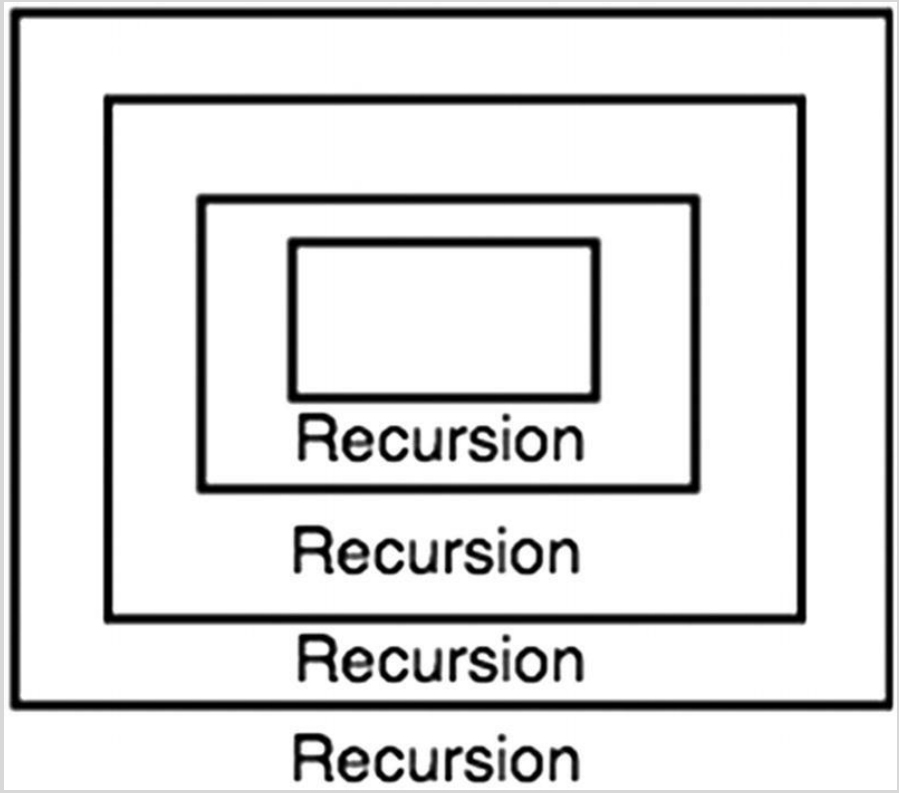
RECURSION

Dr. Megha Ummat

Recursion

- There are many programming concepts that define themselves. Such definitions are called recursive definitions.
- Recursive definitions are primarily used to define infinite sets as giving a complete list of elements is impossible for such as set.
- Recursive definitions give a more efficient way to determine if an object belongs to a set.



Recursion

- A recursive definition consists of two parts:

i. Anchor case or Ground case: The basic elements that are the building blocks of all other elements of the set are listed.

ii. Rules for generating new objects: Rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects.

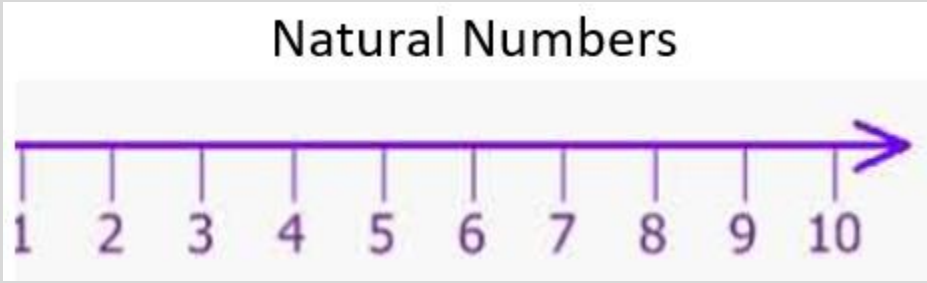
Recursion

Recursive definition to construct the set of natural numbers

1. $0 \in \mathbf{N}$;

2. if $n \in \mathbf{N}$, then $(n + 1) \in \mathbf{N}$;

3. there are no other objects in the set **N**.



According to these rules, the set of natural numbers N consists of the following items: 0, 0 + 1, 0 + 1 + 1, 0 + 1 + 1 + 1, and so on.

Recursion

Another recusive definition for natural numbers

1. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ∈ **N**;

2. if $n \in \mathbf{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbf{N}$;

3. these are the only natural numbers.

Recursion

- Recursive definitions serve two purposes:
  - *Generating* new elements
  - *Testing* whether an element belongs to a set
- In the case of testing, the problem is solved by reducing it to a simpler problem, and if the simpler problem is still too complex it is reduced to an even simpler problem, and so on, until it is reduced to a problem indicated in the anchor.
- For instance, is 123 a natural number? According to the second condition of the definition introducing the set **N**, 123 ∈ **N** if 12 ∈ **N** and the first condition already says that 3 ∈ **N**; but 12 ∈ **N** if 1 ∈ **N** and 2 ∈ **N**, and they both belong to **N**.

Recursion Definition

- Recursive definitions are frequently used to define functions and sequences of numbers.
- For instance, the factorial function, !, can be defined in the following manner:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ (anchor)} \\ n \cdot (n-1)! & \text{if } n > 0 \text{ (inductive step)} \end{cases}$$

- Using this definition, we can generate the sequence of numbers 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, . . . which includes the factorials of the numbers 0, 1, 2, . . . , 10, . . .

Recursive Definition

Another example is the definition

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n-1) + \dfrac{1}{f(n-1)} & \text{if } n > 0 \end{cases}$$

Which generates the sequence of rational numbers

$$1, 2, \frac{5}{2}, \frac{29}{10}, \frac{941}{290}, \frac{969{,}581}{272{,}890}, \ldots$$

Recursive Definition

- Recursive definitions of sequences have one undesirable feature: to determine the value of an element s_n of a sequence, we first have to compute the values of some or all of the previous elements, $s_1, \ldots, s_{n-1}$.
- For example, calculating the value of 3! requires us to first compute the values of 0!, 1!, and 2!.
- If we can find an equivalent definition or formula that makes no references to other elements of the sequence, this undesirability can be solved.
- But finding such a formula is difficult and cannot always be solved.

Recursive Definition

- For example, a definition of the sequence *g,*

$$g(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2 \cdot g(n-1) & \text{if } n > 0 \end{cases}$$

can be converted to a simple formula $g(n) = 2^n$

- The language element statement is defined recursively

```
<statement> ::= while (<expression>) <statement> |
                if (<expression>) <statement> |
                 if (<expression>) <statement>else<statement> |
                . . .
```

Advantage of recursive definition

- The advantage of a recursive definition is that , no effort is needed to make the transition from a recursive definition of a function to its implementation in C++.

```
unsigned int factorial (unsigned int n) {
  if (n == 0)
       return 1;
  else return n * factorial (n - 1);
}
```

Function Calls and Recursive Implementation

What happens when a function is called?

If the function has formal parameters, they have to be initialized to the values passed as actual parameters.

The system has to know where to resume execution of the program after the function has finished.

Function Calls and Recursive Implementation

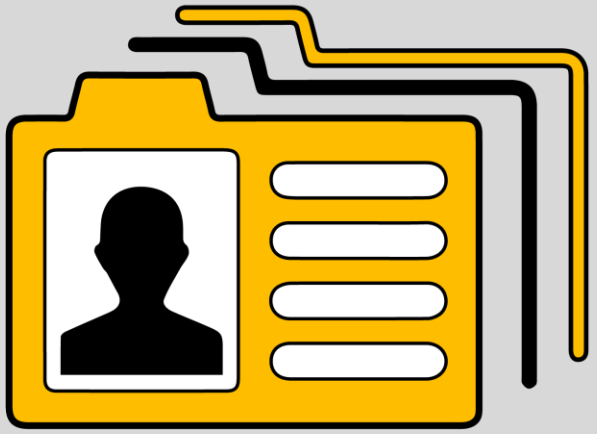
- The function can be called by other functions or by the main program (the function `main()`). Therefore, the return address should be stored in main memory in a place set aside for return addresses.
- Since, we do not know in advance how much space might be needed, and allocating too much space for that purpose alone is not efficient, we use dynamic allocation of memory.
- Besides for a function call, more information has to be stored than just a return address. Therefore, dynamic allocation using the run-time stack is a much better solution.

What information should be stored when a function is called?

- First, local variables must be stored. If function `f1()`, which contains a declaration of a local variable `x`, calls function `f2()`, which locally declares the variable `x`, the system has to make a distinction between these two variables `x`.

What information should be stored when a function is called?

- Basically, the state of each function including main() should be stored.
- The state of each function is characterized by
  - Contents of all automatic variables
  - Values of the function's parameters
  - Return addresses indicating where to restart in the calling function
- The data area containing all this information is called an **activation record** or a **stack frame** and is allocated on the runtime stack.



Activation Record

- An activation record exists for as long as a function owning it is executing.
- This record is a private pool of information for the function, a repository that stores all information necessary for its proper execution and how to return to where it was called from.
- Activation records usually have a short life span because they are dynamically allocated at function entry and deallocated upon exiting.
- Only the activation record of main() outlives every other activation record.

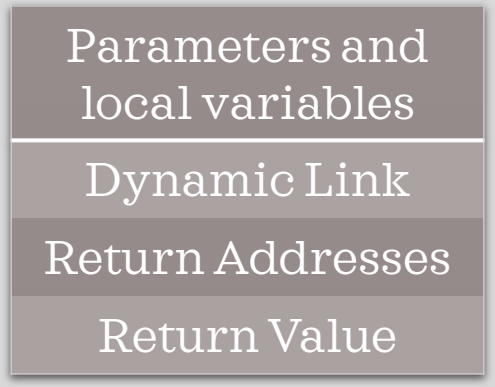
Activation Record

- An activation record usually contains the following information:

■ Values for all parameters to the function, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items.

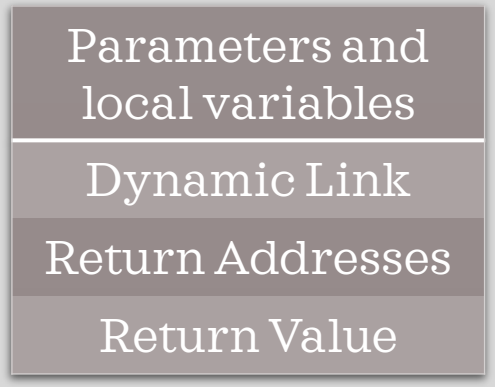
■ Local variables that can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored.

■ The return address to resume control by the caller, the address of the caller's instruction immediately following the call.

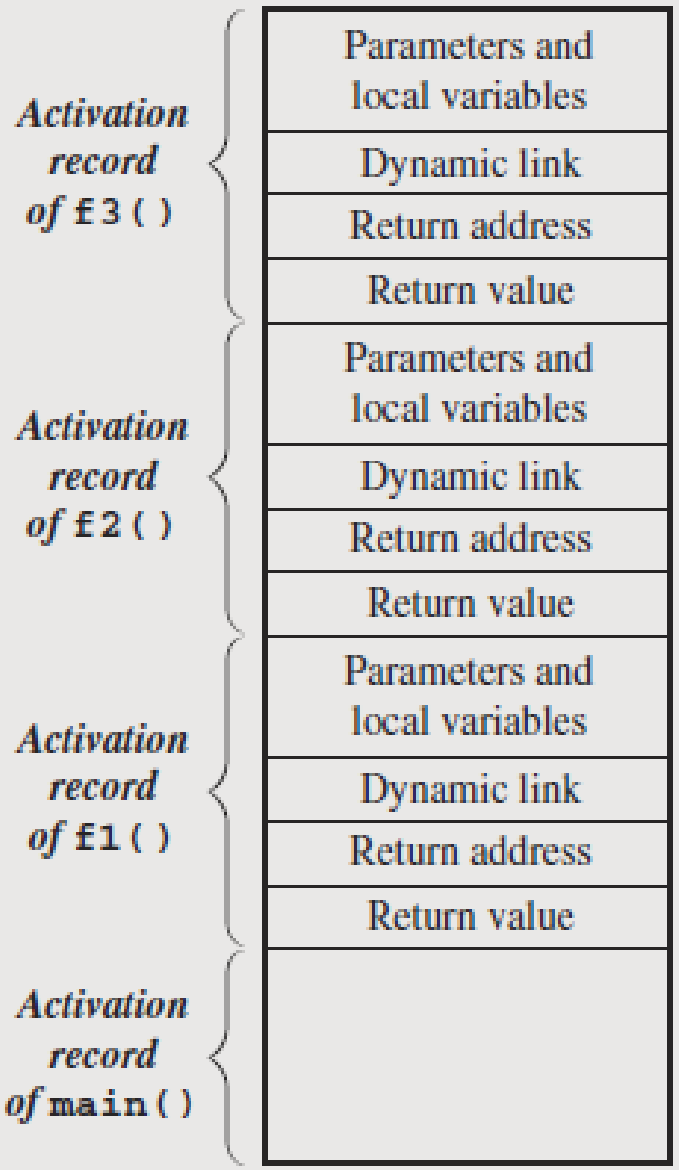


Activation Record

- A dynamic link, which is a pointer to the caller's activation record.
- The returned value for a function not declared as `void`. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.
  - If a function is called either by main() or by another function, then its activation record is created on the run-time stack. The run-time stack always reflects the current state of the function.



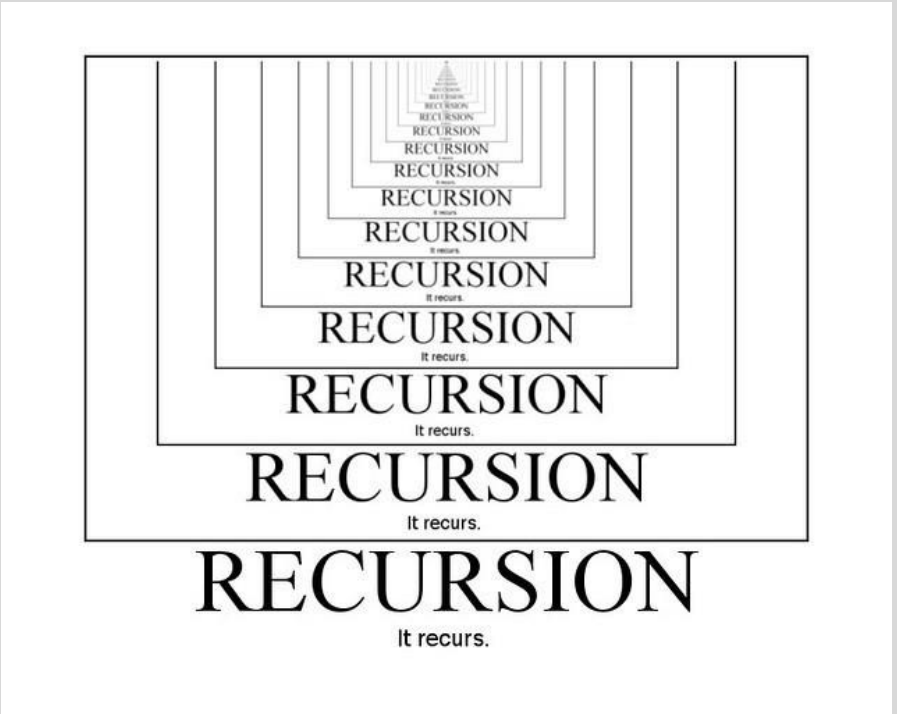
Suppose that `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` in turn calls `f3()`. If `f3()` is being executed, then the state of the run-time stack is as shown in Figure.



Activation Record

- By the nature of the stack, if the activation record for `f3()` is popped by moving the stack pointer right below the return value of `f3()`, then `f2()` resumes execution and now has free access to the private pool of information necessary for reactivation of its execution.
- On the other hand, if `f3()` happens to call another function `f4()`, then the run-time stack increases its height because the activation record for `f4()` is created on the stack and the activity of `f3()` is suspended.

- Creating an activation record whenever a function is called allows the system to handle recursion properly.
- Recursion is calling a function that happens to have the same name as the caller.
- A recursive call is not literally a function calling itself, but rather an instantiation of a function calling another instantiation of the same original.
- These invocations are represented internally by different activation records and are thus differentiated by the system.



Anatomy of a Recursive Call

- **Power function**: A function that defines raising any number x to a non-negative integer power n.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

- A C++ function for computing x^n can be directly written is as follows:

```
/* 102 */ double power (double x, unsigned int n) {
/* 103 */       if (n == 0)
/* 104 */             return 1.0;
              // else
/* 105 */             return x * power(x,n-1);
            }
```

Anatomy of a Recursive Call

- Using this definition, the value of x^4 can be computed in the following way:

Call 1	$x^4 = x.\ x^3$	$=x.x.x.x$
Call 2	$x.x^2$	$=x.x.x$
Call 3	$x.x^1$	$=x.x$
Call 4	$x.x^0$	$=x.1 =x$
Call 5	1	

- The repetitive application of the inductive step eventually leads to anchor, which is the last step in the chain of recursive calls.
- The anchor produces 1 as a result of x^0 ; the result is passed back to previous recursive call.
- Now, the call whose execution is pending returns the result, x.1 =x.
- The third call, which has been waiting for this result, computes its own result, namely x.x and returns it.
- In this way, each new call increases the level of recursion.

Anatomy of a Recursive Call

```
call 1          power(x,4)
call 2                power(x,3)
call 3                      power(x,2)
call 4                            power(x,1)
call 5                                  power(x,0)
```

Anatomy of a Recursive Call

```
call 1                  power(x,4)
call 2                      power(x,3)
call 3                          power(x,2)
call 4                              power(x,1)
call 5                                  power(x,0)
call 5                                  1
call 4                              x
call 3                          x · x
call 2                      x · x · x
call 1                  x · x · x · x
```

Anatomy of a Recursive Call

- When a function is executed it keeps track of all calls on its run time stack.
- Each line of code is assigned a number by the system, and if a line is a function call, then its number is a return address. The address is used by the system to remember where to resume execution after the function has completed.
- For this example, assume that the lines in the function power() are assigned the numbers 102 through 105 and that it is called main() from the statement.

```
            int main()
            { ...
/* 136 */    y = power(5.6,2);
              ...
            }
```

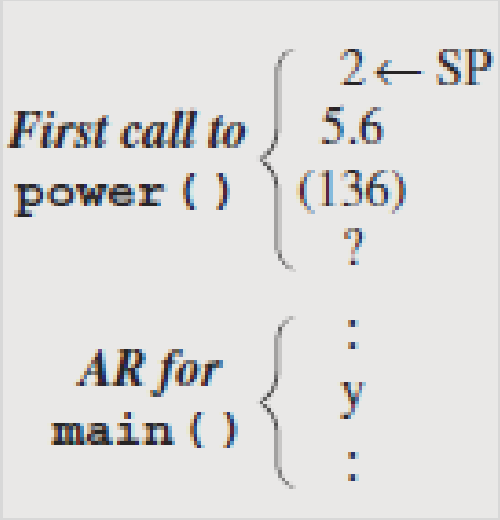
Anatomy of a Recursive Call

- A trace of recursive calls is as follows:

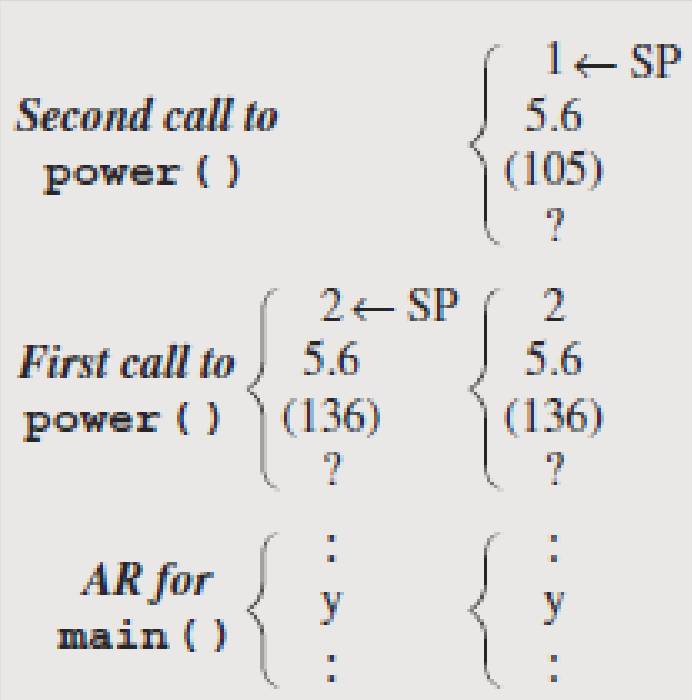
```
call 1                  power(5.6,2)
call 2                        power(5.6,1)
call 3                              power(5.6,0)
call 3                              1
call 2                        5.6
call 1                  31.36
```

Anatomy of a Recursive Call

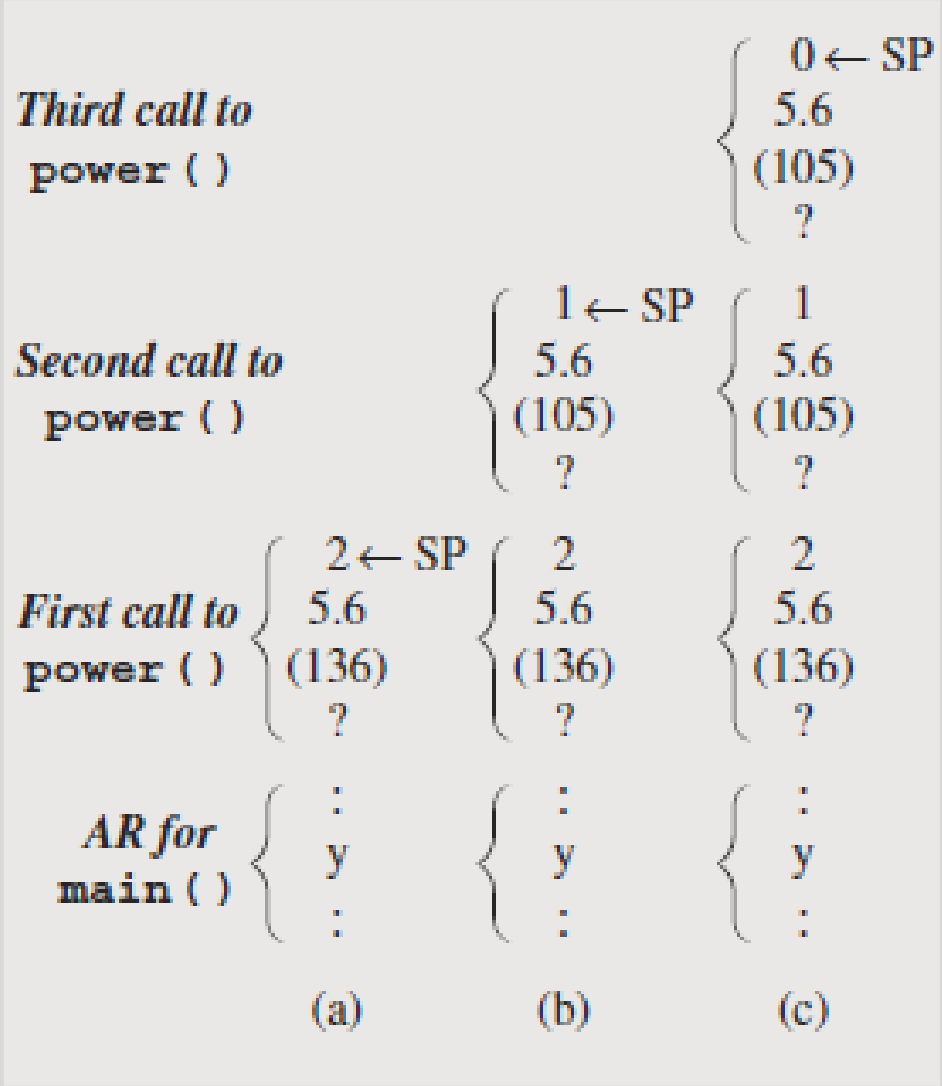
- When the function is invoked for the first time, four items are pushed onto the run-time stack:
  - return address 136
  - actual parameters 5.6 and 2
  - one location reserved for the value returned by power().
- SP is a stack pointer, AR is an activation record, and question marks (?) stand for locations reserved for the returned values.
- The addresses are represented in brackets (), to distinguish them from actual values.



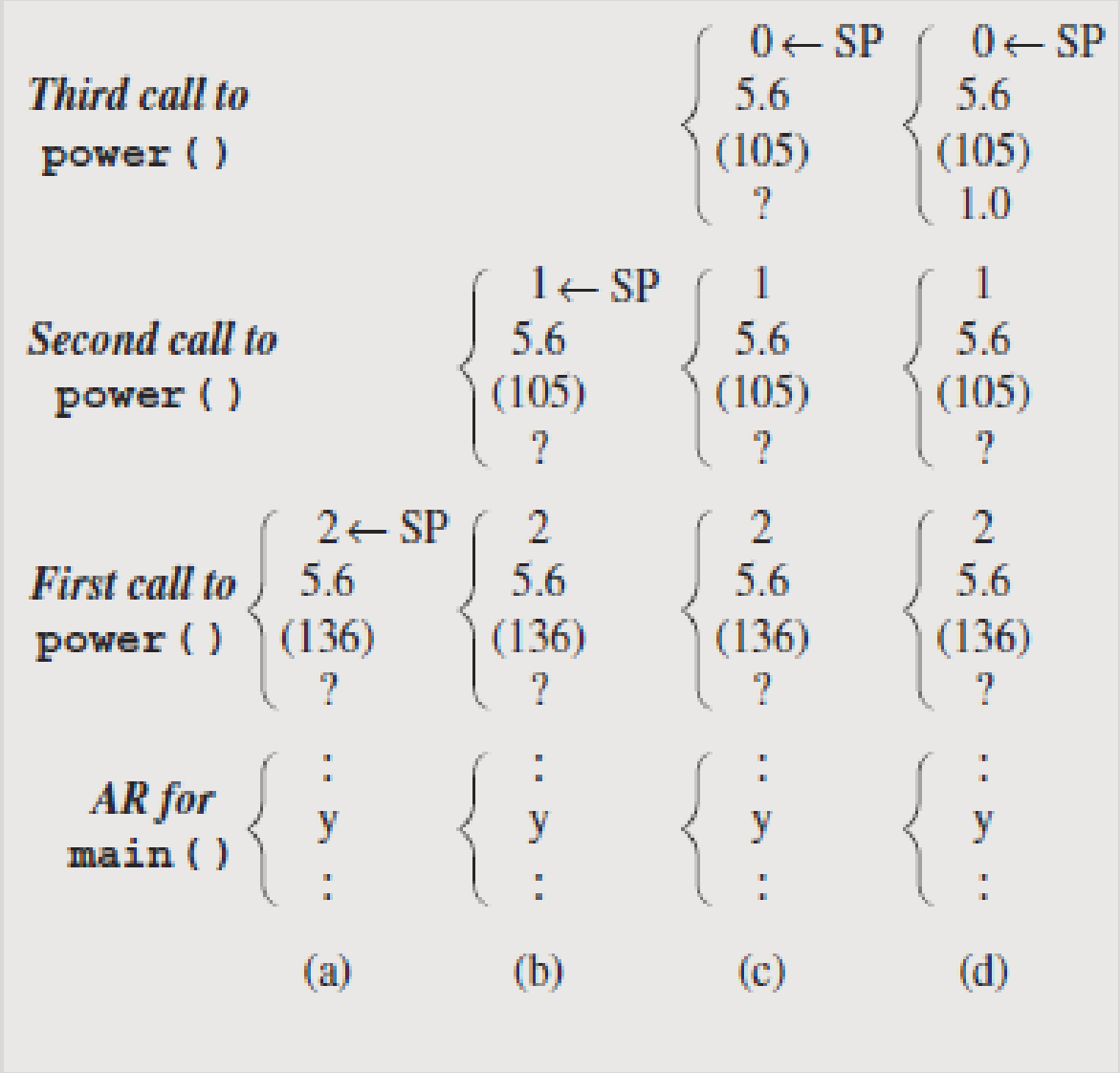
- Now the function power() is executed. First, the value of the second argument, 2, is checked, and power() tries to return the value of 5.6 · power(5.6,1) because that argument is not 0.
- This cannot be done immediately because the system does not know the value of power(5.6,1); it must be computed first.
- Therefore, power() is called again with the arguments 5.6 and 1. But before this call is executed, the runtime stack receives new items, and its contents are shown in Figure.



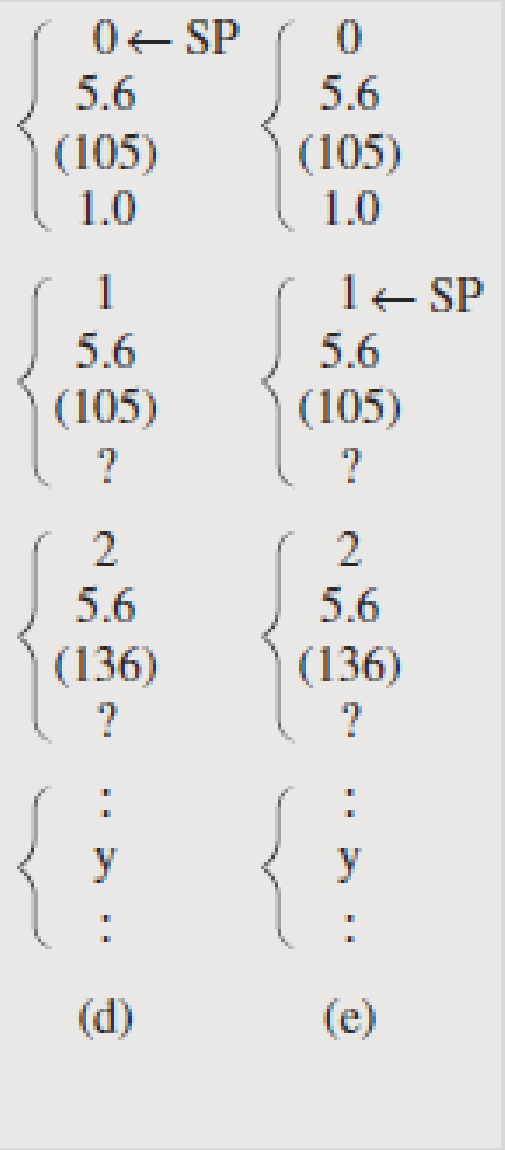
- Again, the second argument is checked to see if it is 0. Because it is equal to 1, `power()` is called for the third time, this time with the arguments 5.6 and 0.
- Before the function is executed, the system remembers the arguments and the return address by putting them on the stack, not forgetting to allocate one cell for the result.

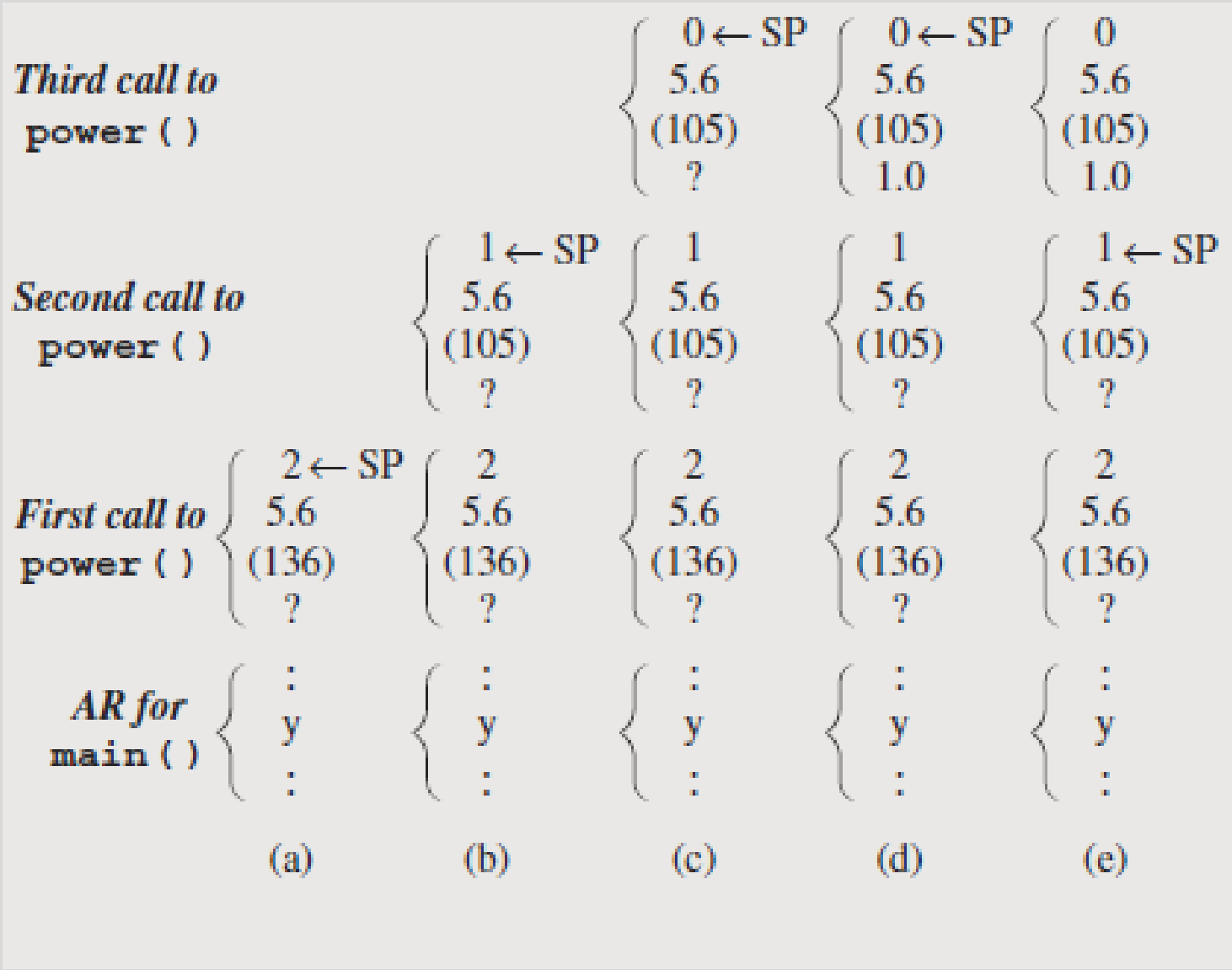


- We check again if the second argument equal to zero?
- Because it finally is, a concrete value—namely, 1.0—can be returned and placed on the stack, and the function is finished without making any additional calls.

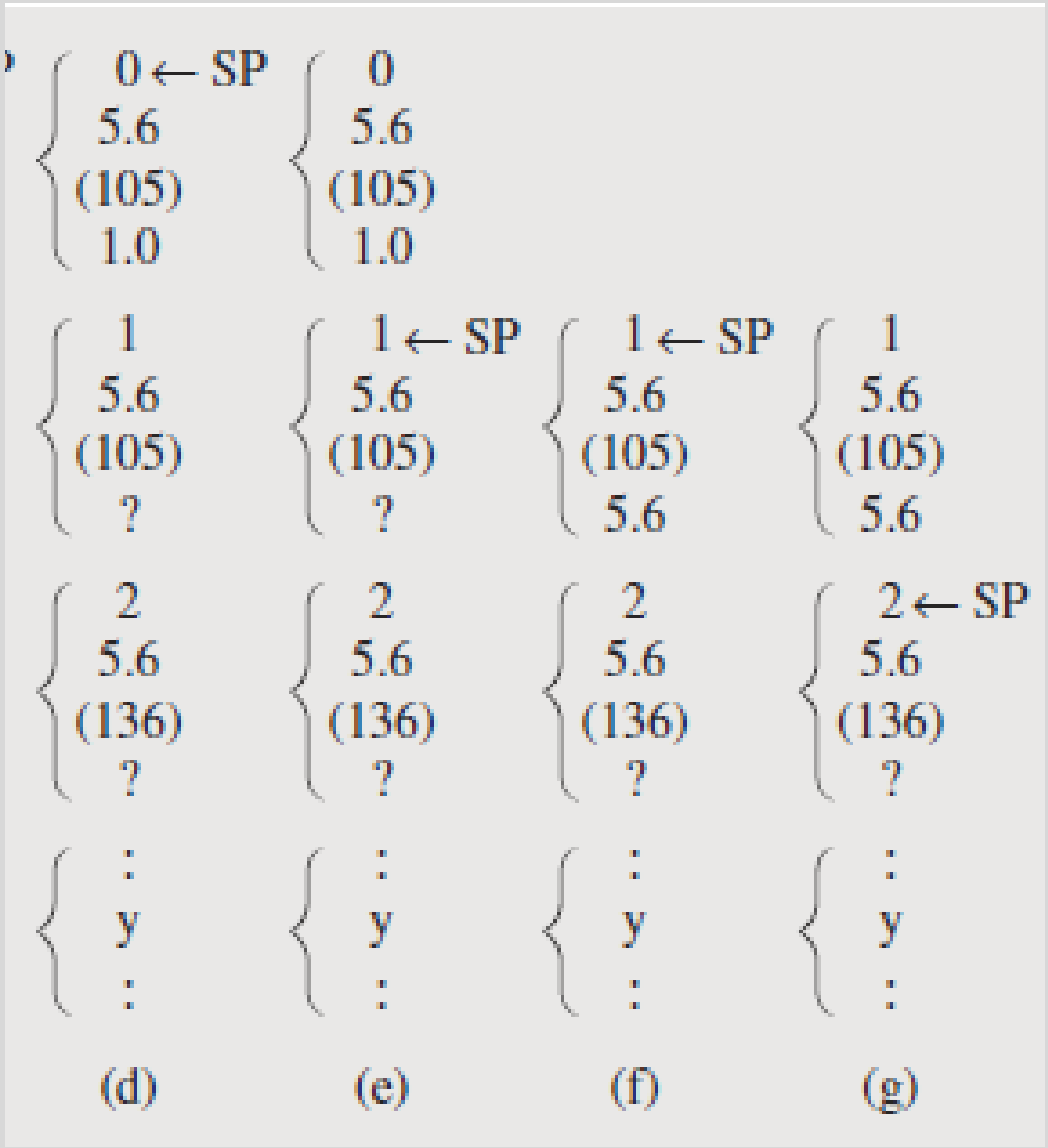


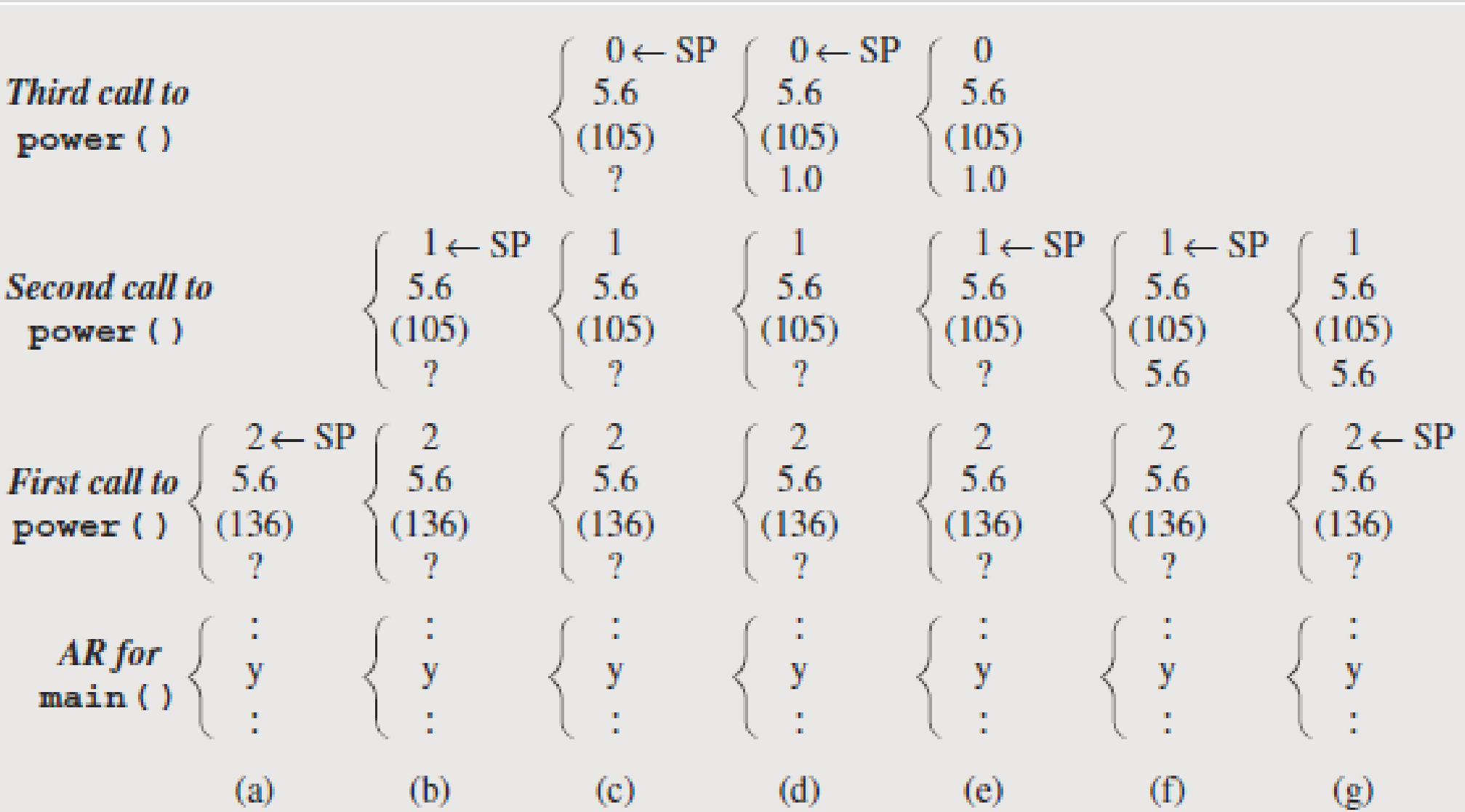
- At this point, there are two pending calls to power() on the run-time stack.
- The system first eliminates the activation record of `power()` that has just finished. This is performed logically by popping all its fields (the result, two arguments, and the return address) off the stack.
- We say “logically” because physically all these fields remain on the stack and only the SP is decremented appropriately.
- This is important because we do not want the result to be destroyed since it has not been used yet.
- Before and after completion of the last call of `power()`, the stack looks the same, but the SP’s value is changed.



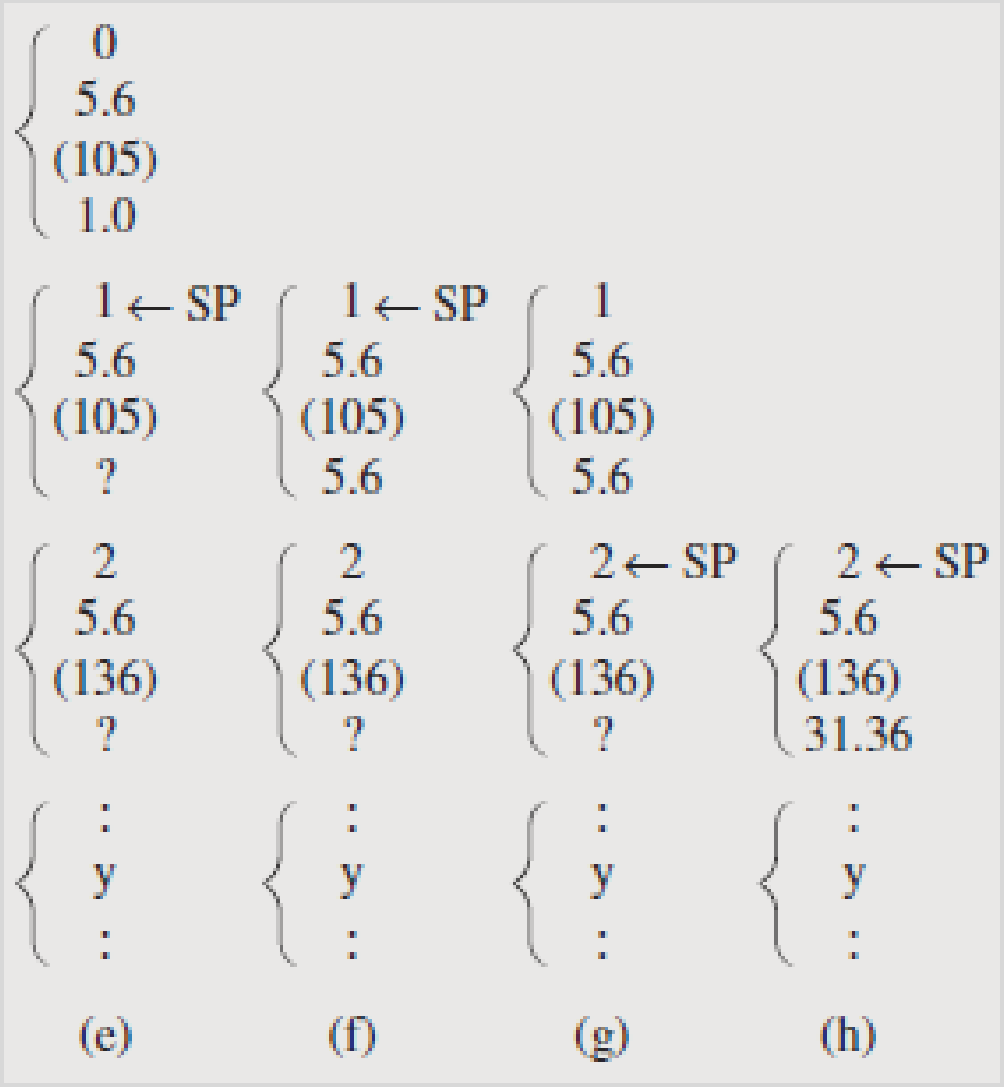

Third call to power ( )
Second call to power ( )
First call to power ( )
AR for main ( )
0 ← SP
5.6
(105)
?
1.0
1 ← SP
2 ← SP
(136)
y
(a)
(b)
(c)
(d)
(e)

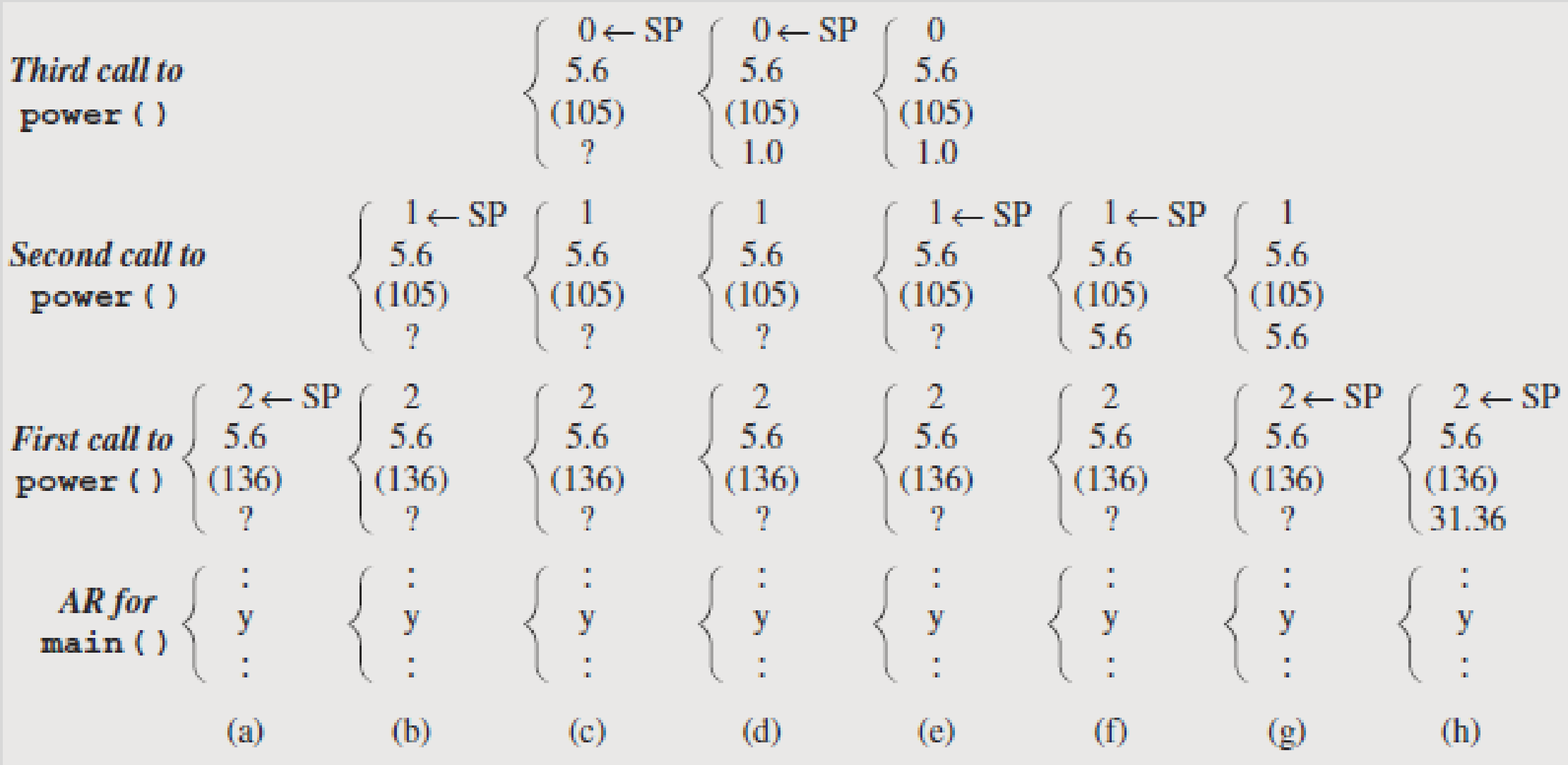
- Now the second call to `power()` can complete because it waited for the result of the call `power(5.6,0)`.
- This result, 1.0, is multiplied by 5.6 and stored in the field allocated for the result.
- After that, the system can pop the current activation record off the stack by decrementing the SP, and it can finish the execution of the first call to `power()` that needed the result for the second call.




Third call to power ( )
0 ← SP
5.6
(105)
?
0 ← SP
5.6
(105)
1.0
0
5.6
(105)
1.0
Second call to power ( )
1 ← SP
5.6
(105)
?
1
5.6
(105)
?
1
5.6
(105)
?
1 ← SP
5.6
(105)
?
1 ← SP
5.6
(105)
5.6
1
5.6
(105)
5.6
First call to power ( )
2 ← SP
5.6
(136)
?
2
5.6
(136)
?
2
5.6
(136)
?
2
5.6
(136)
?
2
5.6
(136)
?
2 ← SP
5.6
(136)
?
AR for main ( )
:
y
:
:
y
:
:
y
:
:
y
:
:
y
:
:
y
:
(a)
(b)
(c)
(d)
(e)
(f)
(g)

- At this moment, `power()` can finish its first call by multiplying the result of its second call, 5.6, by its first argument, also 5.6.
- The system now returns to the function that invoked `power()`, and the final value, 31.36, is assigned to `y`.
- Right before the assignment is executed, the content of the stack looks like Figure 5.2h.




Third call to power ( )
Second call to power ( )
First call to power ( )
AR for main ( )
0 ← SP
5.6
(105)
?
1.0
0
1 ← SP
1
2 ← SP
2
(136)
31.36
y
(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)

Non-recursive version of power()

```
double nonRecPower(double x, unsigned int n) {
  double result = 1;
  for (result = x; n > 1; --n)
       result *= x;
  return result;
}
```

The recursive version is more intuitive, increases program readability, improves self-documentation, and simplifies coding.

Tail Recursion

- Tail recursion is characterized by the use of only one recursive call at the very end of a function implementation.
- In other words, when the call is made, there are no statements left to be executed by the function; the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. An Example:

```
void tail(int i) {
  if (i > 0) {
    cout << i << '';
    tail(i-1);
  }
}
```

Tail Recursion

- What will be the output of tail(3) ?

Call 1	tail(3)	3
Call 2	tail (2)	3 2
Call 3	tail (1)	3 2 1

Tail Recursion

- Tail recursion is simply a glorified loop and can be easily replaced by one.
- In this example, it is replaced by substituting a loop for the `if` statement and decrementing the variable `i` in accordance with the level of recursive call.

```
void iterativeEquivalentOfTail(int i) {
  for ( ; i > 0; i--)
    cout << i << '';
}
```

Tail Recursion

- The following example is of non-tail recursion.

```
void nonTail(int i) {
  if (i > 0) {
    nonTail(i-1);
    cout << i << '';
    nonTail(i-1);
  }
}
```

- There is no advantage as such of using tail recursion over iteration in languages such as C++, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion is important.

Nontail Recursion

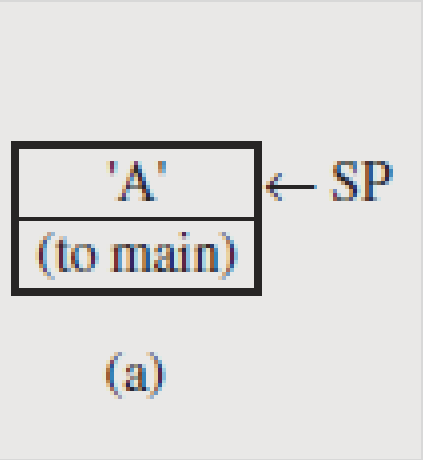
- Another problem that can be implemented in recursion is printing an input line in reverse order.

```
/* 200 */ void reverse() {
              char ch;
/* 201 */      cin.get(ch);
/* 202 */      if (ch != '\n') {
/* 203 */          reverse();
/* 204 */          cout.put(ch);
              }
          }
```

- main() calls reverse and the input is the string “ABC”.
- Draw appropriate activation records for the function calls.

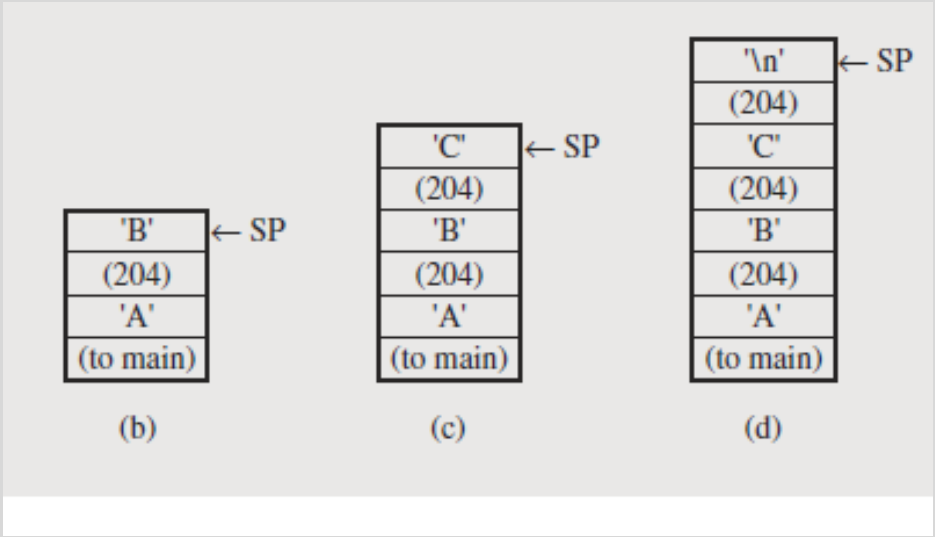
Nontail Recursion

- First, an activation record is created with cells for the variable `ch` and the return address. There is no need to reserve a cell for a result, because no value is returned, which is indicated by using `void` in front of the function's name.
- The function `get()` reads in the first character, "A."
- Figure shows the contents of the run-time stack right before `reverse()` calls itself recursively for the first time.



Nontail Recursion

- The second character is read in and checked to see if it is the end-of-line character, and if not, `reverse()` is called again.
- But in either case, the value of `ch` is pushed onto the run-time stack along with the return address.
- Note that the function is called as many times as the number of characters contained in the input string, including the end-of-line character. In our example, `reverse()` is called four times.



Nontail Recursion

- On the fourth call, `get()` finds the end-of-line character and `reverse()` executes no other statement.
- The system retrieves the return address from the activation record and discards this record by decrementing SP by the proper number of bytes.
- Execution resumes from line 204, which is a print statement. Because the activation record of the third call is now active, the value of `ch`, the letter “C,” is output as the first character.

Nontail Recursion

- Next, the activation record of the third call to `reverse()` is discarded and now SP points to where “B” is stored. The second call is about to be finished, but first, “B” is assigned to `ch` and then the statement on line 204 is executed, which results in printing “B” on the screen right after “C.”
- Finally, the activation record of the first call to `reverse()` is reached. Then “A” is printed, and what can be seen on the screen is the string “CBA.”
- The first call is finally finished and the program continues execution in `main()`.

Non-recursive version of the same function

```
void simpleIterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.getline(stack,80);
  for (top = strlen(stack) - 1; top >= 0; cout.put(stack[top--]));
}
```

Non-recursive version of the same function

- If we are not supplied with strlen() and getline() functions, then our iterative function has to be implemented differently.
- The while loop replaces getline() and the autoincrement variable top replaces strlen().
- The variable named st stands for stack. We are just making explicit what is done implicitly by the system. Our stacks takes over the runt-time stack's duty.

```
void iterativeReverse() {
  char stack[80];
  register int top = 0;
  cin.get(stack[top]);
  while(stack[top]!='\n')
    cin.get(stack[++top]);
  for (top -= 2; top >= 0; cout.put(stack[top--]));
}
```

Non-recursive version of the same function

- We can have a global stack object st outside the function, with the declaration

  stacks<char> st

```
void nonRecursiveReverse() {
    int ch;
    cin.get(ch);
    while (ch != '\n') {
        st.push(ch);
        cin.get(ch);
    }
    while (!st.empty())
        cout.put(st.pop());
}
```

One way or the other, the transformation of nontail recursion into iteration usually involves the explicit handling of a stack.