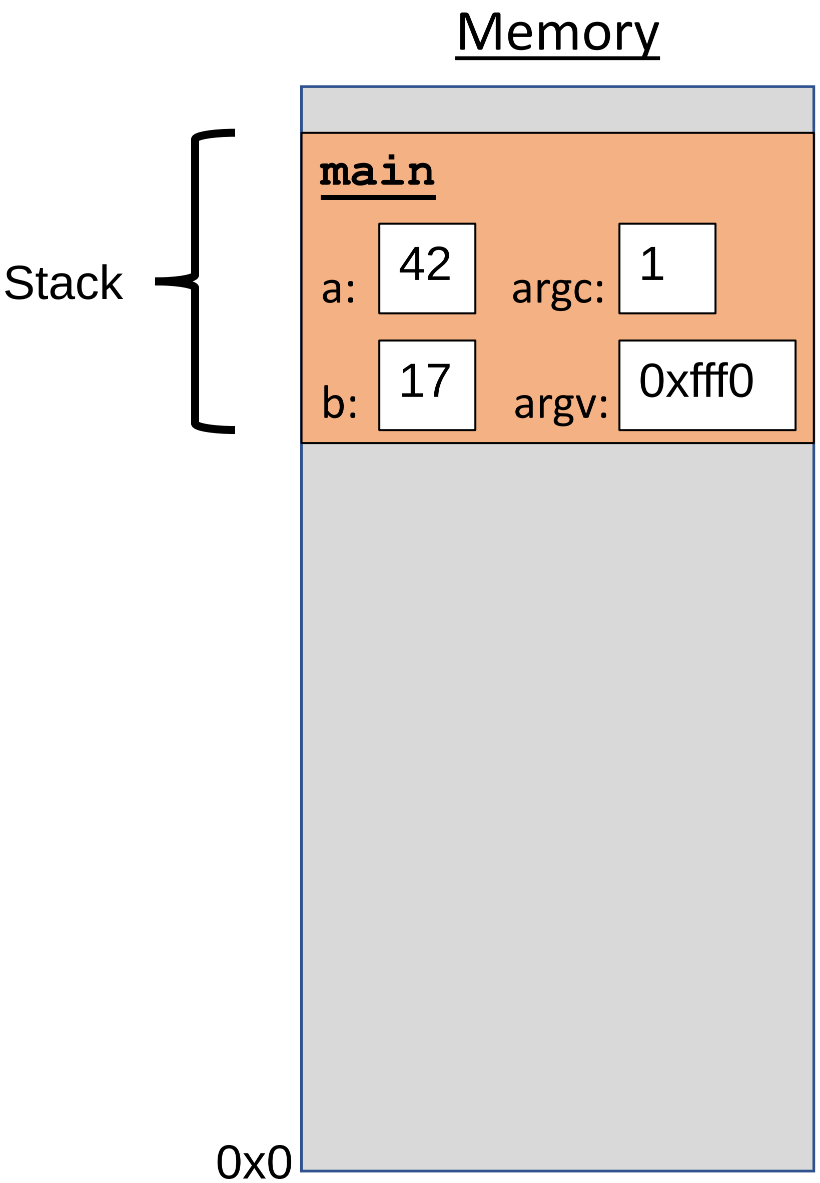


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

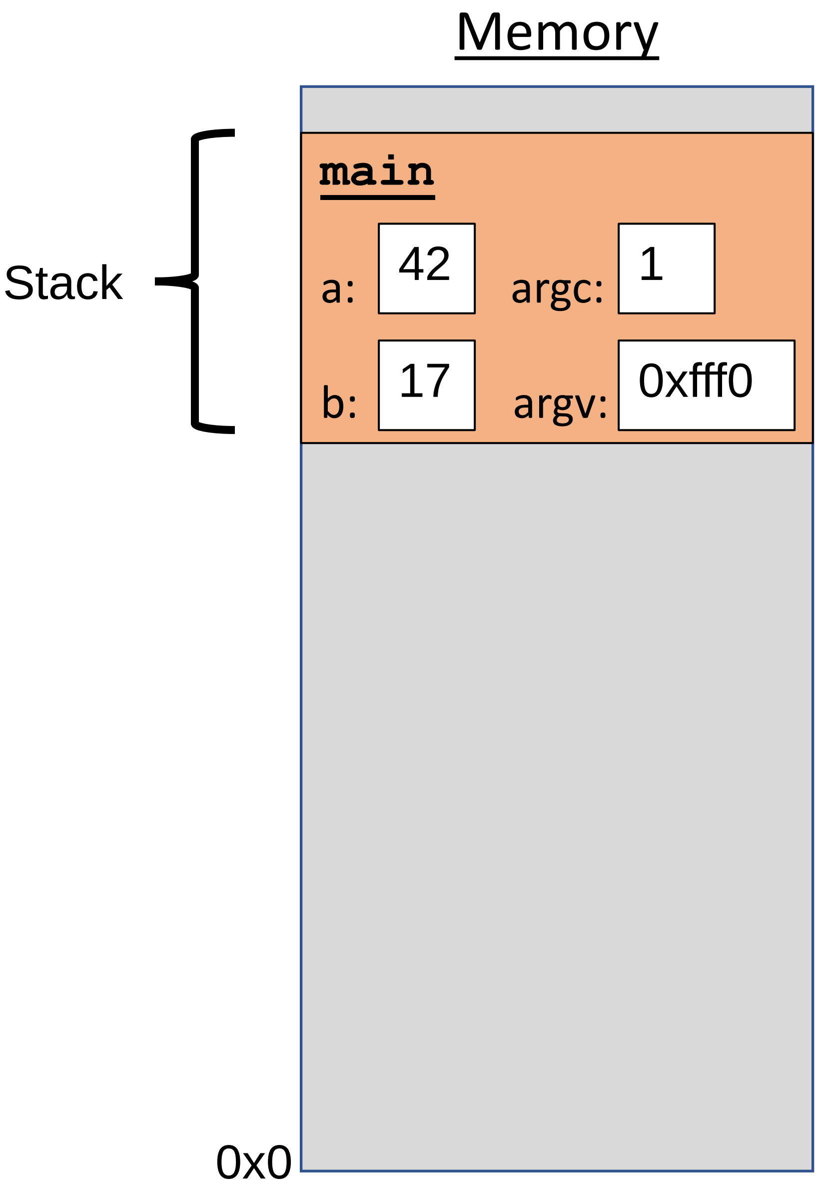


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

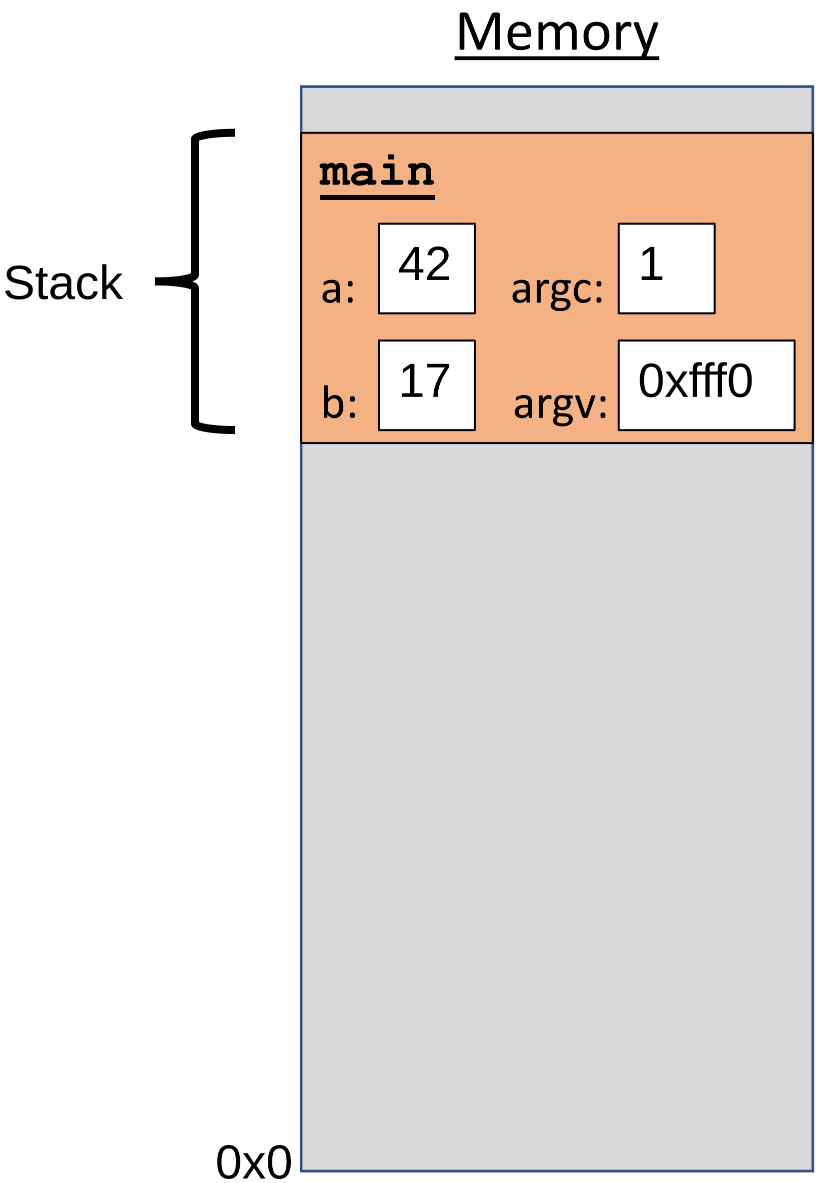


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

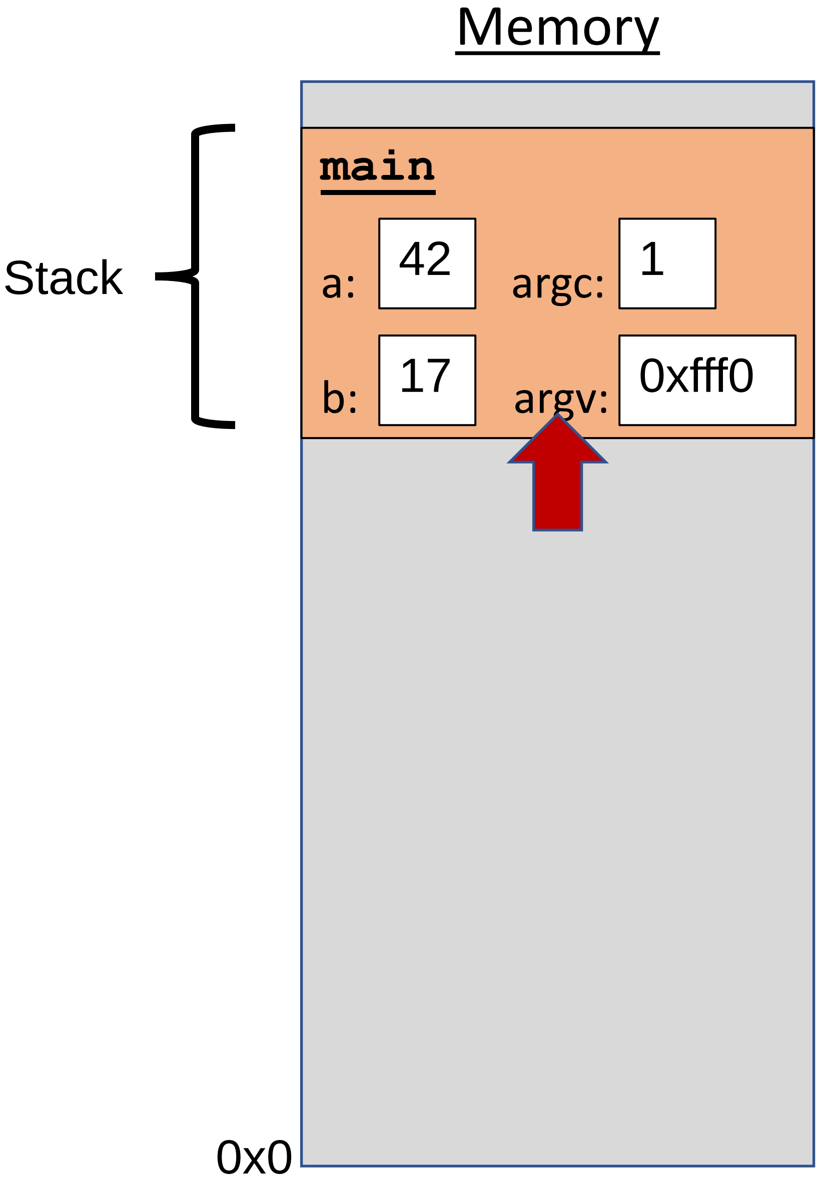


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```

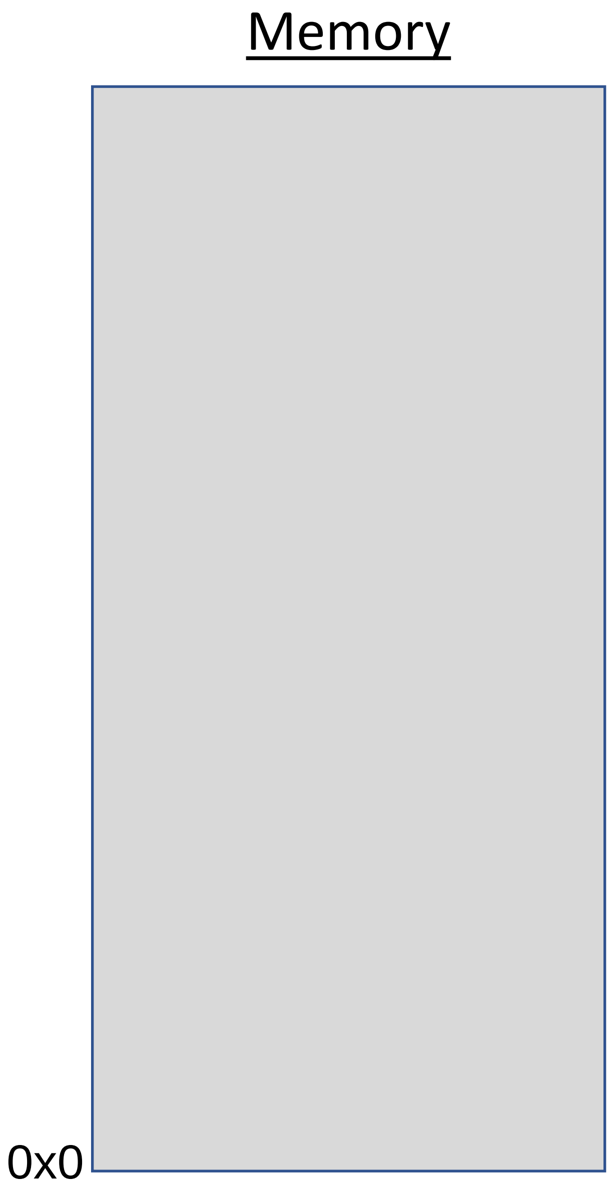


The Stack

```
void func2() {
    int d = 0;
}

void func1() {
    int c = 99;
    func2();
}

int main(int argc, char *argv[]) {
    int a = 42;
    int b = 17;
    func1();
    printf("Done.");
    return 0;
}
```



Recursion

A process by which a function calls itself repeatedly.

- **Either directly.**
  - **F calls F.**
- **Or cyclically in a chain.**
  - **F calls G, G calls H, and H calls F.**

Used for repetitive computations in which each action is stated in terms of

a previous result. **fact(n) = n * fact (n-1)**

Basis and Recursion

For a problem to be written in recursive form, two conditions are to be satisfied:

- **It should be possible to express the problem in recursive form.**
- **The problem statement must include a stopping condition**

fact(n) = 1,	**if n = 0**	**/* Stopping criteria */**
= n * fact(n – 1),	**if n > 0**	**/* Recursive form */**

Examples:

- **Factorial:**

  fact(0) = 1

  fact(n) = n * fact(n − 1), if n > 0

- **GCD (assume that m and n are non-negative and m ≥ n): gcd (m, 0) = m**

  gcd (m, n) = gcd (n, m%n) , if n > 0

- **Fibonacci sequence (0,1,1,2,3,5,8,13,21,…) fib (0) = 0**

  fib (1) = 1

  fib (n) = fib (n − 1) + fib (n − 2), if n > 1

Example 1 :: Factorial

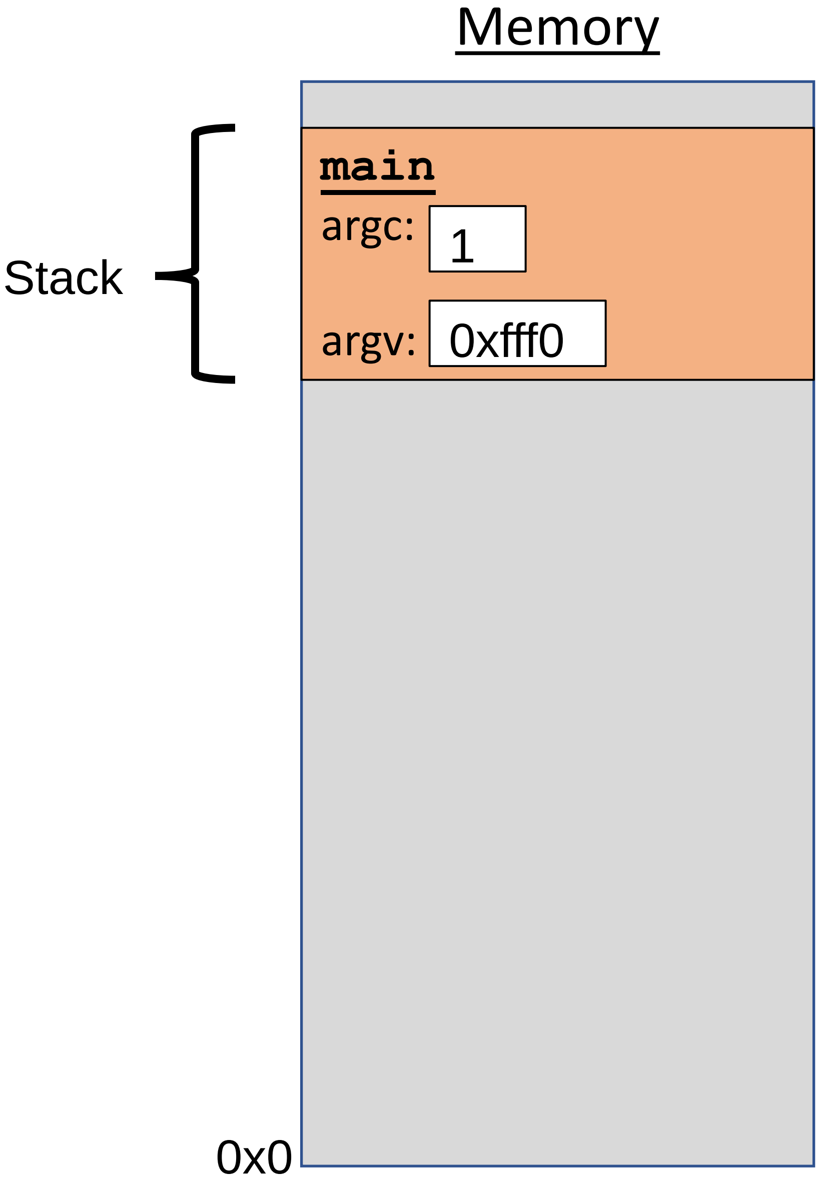
```
int fact ( int n)
{
   if  (n = = 1)
      return();
      else
      return(n * fact(n − 1));
}
```

The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

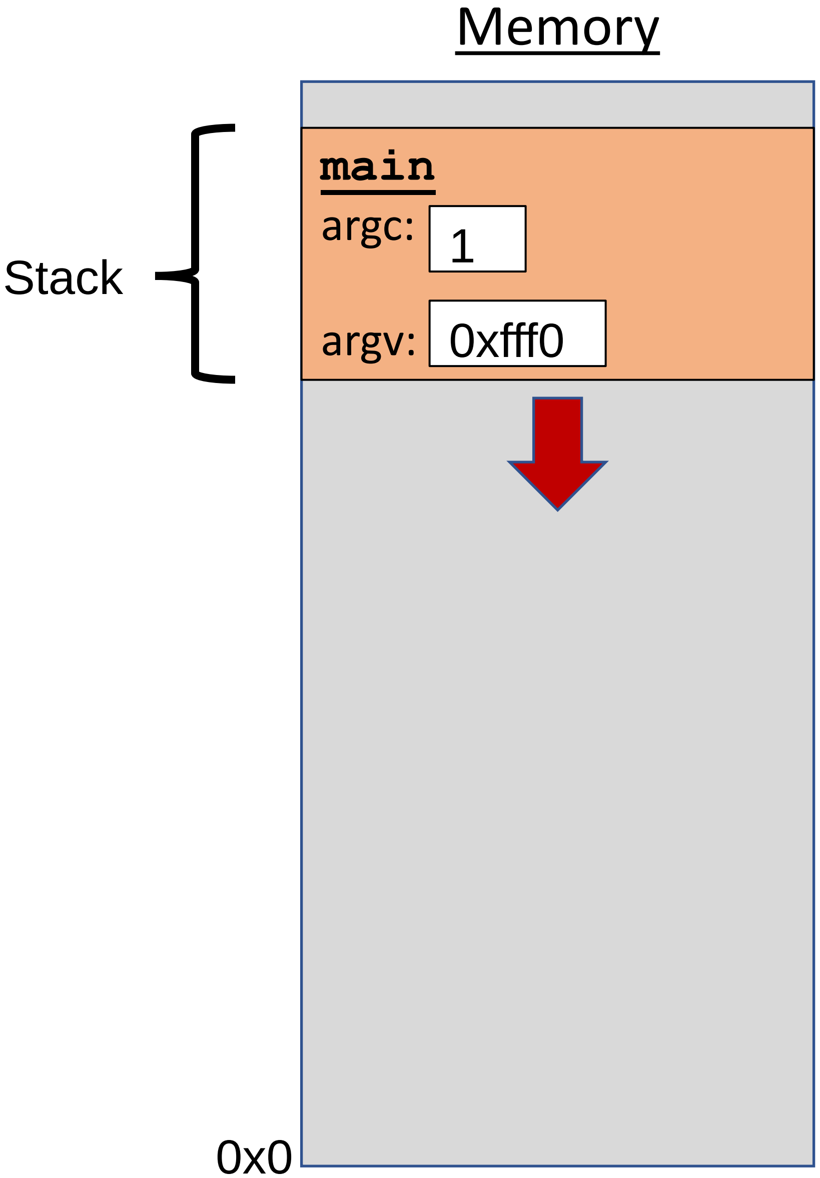


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

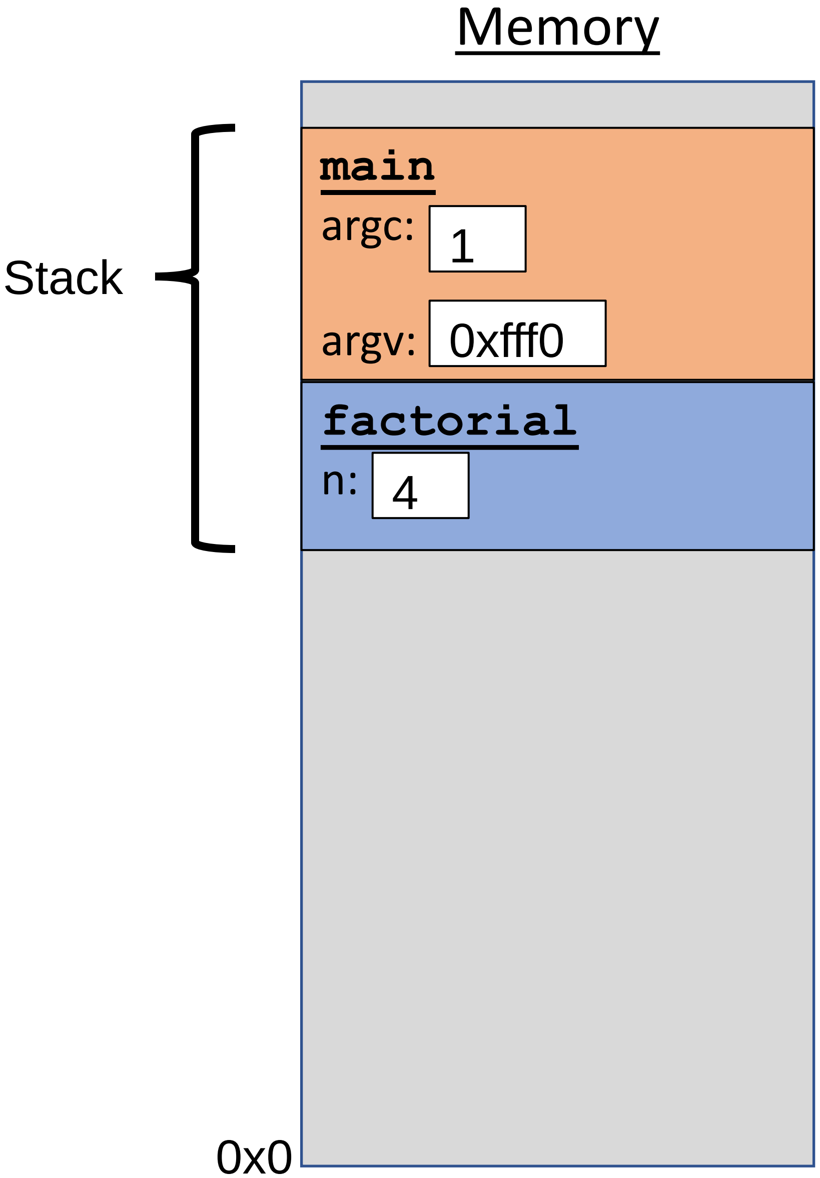


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

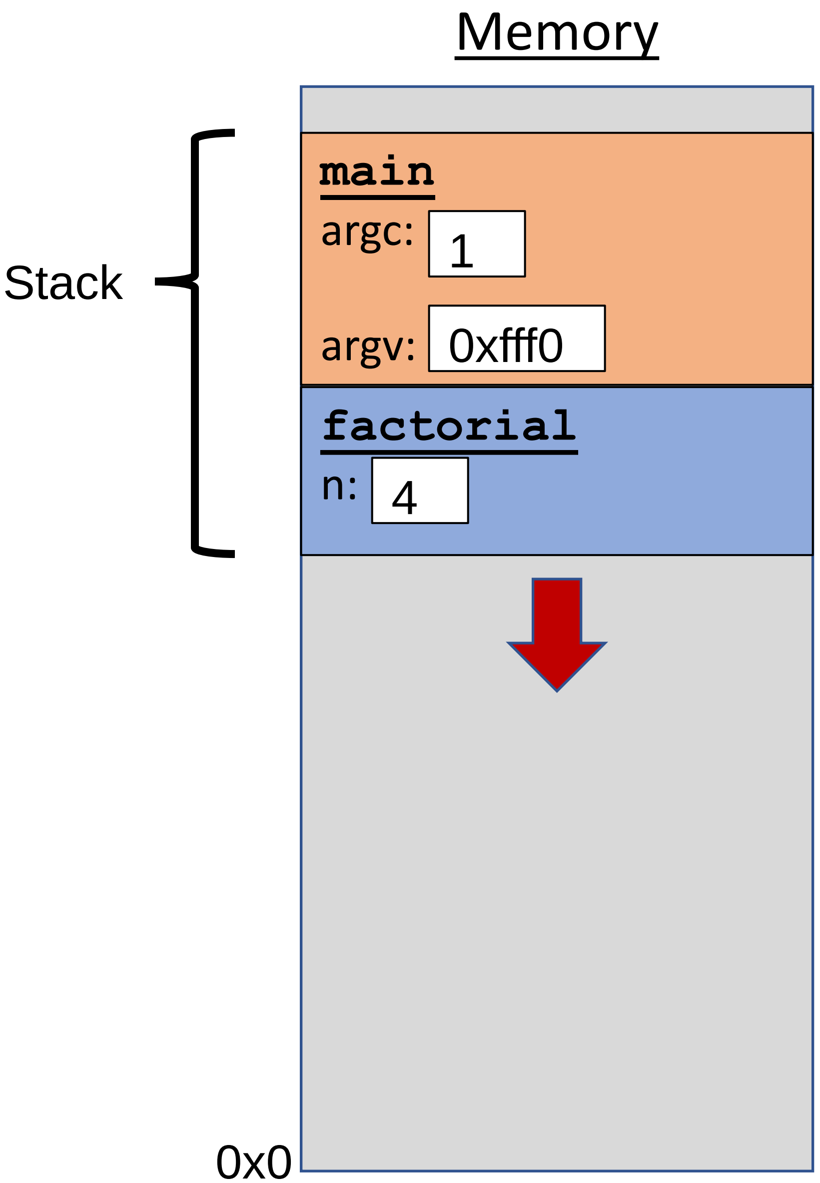


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

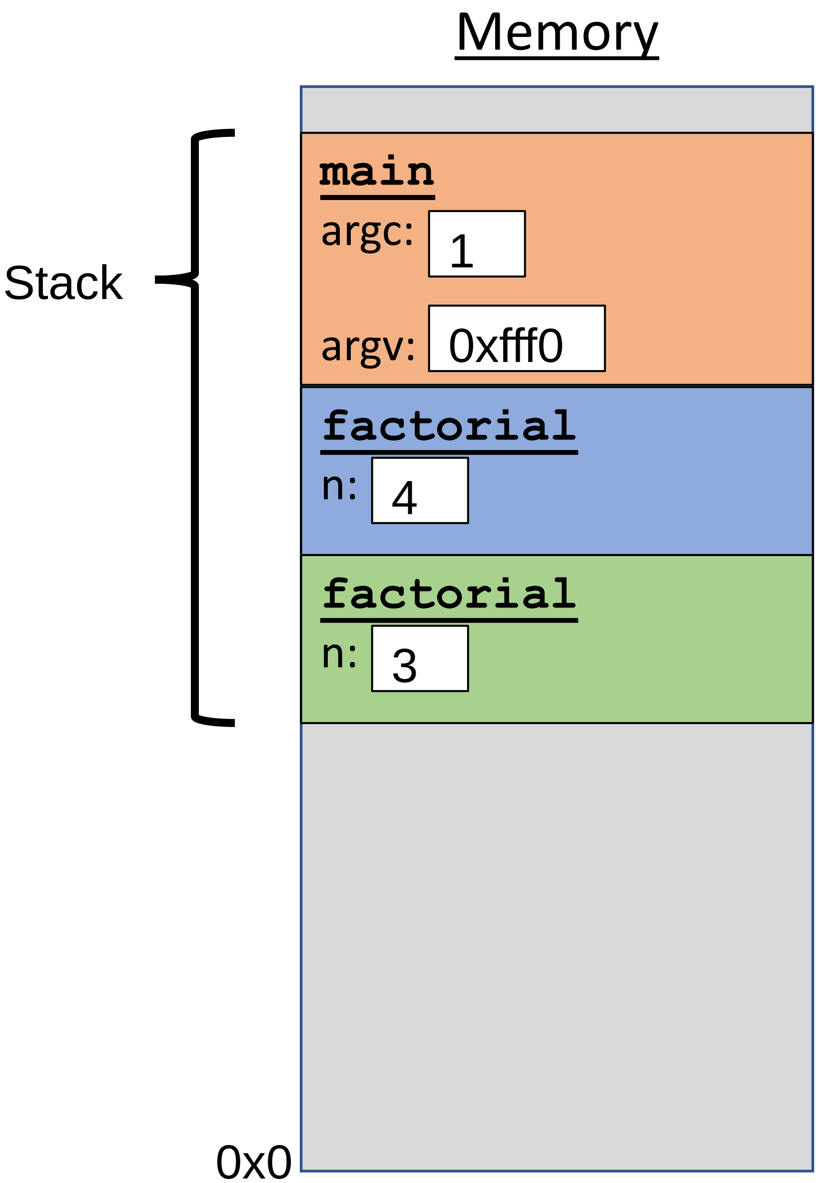


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

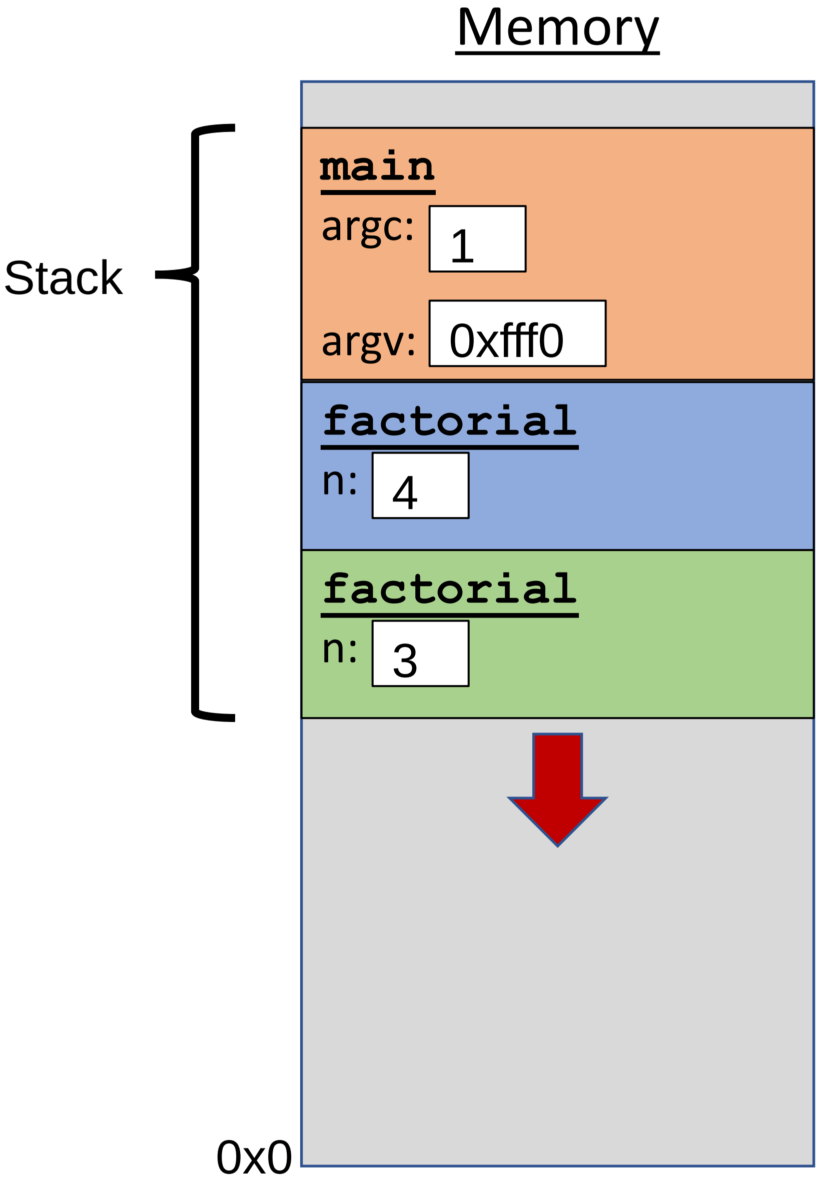


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

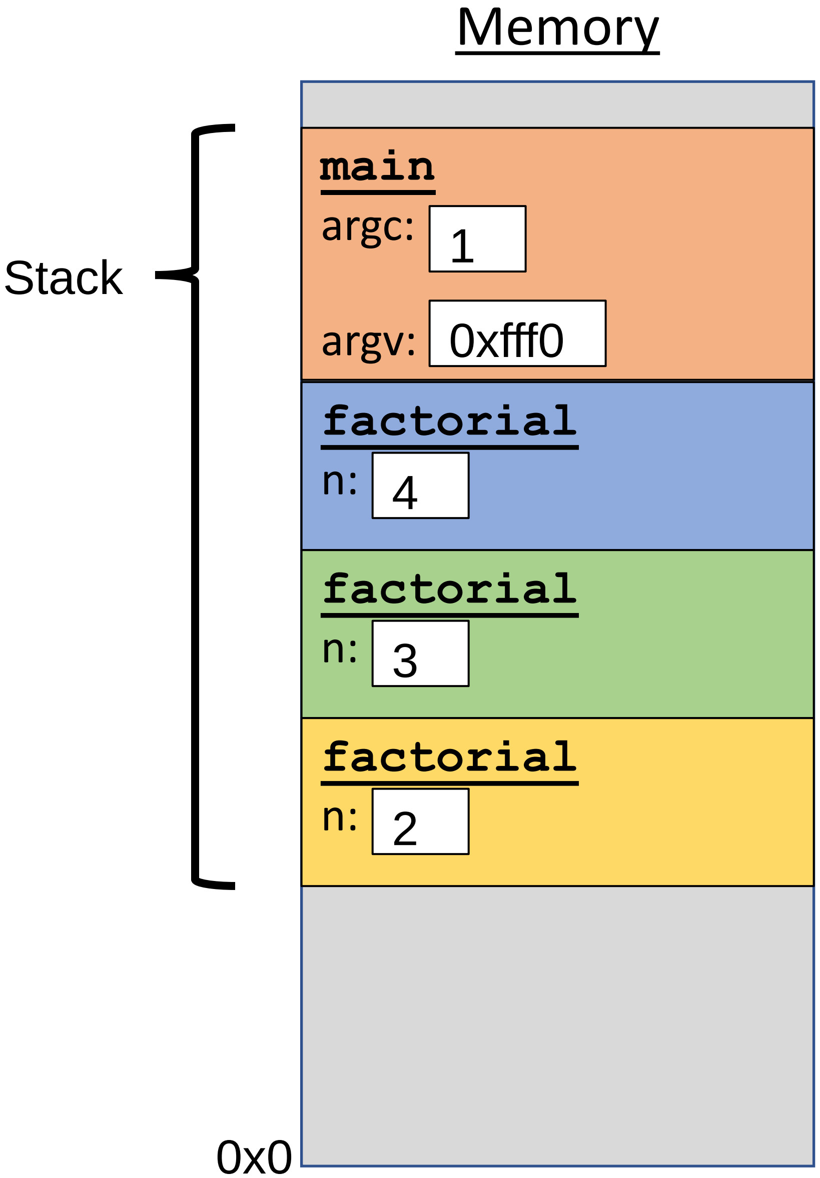


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

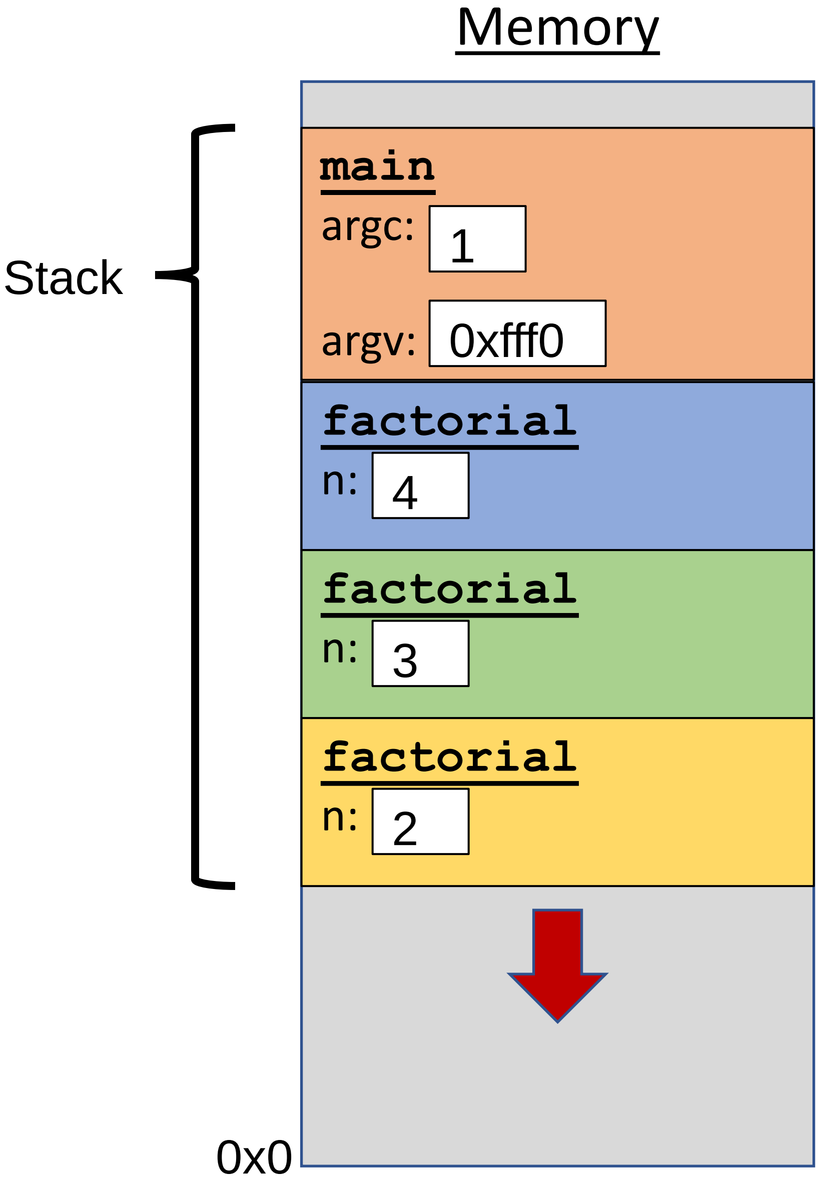


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

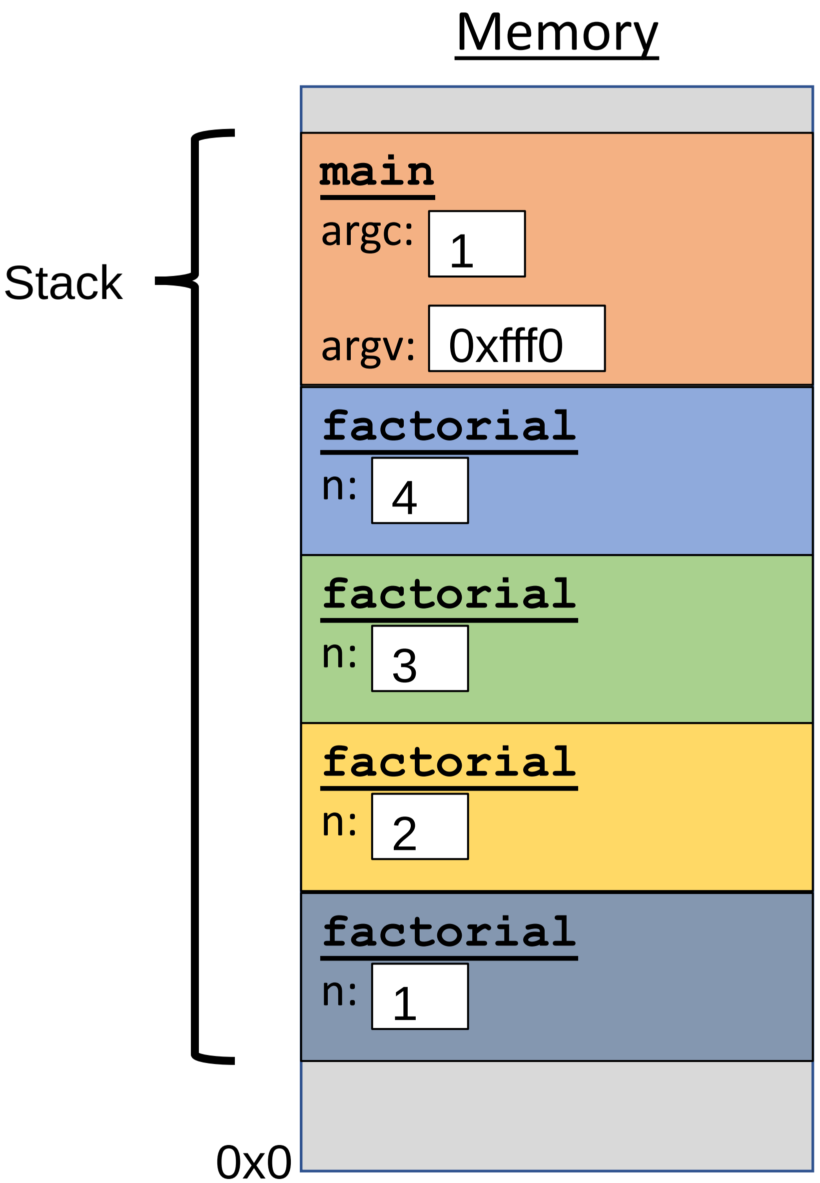


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

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

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

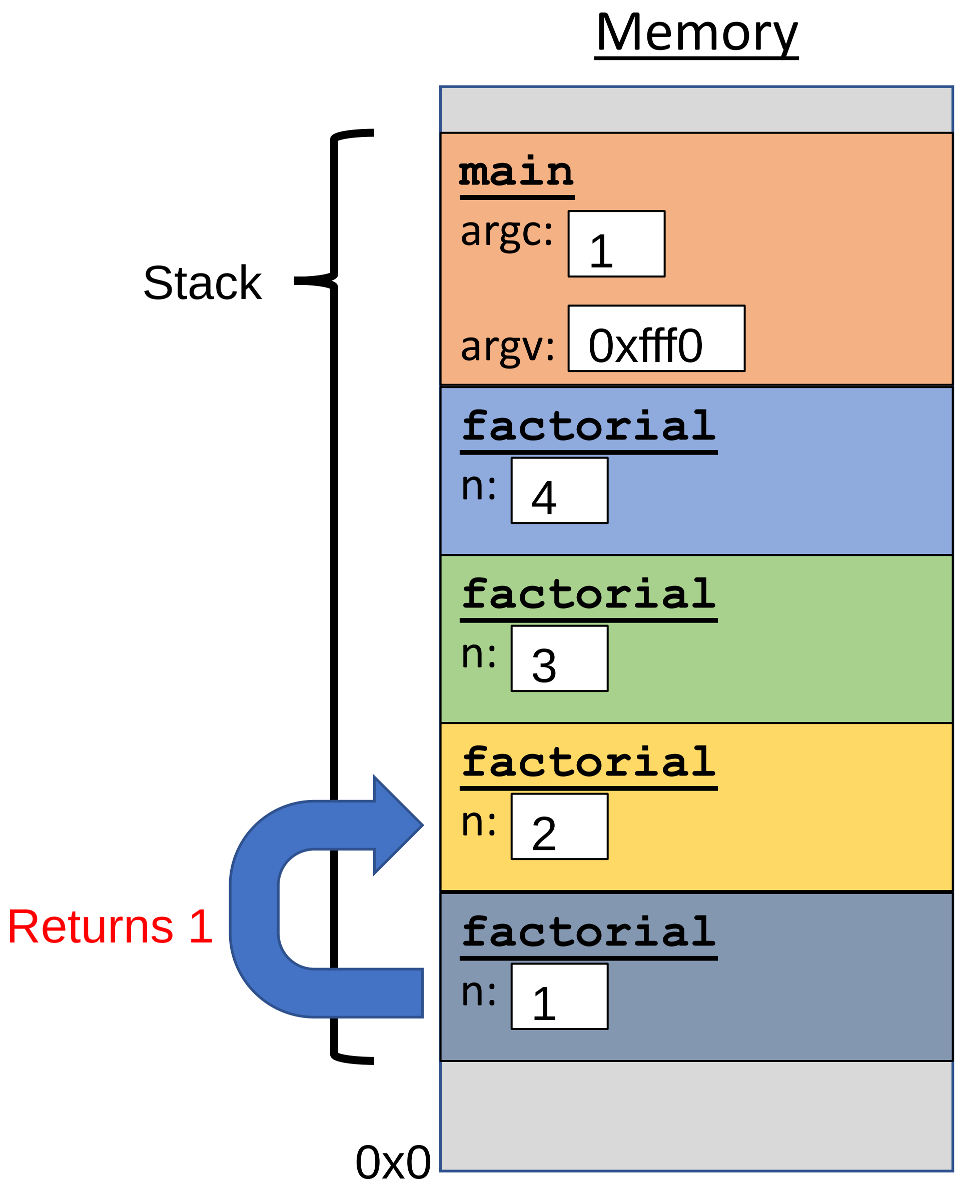


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

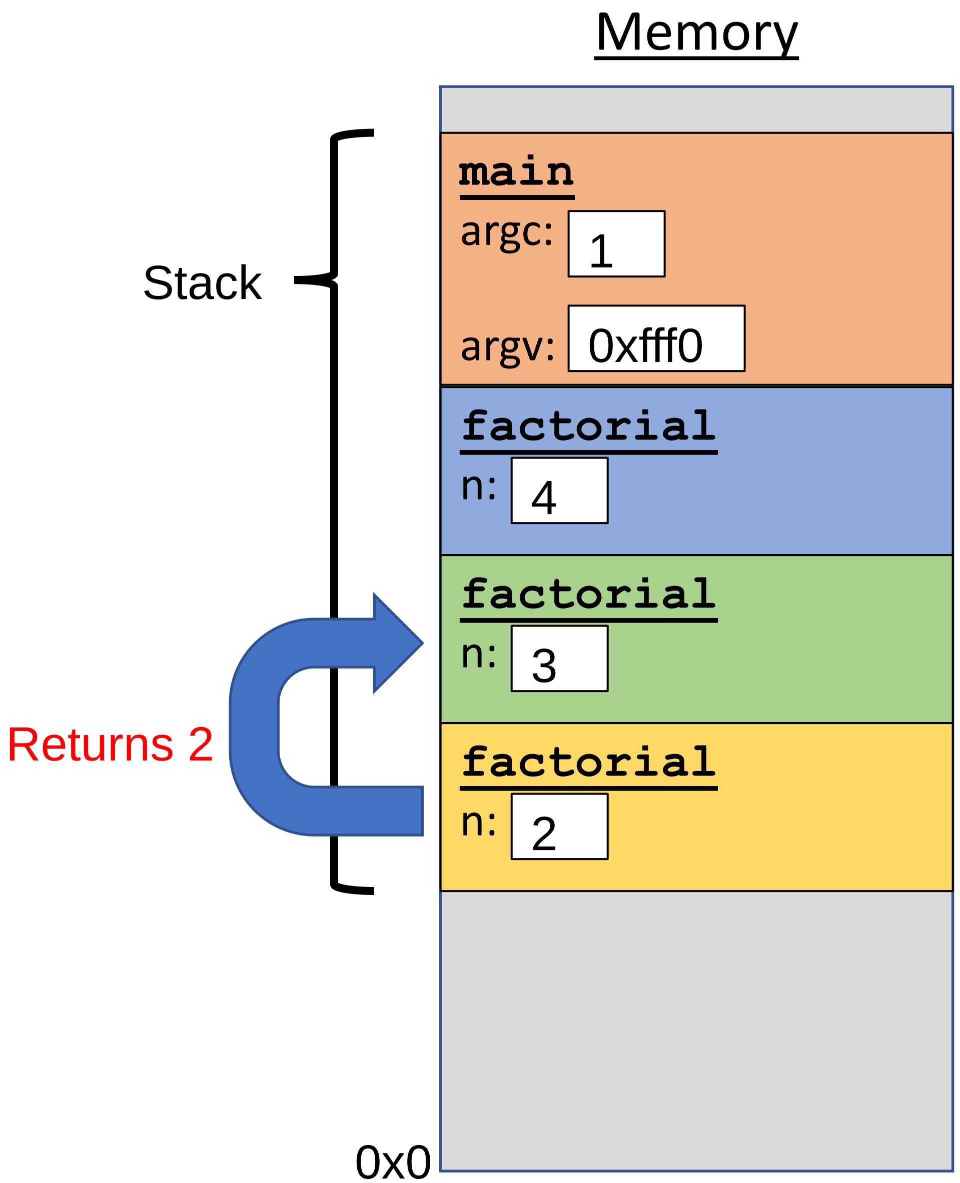


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

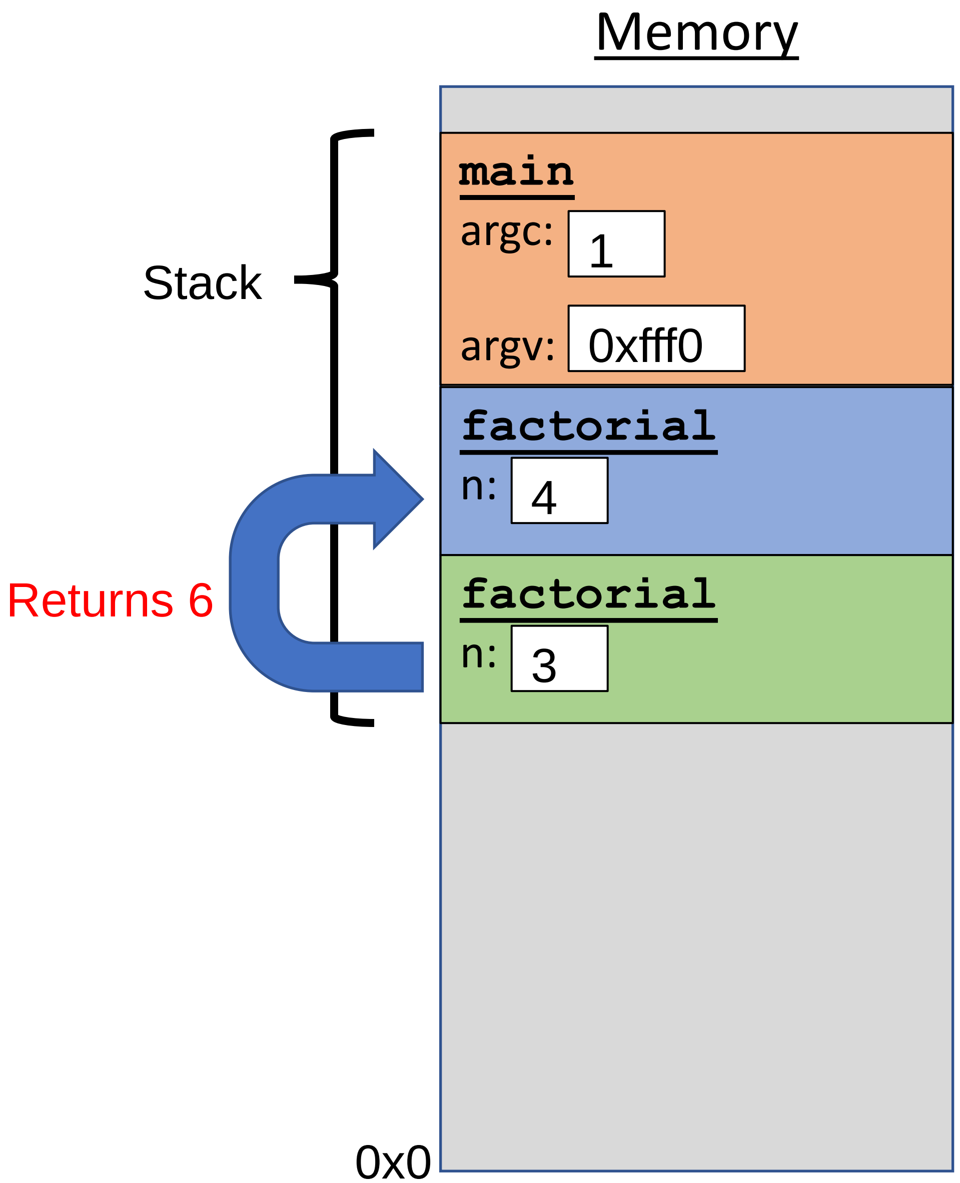


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

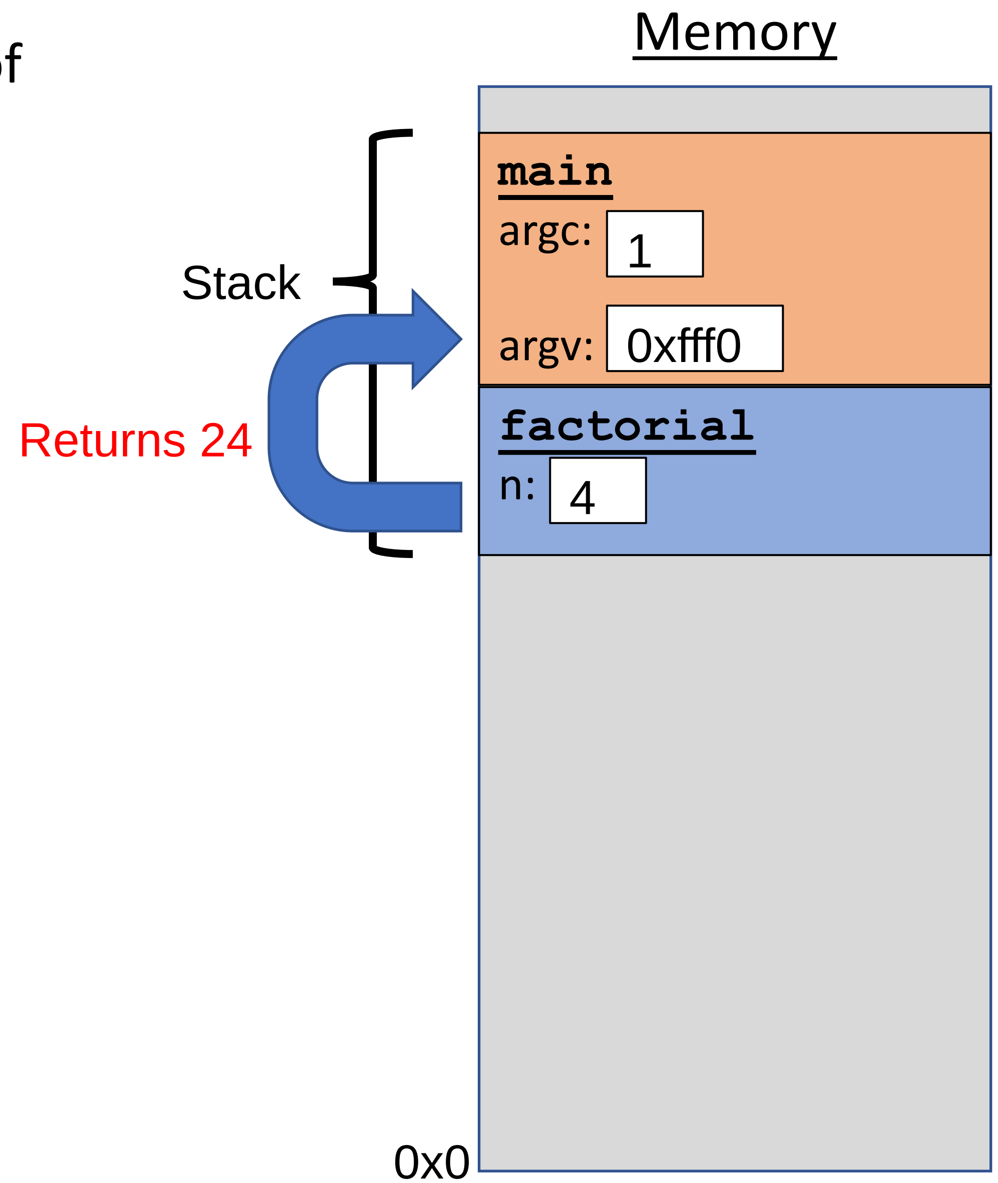


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

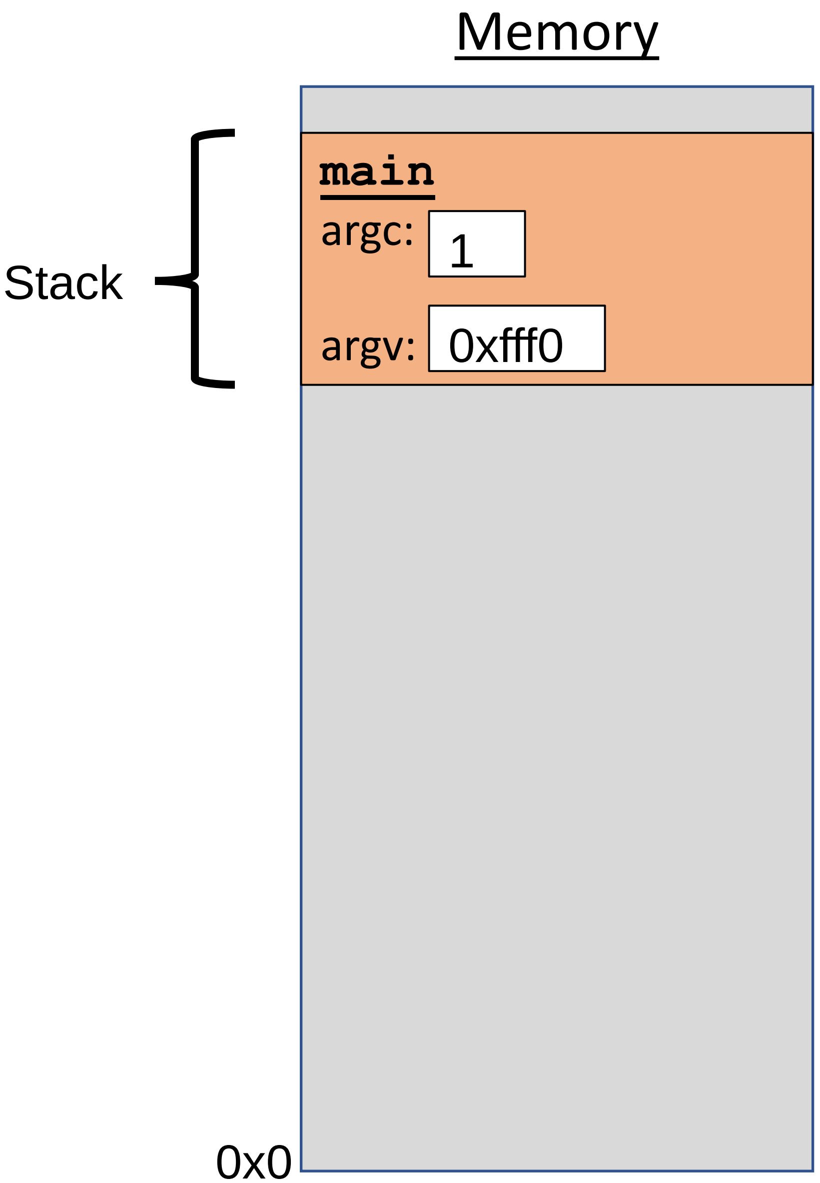


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

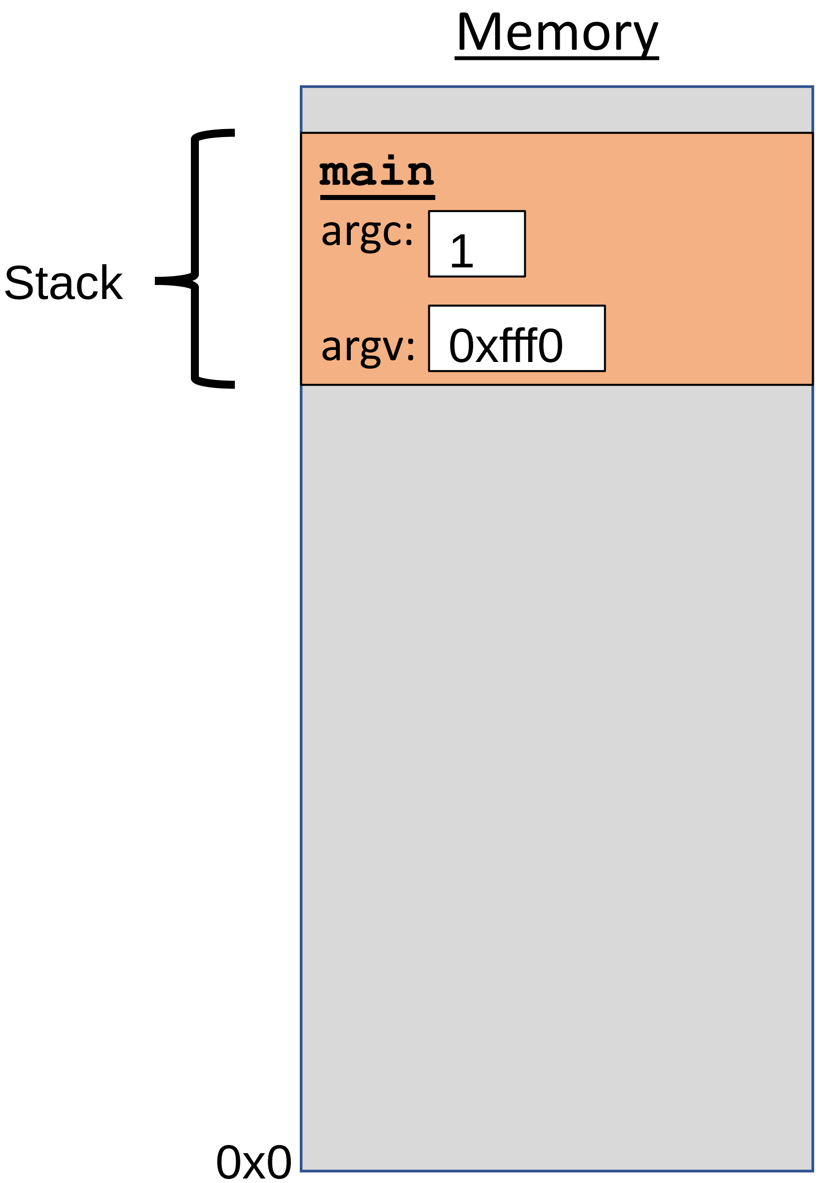


The Stack

Each function **call** has its own *stack frame* for its own copy of variables.

```
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n – 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```



The Stack

- The stack behaves like a…well…stack! A new function call **pushes** on a new frame. A completed function call **pops** off the most recent frame.
- *Interesting fact:* C does not clear out memory when a function's frame is removed. Instead, it just marks that memory as usable for the next function call. This is more efficient!
- A *stack overflow* is when you use up all stack memory. E.g. a recursive call with too many function calls.
- What are the limitations of the stack?

Example 1 :: Factorial Execution

fact(4)

if (4 = = 1) return (1);
else return (4 * fact(3));

24

if (3 = = 1) return (1); else return (3 * fact(2));

6

if (2 = = 1) return (1); else return (2 * fact(1));

2

if (1 = = 1) return (1); else return (1 * fact(0));

1

```
int fact ( int n)
{
   if (n = = 1) return (1);
   else
   return (n * fact(n – 1) );
}
```

Example 2 :: Fibonacci number

Fibonacci number f(n) can be defined as:

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2), \text{ if } \quad n > 1$$

- **The successive Fibonacci numbers are:**

  0, 1, 1, 2, 3, 5, 8, 13, 21, …..

```
int  f (int n)
{
    if  (n < 2)  return (n);
    else return ( f(n − 1) + f(n − 2) );
}
```

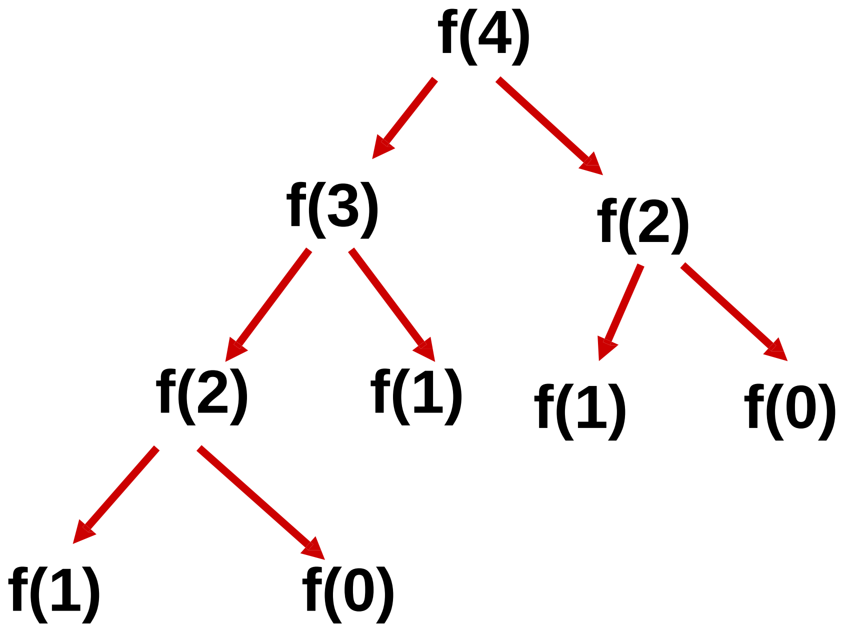
Tracing Execution

```
int  f (int n)
{
    if  (n  < 2)      return (n);
    else return ( f(n − 1) + f(n −
    2) );
}
```

How many times is the function called when evaluating f(4) ?

Inefficiency:

- **Same thing is computed several times.**



called 9 times

How are recursive calls implemented?

What we have seen ….

- **Activation record gets pushed into the stack when a function call is made.**
- **Activation record is popped off the stack when the function returns.**

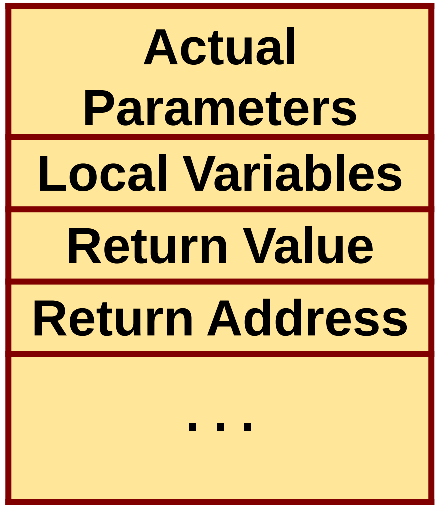
In recursion, a function calls itself.

- **Several function calls going on, with none of the function calls returning back.**
    - **Activation records are pushed onto the stack continuously.**
    - **Large stack space required.**

- Activation records keep popping off, when the termination condition of recursion is reached.

We shall illustrate the process by an example of computing factorial.

- Activation record looks like:

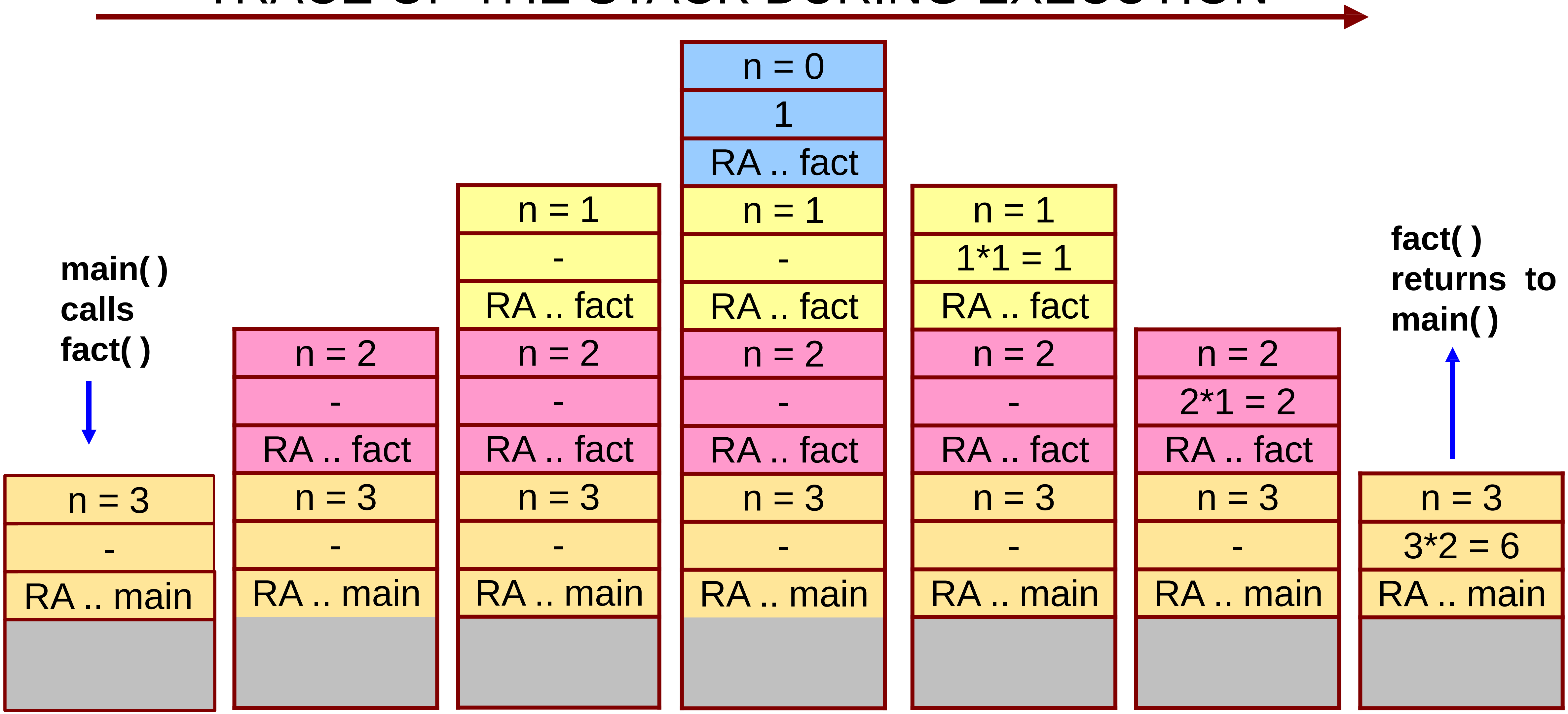


Example:: main( ) calls fact(3)

```
main()
{
  int n;  n = 3;
  printf (“%d \n”, fact(n) );
}
```

```
int fact (n)  int  n;
{
        if  (n = = 0)
                return (1);
        else
                return (n * fact(n-1));
}
```

TRACE OF THE STACK DURING EXECUTION



Do Yourself

Trace the activation records for the following version of Fibonacci sequence.

```
#include <stdio.h>
int  f (int n)
{
    int a, b;
    if  (n  < 2)       return (n);
    else  {
X    a = f(n-1);
Y    b = f(n-2);
     return (a+b);  }
}

main( ) {
   printf(“Fib(4) is: %d \n”, f(4));
}
```

Actual Parameters (n)
Local Variables (a, b)
Return Value
Return Address (either main or f)

Some points to note

Every recursive program can also be written without recursion

- **Tail Recursion: Last thing a recursive function does is making a single recursive call (of itself) at the end.**
- **Easy to replace tail recursion by a loop.**
- **In general, removal of recursion may be a very difficult task (even if you have your own recursion stack).**

Recursion can be helpful in many situations

- **Better readability**
- **Ease of programming**
- **Sometimes, recursion gives best-possible or best-known algorithms to solve problems**

Recursion can also be a killer

- **You solve the same subproblem multiple times (Example: Fibonacci numbers)**
- **Every recursive call incurs a (small) overhead**

Use recursion with caution

Example of tail recursion

Not a tail recursion:

```
int sum1 ( int n )
{
        if (n == 0) return 0;
        return n + sum1(n–1);
}
```

Tail recursion:

```
int sum2 ( int n, int partialsum )
{
        if (n == 0) return partialsum;
        return sum2(n – 1, n + partialsum);
}
```

Call from main() as:

```
scanf("%d", &N);
s = sum2(N, 0);
```

Equivalent iterative function:

```
int sum3 ( int n )
{
        int partialsum= 0;
        while (n > 0) {
                partialsum = n + partialsum;
                n = n – 1;
        }
        return partialsum;
}
```

Important things to remember

- **Think how the current problem can be solved if you can solve exactly the same problem on one or more smaller instance(s).**
- **Do NOT think how the problem will be solved on smaller instances, just call the function recursively and assume that the recursive calls do their jobs correctly.**
- **Do NOT forget to include the base cases to solve the problem on *smallest* instances.**
- **This is basically mathematical induction applied to programming.**

- **When you write a recursive function**
    - **First, write the terminating/base condition**
    - **Then, write the rest of the function**
    - **Always double-check that you have both**

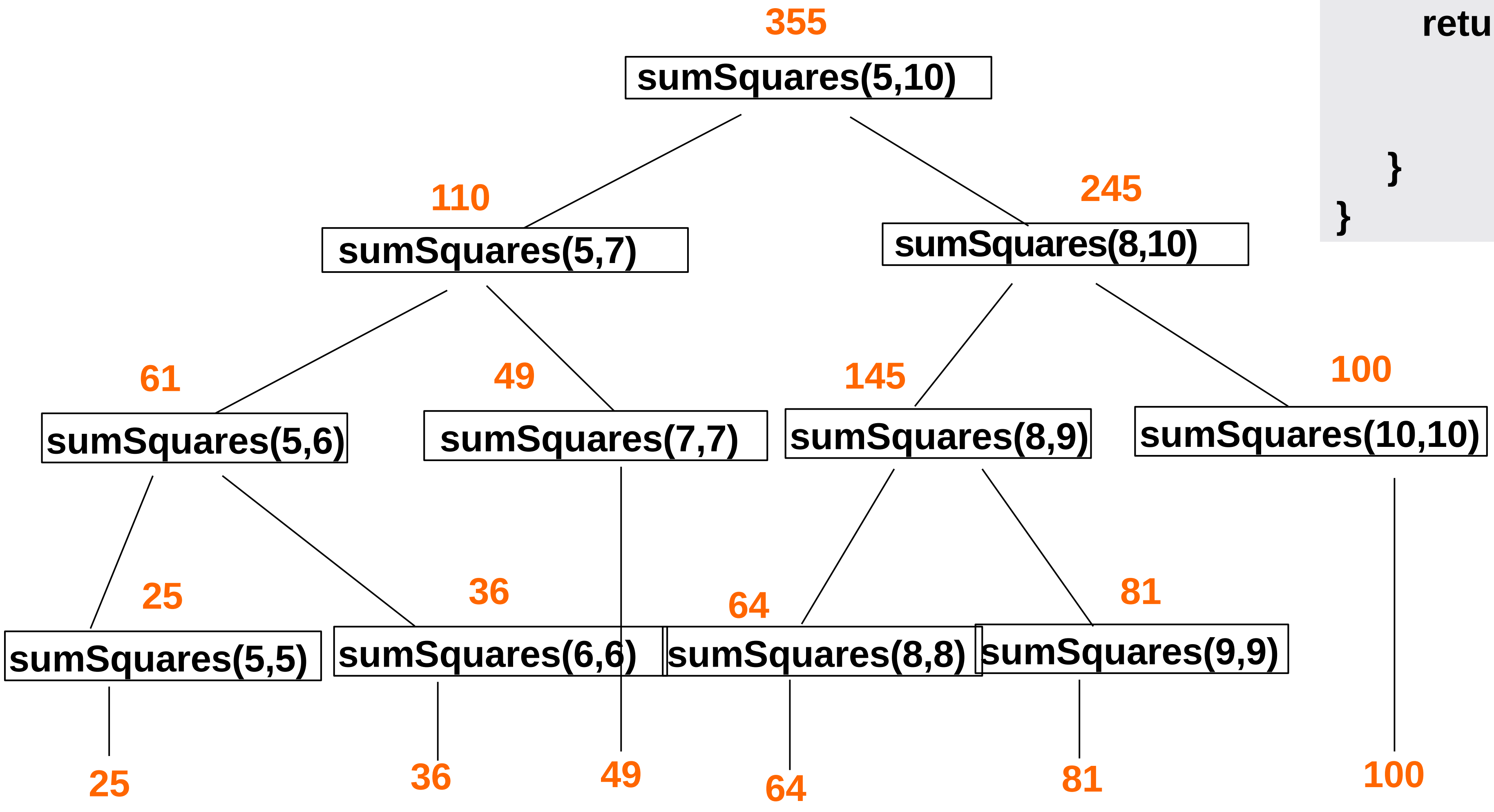
Example: Sum of Squares

Write a function that takes two integers m and n as arguments, and computes and returns the sum of squares of every integer in the range [m:n], both inclusive.

```
int sumSquares (int m, int n)
{
    int middle ;
    if (m == n)
    return(m*m);
    else
    {
        middle = (m+n)/2;
        return (sumSquares(m,middle) + sumSquares(middle+1,n));
     }
}
```

Annotated Call Tree

```
int sumSquares (int m, int n)
{
    int middle ;
    if (m == n) return(m*m);
    else {
        middle = (m+n)/2;
        return (sumSquares(m,middle)
                    +
                    sumSquares(middle+1,n));
    }
}
```



Example: Printing the digits of an integer in reverse

Print the last digit, then print the remaining number in reverse

- **Ex: If integer is 743, then reversed is print 3 first, then print the reverse of 74**

```
void printReversed ( int i )
{
    if (i < 10)       {
      printf("%d\n", i); return;
    }
    else {
       printf("%d", i%10);
       printReversed(i/10);
    }
}
```

Example: Printing your name in reverse

```
#include <stdio.h>

void readandprint ()
{
	char c;

	scanf("%c", &c);
	if (c == '\n') return;
	readandprint();
	printf("%c", c);
}

int main ()
{
	printf("Enter your name and hit return: ");
	readandprint();
	printf("\n");
}
```

Output

```
Enter your name and hit return:
Jane Doe  eoD enaJ
```

Exercise: Rewrite this code so that the output looks as follows:

```
Enter your name and hit return:
Jane Doe  Your name in reverse:
eoD enaJ
```

Counting Zeros in a Positive Integer

Check last digit from right

- **If it is 0, number of zeros = 1 + number of zeroes in remaining part of the number**
- **If it is non-0, number of zeros = number of zeroes in remaining part of the number**

```
int zeros (int number)
{
    if(number < 10) return 0;
    if (number % 10 == 0)
             return( 1 + zeros(number/10) );
    else
      return( zeros(number/10) );
}
```

Common Errors in Writing Recursive Functions

Non-terminating Recursive Function (Infinite recursion)

- **No base case**
- **The base case is never reached**

```
int badFactorial(int x) {
  return x *
  badFactorial(x-1);
}
int badSum2(int x)
{
  if(x==1) return 1;
  return(badSum2(
  x--));
}
```

```
int
  anotherBadFactorial(int
  x) {  if(x == 0)
    return
  1;  else
    return x*(x-1)*anotherBadFactorial(x-2);
    // When x is odd, base case is never
    reached!!
}
```

Common Errors in Writing Recursive Functions

Mixing up loops and recursion

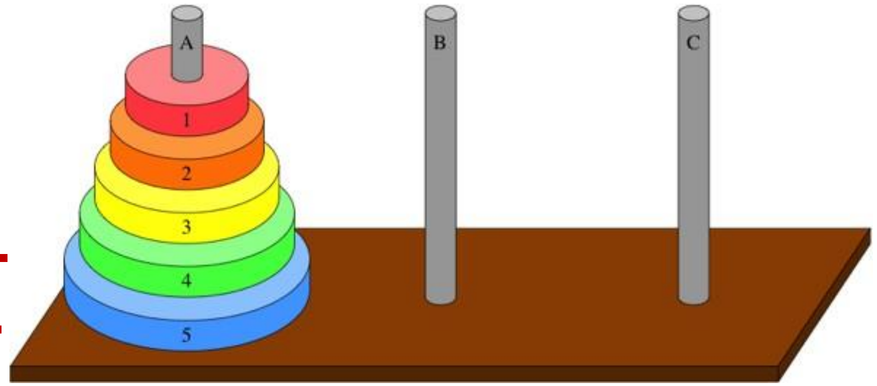
```
int anotherBadFactorial
      (int x) {  int i, fact = 0;
      if (x == 0) return 1;
      else {
            for (i=x; i>0; i=i-1) {
                 fact = fact + x*anotherBadFactorial(x-1);
            }
            return fact;
      }
}
```

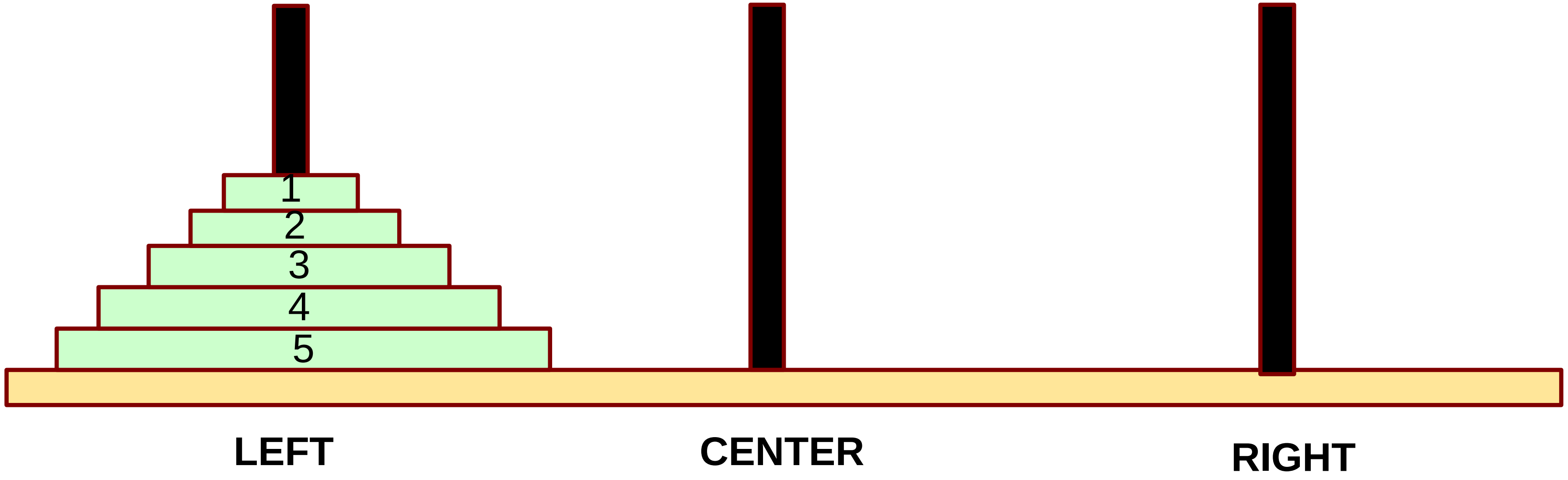
In general, if you have recursive function calls within a loop, think carefully if you need it. Most recursive functions you will see in this course will not need this

Example :: Towers of Hanoi Problem

The problem statement:

- **Initially all the disks are stacked on the LEFT pole.**
- **Required to transfer all the disks to the RIGHT pole.**
  - **Only one disk on the top can be moved at a time.**
  - **A larger disk cannot be placed on a smaller disk.**
- **CENTER pole is used for temporary storage of disks.**

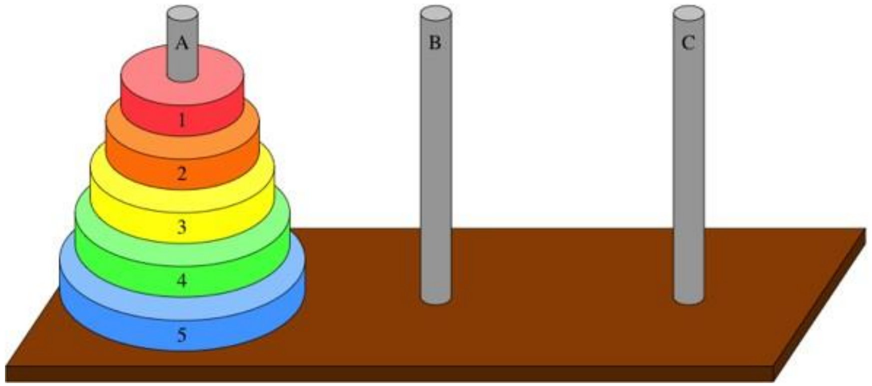




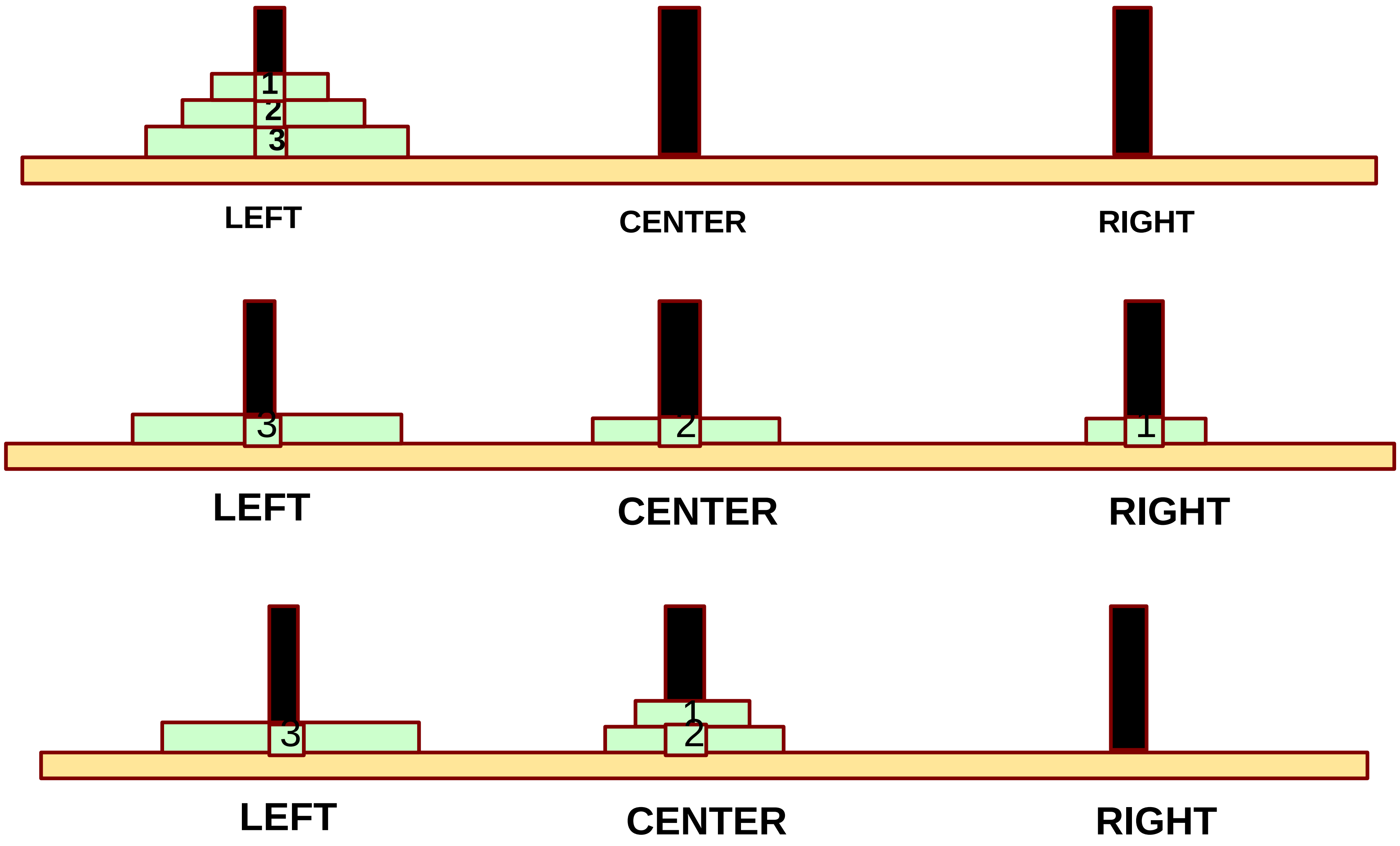
Recursive Formulation

Recursive statement of the general problem of n disks.

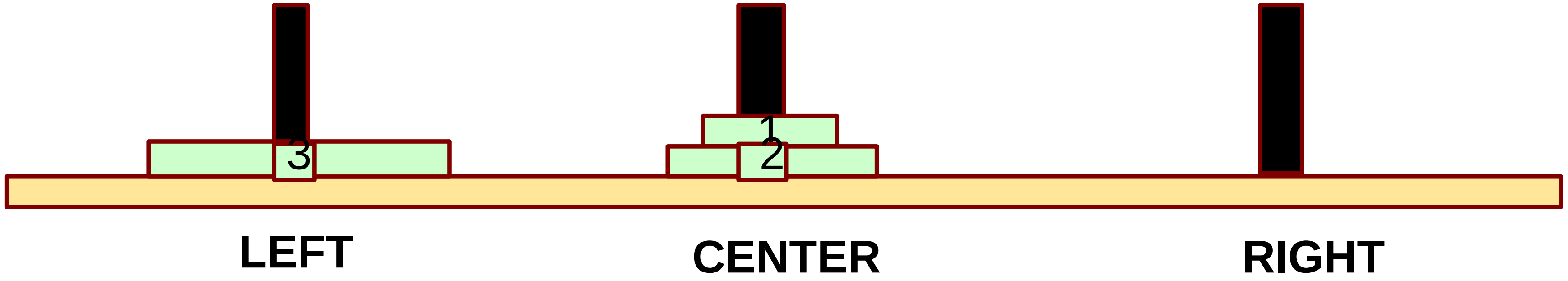
- **Step 1:**
  - **Move the top (n-1) disks from LEFT to CENTER.**
- **Step 2:**
  - **Move the largest disk from LEFT to RIGHT.**
- **Step 3:**
  - **Move the (n-1) disks from CENTER to RIGHT.**

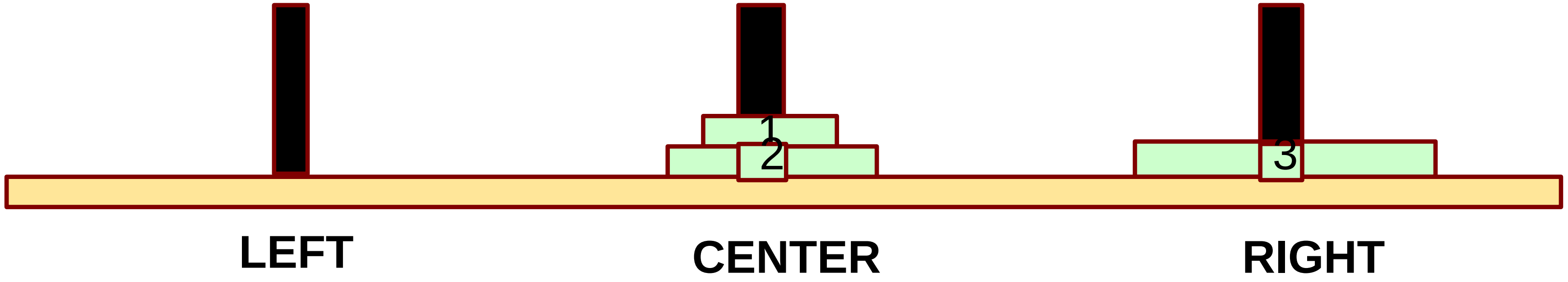


Phase-1: Move top n – 1 from LEFT to CENTER

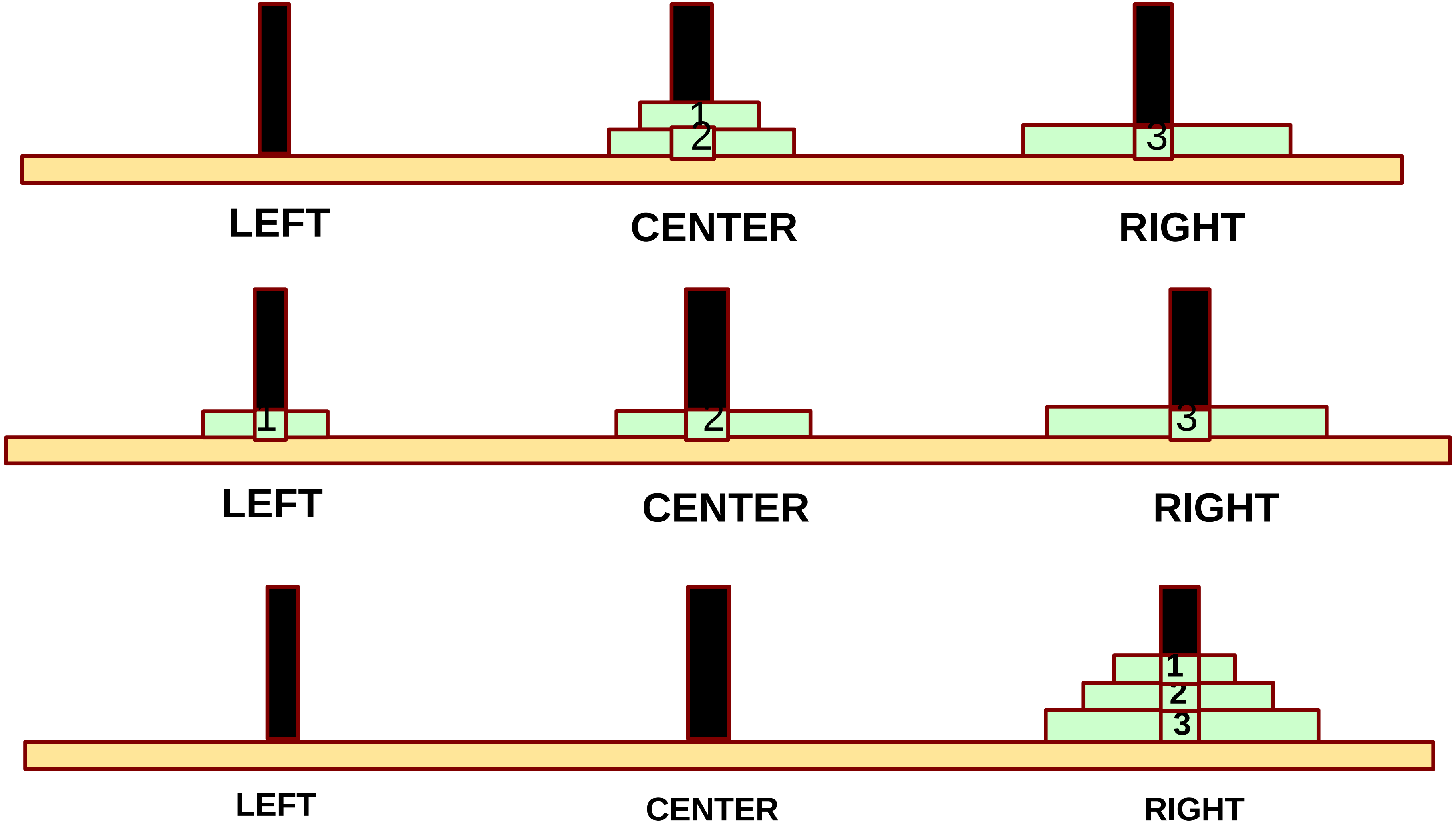


Phase-2: Move the n^{th} disk from LEFT to RIGHT





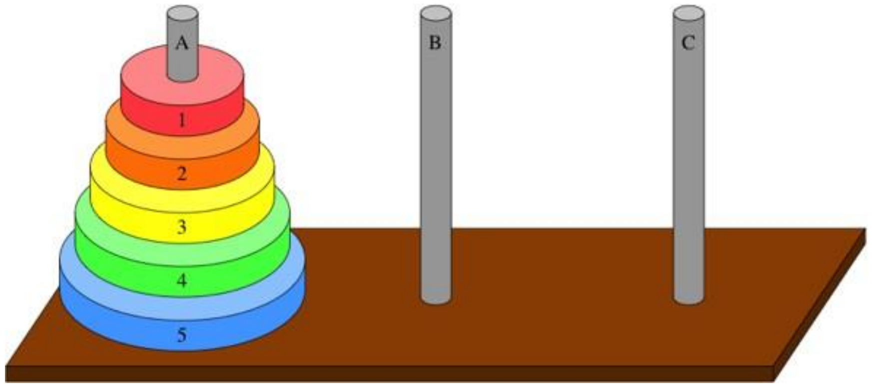
Phase-3: Move top n – 1 from CENTER to RIGHT



```
#include <stdio.h>
void transfer (int n, char from, char to, char temp);

int main( )
{           int n; /* Number of disks */
            scanf (“%d”, &n);
            transfer (n, ‘L’, ‘R’, ‘C’);
            return 0;
}

void transfer (int n, char from, char to, char temp)
{
            if (n > 0) {
                        transfer (n-1, from, temp, to);
                        printf (“Move disk %d from %c to %c \n”, n, from, to);
                        transfer (n-1, temp, to, from);
            }
            return;
}
```



With 3 discs

Telnet 144.16.192.60

```
3
Move disk 1 from L to R
Move disk 2 from L to C
Move disk 1 from R to C
Move disk 3 from L to R
Move disk 1 from C to L
Move disk 2 from C to R
Move disk 1 from L to R
[isg@facweb temp]$
```

With 4 discs

Telnet 144.16.192.60

```
4
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
Move disk 3 from L to C
Move disk 1 from R to L
Move disk 2 from R to C
Move disk 1 from L to C
Move disk 4 from L to R
Move disk 1 from C to R
Move disk 2 from C to L
Move disk 1 from R to L
Move disk 3 from C to R
Move disk 1 from L to C
Move disk 2 from L to R
Move disk 1 from C to R
[isg@facweb temp]$
```

Recursion versus Iteration

Repetition

- **Iteration: explicit loop**
- **Recursion: repeated nested function calls**

Termination

- **Iteration: loop condition fails**
- **Recursion: base case recognized** **Both**

can have infinite loops

Balance

- **Understand the benefits / penalties of recursion in terms of**
  - **Ease of implementation**
  - **Readability**
  - **Performance degradation / performance enhancement**
- **Take an educated decision**

More Examples

What do the following programs print?

```
void foo( int n  )
{
     int data;
     if ( n == 0 )
     return;
     scanf("%d",
     &data);
     foo ( n – 1 );
     printf("%d\n", data);
}
main ( )
{    int k = 5;
     foo ( k );
}
```

```
void foo( int n  )
{
     int data;
     if ( n == 0 ) return;
     foo ( n – 1 );
     scanf("%d",
     &data);
     printf("%d\n",
     data);
}
main ( )
{    int k =
     5;  foo
     ( k );
}
```

```
void foo( int n  )
{
     int data;
     if ( n == 0 )
     return;
     scanf("%d",
     &data);
     printf("%d\n", data);
     foo ( n – 1 );
}
main ( )
{    int k =
     5;  foo
     ( k );
}
```

Printing cumulative sum -- *will this work?*

- int foo(int n)
- {
  - int data, sum ;
  - if (n == 0) return 0;
  - scanf(“%d”, &data);
  - sum = data + foo (n – 1); printf(“%d\n”, sum);
  - return sum;
- }
- main ( ) {
  - int k = 5; foo (k);
- }

Input: 1 2 3 4 5

Output: 5 9 12 14 15

How to rewrite this so that the output is: 1 3 6 10 15 ?

Printing cumulative sum (two ways)

```
int foo( int n  )
{
     int data, sum ;
     if ( n == 0 ) return 0;
     sum = foo ( n – 1 );
     scanf(“%d”, &data);
     sum = sum + data;
     printf(“%d\n”, sum);
     return sum;
}
main ( ) {
     int k = 5;
     foo ( k );
}
```

Input: 1 2 3 4 5

Output:1 3 6 10 15

```
void foo( int n, int sum  )
{
     int data ;
     if ( n == 0 ) return 0;
     scanf(“%d”, &data);
     sum = sum + data;
     printf(“%d\n”, sum);
     foo( k – 1, sum ) ;
}
main ( ) {
     int k = 5;
     foo ( k, 0 );
}
```

Paying with fewest coins

- **A country has coins of denomination 3, 5 and 10, respectively.**
- **We are to write a function canchange(k) that returns –1 if it is not possible to pay a value of k using these coins.**
  - **Otherwise it returns the minimum number of coins needed to make the payment.**

- **For example, canchange(7) will return –1.**
- **On the other hand, canchange(14) will return 4 because 14 can be paid as 3+3+3+5 and there is no other way to pay with fewer coins.**
- **Finally, 15 can be changed as 3+3+3+3+3, 5+5+5, 5+10, so canchange(15) will return 2.**

Paying with fewest coins

```
int canchange( int k )
{
        int a;
        if (k==0) return 0;
        if (____________) return 1;
        if (k < 3)________________;

        a = canchange(__________); if (a > 0) return__________;
        a = canchange(k – 5); if (a > 0) return____________;
        a = canchange( __________); if (a > 0) return
        _______________________;  return –1;
}
```

Paying with fewest coins

```
int canchange( int k )
{
        int a;
        if (k==0) return 0;
        if ( (k ==3) || (k == 5) || (k == 10) )
        return 1;  if (k < 3) return –1  ;

        a = canchange( k – 10 ); if (a > 0)
        return a+1 ;  a = canchange( k – 5 );
        if (a > 0) return a+1 ;  a =
        canchange( k – 3 ); if (a > 0) return
        a+1 ;  return –1;
}
```

Exercise: Rewrite this code if the denominations are 3, 8, and 10. Do you see a problem? Repair it.

Practice Problems

1. Write a recursive function to search for an element in an array
2. Write a recursive function to count the digits of a positive integer (do also for sum of digits)
3. Write a recursive function to reverse a null-terminated string
4. Write a recursive function to convert a decimal number to binary
5. Write a recursive function to check if a string is a palindrome or not
6. Write a recursive function to copy one array to another

- Note:
  - **For each of the above, write the main functions to call the recursive function also**
  - **Practice problems are just for practicing recursion, recursion is not necessarily the most efficient way of doing them**