Recursion: Mechanisms

Paolo Camurati
Dip. Automatica e Informatica
Politecnico di Torino

The stack

- A stack in an Abstract Data Type (ADT) supporting the following operations
  - Push: Insert object on top
  - Pop: Read and delete from top the last-inserted object
- Terminology
  - This strategy is called LIFO (Last In First Out)

The stack

- The stack frame is the data structure containing at least
  - Formal parameters
  - Local variables
  - The return address when function execution is over
  - The pointer to the function’s code
  - The stack frame is created when the function is called and destroyed when it is over

The stack

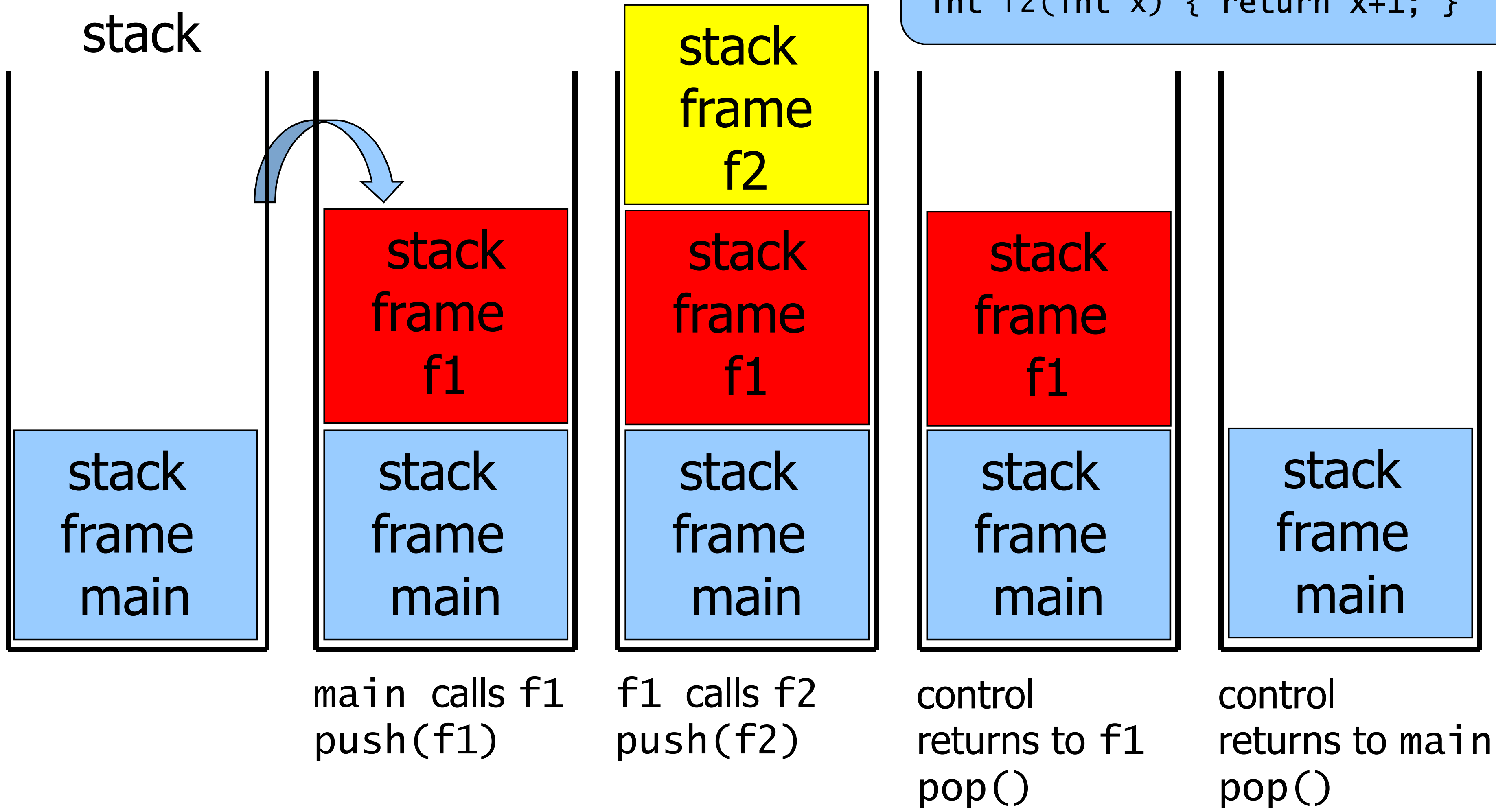
- Stack frames are stored in the system stack
- The system stack has a predefined amount of memory available
  - When it goes beyond the space allocated to it, a **stack overflow** occurs
- The stack grows from larger to smaller addresses (thus upwards)
- The **stack pointer SP** is a register containing the address of the first available stack frame

Example

```
int f1(int x);
int f2(int x);

main() {
    int x, a = 10;
    x = f1(a);
    printf("x is %d \n", x);
}

int f1(int x) { return f2(x); }

int f2(int x) { return x+1; }
```

Example

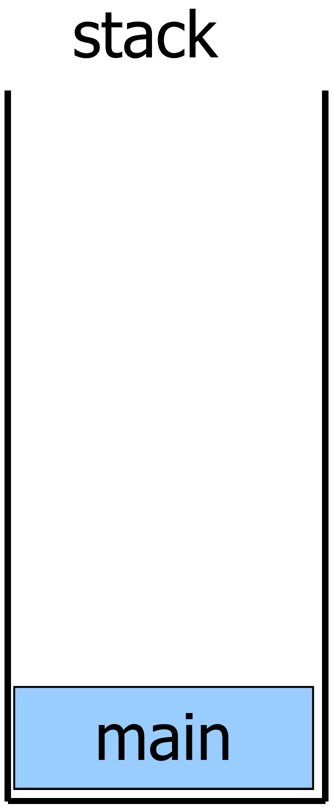
```
int f1(int x);
int f2(int x);
main() {
   int x, a = 10;
   x = f1(a);
   printf("x is %d \n", x);
}
int f1(int x) { return f2(x); }
int f2(int x) { return x+1; }
```



Recursive functions

- Calling and called functions coincide, but operate on different data
- The system stack is used **as** in any function call
- Too many recursive calls may result in stack overflow

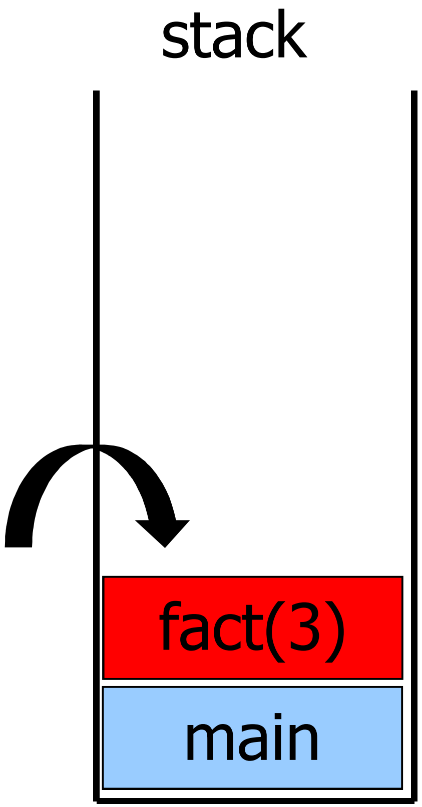
Example 1: Computing 3!



Initial configuration

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```

Example 1: Computing 3!



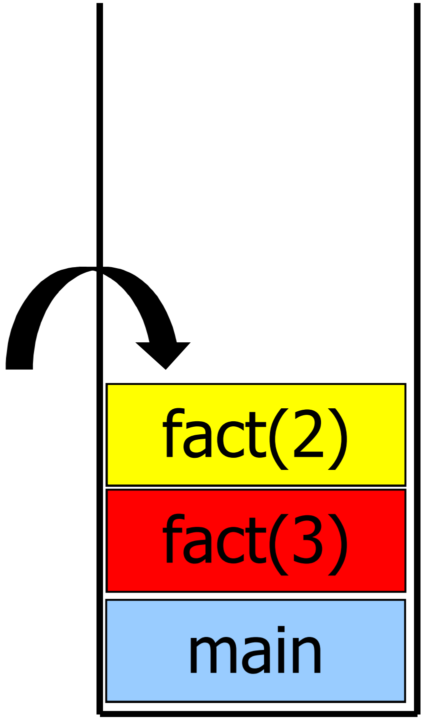
main calls
fact(3)

$3! = 3*2!$

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```

Example 1: Computing 3!

stack



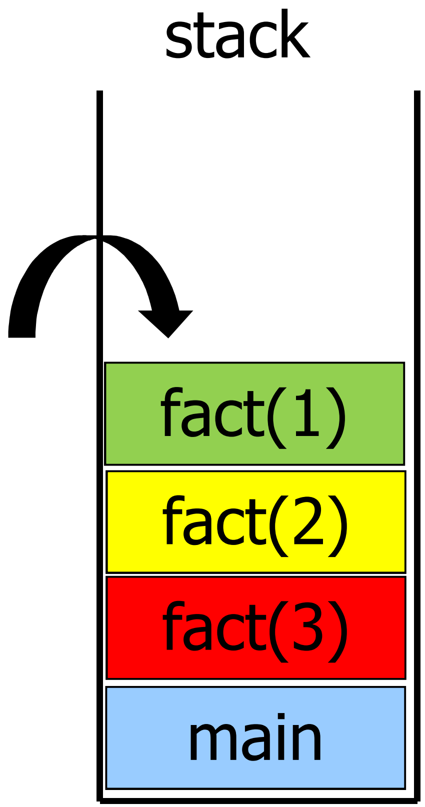
fact(3) calls fact(2)

3! = 3*2!

2! = 2*1!

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```

Example 1: Computing 3!



`fact(2) calls fact(1)`

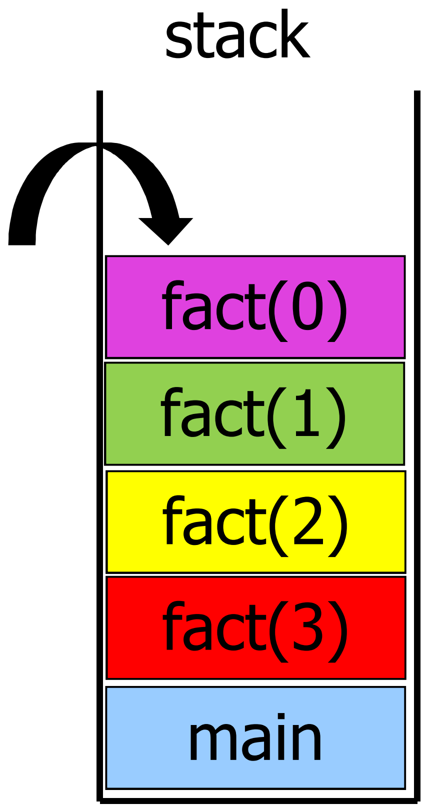
$3! = 3*2!$

$2! = 2*1!$

$1! = 1*0!$

```
main() {
   long n;
   printf("Input n:  ");
   scanf("%d", &n);
   printf("%d %d \n",n, fact(n));
}
long fact(long n) {
   if(n == 0)
     return(1);
   return(n * fact(n-1));
}
```

Example 1: Computing 3!



`fact(1) calls fact(0)`

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```

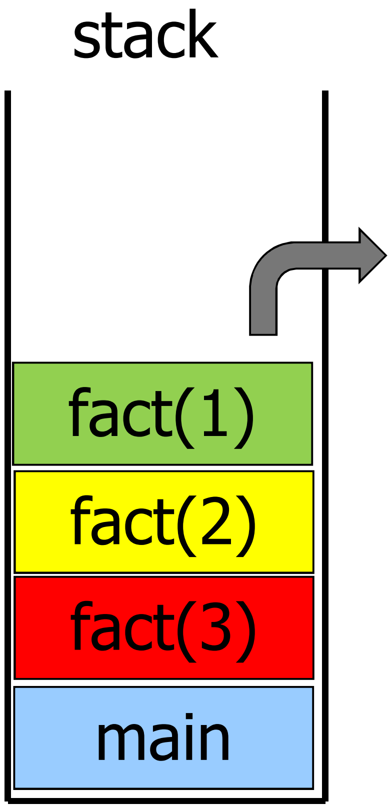
3! = 3*2!

2! = 2*1!

1! = 1*0!

0!

Example 1: Computing 3!



fact(0) terminates, returns value 1 and returns control to fact(1)

$3! = 3*2!$

$2! = 2*1!$

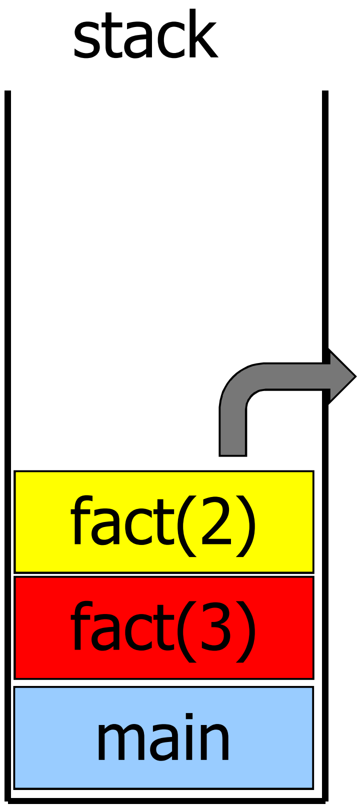
$1! = 1*0!$

$0! = 1$

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```

Example 1: Computing 3!

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```



3! = 3*2!

2! = 2*1!

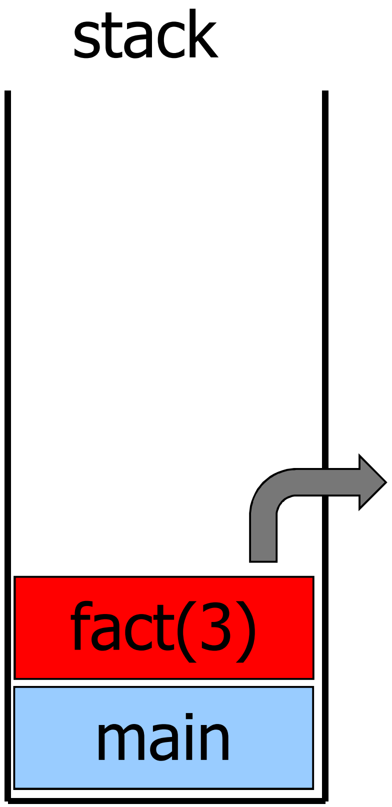
1! = 1*0! = 1

0! = 1

`fact(1) terminates, returns value 1 and  returns control to fact(2)`

Example 1: Computing 3!

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```



$3! = 3*2!$

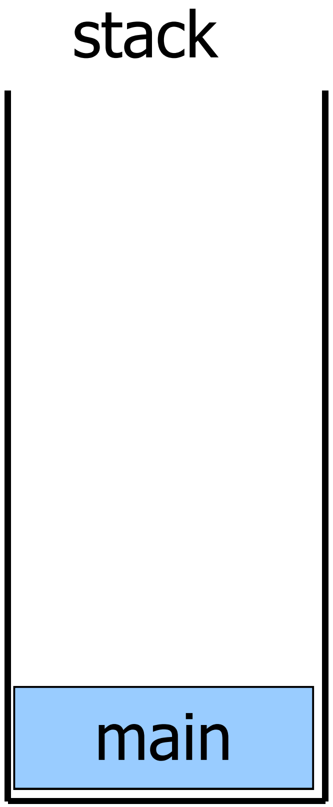
$2! = 2*1! = 2$

$1! = 1*0! = 1$

$0! = 1$

fact(2) terminates, returns value 2 and returns control to fact(3)

Example 1: Computing 3!



fact(3) terminates, returns value 6 and returns control to main

```
main() {
  long n;
  printf("Input n:  ");
  scanf("%d", &n);
  printf("%d %d \n",n, fact(n));
}
long fact(long n) {
  if(n == 0)
    return(1);
  return(n * fact(n-1));
}
```

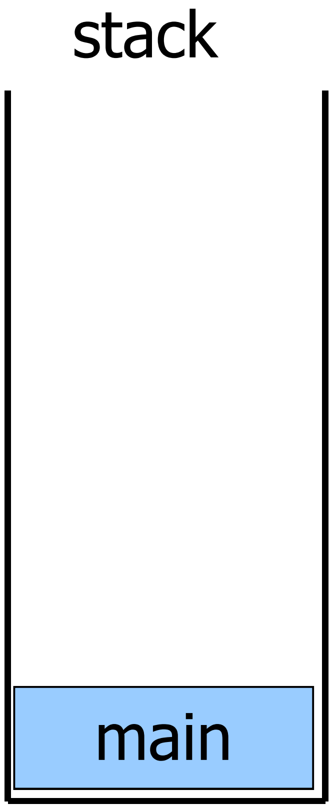
3! = 3*2! = 6

2! = 2*1! = 2

1! = 1*0! = 1

0! = 1

Example 2: reverse_print of “abc”



`Initial configuration`

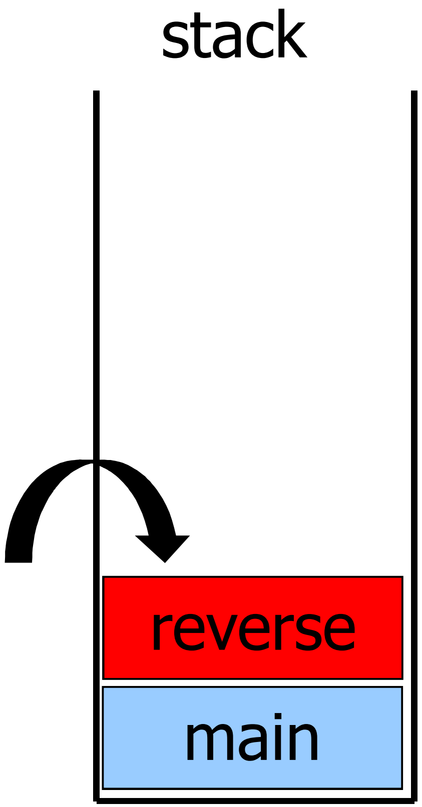
Input: “abc”

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

str

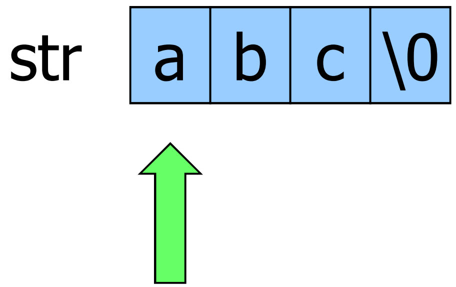
a	b	c	\0

Example 2: reverse_print of "abc"

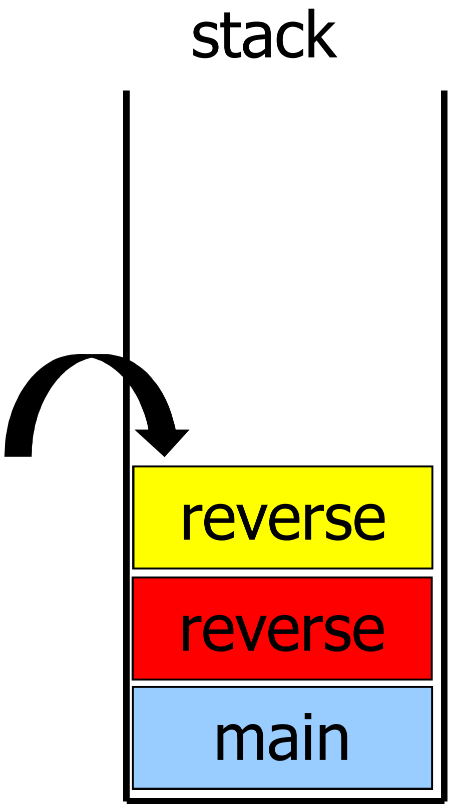



```
main calls
reverse_print
```

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

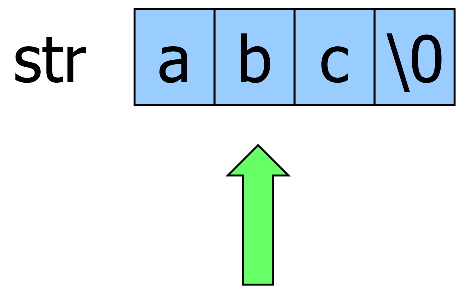


Example 2: reverse_print of “abc”

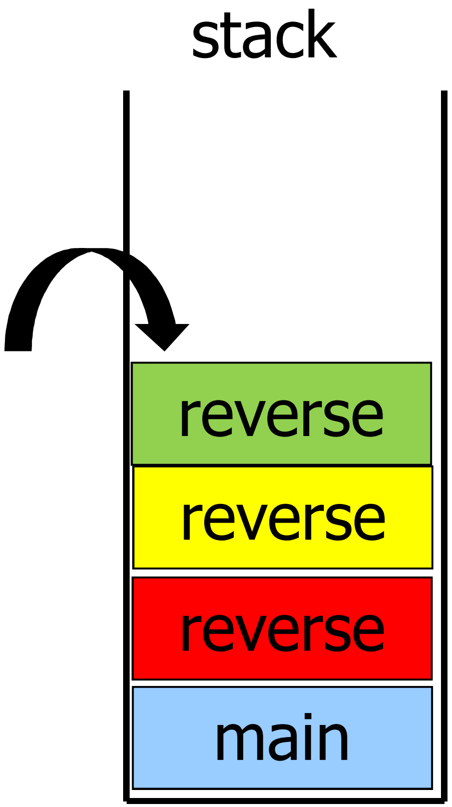


reverse_print calls itself

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

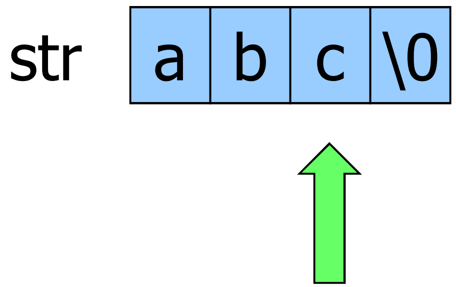


Example 2: reverse_print of "abc"

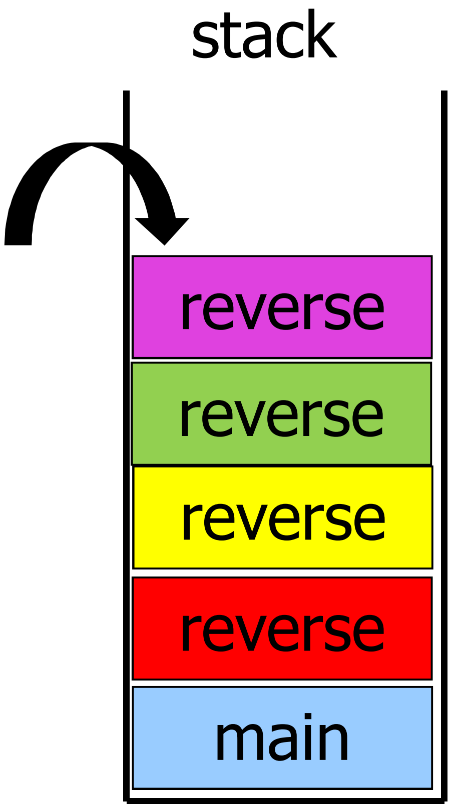


reverse_print calls itself

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

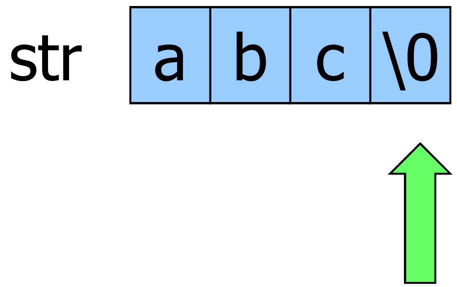


Example 2: reverse_print of “abc”



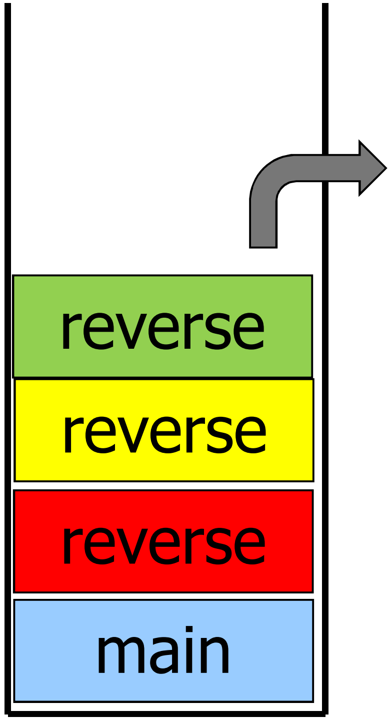
reverse_print
calls itself

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```



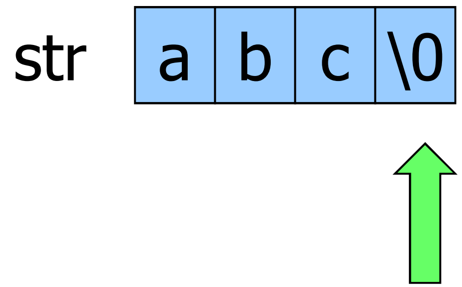
Example 2: reverse_print of “abc”

stack



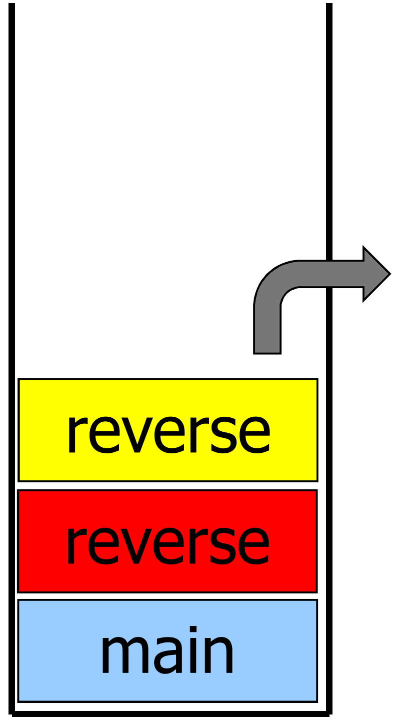
termination
condition
return

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```



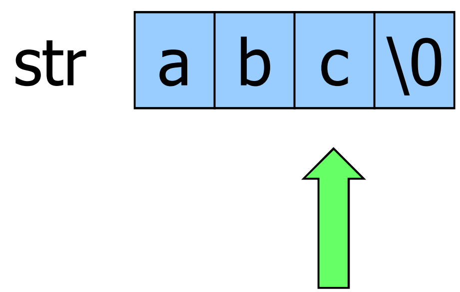
Example 2: reverse_print of "abc"

stack




```
print c
return
```

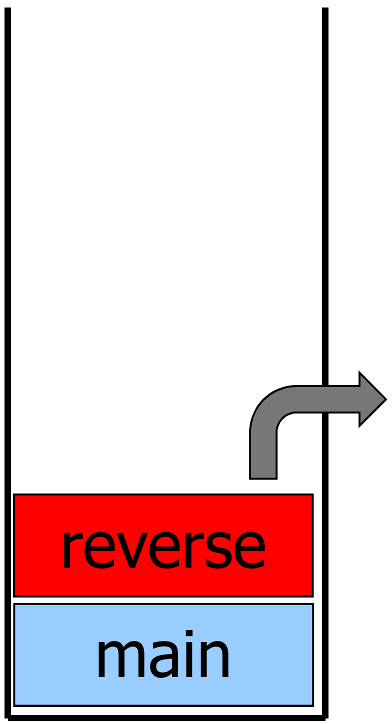
```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```



Output: "c"

Example 2: reverse_print of "abc"

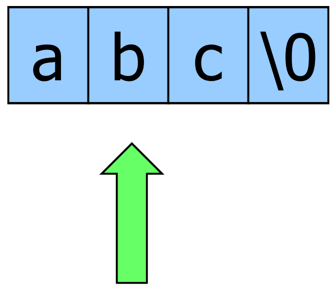
stack




```
print b
return
```

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

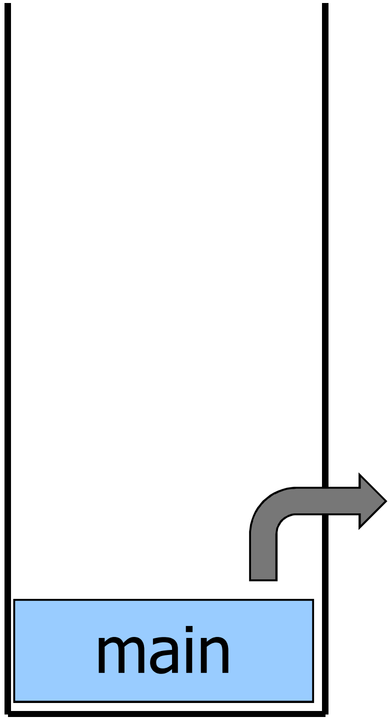
str



Output: "cb"

Example 2: reverse_print of “abc”

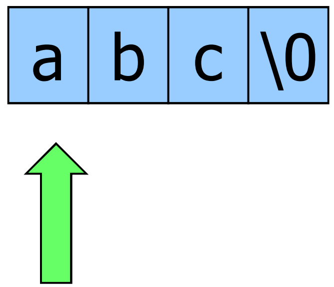
stack




```
print a
return
```

```
main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

str



Output: “cba”

Emulating recursion

- Recursion
  - May be memory-consuming
  - Is somehow equivalent to looping
- All recursive programs may be implemented in iterative form as well
  - There is a duality between recursion and iteration
- The best solution (efficiency and clarity of code) depends on the problem

Emulating recursion

- Recursion may be emulated dealing explicitly with a stack
- The best solution (efficiency and code clarity) depends on problem
- Try to remain at the highest possible abstraction level

User stack
Versus
System stack

Duality recursion - iteration

Factorial:

5! = 1*2*3*4*5 = 120

```
long fact(long n) {
  long tot = 1;
  int i;
  for (i=2; i<=n; i++)
    tot = tot * i;
  return(tot);
}
```

No stack needed!

Duality recursion - iteration

Fibonacci:

$FIB_0 = 0$

$FIB_1 = 1$

$FIB_2 = FIB_0 + FIB_1 = 1$

$FIB_3 = FIB_1 + FIB_2 = 2$

$FIB_4 = FIB_2 + FIB_3 = 3$

$FIB_5 = FIB_3 + FIB_4 = 5$

```
long fib(long n) {
  long f1p=1, f2p=0, f;
  int i;
  if(n == 0 || n == 1)
    return(n);
  f = f1p + f2p; /* n==2 */
  for(i=3; i<= n; i++) {
     f2p = f1p;
     f1p = f;
     f = f1p+f2p;
   }
 return(f);
}
```

No stack needed!

Duality recursion - iteration

Binary search:

```
int BinSearch
   (int v[], int l, int r, int k) {
          int m;
          while((r-l) != 0) {
             m = (l+r) / 2;
             if(v[m] >= k)
                r = m;
             else
         l = m+1;
             }
          if(v[l]==k)
      return(l);
          else
      return(-1);
}
```

No stack needed!

Emulating recursion with a user stack

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

```
long fact(long n) {
  long fact = 1;
  stack_t stack;
  stack = stack_init ();
  while (n>0) {
    stack_push (stack, n);
    n--;
  }
  while (stack_size (stack) > 0) {
    n = stack_pop (stack);
    fact = n * fact;
  }
  return fact;
}
```

ADT stack_t
(a stack)

Tail-recursive functions

- In traditional recursion recursive (traditional model)
  - Recursive salls are performed first
  - Then the return value is used to compute the result
  - The final result is obtained after all calls have terminated, i.e., the program has returned from every recursive call
- Tail-recursion (or tail-end recursion) is a particular case of recursion

Tail-recursive functions

In tail recursive function, the recursive call is the last operation to be executed, except for return

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

This function is not tail-recursive because the product can be executed only after returning from the recursive call

```
fact(3)
3 * fact(2)
3 * (2 * fact(1))
3 * (2 * (1 * fact(0)))
3 * (2 * (1 * 1))
```

A stack is needed

Tail-recursive functions

Tail-recursive version

```
long fact_r(long n, long f) {
  if (n == 0)
    return(f);
  return fact_tr(n-1, n*f);
}
```

This function is tail-recursive because the product is executed before the recursive call

```
fact_tr(3,1)
fact_tr(2,3)
fact_tr(1,6)
fact_tr(0,6)
```

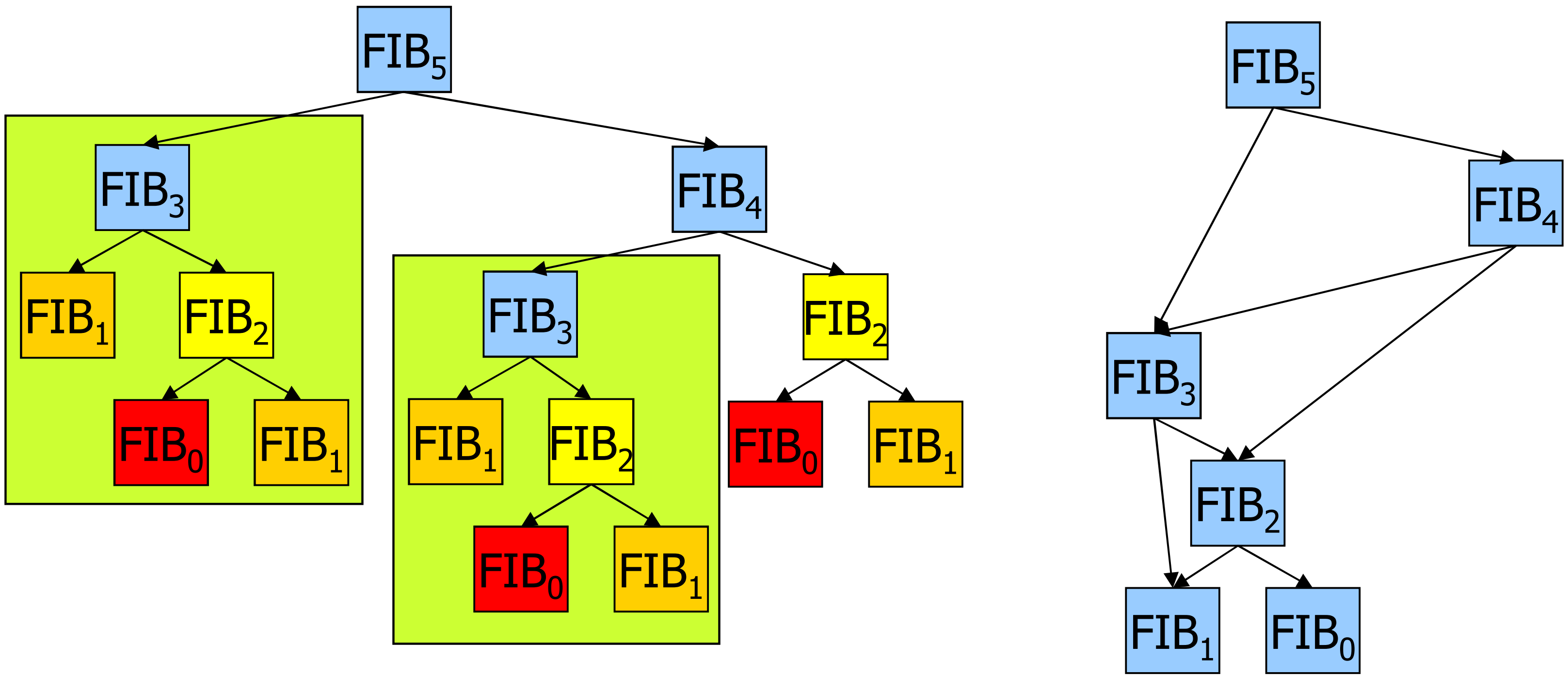
A stack is not necessary

Tail-recursive functions

- In tail recursive functions
  - Calculations are performed first
  - Recursive calls are done after
  - Current results are passed to future calls
  - Current stack frame is not needed anymore
    - Recursion can be substituted by a simple jump (**tail call elimination**)
    - A proper compiler or language (Prolog, Lisp, etc.) may recognize tail recursive functions and it may optimize their code
    - Stack overflows does not happen anymore

Limits of the divide and conquer paradigm

- Assumption: independent subproblems
- Memory occupation



Limits of the divide and conquer paradigm

- An alternative paradigm is Dynamic Programming
  - Stores solutions to subproblems as soon as they are found
  - Before it solves a subproblem, it checks whether it has already been solved
  - Better than divide and conquer for shared subproblems

Limits of the divide and conquer paradigm

- Dynamic Programming procedes
  - Bottom-up, whereas divide and conquer is top-down
  - Dynamic programming is called recursion with storage or memoization