Recursion

Mechanisms

Paolo Camurati and Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

The stack

The stack was previously introduced in the discrete mathematics unit

- In computer science a stack in an Abstract Data Type (ADT) that serves as a collection of elements
- A stack supports the following operations
  - Push
    - Insert object on top
  - Pop
    - Read and delete from top the last-inserted object
  - This reading/writing strategy is called LIFO (Last-In First-Out)

The stack

- A programmer can implement its own stacks
- The operating system (or any application) can use its own stack as well
  - At the C language level, the stack is the data structure containing at least
    - Formal parameters
    - Local variables
    - The return address when the function execution is over
    - The pointer to the function’s code

The stack

- All these pieces of data form a **stack frame**
  - A new stack frame is created when the function is called and the same stack frame is destroyed when the function is over
- 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

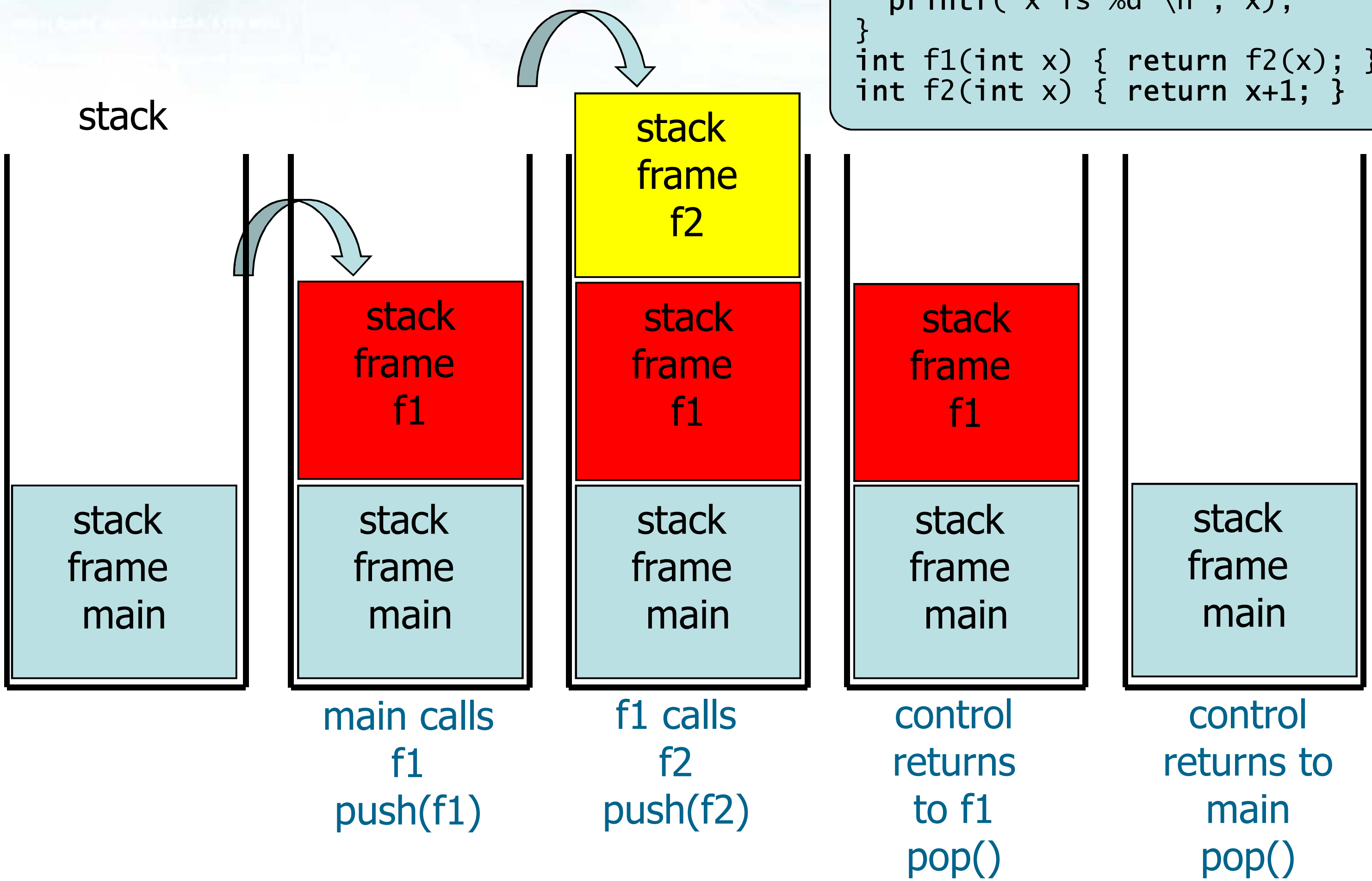
Example

- Let us analyze the stack structure during the execution of the following program

```
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;
}
```

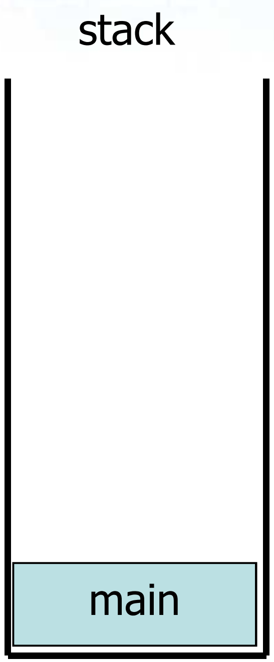
```
int f1(int x);
int f2(int x);
main() {
  int x, a = 10;
  x = f1(a);
  printf("x is %d \n", x);
}
int f1(int x) { return f2(x); }
int f2(int x) { return x+1; }
```



Recursive functions

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

Example 1



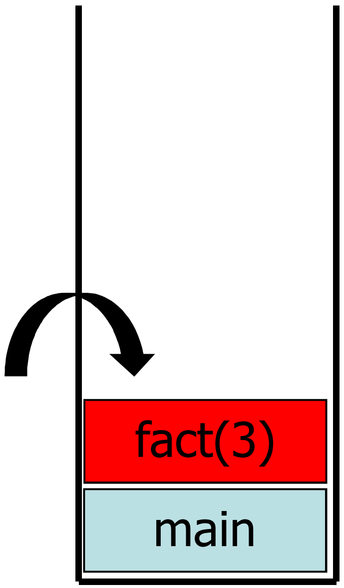
Initial configuration

```
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

```
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));
}
```

stack

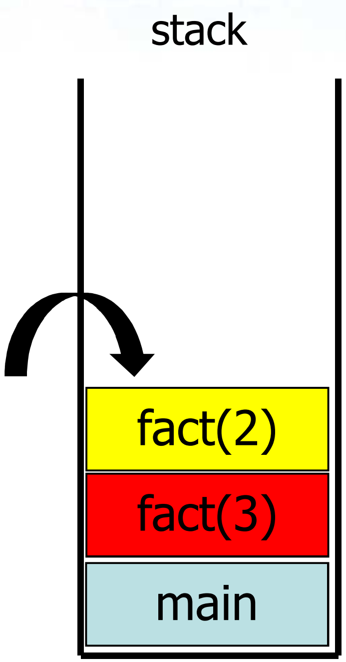


main calls fact(3)

3! = 3*2!

Example 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));
}
```



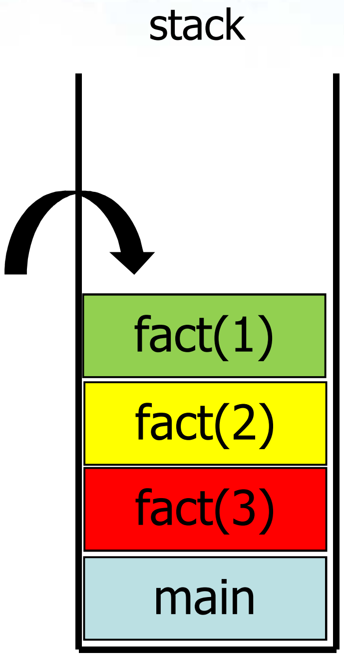
fact(3) calls fact(2)

$$3! = 3*2!$$

$$2! = 2*1!$$

Example 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));
}
```



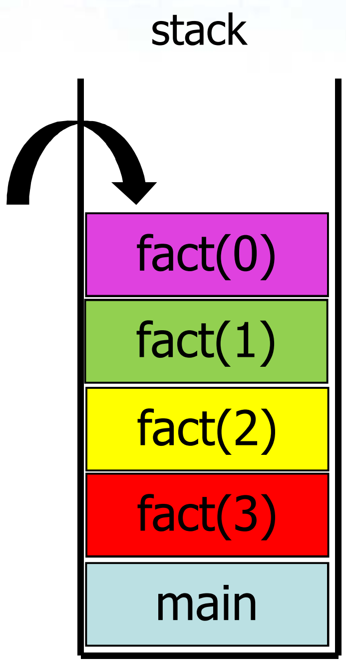
fact(2) calls fact(1)

$3! = 3*2!$

$2! = 2*1!$

$1! = 1*0!$

Example 1



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!

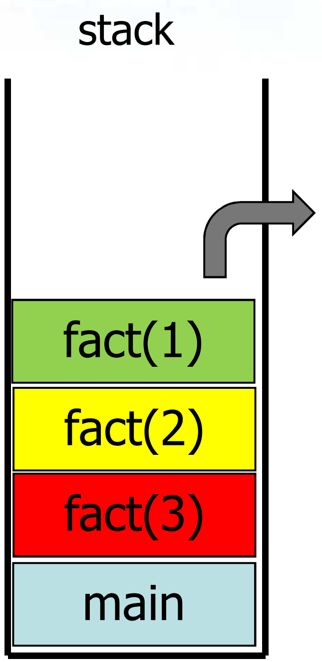
2! = 2*1!

1! = 1*0!

0! =

Example 1

```
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));
}
```



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

3! = 3*2!

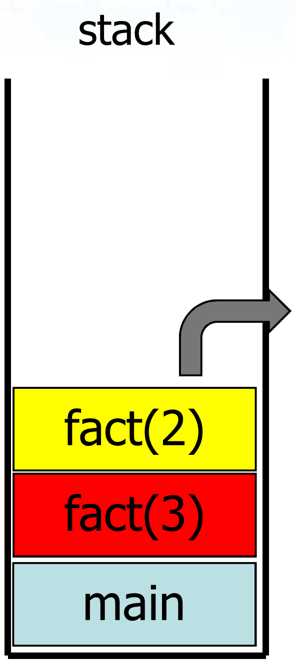
2! = 2*1!

1! = 1*0!

0! = 1

Example 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));
}
```



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

$3! = 3*2!$

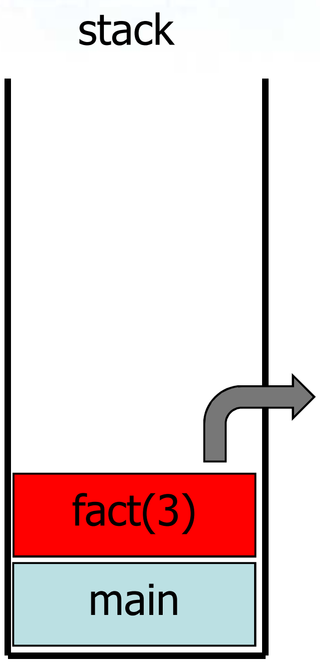
$2! = 2*1!$

$1! = 1*0! = 1$

$0! = 1$

Example 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));
}
```



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

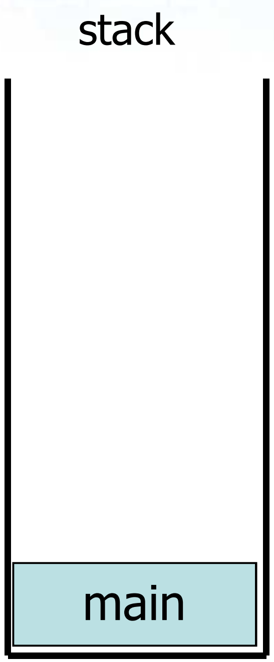
3! = 3*2!

2! = 2*1! = 2

1! = 1*0! = 1

0! = 1

Example 1



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

Recursion versus iteration

- 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
- Try to remain at the highest possible abstraction level

Duality recursion - iteration

- Factorial iterative computation
  - 5! = 1*2*3*4*5 = 120
  - The implementation may be iterative and recursive as well
  - There is no need to use a stack

```
long fact(long n) {
  long tot = 1;
  int i;

  for (i=2; i<=n; i++)
    tot = tot * i;

  return(tot);
}
```

Duality recursion - iteration

- Fibonacci iterative computation
  - 0 1 1 2 3 5 8 13 21 ...
    - F(0) = 0
    - F(1) = 1
    - F(2) = F(0) + F(1) = 1
    - F(3) = F(1) + F(2) = 2
    - F(4) = F(2) + F(3) = 3
    - F(5) = F(3) + F(4) = 5
  - The implementation may be iterative and recursive as well
  - There is no need to use a stack

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

Duality recursion - iteration

- Binary search
  - The implementation may be iterative and recursive as well
  - There is no need to use a stack

```
int BinarySearch (
  int v[], int l, int r, int k) {
  int c;

  while (l<=r){
    c = (int) ((l+r) / 2);
    if (k == v[c]) {
      return(c);
    }
    if (k < v[c]) {
      r = c-1;
    } else {
      l = c+1;
    }
  }
  return(-1);
}
```

Emulating recursion

- Recursion may be emulated explicitly dealing with a stack
  - Recursion is realized using the system stack to store and restore the local status
  - It is always possible to emulate recursion through iterations using a user-defined stack
    - The user stack mimics the system stack
    - It is manipulated by the programmer to store and restore information (function stack frames) as the system does into the system stack

Emulating recursion

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

Original recursive function

The ADT stack_t a user stack

```
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;
}
```

Non recursive (iterative) function emulating recursion using a user stack

Tail-recursive functions

- In traditional (model) recursive function
  - Recursive calls 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))
```

The system stack is required

Tail-recursive functions

- Tail-recursive version of the factorial function

```
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)
```

The system stack is **not** required

Tail-recursive functions

- In tail recursive functions
  - Calculations are performed first
  - Recursive calls are done after
  - Current results are passed to future calls
    - The return value of any given recursive step is the same as the return value of the next recursive call
    - The consequence of this is that once you are ready to perform your next recursive step, you do not need the current stack frame any more

Tail-recursive functions

- 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 and there is no limit to the number of recursive calls that can be made
- Tail recursion is essentially equivalent to looping
- Tail recursion only applies if there are no instructions that follow the recursive call

Solution

```
void print (char *s) {
  if (*s == '\0') {
    return;
  }
  printf (“%c”, *s);
  print (s+1);
  return;
}
```

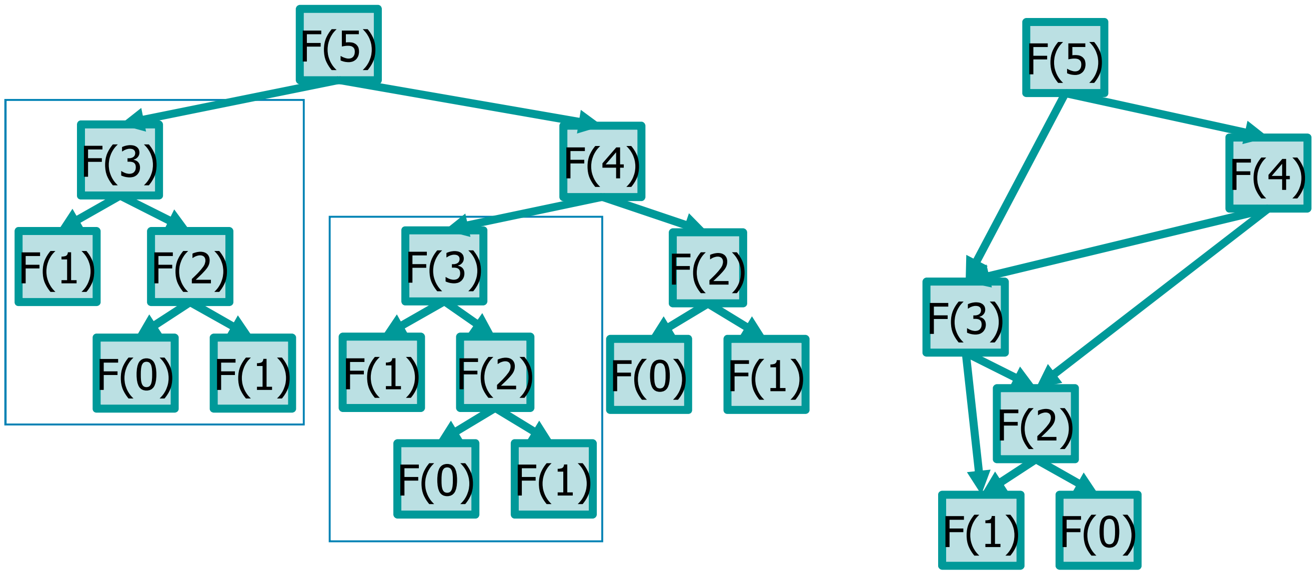
Printing a string:
There are no instructions that follow the recursive call.
The compiler may understand this and it may avoid the stack. This function is tail recursive.

```
void reverse_print (char *s) {
  if (*s == '\0') {
    return;
  }
  reverse_print(s+1);
  printf (“%c”, *s);
  return;
}
```

Reverse printing a string:
There are instructions that follow the recursive call.
The stack cannot be avoided.
This function is not tail recursive.

Limits of the recursion

- Disadvantages
  - The number of recursions is limited by the stack size
    - The stack consume memory
  - Sub-problems may not be independent, and recomputations may occur leading to inefficiency



Limits

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

Fibonacci with Dynamic Programming

How can we improve Fibonacci from $O(2^n)$ to $O(n)$?

```
int main (void) {
  int *known, i, n;
  fprintf(stdout, "Input n: ");
  scanf("%d", &n);
  known = (int *) malloc ((n+1)*sizeof(int));
  if (known==NULL){
    fprintf (stderr, "Memory allocation error.\n");
    exit(EXIT_FAILURE);
  }
  for (i=0; i<=n; i++) {
    known[i] = -1;
  }
  fprintf(stdout, "Fibonacci %d-th term = %d\n",
    n, fib_dp(n, known));
  free(known);
  return EXIT_SUCCESS;
}
```

We define an array **know**

Fibonacci with Dynamic Programming

```
int fib_dp (int n, int *known) {
  if (known[n] < 0) {
    if (n==0 || n==1) {
      known[n] = n;
    } else {
      known[n] = fib_dp (n-1, known) +
                 fib_dp (n-2, known);
    }
  }

  return known[n];
}
```

We avoid all recomputations

We store partial results into array **know**