Functions Part 2

Topics

- Type Casting
- Revisitd: `void` and `return`
- Revisited: PostFix and Prefix ++, --
- Recursion

Type-Casting

- You can force type conversions
- Use the type name as if it were an expression in parenthesis as in:

  (int) x

  which casts the content of x into an integer
- Using int(x) also works but let's stick to classical C
- Why cast?

Automatic Type Conversion

- When you assign an int to a float, C internally casts the float literal into an int
- Casting to int causes loss of data since fractions are dropped, hence 5.6 becomes 5
- When there is potential loss of data you will get a warning, but the code compiles
- You need to worry about cases like 4/9 more than cast variables in expressions, since 4/9 is 0 right?
- C produces outcomes based on operand types, hence: div=4/9 where div is float, results in float being 0.000000, which is where you would consider casting

Type-Casting

- More examples:

```
double answer1,answer2;
int i=5;
answer1 = (double)i/4;
answer2 = (double)(i/4);
```

//what's the difference between 1 and 2?
// Try it and you will see

In the book – One Small Section

So you don't have to go back and forth (section Type Casting under Simple Math:

Cast operators

A cast expression is a unary expression. A parenthesized type name followed by a cast expression is a cast expression. The parenthesized type name has the effect of forcing the cast expression into the type specified by the type name in parentheses. For arithmetic types, this either does not change the value of the expression, or truncates the value of the expression if the expression is an integer and the new type is smaller than the previous type.

An example of casting a float as an int:

```
float pi = 3.141592;
int truncated_pi = (int)pi; // truncated_pi == 3
```

An example of casting a char as an int:

```
char my_char = 'A';
int my_int = (int)my_char; // my_int == 65, which is the ASCII value of 'A'
```

Additional Recommended Reading

- This is classical text on C:
  The C Programming Language by Kernighan and Ritchie
  I call it the C bible!
- Not available for free, so we could not officially use it as a Free Open Educational Resource
- Available as a PDF download
- Watch out for copy-write
- On Casting, see chapter 2, section 2.7

`void` Functions

- Functions need not always return A result
- A return type of `void` indicates A function that returns no result
  - `return` statement contains no expression
  - `return` statement assumed at end of function
- In other languages, `void` functions are called subroutines or procedures

`void` Functions

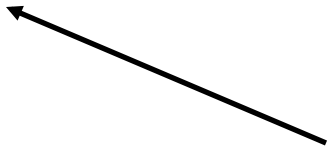
- Examples:

```
void horizontal_line( ) {
 printf( "\n-------------\n“ );
 return;
}
void sayGoodnightGracie( ) {
 printf( "Goodnight, Gracie“ );
}
```

`void` Functions

- Examples:

```
void horizontal_line( ) {
 printf( "\n-------------\n“ );
 return;
}
void sayGoodnightGracie( ) {
 printf( "Goodnight, Gracie“ );
}
```



return statement is assumed

Another Use For `void`

- `void` means “nothing” so you sometimes see it stated when there is nothing else to say
- For example:

```
void horizontal_line( void ){
 printf( "\n----------\n“ );
}
```

`return` Statement

- A function may contain multiple `return` statements

```
int max( int a, int b) {
     if (a < b)
   return b;
     else
   return a;
}
```

- Generally, more readable with just one

`return` Statement

- A function may contain multiple `return` statements

```
int max( int a, int b) {
     int result = 0;
     if (a < b)
   result = b;
     else
   result = a;
     return( result );
}
```

Prefix and Postfix Operators Revisited

- `++` is a shorthand for `+ 1`

```
i++;          i = i + 1;
```

- `--` is a shorthand for `- 1`

```
i--;          i = i - 1;
```

- The operator can come before or after the variable

```
i++;          ++i;
```

Prefix and Postfix Operators

- Prefix operator occurs before expression evaluation
- Postfix operator occurs after statement evaluation

```
int i = 12, j = 10, k = 0;
k = i++ * --j;
//--j then * i then assign to k, then i++
k = --i + ++j;
//subtract 1 from i, and add 1 to j, then i+j
//and then assign to k
```

Next … Presenting

- Recursion
- Our coverage is for you to get used to the idea
- We will not do heavy duty recursion, but for professional programmers it’s a must.
- Recursion is covered in more details in courses like Data Structures − do take one.

What is recursion?

- Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first
- Recursion is a technique that solves a problem by solving a **smaller problem** of the same type

In Code

```
int f(int x)
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

← this line says f(x) is…

← 2 * f(x-1)

In Code

```
int f(int x)   ← this line says f(x) is…
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);  ← 2 * f(x-1)
    return y+1;
 }
}
```

So you see how solving f(x) requires a call to solve f(x-1), hence a smaller problem. Smaller because x-1 is smaller than x

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n\text{-}1)!*n & \text{if } n > 0 \end{cases} \quad (\textit{recursive} \text{ solution})$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n\text{-}1)*n & \text{if } n > 0 \end{cases} \quad (\textit{closed form} \text{ solution})$$

In English

- The factorial of n (let's ignore some details for now) is:
  - n multiplied by the factorial of n-1
- If you're thinking this will never stop… that's because we did not define the end point yet, but we will.

In English – Take 2

- The factorial of n is defined as:
  - Equals to 1 if n ==0
  - Otherwise, it equals to n * factorial of n-1

- The “Equals to 1 if n==0” is what will end the recursion.
- Mathematically, factorial of n is undefined for n<0.

But Wait…

- Can’t we do this with loops?
- Of course, and when the data we deal with is plain numbers, loops are preferred and more efficient.
- Recursion is needed when dealing with data that is not plain numbers, hence Data Structures.
- Since our goal is to understand recursion, let’s stick to numbers so we can focus on the concept.

Coding the factorial function

- Recursive implementation

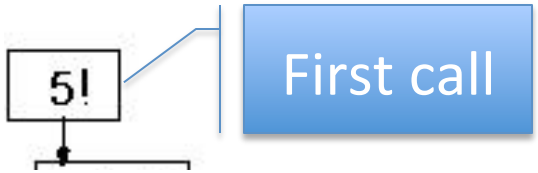
```
int Factorial(int n)
{
 if (n==0)  // this is called the base case
   return 1;
 else
   return n * Factorial(n-1);  //here is the recursive call
}
```

Coding Recursion ABC's

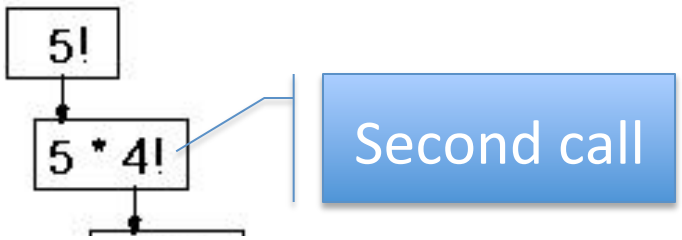
- Check that you have a base case
- Check that you have a recursive call
- Check that the recursive call is to a "smaller" n
- Then check that everything works together by tracing the code on PAPER!!

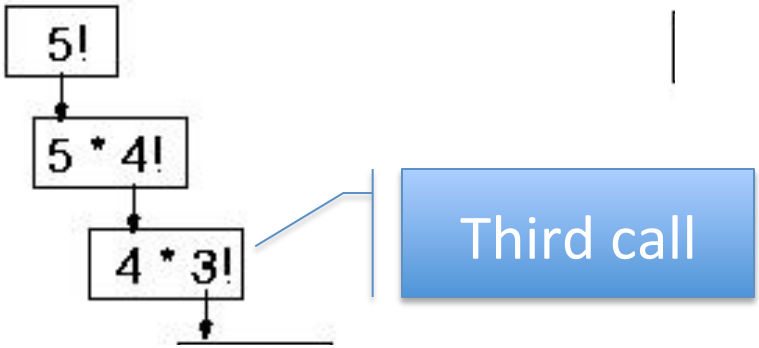
```
int Factorial(int n)
{
 if (n==0)  // this is called the base case
   return 1;
 else
   return n * Factorial(n-1);  //here is the recursive call
}
```

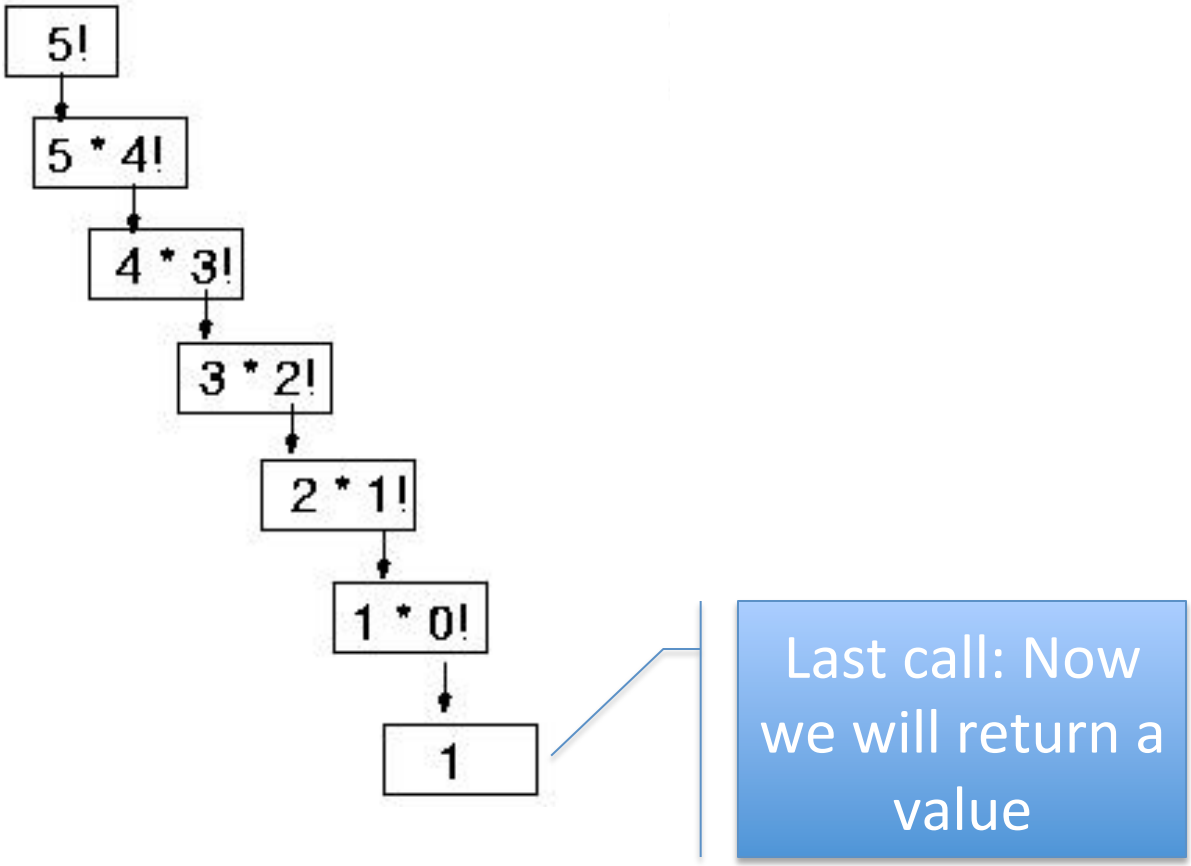
Tracing on paper

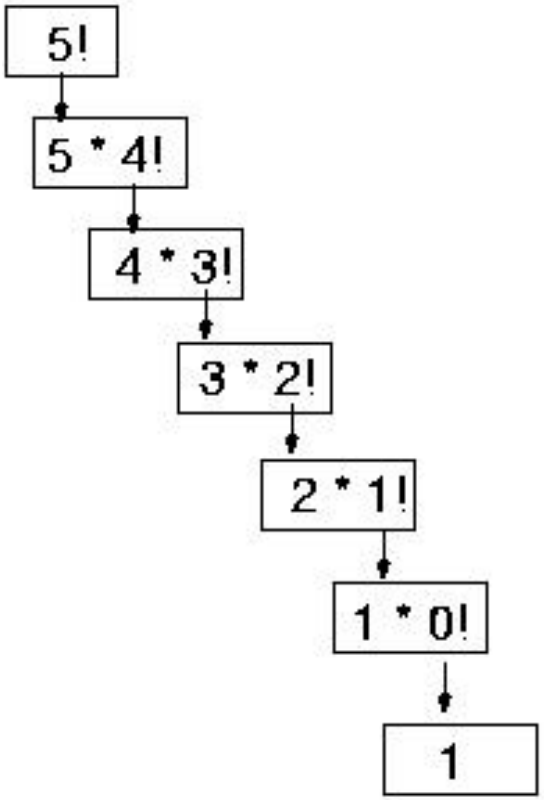


Tracing on Paper

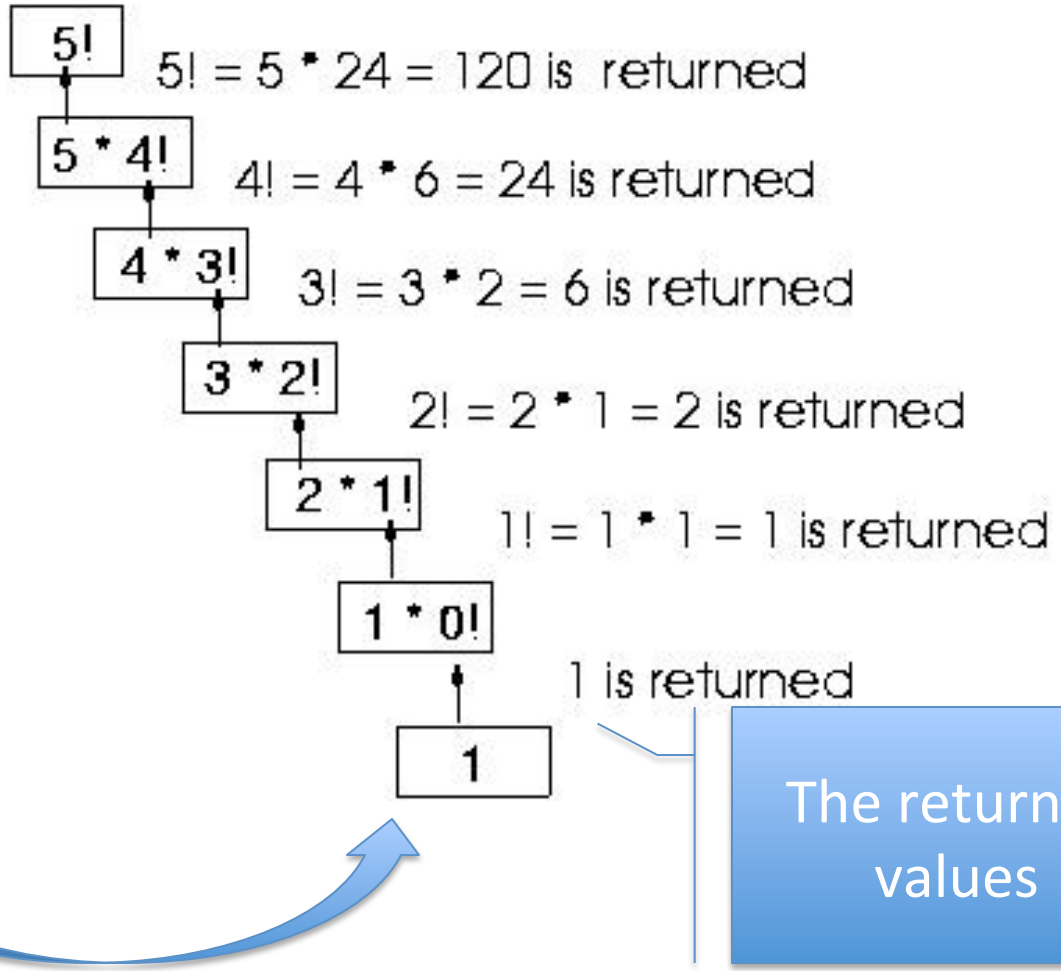



5!
5 * 4!
4 * 3!
Third call


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
Last call: Now we will return a value



Final value = 120



Coding the factorial function - loops

- for loop

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Steps to writing a recursive function

- Determine the base case(s)

  (the one for which you know the answer)
- Determine the general case(s)

  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm

  (Trace on paper)

What happens in RAM?

Assume we have:

```
int a(int w)
{
 return w+w;
}


int b(int x)
{
 int z,y;
 …… // other statements
 z = a(x) + y;

 return z;
}
```

```
int a(int w)
{
 return w+w;
}


int b(int x)   ← assume this is called from main
{
 int z,y;
 …… // other statements
 z = a(x) + y;

 return z;
}
```

An **activation** record is stored into a special area of RAM called stack (**run-time stack**). Think of it as a letter placed on a desk. The letter contains all variable names and their values

```
int a(int w)
{
 return w+w;
}


int b(int x)
{
 int z,y;
 …… // other statements
 z = a(x) + y; ← we will get to this line causing a call to a

 return z;
}
```

The computer has to stop executing function *b* and starts executing function a. Hence, another letter must be placed on the desk, on top of the previous letter. The new letter will contain variables of a, and their values.

```
int a(int w)
{
 return w+w;
}
```

← Now we're here, and the line says "return"…

```
int b(int x)
{
 int z,y;
 …… // other statements
 z = a(x) + y;
 return z;
}
```

```
int a(int w)
{
 return w+w; → Give back w+w
}


int b(int x)
{
 int z,y;
 …… // other statements
 z = a(x) + y;
 return z;
}
```

That means the top letter on the desk will be "thrown" out, but in the letter below it, a(x) will be replaced by whatever the literal outcome is from w+w

```
int a(int w)
{
 return w+w;
}


int b(int x)
{
 int z,y;
 …… // other statements
 z = a(x) + y; ← back here, now z is assigned to some literal
 return z;
}
```

Those letters…

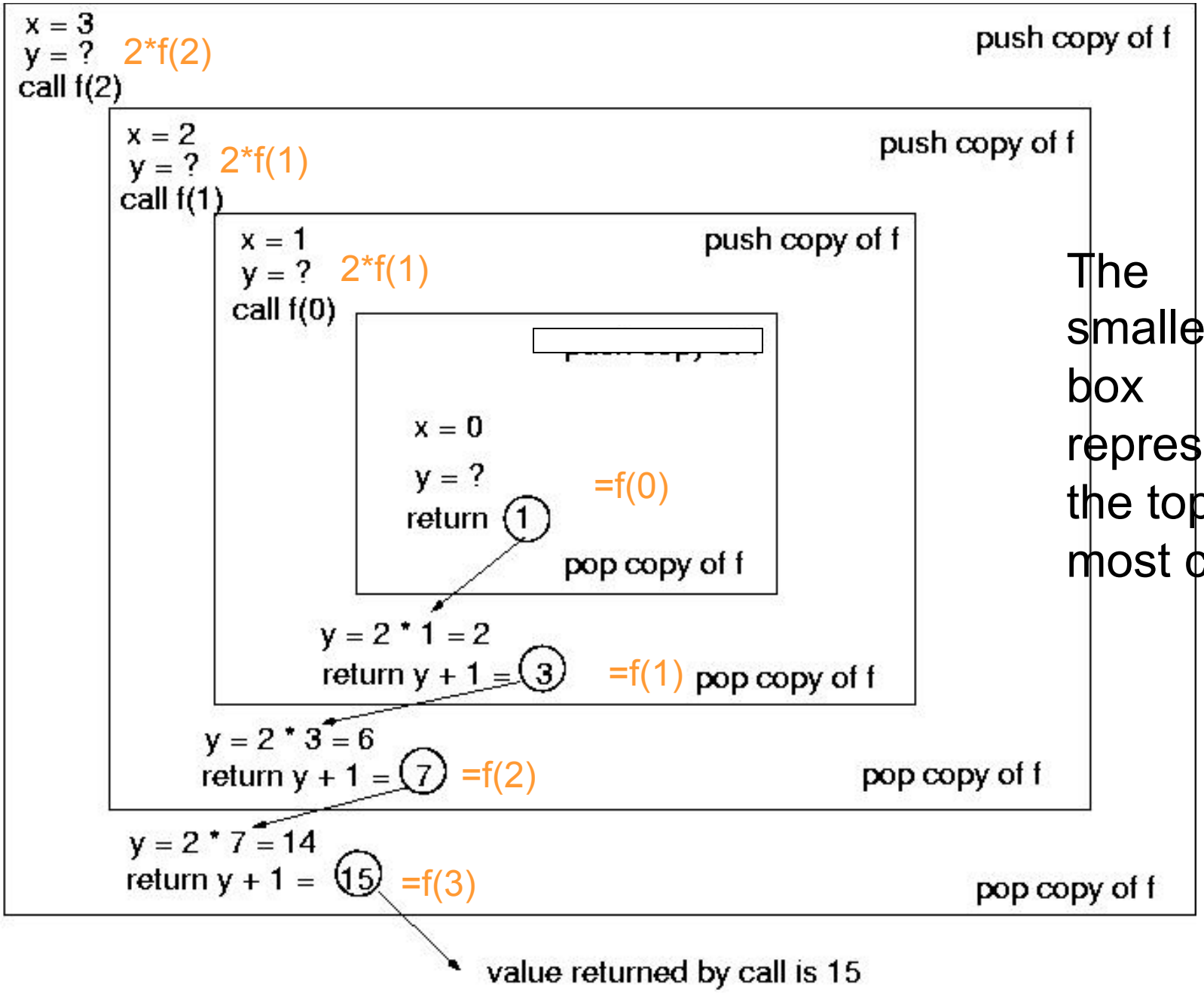
- The desk where the letters are placed is called the call stack.
- It's a stack because every new letter will be placed on top of the previous letter
- Every time there is a "return" the current top entry in the stack is popped out, and the return value, if any, is placed in the call that activated that new letter
- What is the bottom-most call (letter) in the stack made to?.....yeah, it's **main**

When a recursive function is called?

- Except the fact that the calling and called functions have the same name, there is really no difference between recursive and nonrecursive calls

```
int f(int x)
{
 int y;

 if(x==0)
  return 1;
 else {
   y = 2 * f(x-1);
   return y+1;
}
}
```



The smallest box represents the top most call

Congratulations

- You know what a call stack is
- And can code simple recursive functions
- Now read the book's Recursion section under Procedures and functions.