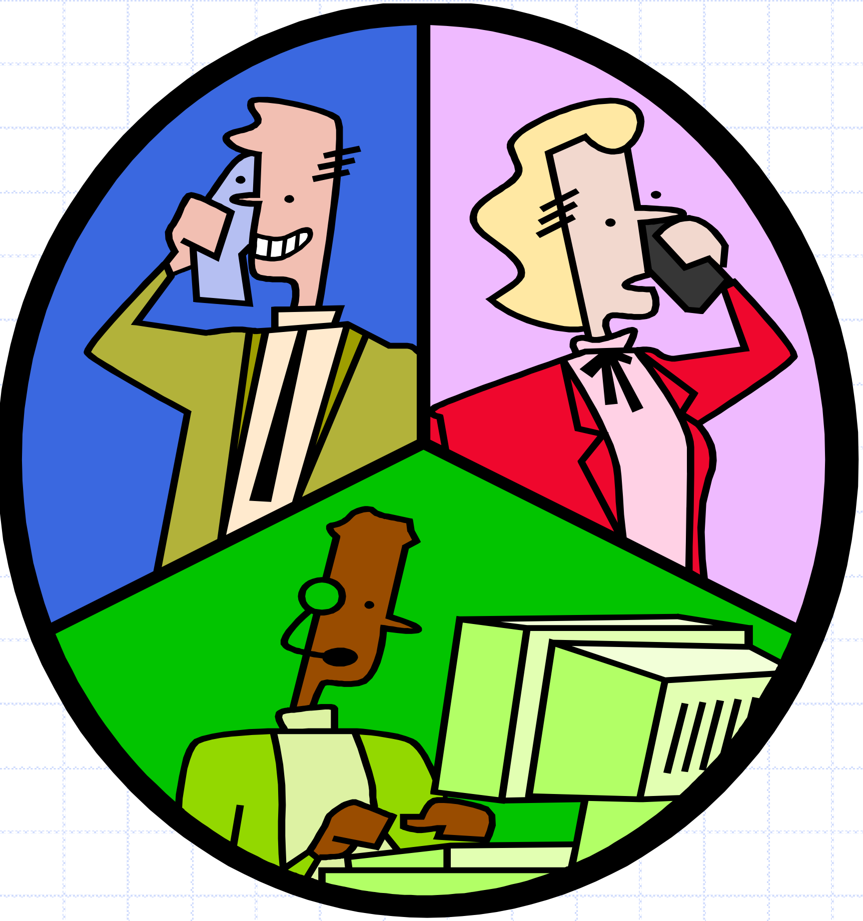
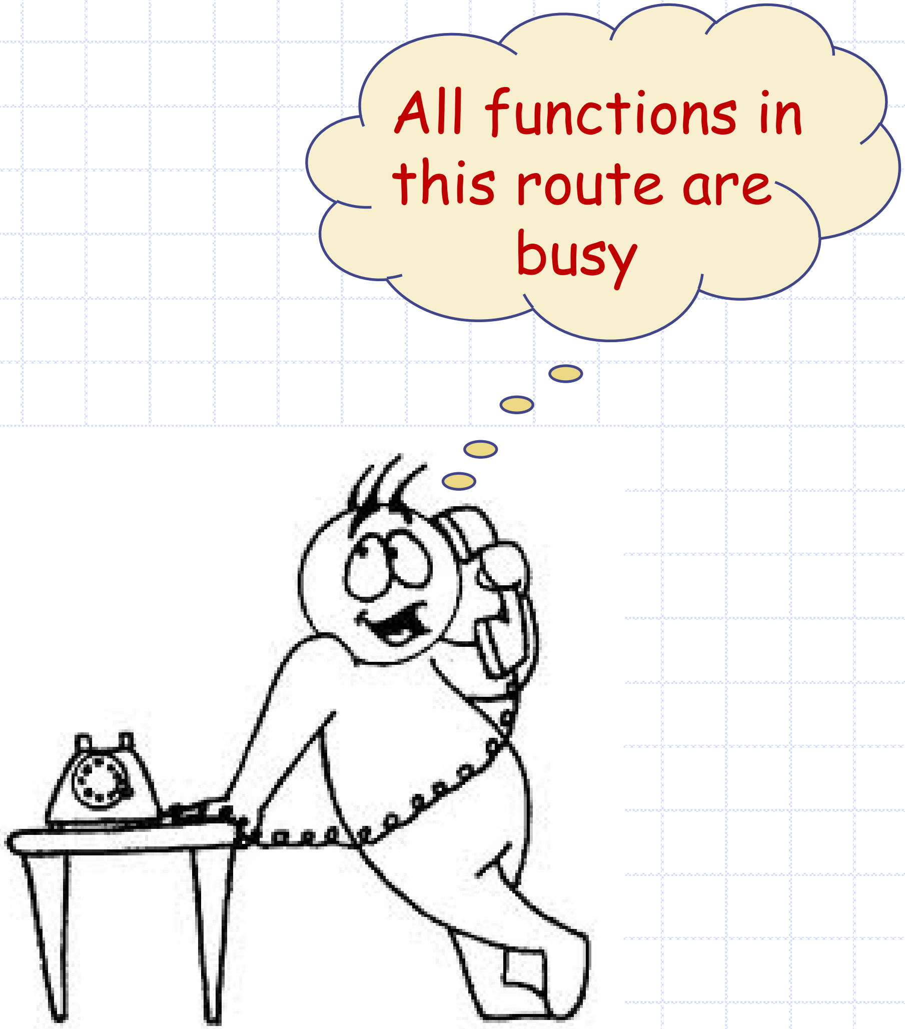
Returning from a function: return statement

- When a return statement is encountered in a function definition
  - control is immediately transferred back to the statement making the function call in the parent function.
- A function in C can return only ONE value or NONE.
  - Only one return type (including void)

Execution of a Function: Steps





```
1   #include <stdio.h>
2   int max(int a, int b) {
3      if (a > b)
4          return a;
5     else
6          return b;
7   }

8   int main () {
9          int x;
10         x = 10a max(6, 4);
11         printf("%d",x);
12         return 0;
13  }
```

- Steps when a function is called: max(6,4) in step 10a.
- Allocate space for (i) **return value**, (ii) **store return address** and (iii) **pass parameters.**

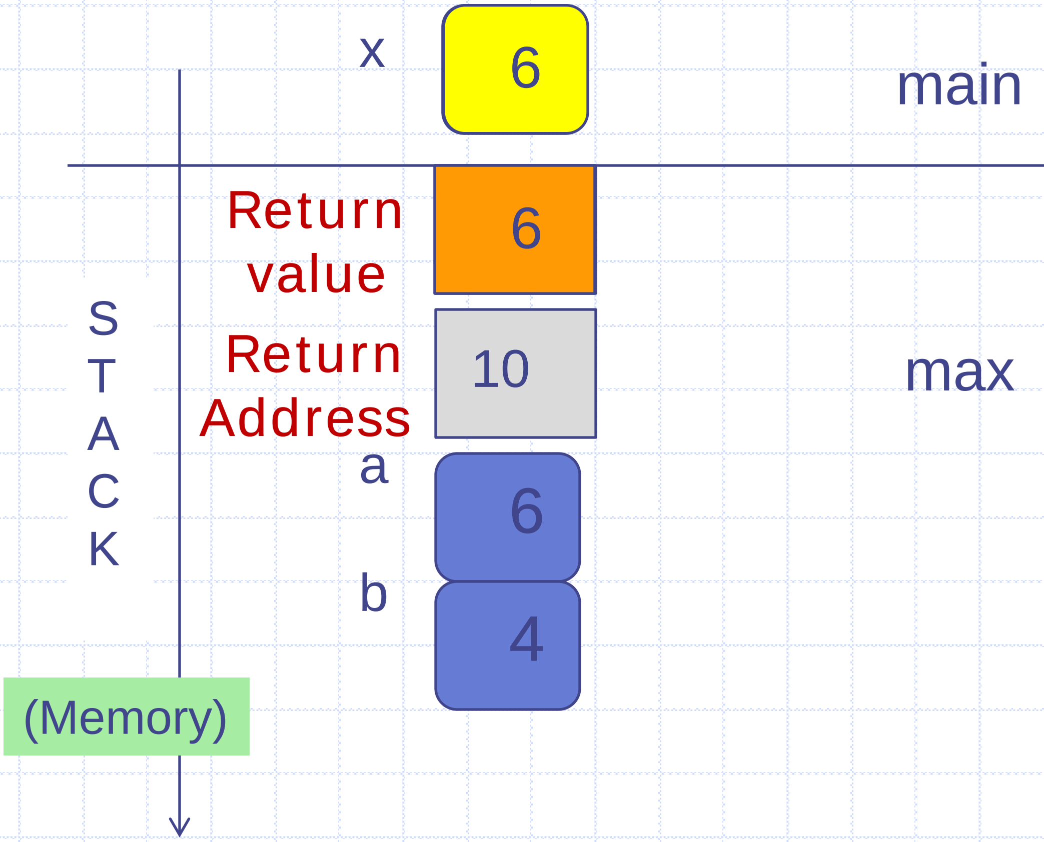
1. Create a box informally called ``Return value'' of same type as the return type of function.
2. Create a box and store the location of the next instruction in the calling function (main)—return address. Here it is 10 (why not 11?). Execution resumes from here once function terminates.
3. Parameter Passing- Create boxes for each formal parameter: a, b here. Initialize them using actual parameters, 6 and 4.

Calling max(6,4):

1. Allocate space for return value.
2. Store return address (10).
3. Pass parameters.

```
1   #include <stdio.h>
2   int max(int a, int b) {
3       if (a > b)
4           return a;
5       else
6           return b;
7   }

8   int main () {
9           int x = -1;
10          x      max(6, 4);
11          printf("%d",x);
12          return 0;
13  }
```



After completing max(), execution in main() will re-start from address 10.

Stack

- We referred to stack.
- A stack is just a part of the memory of the program that grows in one direction only.
- The memory (boxes) of all variables defined as actual parameters or local variables reside on the stack.
- The stack grows as functions call functions and shrinks as functions terminate.

Function Declaration- **Prototype**

- A function declaration is a statement that tells the compiler about the different properties of that function
  - name, argument types and return type of the function
- Structure:

```
return_type function_name (list_of_args);
```

- Looks very similar to the first line of a function definition, but NOT the same
  - has semicolon at the end instead of BODY

Function Declaration

```
return_type function_name (list_of_args);
```

- Examples:
  - int max(int a, int b);
  - int max(int x, int y);
  - int max(int , int);

All 3 declarations are equivalent! Since there is no BODY here, argument names do not matter, and are optional.

- Position in program: Before the call to the function
  - allows compiler to detect inconsistencies
  - Header files (stdio.h, math.h,...) contain declarations of frequently used functions
  - #include <...> just copies the declarations

Evaluating expressions

- Associativity and Precedence precisely define what an expression means, e.g.,

  a * b – c / d

  is same as:

  (a * b) – (c / d)

- But how exactly are expressions evaluated?

Evaluating expression (a*b) – (c/d)

- Computers evaluate one operator at a time.
- Above expr may be evaluated as

```
t1 = c/d;
t2 = a*b;
t3 = t2 – t1;
```

  - t1, t2, t3 are temporary variables, created by the compiler (not by programmer). They are temporary, meaning, their lifetime is only until use.

Evaluating expression (a*b) – (c/d)

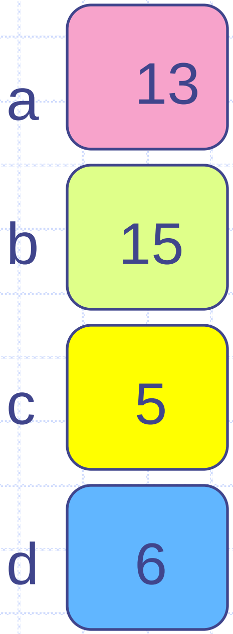
- The order of evaluation of arguments for most C operators is not defined
    - Exceptions are && and ||
    - Remember: short circuit evaluation
- Above expr may also be evaluated as

```
t1 = a*b;
t2 = c/d;
t3 = t1 – t2;
```

    - Do not depend on order of evaluation of args, except for && and ||
    - Same holds true for function arguments

```
int main () {
    int a =4, b = 2, c = 5, d = 6;
    a = a + b *c-d/a;
    b = b-(c-d)*a;
    printf(“%d %d”,a,b);
    return 0; }
```

State of the program just prior to printf


a 13

b 15

c 5

d 6

Output

```
13  15
```

Since each function call is also an expression, statements having mix of function calls and operators are also evaluated in a similar fashion.

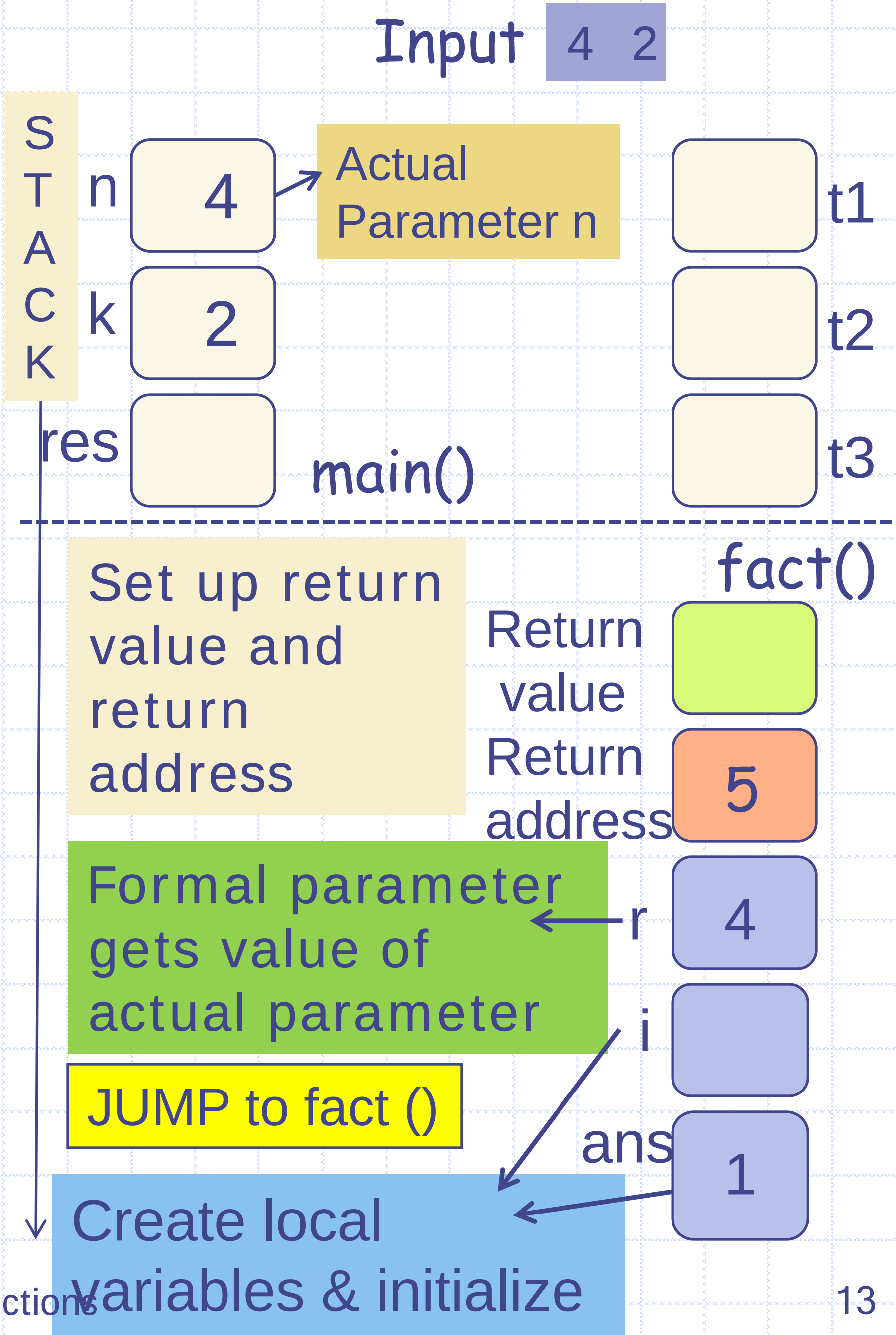
```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
     int i;
     int  ans=1;
     for (i=0; i < r; i=i+1) {
           ans = ans *(i+1);
     }
     return ans;
}
```

```
int main () {
     int n, k;
     int res;
     scanf(“%d%d”,&n,&k);
     res = (fact(n)/ fact(k))/fact(n-k);
     printf(“%d choose %d is”,n,k);
     printf(“%d\n”,res);
     return 0;
}
```

- Define a factorial function.
- Use to calculate ^nC_k
- Let us trace the execution of main().
- Add temporary variables for expressions and intermediate expressions in main for clarity.

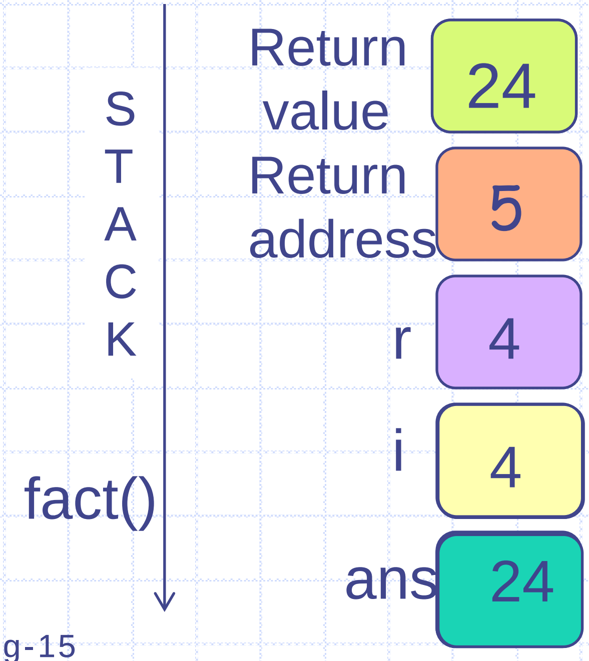
```
int fact(int r) {   /* calc. r! */
int i; int ans =1 ;
   /*  code here */
}
```

```
int main () {
     int n, k;
     int res;
     scanf("%d%d",&n,&k);
/* Adding temporary vars
and Intermediate exprs. */
          int          t1, t2, t3;
          t1 =         fact(n);
          t2 =         fact(k):
          t3 =         fact(n-k);
          res =    (t1/t2)/t3;
     printf("%d choose %d is");
     printf("%d\n", res);
     return 0;
}
```


Input 4 2
STACK
n 4
k 2
res
Actual Parameter n
t1
t2
t3
main()
fact()
Set up return value and return address
Return value
Return address 5
Formal parameter gets value of actual parameter
r 4
i
ans 1
JUMP to fact ()
Create local variables & initialize

```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
     int i;
     int  ans=1  ;
     for ( i=0;   i < r;    i=i+1) {
        ans = ans *(i+1);
     }
     return ans;
}
```

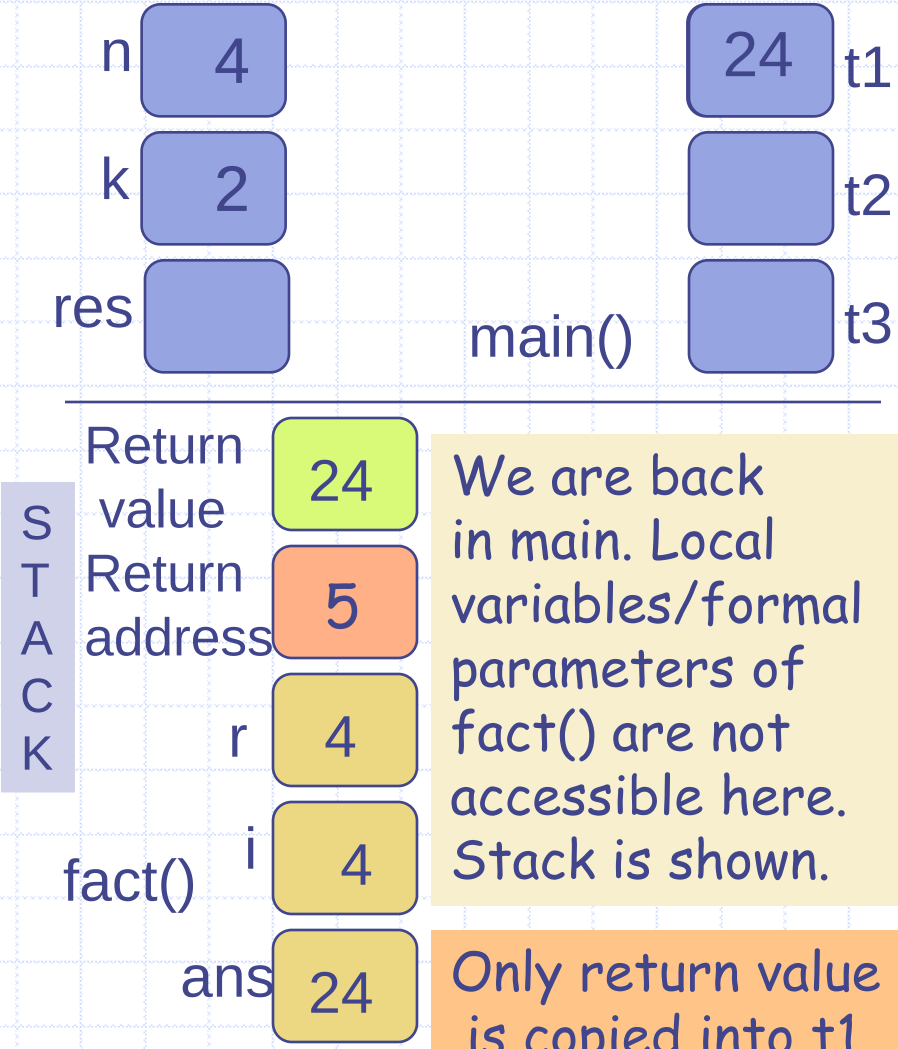
We have jumped to fact() and prepared the stack for the call. Parameters are passed, return addr is stored and local variables are initialized. Now we are ready to execute.



Assign to return value

Now jump to return address

```
1  int main () {
2       int n, k;
3       int res;
4       scanf("%d%d",&n,&k);
   /* Adding temporary vars
    and Intermediate exprs. */
              int        t1, t2, t3;
5             t1 =       fact(n);
6             t2 =       fact(k):
7             t3 =       fact(n-k);
8             res =   (t1/t2)/t3;
        printf("%d choose %d is");
9       printf("%d\n",res);
        return 0;
   }
```



We are back in main. Local variables/formal parameters of fact() are not accessible here. Stack is shown.

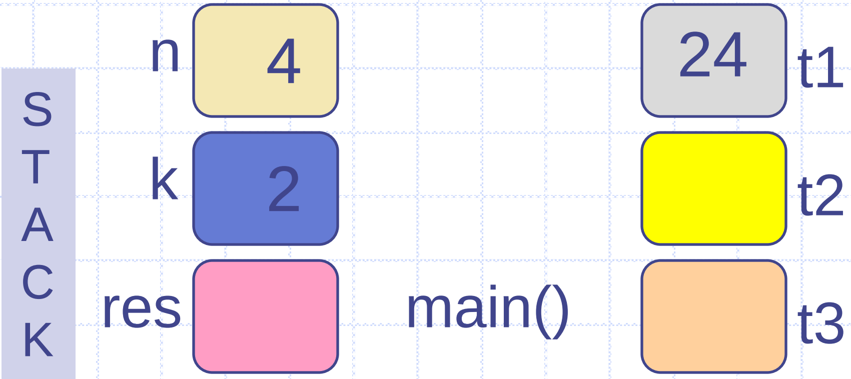
Only return value is copied into t1.

Control jumps to statement 5, since that was the return address.

After copying return value, assume that the stack for fact() is wiped clean.

```
int main () {
      int n, k;
      int res;
      scanf("%d%d",&n,&k);
/* Adding temporary vars
 and Intermediate exprs. */
           int         t1, t2, t3;
           t1  =       fact(n);
           t2  =  ⇨    fact(k):
           t3  =       fact(n-k);
           res  =    (t1/t2)/t3;
      printf("%d choose %d is");
      printf("%d\n",res);
      return 0;
}
```

After copying return value, assume that the stack for fact() is wiped clean.



STACK

n 4

k 2

res

main()

24 t1

t2

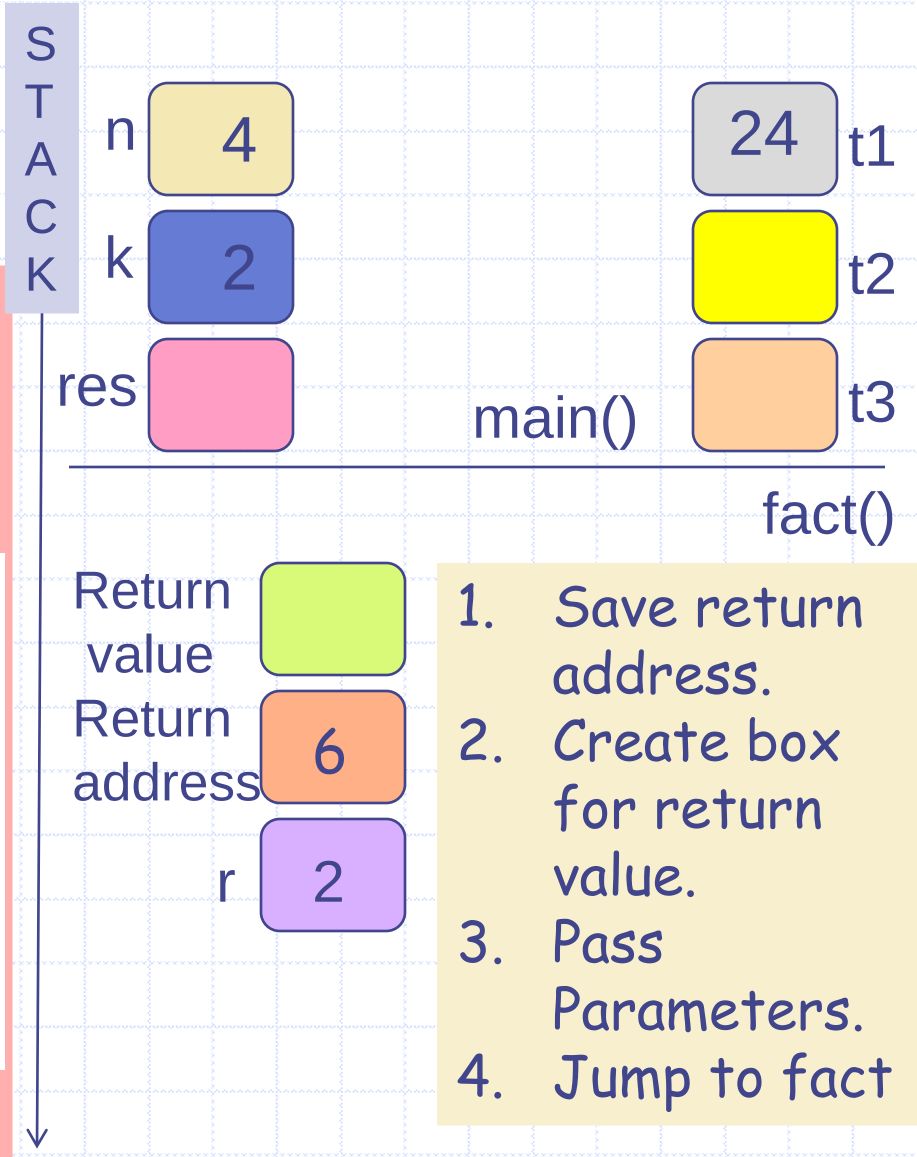
t3

The next statement is another function call: fact(k). Prepare stack for new call.

1. Save return address.
2. Create box for return value.
3. Pass Parameters: Create boxes corresponding to formal parameters. Copy values from actual parameters.
4. Jump to called function.
5. Create/initialize local variables.

```
int fact(int r) {   /* calc. r! */
    int i; int ans =1 ;
      /*  code here */
}
```

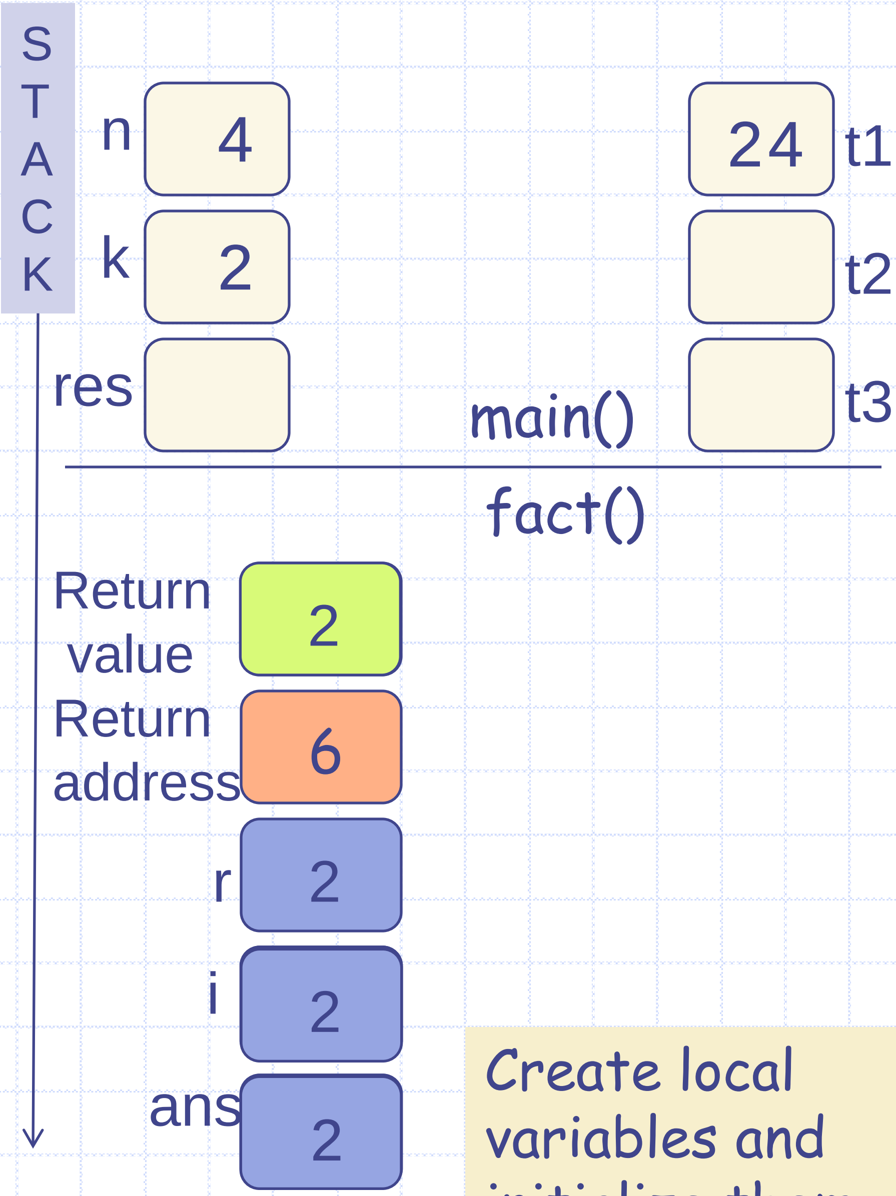
```
1  int main () {
2       int n, k;
3       int res;
4       scanf("%d%d",&n,&k);
   /* Adding temporary vars
   and Intermediate exprs. */
            int      t1, t2, t3;
5           t1 =     fact(n);
6           t2 =     fact(k):
7           t3 =     fact(n-k);
8           res =   (t1/t2)/t3;
        printf("%d choose %d is");
9       printf("%d\n",res);
        return 0;
   }
```



1. Save return address.
2. Create box for return value.
3. Pass Parameters.
4. Jump to fact

```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
     int i;
     int  ans=1;

     for (i=0;  i < r; i=i+1) {
           ans = ans *(i+1);
     }
     return ans;
}
```

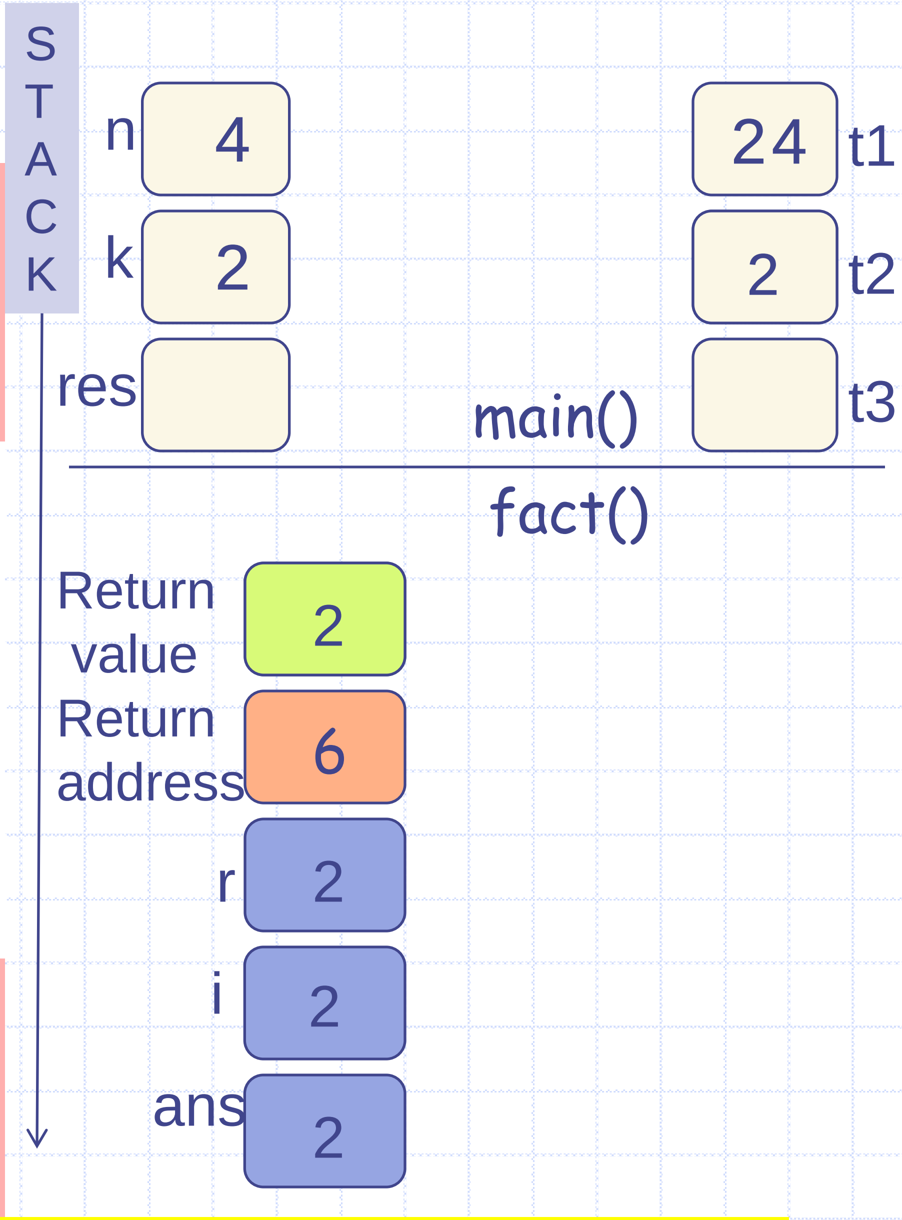


Assign value of ans to box for return value.

Create local variables and initialize them

Now jump to return address

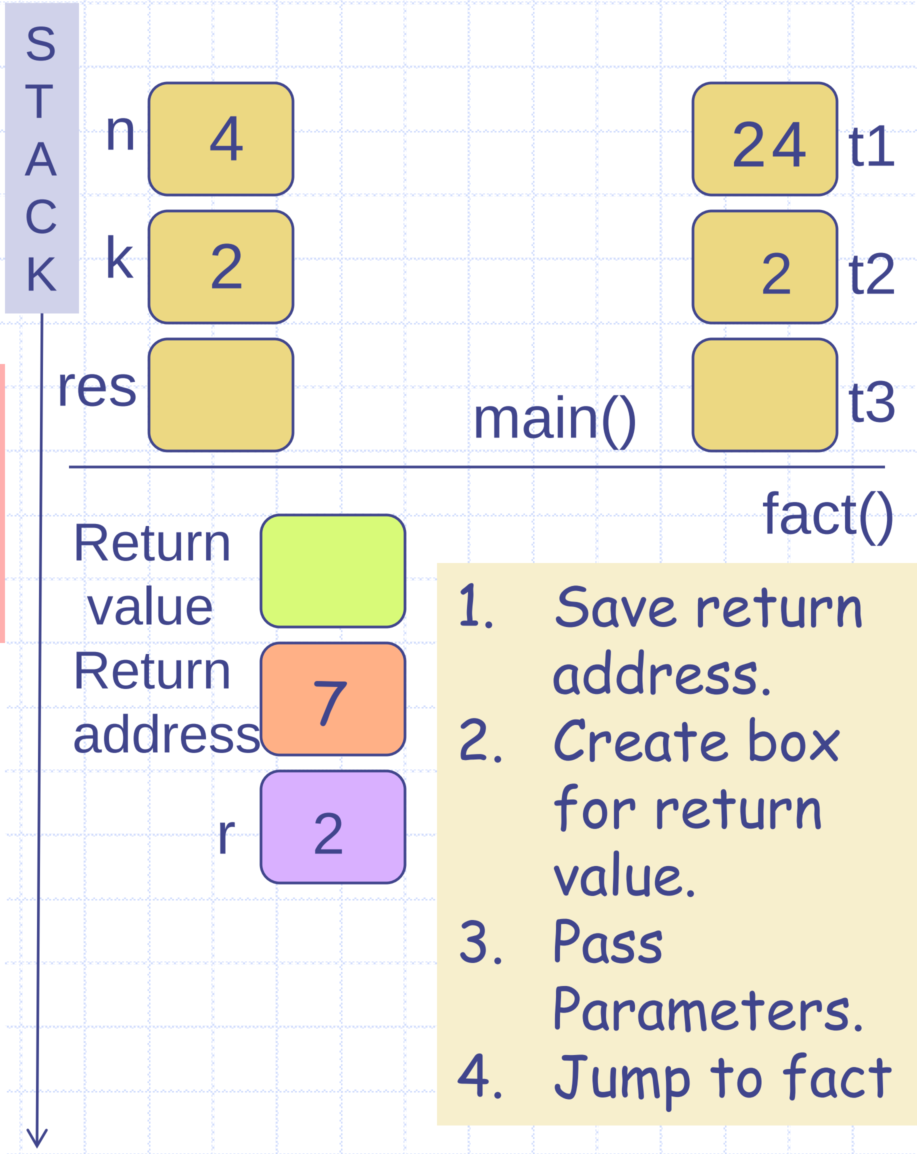
```
1 int main () {
2       int n, k;
3       int res;
4       scanf(“%d%d”,&n,&k);
/* Adding temporary vars
and Intermediate exprs. */
             int      t1, t2, t3;
5            t1 =     fact(n);
6            t2 =     fact(k):
7            t3       fact(n-k);
8            res =  (t1/t2)/t3;
        printf(“%d choose %d is”);
9       printf(“%d\n”,res);
        return 0;
}
```



This is another function call. So we prepare stack. Earlier entries for fact() is erased.

```
int fact(int r) {   /* calc. r! */
    int i; int ans =1 ;
      /*  code here */
}
```

```
1 int main () {
2       int n, k;
3       int res;
4       scanf(“%d%d”,&n,&k);
  /* Adding temporary vars
   and Intermediate exprs. */
           int        t1, t2, t3;
5          t1 =       fact(n);
6          t2 =       fact(k):
7          t3         fact(n-k);
8          res =    (t1/t2)/t3;
        printf(“%d choose %d is”);
9       printf(“%d\n”,res);
        return 0; }
```



1. Save return address.
2. Create box for return value.
3. Pass Parameters.
4. Jump to fact

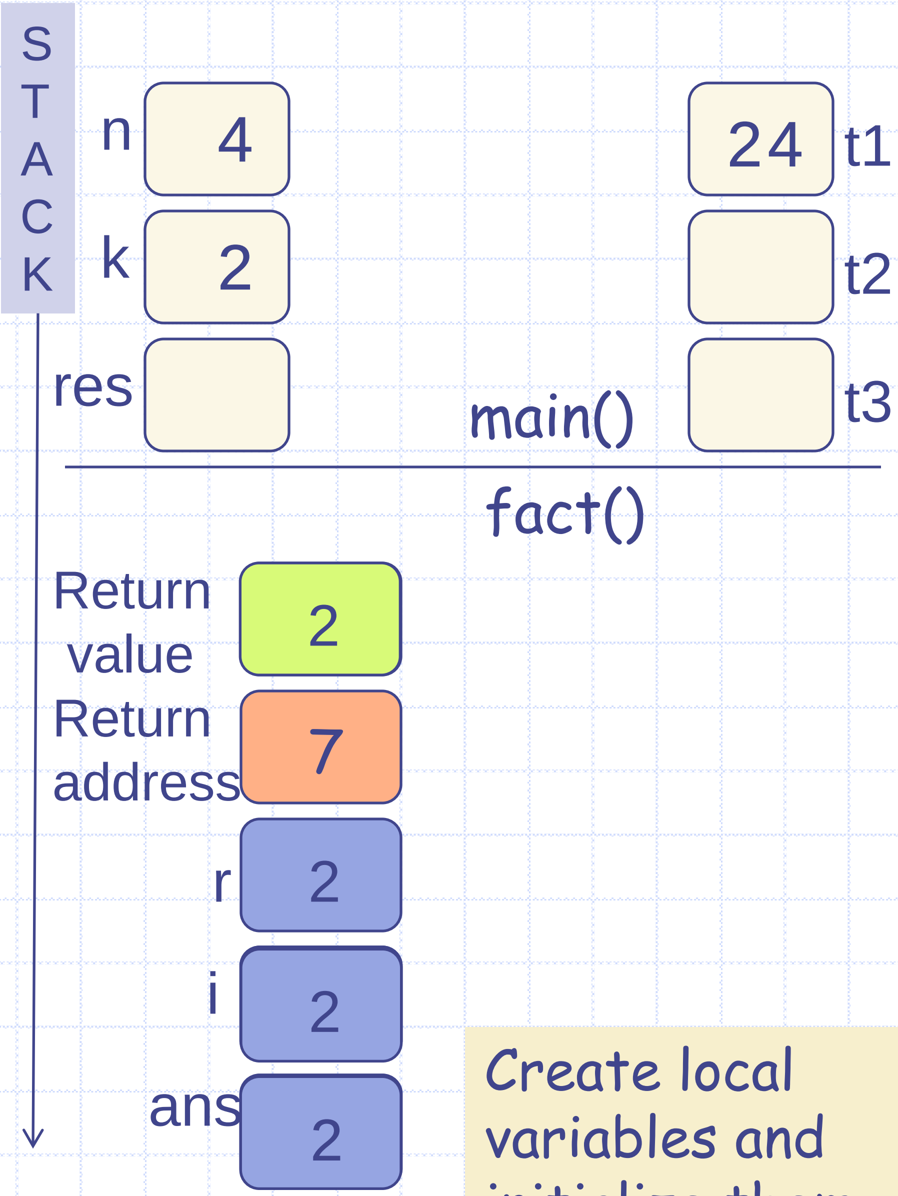
Previous stack for fact() has been erased.

```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
     int i;
     int  ans=1;

     for (i=0;  i < r; i=i+1) {
       ans = ans *(i+1);
     }
     return ans;
}
```

Assign value of ans to box for return value.

Now jump to return address



STACK

n	4	24	t1
k	2		t2
res			t3

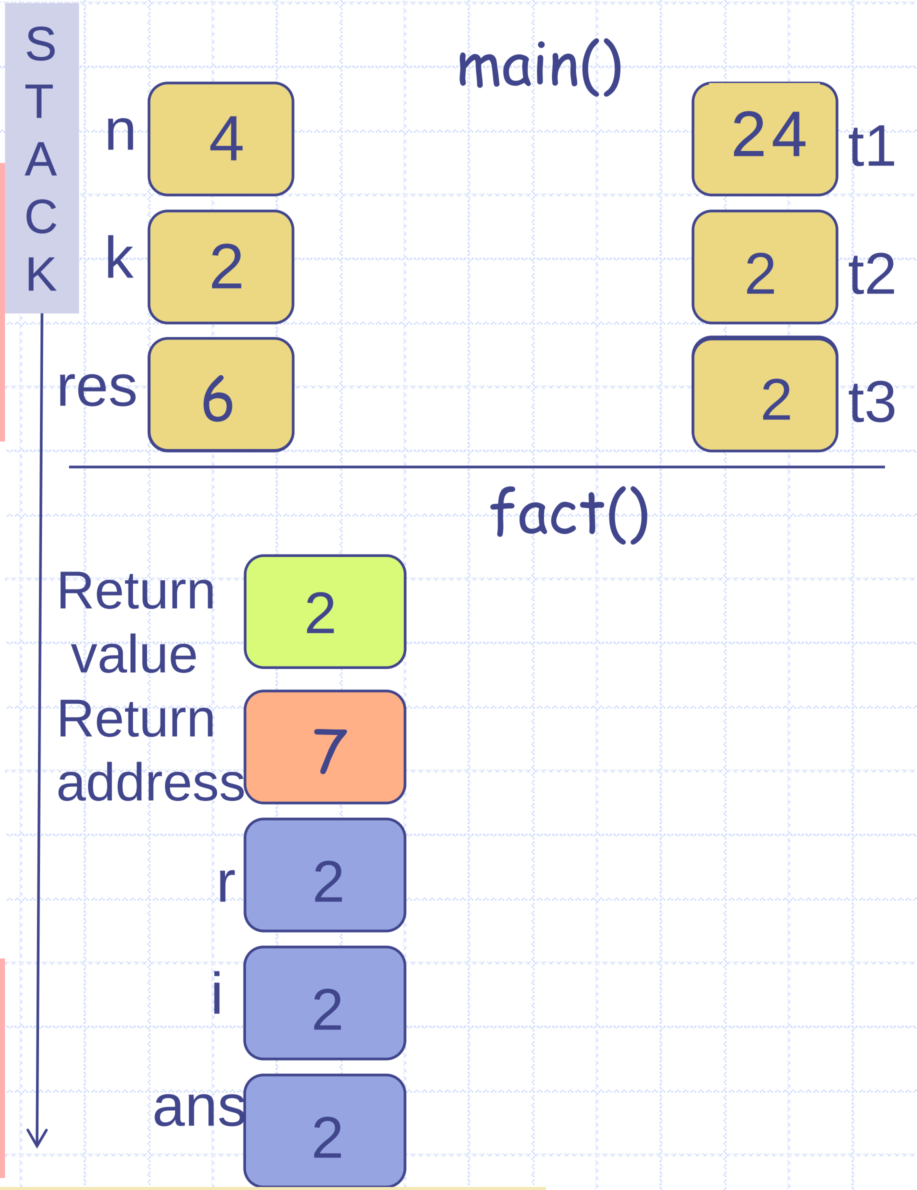
main()

fact()

Return value	2
Return address	7
r	2
i	2
ans	2

Create local variables and initialize them

```
1 int main () {
2       int n, k;
3       int res;
4       scanf("%d%d",&n,&k);
  /* Adding temporary vars
   and Intermediate exprs. */
          int       t1, t2, t3;
5         t1 =      fact(n);
6         t2 =      fact(k);
7         t3 =      fact(n-k);
8         res =   (t1/t2)/t3;
        printf("%d choose %d is ");
9       printf("%d\n",res);
        return 0; }
```



STACK

main()

n 4

k 2

res 6

24 t1

2 t2

2 t3

fact()

Return value 2

Return address 7

r 2

i 2

ans 2

Stack for fact() is erased after return value is copied.

4 choose 2 is 6

Nested Function Calls

- Functions can call each other
- A declaration or definition (or both) must be visible before the call
  - Help compiler detect any inconsistencies in function use
  - Compiler error, if both (decl & def) are missing

```
#include<stdio.h>
int min(int, int); //declaration
int max(int, int); //of max, min

int max(int a, int b) {
   return (a > b) ? a : b;
}

// a “cryptic” min, uses max
int min(int a, int b) {
   return a + b – max (a, b);
}

int main() {
  printf(“%d”, min(6, 4));
}
```