ESC101: Introduction to Computing

f(unction)

Function Call

- Since a function call is an *expression*
  - it can be used anywhere an expression can be used
  - subject to type restrictions

`printf(“%d”, max(5,3));`	prints 5
`max(5,3) – min(5,3)`	evaluates to 2
`max(x, max(y, z)) == z`	checks if z is max of x, y, z
`if (max(a, b)) printf(“Y”);`	prints Y if max of a and b is not 0.

Returning from a function: Type

- Return type of a function tells the type of the result of function call
- Any valid C type
  - int, char, float, double, …
  - **void**
- Return type is void if the function is not supposed to return any value

```
void print_one_int(int n) {
    printf(“%d”, n);
}
```

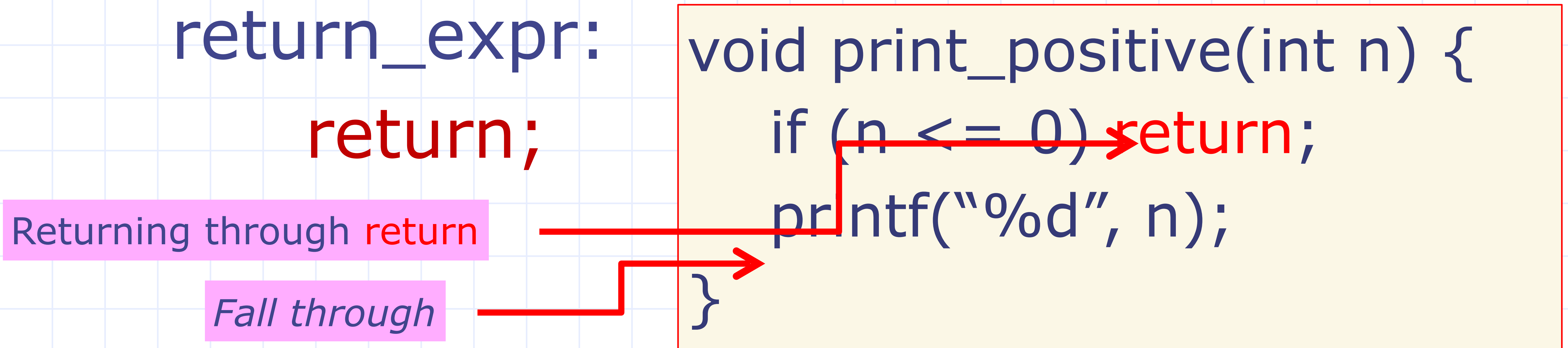
Returning from a function: return statement

- If return type is not void, then the function should return a value:

  return return_expr;

- If return type is void, the function may *fall through* at the end of the body or use a return without return_expr:

  return;

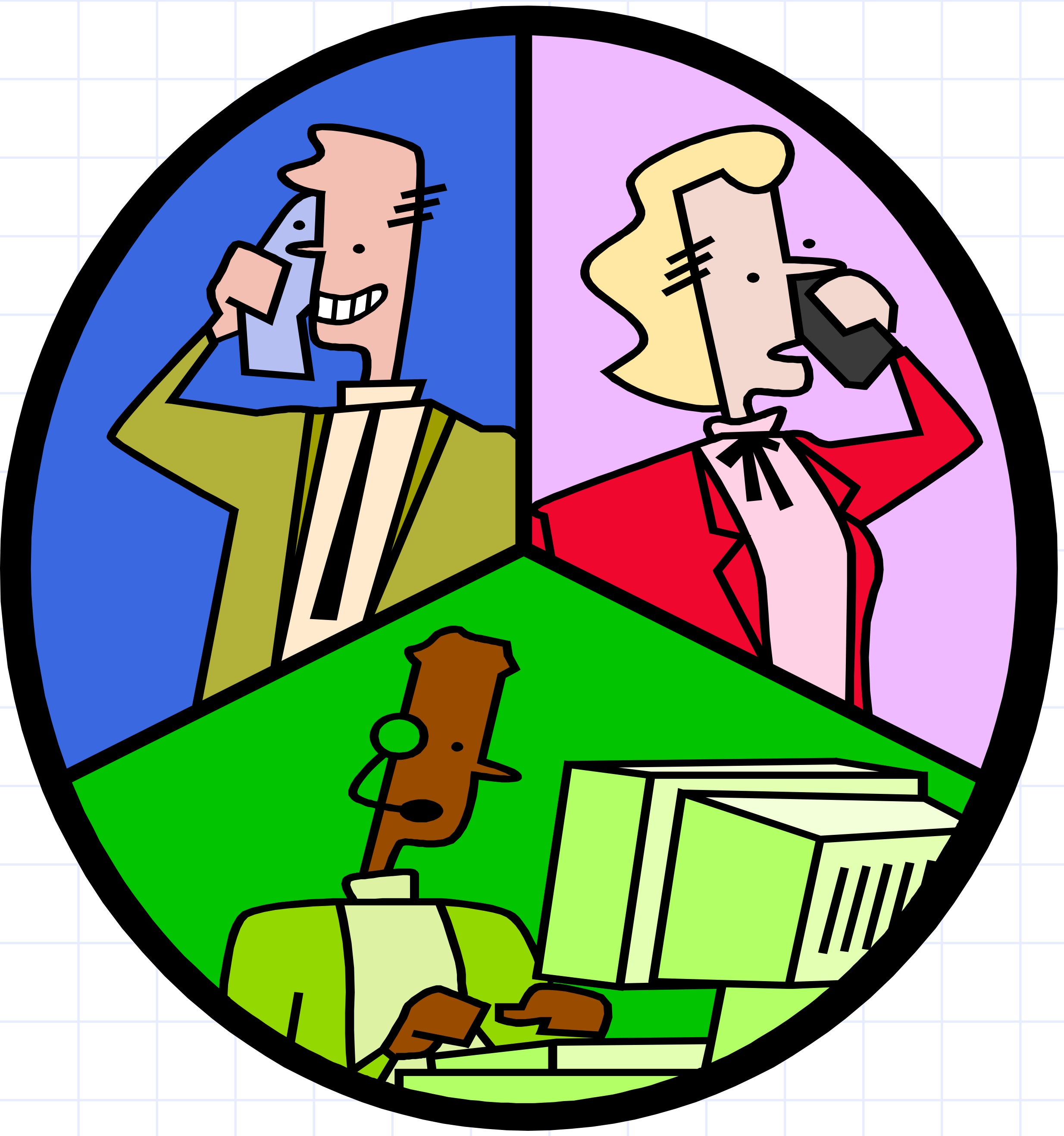


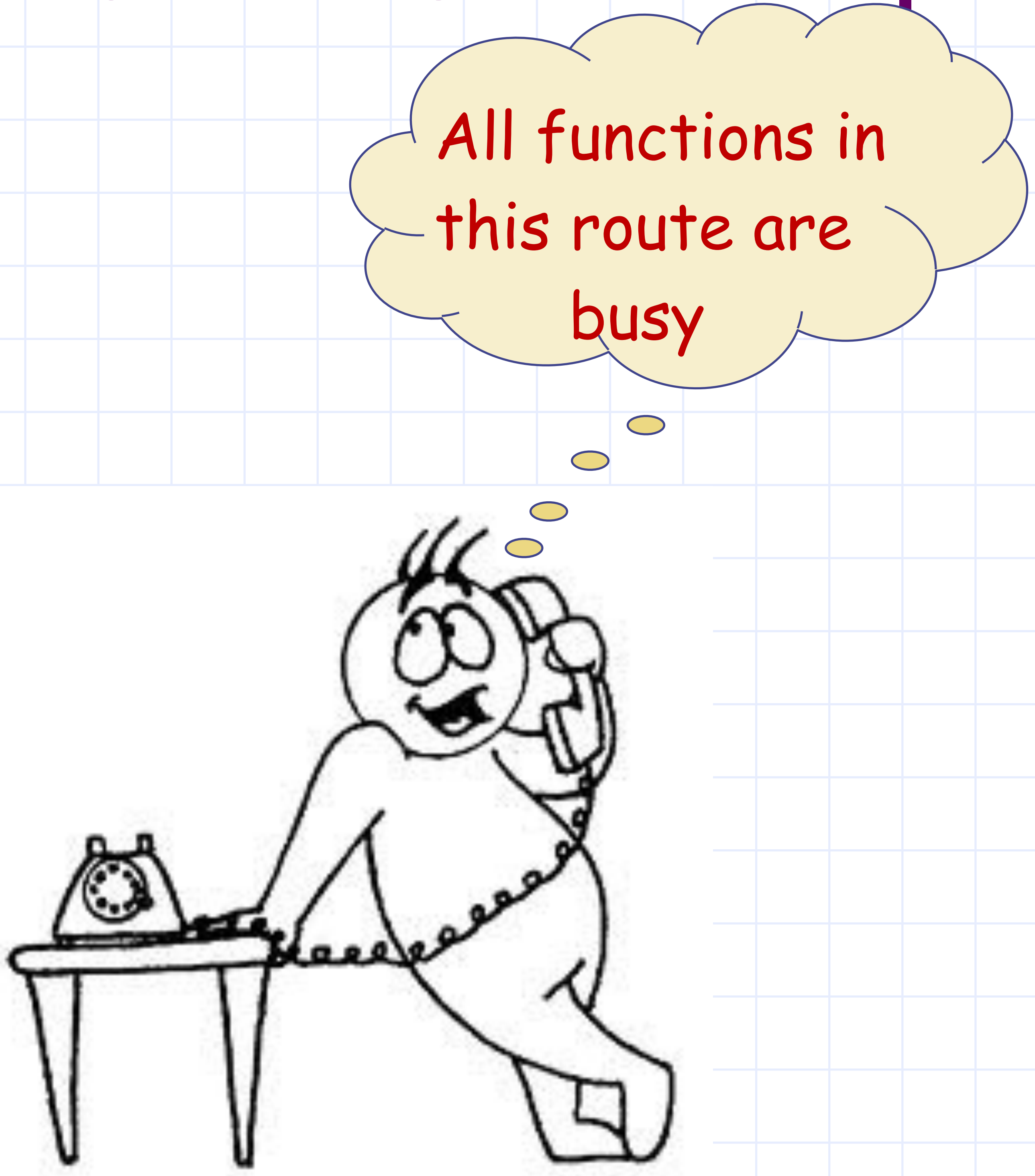
```
void print_positive(int n) {
    if (n <= 0) return;
    printf("%d", n);
}
```

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





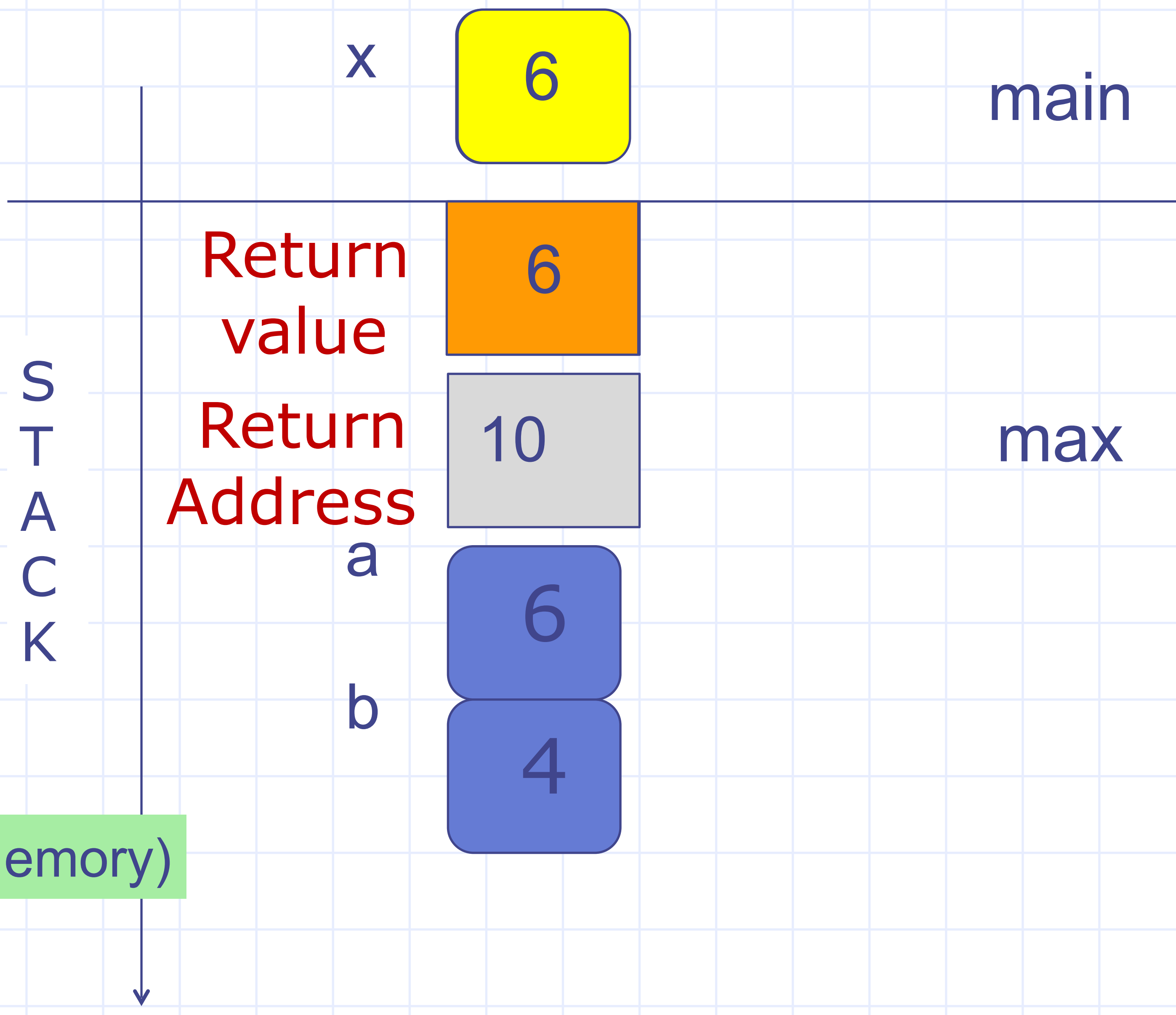
```
1   #include <stdio.h>
2   int max(int a, int b) {
3      if (a > b)
4          return a;
5     else
6          return b;
7   }

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.

```
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 }
```

Calling max(6,4):
1. Allocate space for return value.
2. Store return address (10).
3. Pass parameters.



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

Practice Problem

- Write a function that calculates sum of digits. Given a number, call it repeatedly to check if the sum is equal to 9 or not.
- Input: 99
- Output: Yes (Explanation - 99->18->9)

Solution for Practice problem

```c
#include <stdio.h>
int main()
{

   int sumD;
   int n;
   scanf("%d",&n);
   int flag=0;
   while( n >= 9)
   {
     //Check of sum of digits
   }
   if(flag == 1)
      printf("Yes\n");
   else
      printf("No\n");
   return 0;
}
```

```c
int sumDigits(int n)
{
   int sum=0;
   return sum;
}
```

Solution for Practice problem

```
#include <stdio.h>
int main()
{

   int sumD;
   int n;
   scanf("%d",&n);
   int flag=0;
   while( n >= 9)
   {
     //Check of sum of digits
   }
   if(flag == 1)
      printf("Yes\n");
   else
      printf("No\n");
   return 0;
}
```

```
int sumDigits(int n)
{
   int sum=0;
   while(n>0)
   {
      //calculate sum of
digits
   }
   return sum;
}
```

Solution for Practice problem

```
#include <stdio.h>
int main()
{

   int sumD;
   int n;
   scanf("%d",&n);
   int flag=0;
   while( n >= 9)
   {
     //Check of sum of digits
   }
   if(flag == 1)
      printf("Yes\n");
   else
      printf("No\n");
   return 0;
}
```

```
int sumDigits(int n)
{
   int sum=0;
   while(n>0)
   {
      sum = sum + n%10;
      n = n/10;
   }
   return sum;
}
```

Solution for Practice problem

```
#include <stdio.h>
int main()
{

   int sumD;
   int n;
   scanf("%d",&n);
   int flag=0;
   while( n >= 9)
   {
      sumD = sumDigits(n);
      if(sumD == 9)  {
         flag = 1;
         break;
      }
      n = sumD;
   }
   if(flag == 1)
      printf("Yes\n");
   else
      printf("No\n");
   return 0;
}
```

```
int sumDigits(int n)
{
   int sum=0;
   while(n>0)
   {
      sum = sum + n%10;
      n = n/10;
   }
   return sum;
}
```

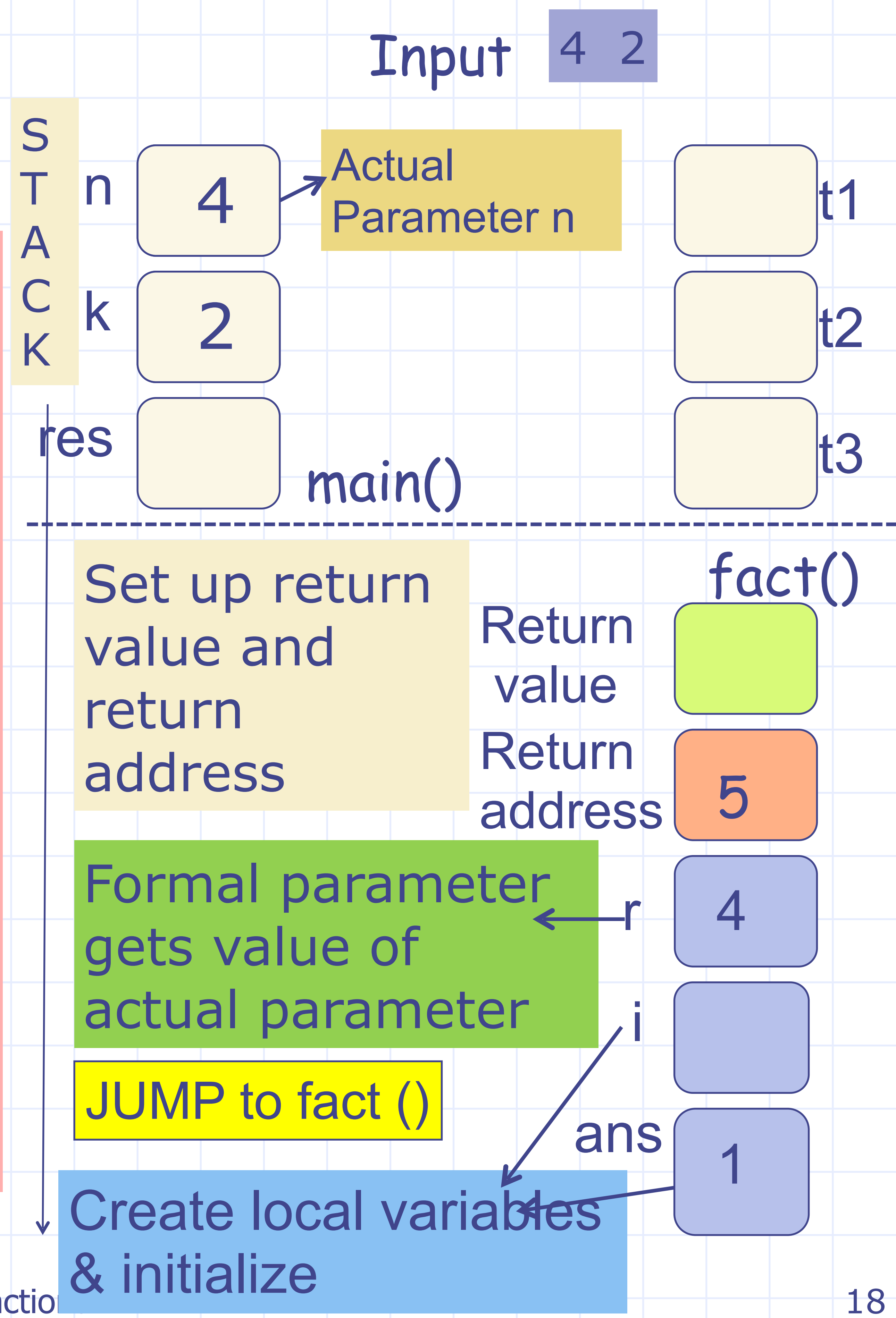
```
# include <stdio.h>
int fact(int r) {   /* calc. r! */
     int i;
     int  ans=1;
     for (i=0; i < r; i=i+1) {
          ans = ans *(i+1);
     }
     return ans;
}
```

```
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 ${}^{n}C_{k}$
- 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 */
}
```

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



Set up return value and return address

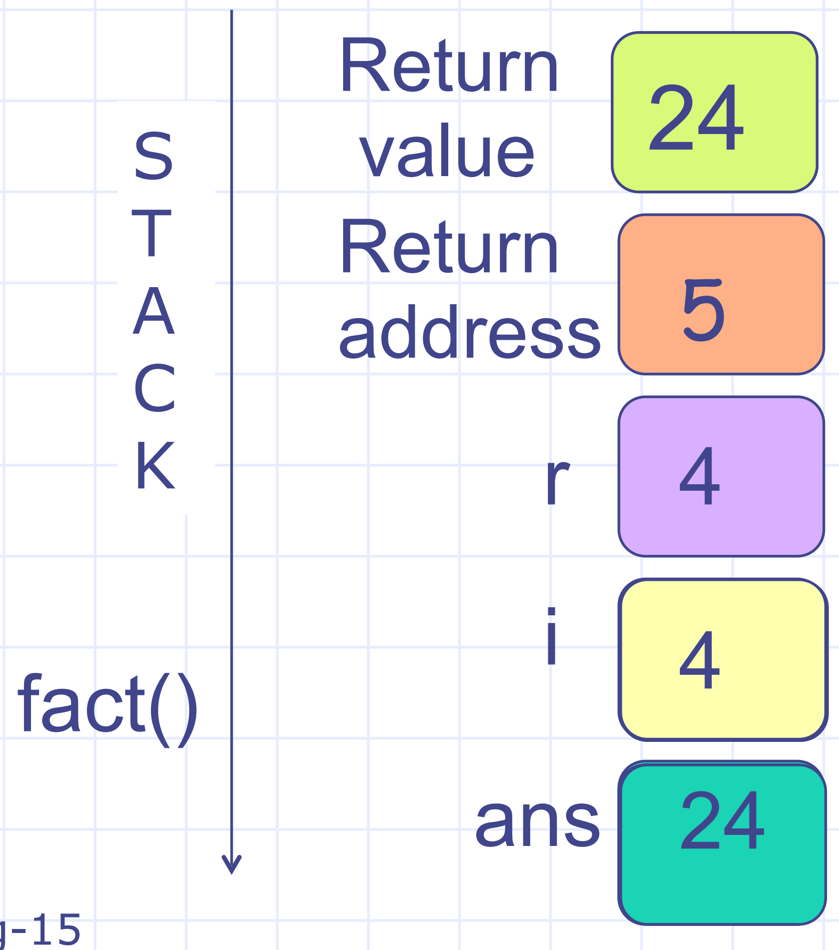
Formal parameter gets value of actual parameter

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.

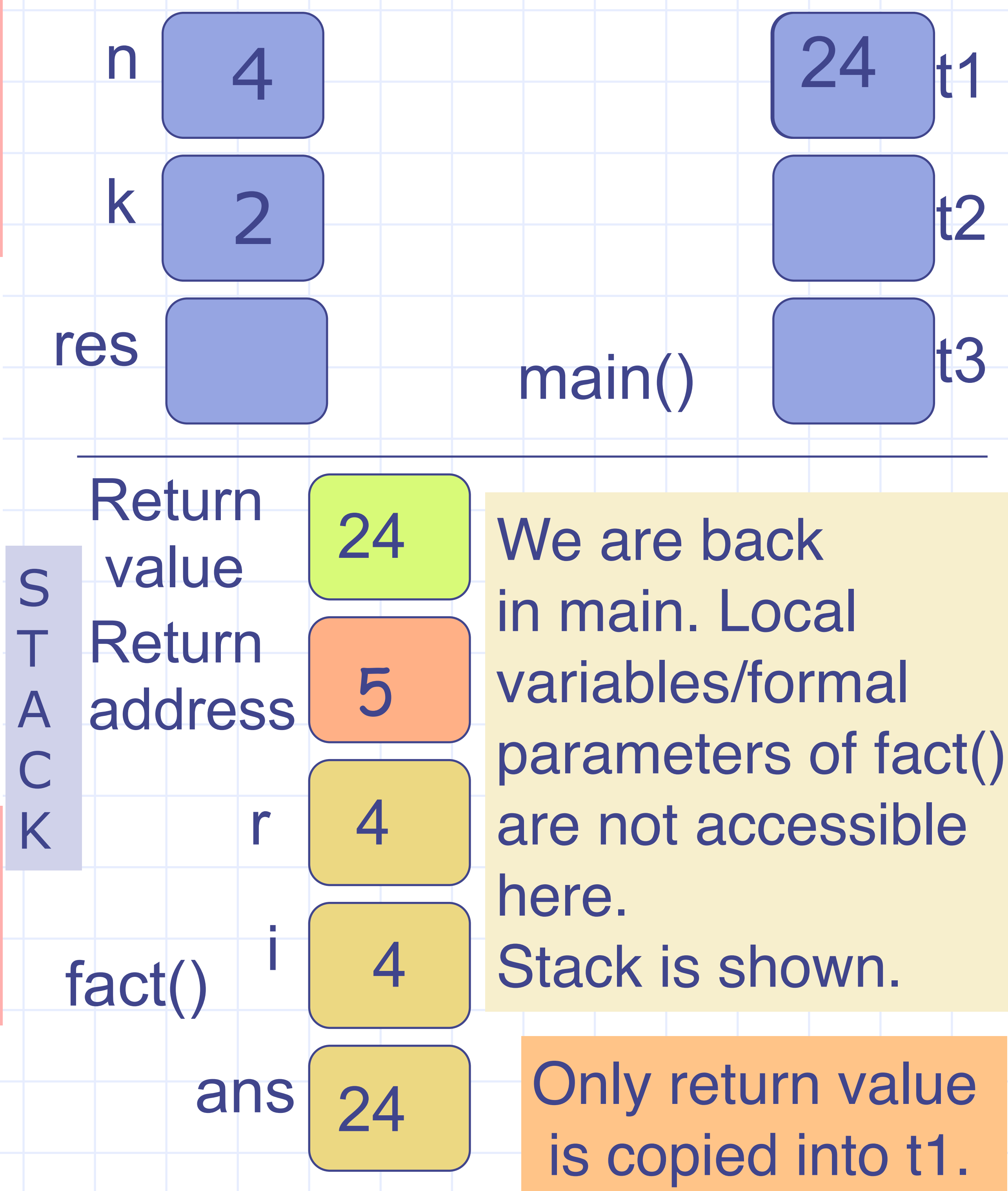


Assign to return value

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. */
5            int        t1, t2, t3;
6            t1 =       fact(n);
7            t2 =       fact(k):
8            t3 =       fact(n-k);
             res =    (t1/t2)/t3;
9       printf("%d\n",res);
        return 0;
   }
```

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

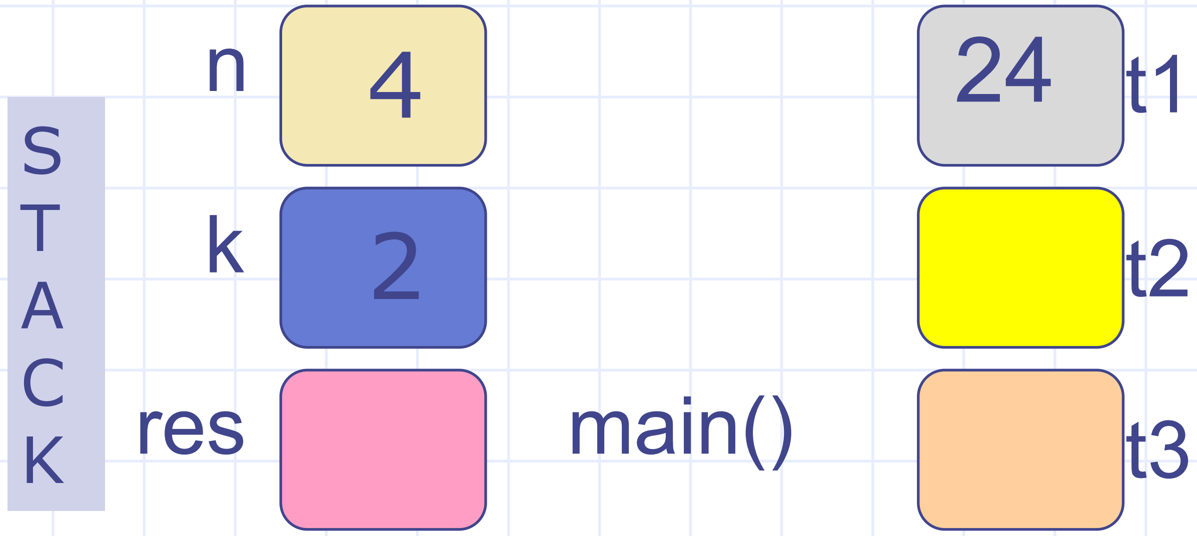


We are back in main. Local variables/formal parameters of fact() are not accessible here. Stack is shown.

Only return value is copied into t1.

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\n",res);
    return 0;
}
```



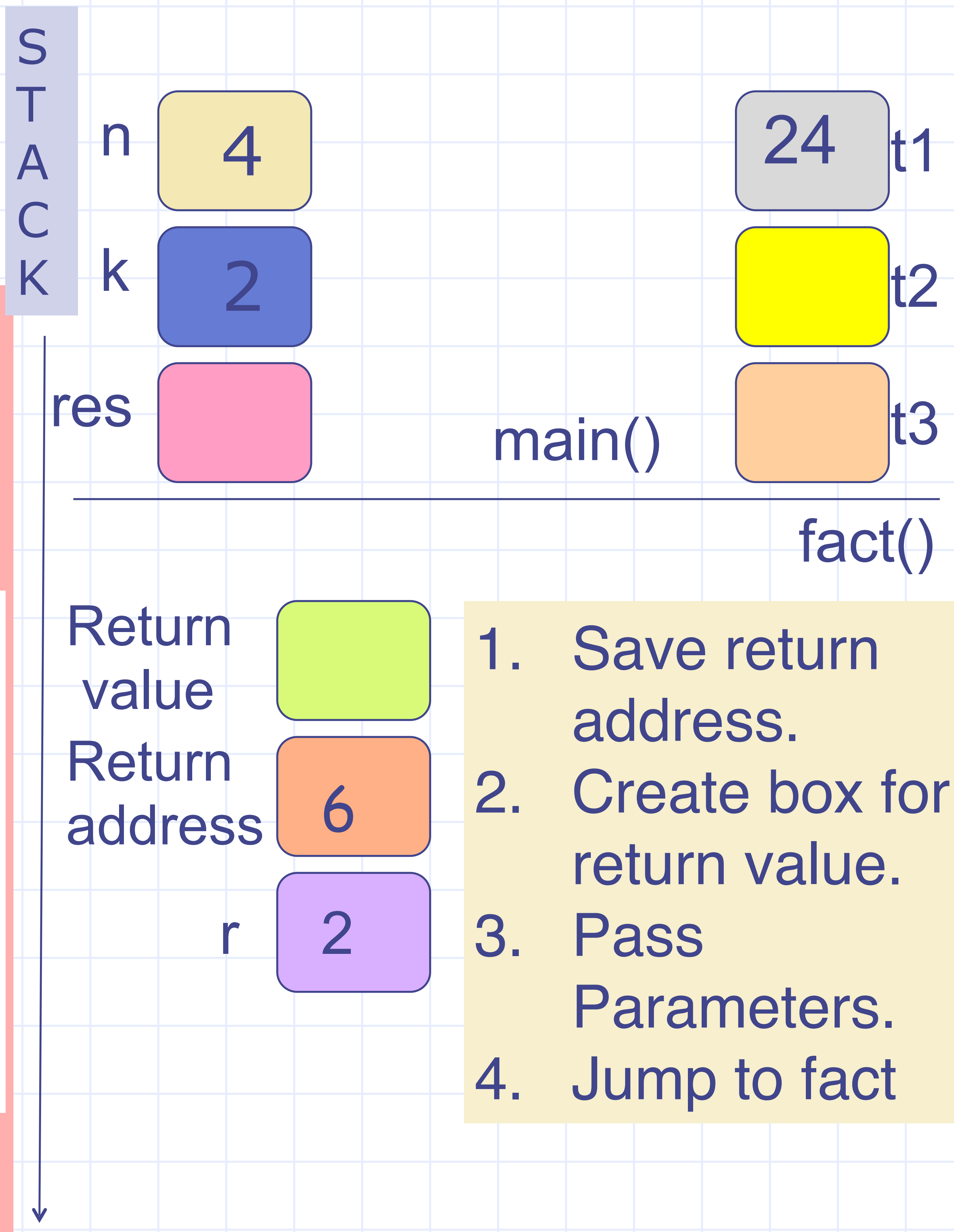
The next statement is another function call: fact(k). Prepare stack for new call.

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.

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

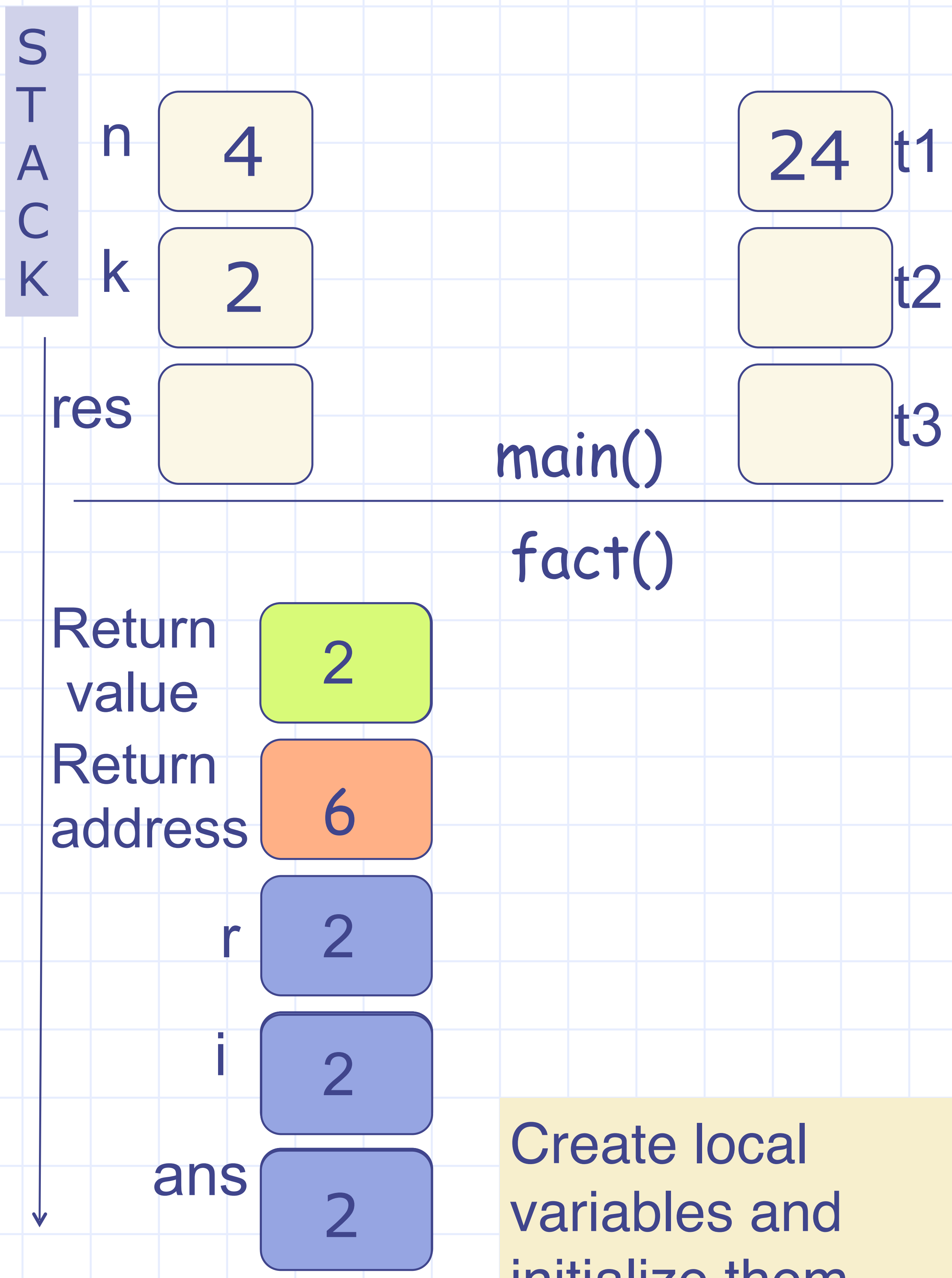
```
int fact(int r) {   /* calc. r! */
  int i; int ans =1 ;
     /*  code here */
}
```

```
1  int main () {
2       int n, k;
3       int res;
4       scanf(“%d%d”,&n,&k);

   /* Adding temporary vars and
   Intermediate exprs. */
5             int        t1, t2, t3;
6             t1 =       fact(n);
7             t2 =       fact(k):
8             t3 =       fact(n-k);
              res =    (t1/t2)/t3;
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;
}
```

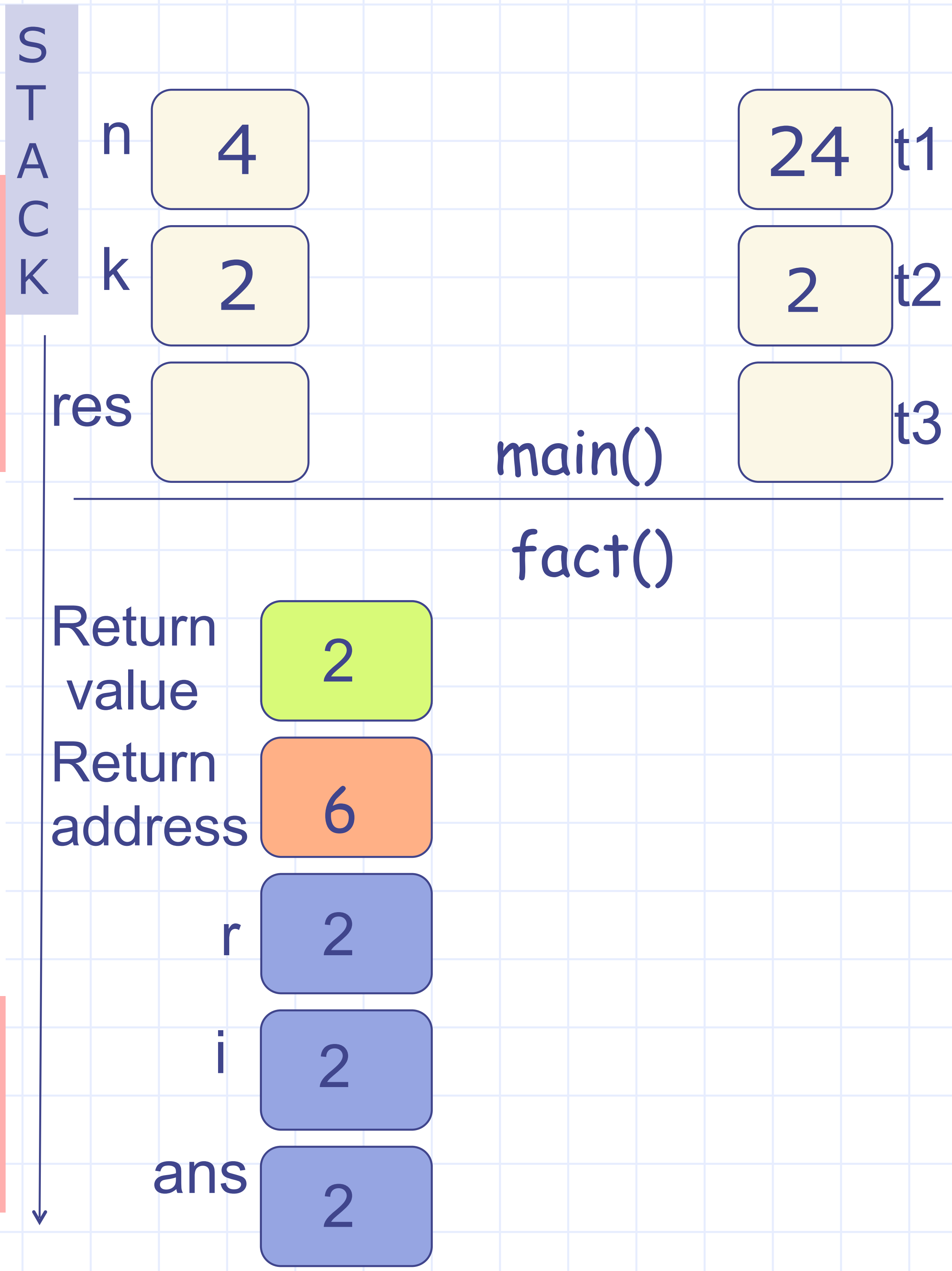


Assign value of ans to box for return value.

Create local variables and initialize them

Now jump to return address

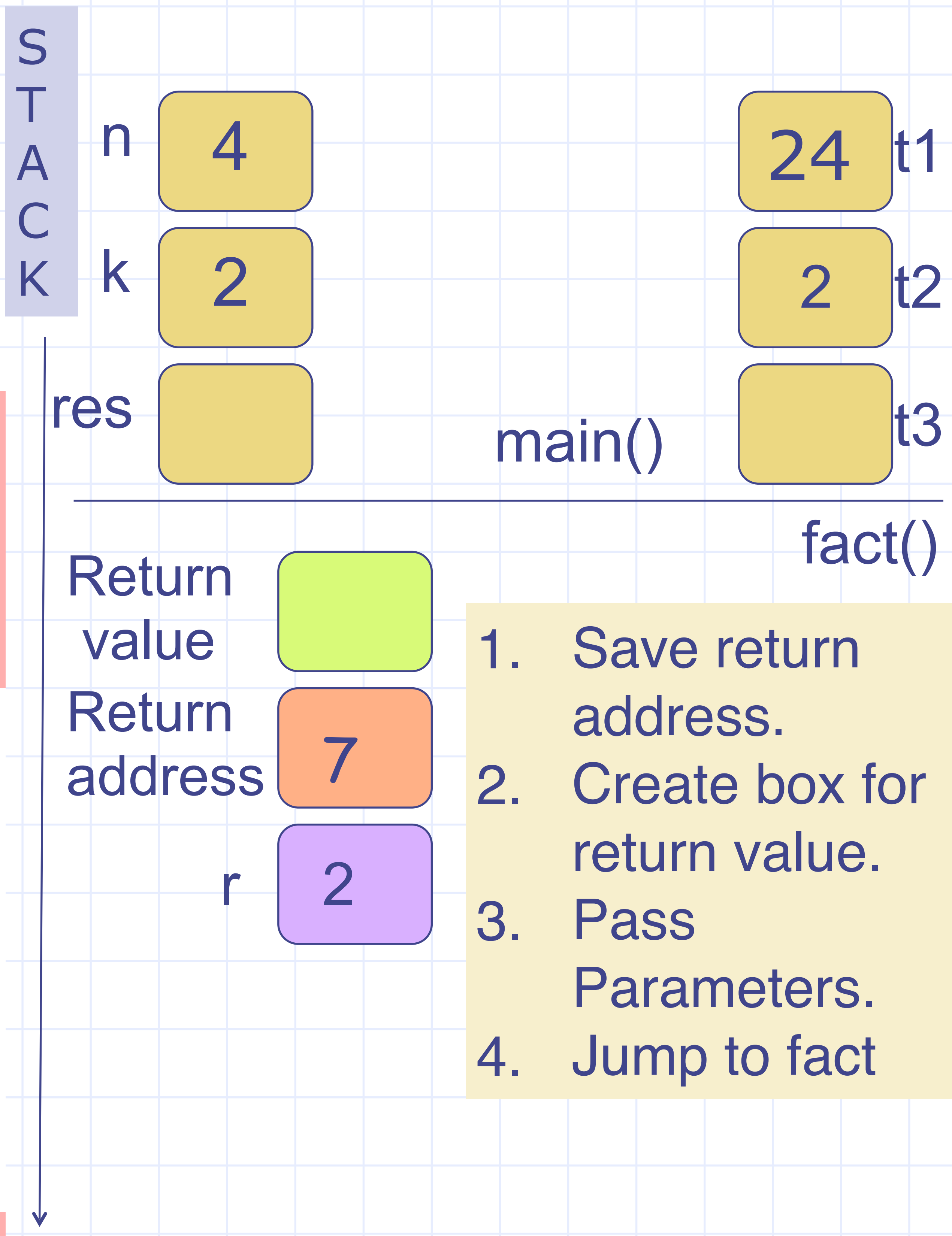
```
1  int main () {
2       int n, k;
3       int res;
4       scanf("%d%d",&n,&k);
   /* Adding temporary vars
   and Intermediate exprs. */
5           int      t1, t2, t3;
6           t1 =      fact(n);
7           t2 =      fact(k):
8           t3 =      fact(n-k);
            res =   (t1/t2)/t3;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

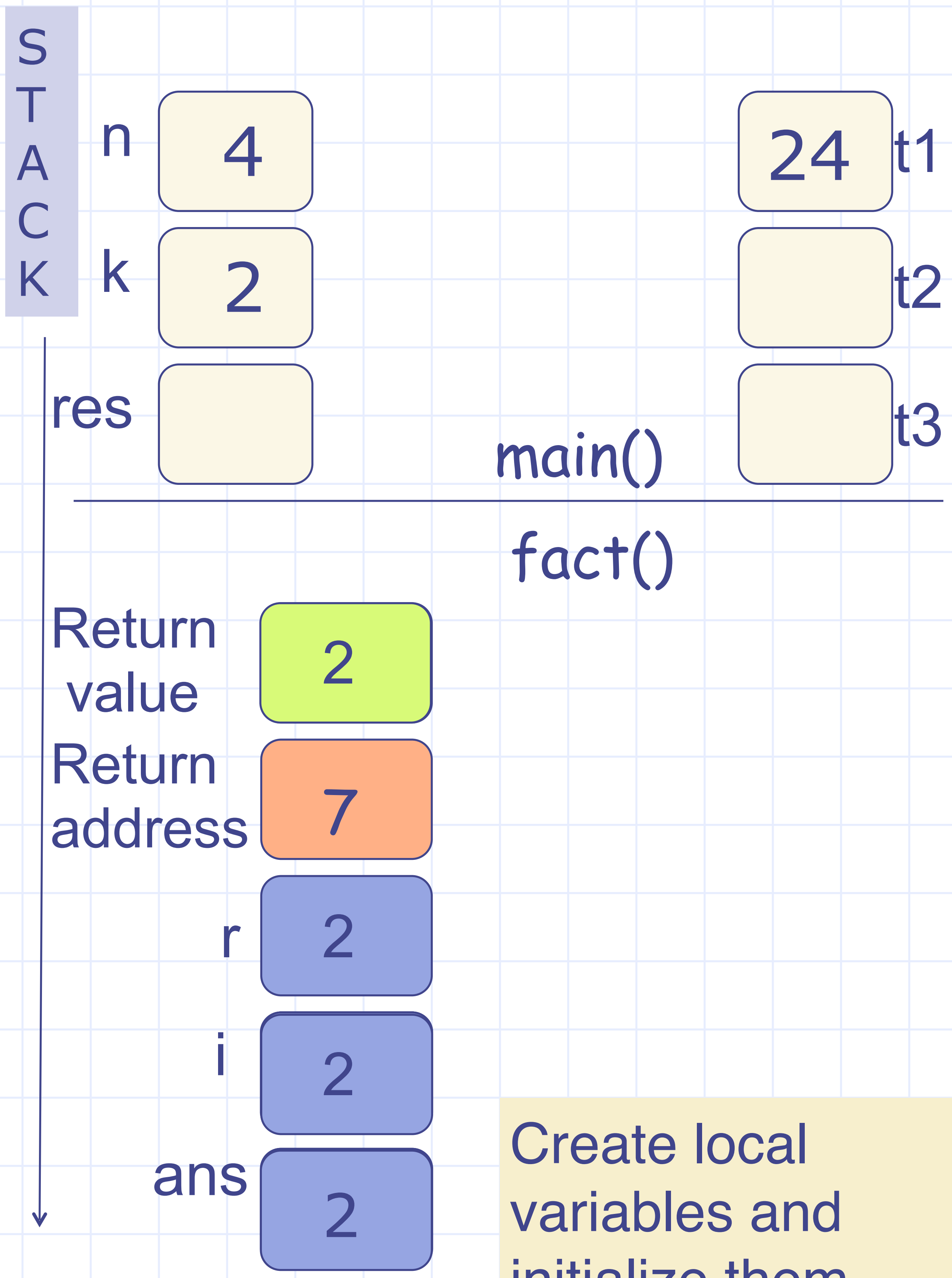
```
1  int main () {
2       int n, k;
3       int res;
4       scanf("%d%d",&n,&k);
   /* Adding temporary vars
    and Intermediate exprs. */
5              int         t1, t2, t3;
6              t1 =        fact(n);
7              t2 =        fact(k):
8              t3 =        fact(n-k);
       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

```c
# 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;
}
```

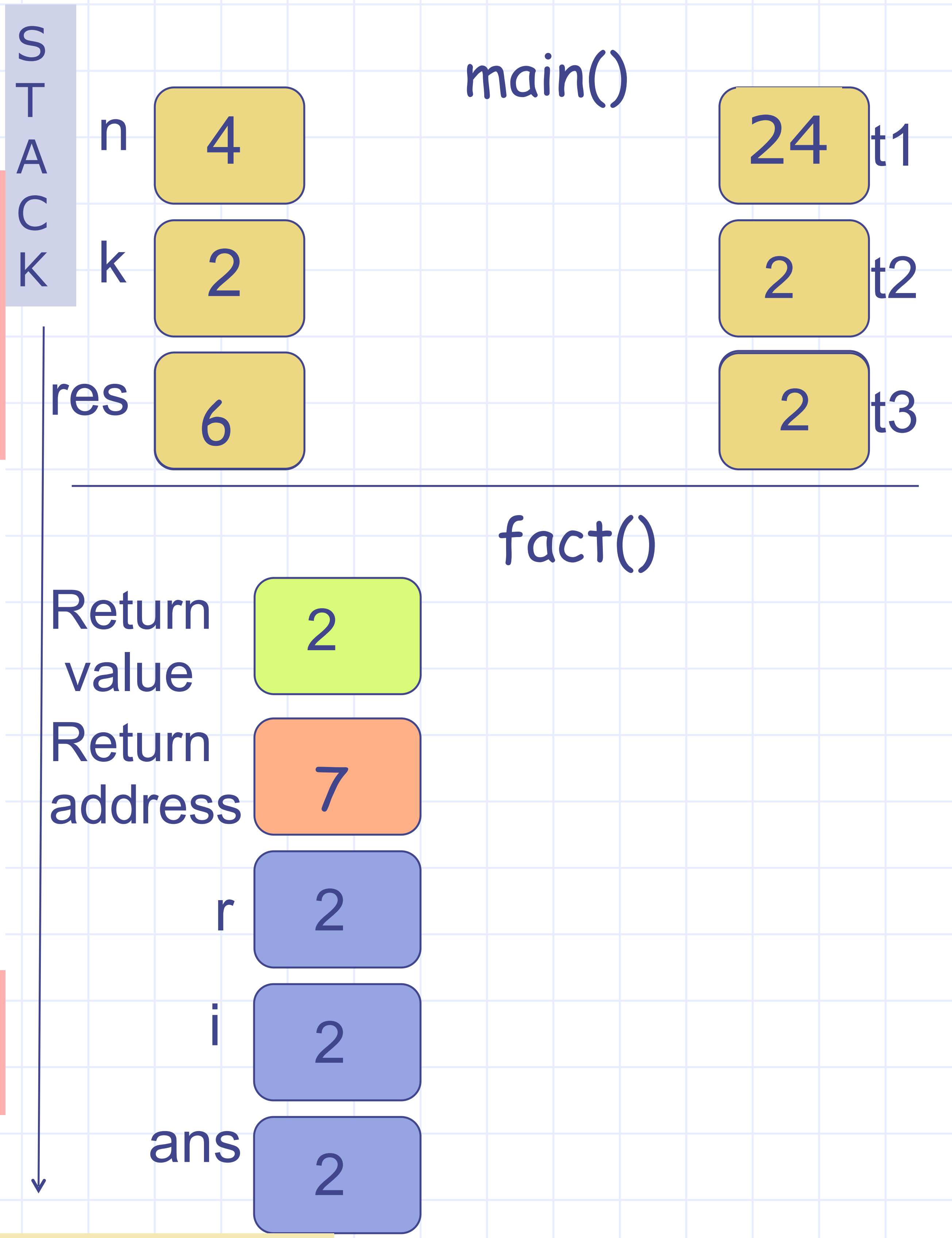


Assign value of ans to box for return value.

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. */
5          int       t1, t2, t3;
6          t1 =      fact(n);
7          t2 =      fact(k):
8          t3 =      fact(n-k);
           res =   (t1/t2)/t3;
       printf("%d\n",res);
       return 0; }
```



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

4 choose 2 is 6

Example

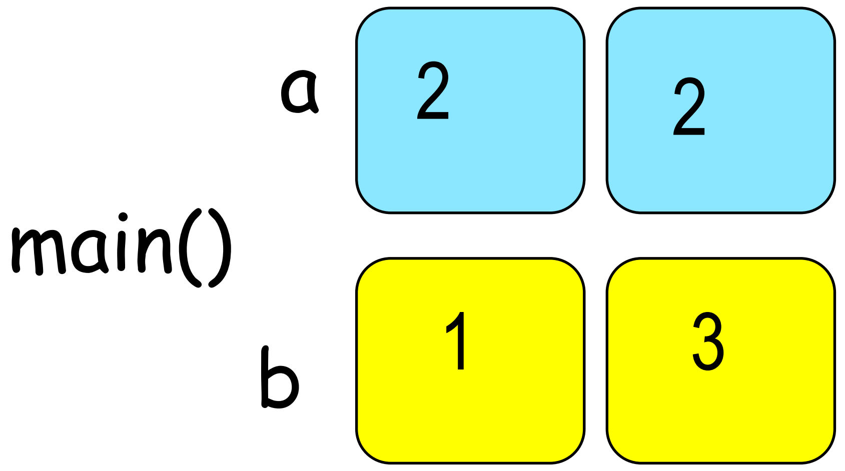
What is printed by the program?

Rule: All arguments are evaluated before function call is made.

```
int f (int a, int b) {
         return b-a;
}
```

Let us evaluate function arguments in left to right order.

```
main () {
    int a = 2, b = 1;

    a = f( a=b+1,    b=a+1);

   printf("%d  %d", a,b);
}
```



Expected Output

1 3

Evaluate f(a=b+1,b=a+1).

How should we evaluate it?

Left-right OR right-left

```
int f (int a, int b) {
       return b-a;
}
```

```
main () {
    int a = 2, b = 1;

    a = f( a=b+1,    b=a+1);

    printf("%d    %d", a,b);
}
```

We used left to right evaluation. Expected output:

1 3

Let us compile and run.

On some machines, output is:

-1 3

a	2	4	-1
b	1	3	

What happened?

The compiler evaluated right to left. Output is

-1 3

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));
}
```

Practice Problem

- Write a function that simulates a bank account. Allow operations Deposit ‘d’ *amount* Withdraw ‘w’ *amount* and add interest at a fixed rate of 10%
- Sample Input format : *InitialAmount* d *amount1* w *amount2*
- Input: 1000 d 100 w 200 i d 300 w 100
- Output: amount = 1190

Solution for Practice problem

```
#include <stdio.h>
int interest(int amnt);
int deposit( int amnt, int sum);
int withdraw( int amnt, int sum);
int main()
{
   int amnt, c, n;
   scanf("%d",&amnt);
   getchar(); // skip a space after reading amount
   while( (c = getchar() )!=EOF)
   {
   }
   printf("amount = %d\n”,amnt);
   return 0;
}
```

Solution for Practice problem

```
#include <stdio.h>
int interest(int amnt);
int deposit( int amnt, int sum);
int withdraw( int amnt, int sum);
int main()
{
   int amnt, c, n;
   scanf("%d",&amnt);
   getchar(); // skip a space after reading amount
   while( (c = getchar() )!=EOF)
   {
      switch(c)
      {
         case 'd': scanf("%d",&n);
                   amnt=deposit(amnt, n);
                   getchar(); break; //skip a space after reading amount
     }
   }
   printf("amount = %d\n",amnt);
   return 0;
}
```

Solution for Practice problem

```
#include <stdio.h>
int interest(int amnt);
int deposit( int amnt, int sum);
int withdraw( int amnt, int sum);
int main()
{
   int amnt, c, n;
   scanf("%d",&amnt);
   getchar(); // skip a space after reading amount
   while( (c = getchar() )!=EOF)
   {
      switch(c)
      {
         case 'd': scanf("%d",&n);
                   amnt=deposit(amnt, n);
                   getchar(); break; //skip a space after reading amount
         case 'w': scanf("%d",&n);
                   amnt=withdraw(amnt, n);
                   getchar();  break; //skip a space after reading amount
         case 'i': amnt=interest(amnt);
                   getchar();  break; //skip a space after reading amount
         default: printf("invalid input\n"); break;
      }
   }
   printf("amount = %d\n",amnt);
   return 0;
}
```

Solution for Practice problem

```
int interest(int amnt)
{
   return (amnt+amnt*10/100.0);
}


int deposit( int amnt, int sum)
{
   return (amnt+sum);
}

int withdraw( int amnt, int sum)
{
   //can have additional checks for seeing amnt is not negative
   return (amnt -sum);
}
```

Next class

- Scope

- Best of Luck for the Lab Exam