



## Chapter Eleven: Recursion

# Chapter Goals

---

- To learn to “think recursively”
- To be able to use recursive helper functions
- To understand the relationship between recursion and iteration
- To understand when the use of recursion affects the efficiency of an algorithm
- To analyze problems that are much easier to solve by recursion than by iteration

*Recursion* is a powerful technique  
for breaking up complex computational  
problems into simpler ones,  
where the “simpler one”  
***is*** the ***solution*** to the whole problem!

# Recursion

A recursive function is a function that calls itself,  
reducing the problem a bit on each call:

```
void solveIt(the-Problem)
{
    . . .
    solveIt(the-Problem-a-bit-reduced) ;
}
```

Of course, there's a lot more to it than this.

# Recursion

---

In recursion,  
the same computation recurs,  
or occurs repeatedly,  
as the problem is solved.

But this is not looping!

Recursion is often the most natural way  
of thinking about a problem,

and there are some computations  
that are very difficult to perform without recursion.

# Recursion

---

Of course, just as you have been doing as you learn  
how to program, you'll have to first learn  
(or relearn)

how to think...

...how to “think recursively”.

# How to think recursively – Triangles of Boxes

Let's begin by solving this problem:

Write a function,  
yes, of course recursive one!,  
that prints a triangle of “boxes”

```
void print_triangle(int side_length) ;
```

```
[]
```

```
[] []
```

```
[] [] []
```

```
[] [] [] []
```



# How to think recursively – Triangles of Boxes

Of course this is easily done with nested loops,  
but we are teaching you how to think recursively.

So think recursively!

```
[]  
[] []  
[] [] []  
[] [] [] []
```

# How to think recursively – Triangles of Boxes

---

How to think recursively?

# How to think recursively – Triangles of Boxes

---

It helps if you are  
(or pretend to be while thinking recursively)  
a bit lazy:

let others do most of the work for you!

# How to think recursively – Triangles of Boxes

Pretend that “someone else”  
has written the function that draws triangles.

Now that problem is solved, analyze the problem,  
looking for a way to reduce the problem and  
use that function to solve the reduced problem:

```
[ ]  
[ ] [ ]  
[ ] [ ] [ ]  
[ ] [ ] [ ] [ ]
```

# How to think recursively – Triangles of Boxes

You can call that function  
(that someone else wrote)  
to solve the reduced problem  
of printing a triangle of side length three,

```

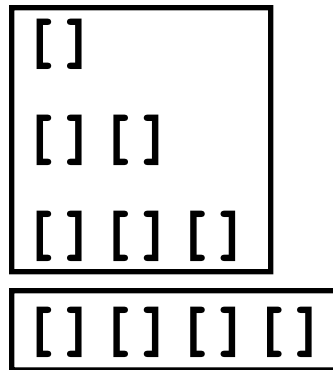
[ ]
[ ] [ ]
[ ] [ ] [ ]
[ ] [ ] [ ] [ ]

```

# How to think recursively – Triangles of Boxes


*then*

you solve the much easier problem of  
printing a line of 4 boxes.

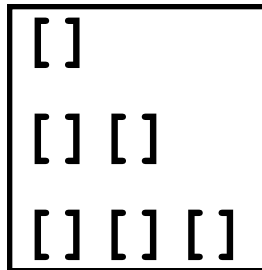


# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)
{
    print_triangle(side_length - 1);
}
```



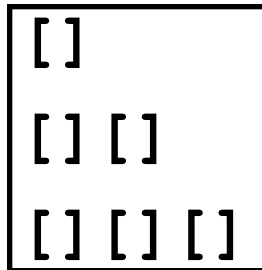
The problem is being reducing by making the length 1 shorter.



```
[]
[] []
[] [] []
```

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)
{
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```



```
[]
[] []
[] [] []
```

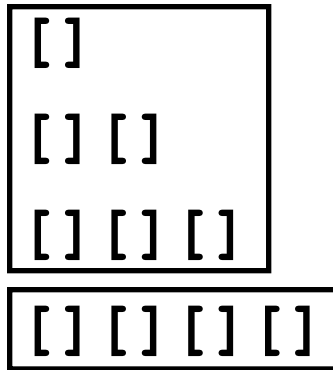


Now that simple **for** statement to draw a line **side\_length** long.



# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)
{
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```



Now that simple **for** statement to draw a line **side\_length** long.

# How to think recursively – Triangles of Boxes

---

BUT WAIT!!!

```
[ ]  
[ ] [ ]  
[ ] [ ] [ ]  
[ ] [ ] [ ] [ ]
```

# How to think recursively – Triangles of Boxes

---

BUT WAIT!!!

[ ]

[ ] [ ]

[ ] [ ] [ ]

BUT WAIT!!!

[ ]

[ ] [ ]

BUT WAIT!!!

[ ]

# How to think recursively – Triangles of Boxes

---

What if the  
side length was...

one?

[ ]

# How to think recursively – Triangles of Boxes

---

Drawing a line of only one box is easy, but...

we certainly don't want to *recurse*,  
reducing the problem to a triangle of

**0** length,

**[]**

# How to think recursively – Triangles of Boxes

---

Drawing a line of only one box is easy, but...

we certainly don't want to *recurse*,  
reducing the problem to a triangle of

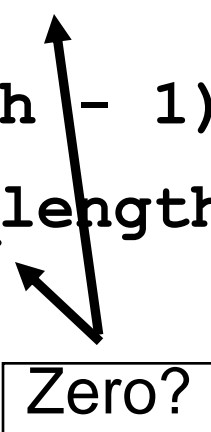
**0** length, do we?!?!?!?

[ ]



# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)
{
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```



The diagram illustrates the recursive process. Two arrows originate from a box labeled "Zero?". One arrow points to the expression "side\_length - 1" in the recursive call "print\_triangle(side\_length - 1);". The other arrow points to the loop condition "i < side\_length" in the "for" loop. This visualizes how the base case of zero is used to determine the number of iterations and recursive steps.

# How to think recursively – Triangles of Boxes

---

When the length of the line is less than one,  
we simply do not want to make the recursive call  
  
at all!

So we must test for:  
the END CONDITION.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

When we are through, stop...

...don't recurse any more.

# How to think recursively – The Two Key Requirements

---

The two keys requirements  
for a successful recursive function:

- Every recursive call must simplify the task in some way.
- There must be special cases to handle the simplest tasks directly so that the function will stop calling itself.

# How to think recursively – Triangles of Boxes

---

We can “trace” this recursive function call:

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

CURRENT CALL: print\_triangle(4)

side\_length: 4

Someone calls print\_triangle(4)

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(4)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **4**

Someone calls `print_triangle(4)`  
-In call `print_triangle(4)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(4)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **4**

Someone calls `print_triangle(4)`

-The call `print_triangle(4)` calls `print_triangle(3)`.



# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(3)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: 3

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

--In call `print_triangle(3)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(3)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **3**

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

--In call `print_triangle(3)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(3)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **3**

Someone calls `print_triangle(4)`

-The call `print_triangle(4)` calls `print_triangle(3)`.

--The call `print_triangle(3)` calls `print_triangle(2)`.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(2)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: 2

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

--The call `print_triangle(3)` calls `print_triangle(2)`.

---In call `print_triangle(2)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(2)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **2**

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

---The call `print_triangle(3)` calls `print_triangle(2)`.

---In call `print_triangle(2)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(2)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **2**

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

--The call `print_triangle(3)` calls `print_triangle(2)`.

---The call `print_triangle(2)` calls `print_triangle(1)`.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(1)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: 1

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

---The call `print_triangle(3)` calls `print_triangle(2)`.

----The call `print_triangle(2)` calls `print_triangle(1)`.

-----In call `print_triangle(1)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(1)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **1**

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

---The call `print_triangle(3)` calls `print_triangle(2)`.

----The call `print_triangle(2)` calls `print_triangle(1)`.

-----In call `print_triangle(1)`



# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(1)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: 1

Someone calls `print_triangle(4)`

-The call `print_triangle(4)` calls `print_triangle(3)`.

--The call `print_triangle(3)` calls `print_triangle(2)`.

---The call `print_triangle(2)` calls `print_triangle(1)`.

----The call `print_triangle(1)` calls `print_triangle(0)`.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(0)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: 0

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

---The call `print_triangle(3)` calls `print_triangle(2)`.

----The call `print_triangle(2)` calls `print_triangle(1)`.

-----The call `print_triangle(1)` calls `print_triangle(0)`.

-----In call `print_triangle(0)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(0)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: 0

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

---The call `print_triangle(3)` calls `print_triangle(2)`.

----The call `print_triangle(2)` calls `print_triangle(1)`.

-----The call `print_triangle(1)` calls `print_triangle(0)`.

-----In call `print_triangle(0)`

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(0)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **0**

Someone calls `print_triangle(4)`

--The call `print_triangle(4)` calls `print_triangle(3)`.

---The call `print_triangle(3)` calls `print_triangle(2)`.

----The call `print_triangle(2)` calls `print_triangle(1)`.

-----The call `print_triangle(1)` calls `print_triangle(0)`.

-----The call `print_triangle(0)` returns, doing nothing.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(1)
{
    if (side_length < 1) { return; }           side_length: 1
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}

[]
```

Someone calls `print_triangle(4)`

-The call `print_triangle(4)` calls `print_triangle(3)`.

--The call `print_triangle(3)` calls `print_triangle(2)`.

---The call `print_triangle(2)` calls `print_triangle(1)`.

----The call `print_triangle(1)` calls `print_triangle(0)`.

-----The call `print_triangle(0)` returns, doing nothing.

----Back in `print_triangle(1)` prints `[]` and returns.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(2)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **2**

[]  
[] []

Someone calls `print_triangle(4)`

-The call `print_triangle(4)` calls `print_triangle(3)`.

--The call `print_triangle(3)` calls `print_triangle(2)`.

---The call `print_triangle(2)` calls `print_triangle(1)`.

----The call `print_triangle(1)` calls `print_triangle(0)`.

-----The call `print_triangle(0)` returns, doing nothing.

----Back in `print_triangle(1)` prints `[]` and returns.

--- Back in `print_triangle(2)` prints `[] []` and returns.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(3)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: 3

```
[]
[] []
[] [] []
```

Someone calls `print_triangle(4)`

- The call `print_triangle(4)` calls `print_triangle(3)`.
- The call `print_triangle(3)` calls `print_triangle(2)`.
- The call `print_triangle(2)` calls `print_triangle(1)`.
- The call `print_triangle(1)` calls `print_triangle(0)`.
- The call `print_triangle(0)` returns, doing nothing.
- Back in `print_triangle(1)` prints `[]` and returns.
- Back in `print_triangle(2)` prints `[] []` and returns.
- Back in `print_triangle(3)` prints `[] [] []` and returns.

# How to think recursively – Triangles of Boxes

```
void print_triangle(int side_length)  CURRENT CALL: print_triangle(4)
{
    if (side_length < 1) { return; }
    print_triangle(side_length - 1);
    for (int i = 0; i < side_length; i++)
    {
        cout << "[]";
    }
    cout << endl;
}
```

side\_length: **4**

```
[]
[] []
[] [] []
[] [] [] []
```

Someone calls `print_triangle(4)`

- The call `print_triangle(4)` calls `print_triangle(3)`.
- The call `print_triangle(3)` calls `print_triangle(2)`.
- The call `print_triangle(2)` calls `print_triangle(1)`.
- The call `print_triangle(1)` calls `print_triangle(0)`.
- The call `print_triangle(0)` returns, doing nothing.
- Back in `print_triangle(1)` prints `[]` and returns.
- Back in `print_triangle(2)` prints `[] []` and returns.
- Back in `print_triangle(3)` prints `[] [] []` and returns.
- Back in `print_triangle(4)` prints `[] [] [] []` and returns.



# How to think recursively – Triangles of Boxes

---

**Ah**

[ ]

[ ] [ ]

[ ] [ ] [ ]

[ ] [ ] [ ] [ ]

# Triangle Numbers

---

We can modify this problem to solve the triangle number problem:

```
[]  
[] []  
[] [] []  
[] [] [] []
```

# Triangle Numbers

We can modify this problem to solve the triangle number problem:

What is the sum of 1's in a triangular pattern?  
Or, what is the area of a triangle of height  $n$ ?

$n$		Triangle number:
1	1	1 (1)
2	1 1	3 (2 + 1)
3	1 1 1	6 (3 + 2 + 1)
4	1 1 1 1	10 (4 + 3 + 2 + 1)

# Triangle Numbers

---

We start by thinking about the end condition:  
when `side_length` is 1,  
the triangle number (the area) is 1 and we are done:

```
int triangle_area(int side_length)
{
    if (side_length == 1) { return 1; }

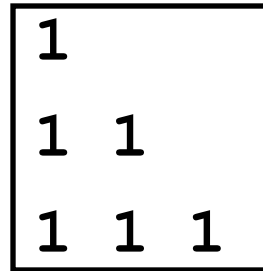
}
```

# Triangle Numbers

What about the reduced problem?

What value should we add to the current side length?

If the current side length is 4,



← this triangle number  
(the smaller triangle)



← plus this side length  
(the current side length)

```
int triangle_area(int side_length)
{
    if (side_length == 1) { return 1; }
    int smaller_side_length = side_length - 1;
    int smaller_area = triangle_area(smaller_side_length);
    return smaller_area + side_length;
}
```

# Triangle Numbers

There might be a problem with this code:

```
int triangle_area(int side_length)
{
    if (side_length == 1) { return 1; }
    int smaller_side_length = side_length - 1;
    int smaller_area = triangle_area(smaller_side_length);
    return smaller_area + side_length;
}
```

What would happen if this function were called with **-1**?

# Triangle Numbers

To make sure this doesn't happen,  
we should add a test that handles this situation:

```
int triangle_area(int side_length)
{
    if (side_length <= 0) { return 0; }

    if (side_length == 1) { return 1; }
    int smaller_side_length = side_length - 1;
    int smaller_area = triangle_area(smaller_side_length);
    return smaller_area + side_length;
}
```

# Triangle Numbers

---

The complete program follows.



# Triangle Numbers

ch11/triangle.cpp

```
/**
    Computes the area of a triangle with a given side length.
    @param side_length the side length of the triangle base
    @return the area
*/
int triangle_area(int side_length)
{
    if (side_length <= 0) { return 0; }
    if (side_length == 1) { return 1; }
    int smaller_side_length = side_length - 1;
    int smaller_area = triangle_area(smaller_side_length);
    return smaller_area + side_length;
}

int main()
{
    cout << "Enter the side length: ";
    int input;
    cin >> input;
    cout << "Area: " << triangle_area(input) << endl;
    return 0;
}
```

# Common Error: Infinite Recursion

---

Consider this function:

```
void forever_young( )  
{  
    cout << "I am ";  
    forever_young();  
    cout << "forever young!";  
}
```

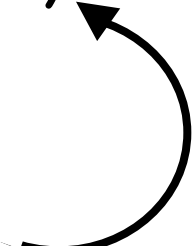
## Common Error: Infinite Recursion

The image consists of a background grid of the words "I am" repeated many times in a small, black, monospaced font. Overlaid on this grid are three large, stylized, handwritten-style phrases in a dark, bold script. The first phrase, "Jane!", is positioned at the top center. The second phrase, "How do I stop", spans across the middle section. The third phrase, "this crazy thing?", is located at the bottom center. The overall composition suggests a theme of self-identity or mental health struggles.

# Common Error: Infinite Recursion

You would get very, very old waiting to be “forever young!”

```
void forever_young( )  
{  
    cout << "I am ";  
    forever_young();  
    cout << "forever young!";  
}
```



# Common Error: Infinite Recursion

---

Infinite recursion is an error, for sure.

Each function call uses some system resources that only a **return** statement releases.

It is very likely that your computer will hang or crash or whatever you call it when it just WON'T STOP.

# Common Error: Infinite Recursion

---

In the previous code,  
the programmer forgot to write the end test.

This is easy to see in that code.

Infinite recursion can also occur  
when the test to end  
never happens.

# Common Error: Infinite Recursion

---

Recall the two key requirements  
for a successful recursive function:

- Every recursive call must simplify the task in some way.
- There must be special cases to handle the simplest tasks directly so that the function will stop calling itself.

Failing to implement these requirements can lead to infinite recursion (and very unpleasant consequences).

Go hang a salami,  
I'm a lasagna hog



A man,  
a plan,  
a canal

— Panama!

Madam,  
I'm Adam

rotor

# Recursion

---



# Thinking Recursively – Palindromes

---

*Palindrome:*

a string that is equal to itself  
when you reverse all characters

# Thinking Recursively – Palindromes

---

The problem:

Write a function to test if a string is a palindrome.

```
bool is_palindrome(string s)
```

# Thinking Recursively – Palindromes

---

We will, of course, be thinking recursively.

How to reduce the problem?

# Thinking Recursively – Palindromes

---

How about :

Remove the first character?

Remove the last character?

Remove both the first and the last character?

Remove a character from the middle?

Cut the string into two halves?



# Thinking Recursively – Palindromes

---

Every palindrome's first half  
is the same as its other (better?) half.

In this problem,  
chopping in half  
seems to be a good way to reduce the problem.

# Thinking Recursively – Palindromes



# Thinking Recursively – Palindromes

"rotor"

(chop)



Neither of these is a palindrome.  
Not sure how chopping in half gets us closer to a way to  
determine a palindromic situation.

# Thinking Recursively – Palindromes

---

One character at a time seems not so good.  
How about chopping off BOTH ends at the same time?

# Thinking Recursively – Palindromes



***Symmetrical***  
chopping.  
Nice!

# Thinking Recursively – Palindromes



oo!

"rotor"

(chop)

"r"

# Thinking Recursively – Palindromes



oo!

**"rotor"**

(chop)

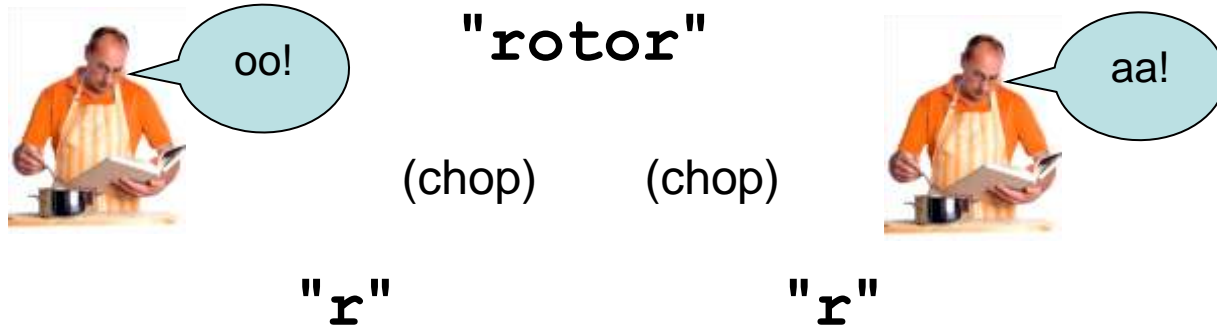
(chop)

**"r"**



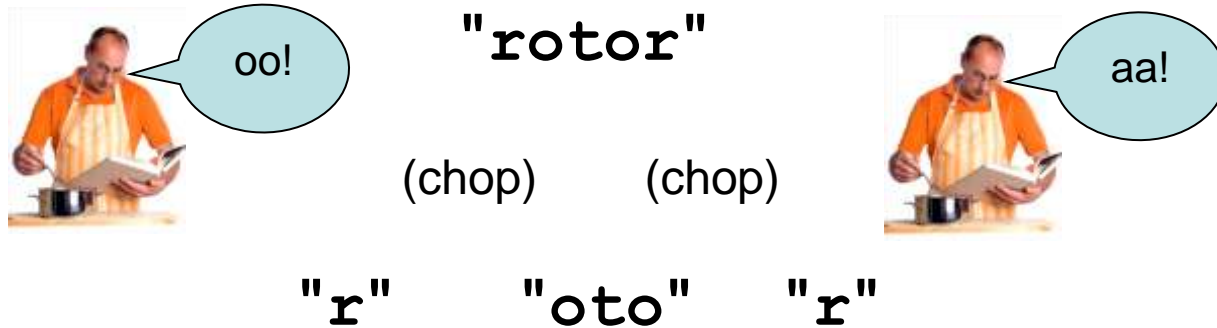
aa!

# Thinking Recursively – Palindromes

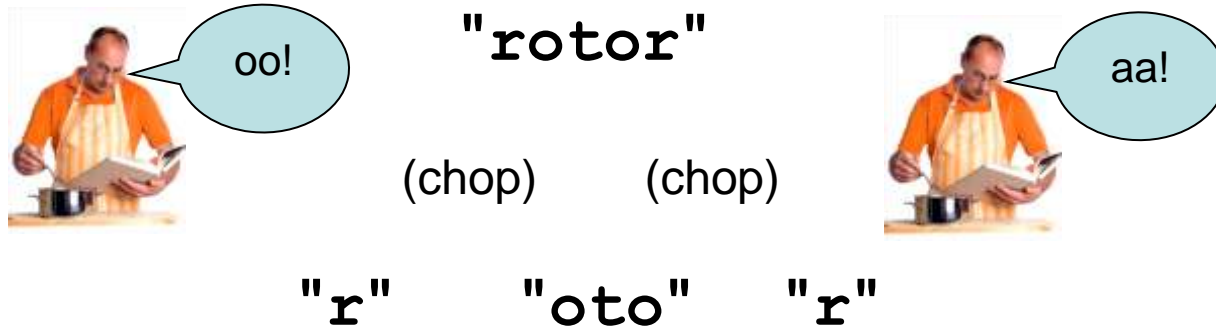




# Thinking Recursively – Palindromes



# Thinking Recursively – Palindromes



The 'r' on both ends means  
this is starting to look like a palindrome.

# Thinking Recursively – Palindromes

"rotor"

(chop)      (chop)

"r"      "oto"      "r"

We can reduce the problem  
to the “middle” of the string for the recursive call.

# Thinking Recursively – Palindromes

`"rotor"`

(chop)      (chop)

`"r"      "oto"      "r"`

So the recursive algorithm is:

# Thinking Recursively – Palindromes

"rotor"

(chop)      (chop)

"r"      "oto"      "r"

If the end letters are the same

AND

`is_palindrome ( the middle word )`

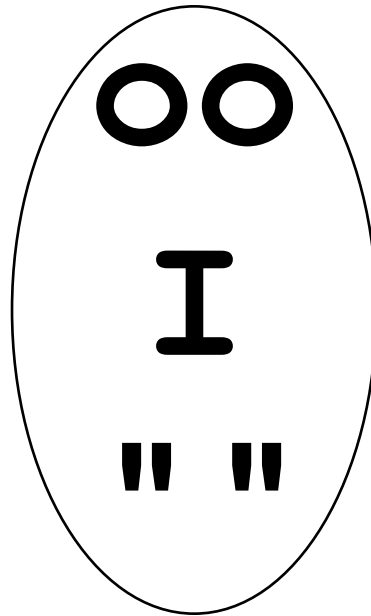
then the string is a palindrome!

# Thinking Recursively – Palindromes

Now we start thinking recursively:

when to end and how to handle those situations?

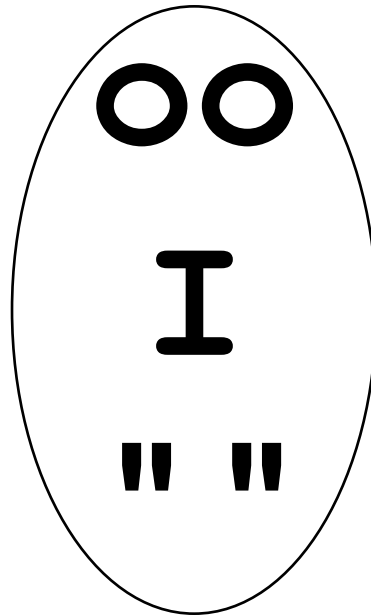
Do not be  
frightened  
- it's not  
really  
a monster!



It's very  
friendly,  
actually  
  
- even when you  
plan to chop it  
into pieces!

# Thinking Recursively – Palindromes

Each of these is a possible ending condition.



# Thinking Recursively – Palindromes

But wait!

"oo" will become "" when we recurse,  
taking off both ends.

oo

(chop)

""

""  
o

""  
o





# Thinking Recursively – Palindromes

That leaves us with two possible end situations:

string of length 0

string of length 1

```
bool is_palindrome(string s)
{
    // Separate case for shortest strings
    if (s.length() == 0 || s.length() == 1 ) { return true; }
```

# Thinking Recursively – Palindromes

Otherwise, we recurse on “the middle”.

```
// Get first and last character, converted to lowercase
char first = tolower(s[0]);
char last = tolower(s[s.length() - 1]);

if (first == last)
{
    string shorter = s.substr(1, s.length() - 2);
    return is_palindrome(shorter);
}
else
{
    return false;
}
}
```

# Thinking Recursively – Palindromes

ch11/palindrome.cpp

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

/**
    Tests whether a string is a palindrome. A palindrome
    is equal to its reverse, for example "rotor" or "racecar".
    @param s a string
    @return true if s is a palindrome
*/
bool is_palindrome(string s)
{
    // Separate case for shortest strings
    if (s.length() == 0 || s.length() == 1 ) { return true; }
    // Get first and last character, converted to lowercase
    char first = s[0];
    char last = s[s.length() - 1];
```

# Thinking Recursively – Palindromes

ch11/palindrome.cpp

```
if (first == last)
{
    string shorter = s.substr(1, s.length() - 2);
    return is_palindrome(shorter);
}
else
{
    return false;
}
}
```

```
int main()
{
    cout << "Enter a string: ";
    string input;
    getline(cin, input);
    cout << input << " is ";
    if (!is_palindrome(input)) { cout << "not "; }
    cout << "a palindrome." << endl;
    return 0;
}
```

# Recursive Helper Functions

---

Sometimes it is easier to find a recursive solution if you change the original problem slightly.

Then the original problem can be solved by calling a recursive helper function.

# Recursive Helper Functions

---

Consider the palindrome problem.

It is a bit inefficient to construct new string objects in every step.

# Recursive Helper Functions

---

Now consider the following change in the problem.

Rather than testing whether the  
entire string is a palindrome,

check whether a substring is a palindrome:

# Thinking Recursively – Palindromes

Check whether a substring is a palindrome:

```
/*  
Tests whether a substring of a string is a palindrome.  
@param s the string to test  
@param start the index of the first character of the substring  
@param end the index of the last character of the substring  
@return true if the substring is a palindrome  
*/  
bool substring_is_palindrome(string s, int start, int end);
```

This function turns out to be even easier  
to implement than the original test.

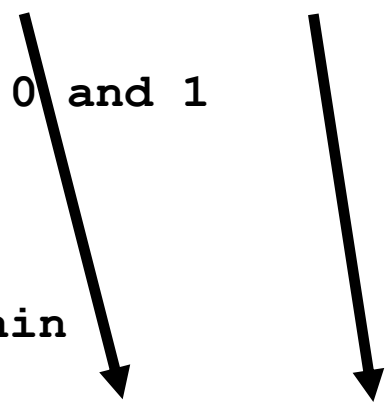


# Thinking Recursively – Palindromes

In the recursive calls, simply adjust the start and end arguments to skip over matching letter pairs.

There is no need to construct new string objects to represent the shorter strings.

```
bool substring_is_palindrome(string s, int start, int end)
{
    // Separate case for substrings of length 0 and 1
    if (start >= end) { return true; }
    if (s[start] == s[end])
    {
        // Test substring that doesn't contain
        // the first and last letters
        return substring_is_palindrome(s, start + 1, end - 1);
    }
    else
    {
        return false;
    }
}
```

Two black arrows originate from the 'start' and 'end' parameters of the recursive call 'substring\_is\_palindrome(s, start + 1, end - 1)'. The arrow from 'start' points to the 'start' parameter of the function signature 'substring\_is\_palindrome(string s, int start, int end)'. The arrow from 'end' points to the 'end' parameter of the same function signature.

# Thinking Recursively – Palindromes

---

Now that the helper function is written,  
don't forget to solve the problem!

# Thinking Recursively – Palindromes

Provide a function that solves the problem by calling the helper function.

This will be an easy one:

```
bool is_palindrome(string s)
{
    return substring_is_palindrome(s, 0, s.length() - 1);
}
```

This function is not recursive but the helper function is.

# The Efficiency of Recursion

---

As you have seen in this chapter,  
recursion can be a powerful tool for implementing  
complex algorithms.

On the other hand,  
recursion can lead to algorithms that perform poorly.

# The Efficiency of Recursion

The Fibonacci sequence  
is a sequence of numbers defined by the equation

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

# The Efficiency of Recursion

Each value in the sequence is  
the sum of the two preceding values.

The first ten terms of the sequence are:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...

The next entry would be  $34 + 55$ , or 89.  
This sequence continues forever.

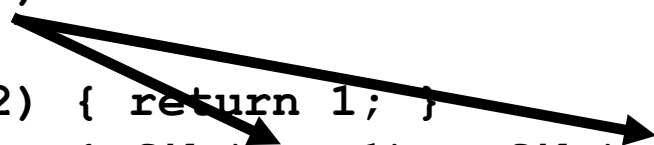
In code:

# Thinking Recursively – Palindromes

ch11/fibtest.cpp

```
#include <iostream>
using namespace std;
/**
    Computes a Fibonacci number.
    @param n an integer (non-negative)
    @return the nth Fibonacci number
*/
int fib(int n)
{
    if (n <= 2) { return 1; }
    else return { fib(n - 1) + fib(n - 2); }
}

int main()
{
    cout << "Enter n: ";
    int n;
    cin >> n;
    int f = fib(n);
    cout << "fib(" << n << ") = " << f << endl;
    return 0;
}
```

A diagram illustrating the recursive calls in the fib function. Two arrows originate from the parameter 'n' in the function signature 'int fib(int n)'. One arrow points to the 'n' in the recursive call 'fib(n - 1)' within the 'else' branch. The other arrow points to the 'n' in the recursive call 'fib(n - 2)' within the same 'else' branch.

# The Efficiency of Recursion

Very simple and works perfectly,

but,

get out your stopwatches!

*click*





# The Efficiency of Recursion

Run this program with  $n = 3$ .  
Too fast to time.

Try 15.

There appears to be a bit of a pause between outputs.



# The Efficiency of Recursion

Try  $n = 30, 40, 50$ .

There is a very noticeable pause between outputs and it seems to be getting longer as  $n$  gets larger.



# The Efficiency of Recursion



We will modify the code to output trace messages:

# Thinking Recursively – Palindromes

ch11/fibtrace.cpp

```
int fib(int n)
{
    cout << "Entering fib: n = " << n << endl;
    int f;
    if (n <= 2) { f = 1; }
    else { f = fib(n - 1) + fib(n - 2); }
    cout << "Exiting fib: n = " << n
        << " return value = " << f << endl;
    return f;
}

int main()
{
    cout << "Enter n: ";
    int n;
    cin >> n;
    int f = fib(n);
    cout << "fib(" << n << ") = " << f << endl;
    return 0;
}
```

# The Efficiency of Recursion

The output:

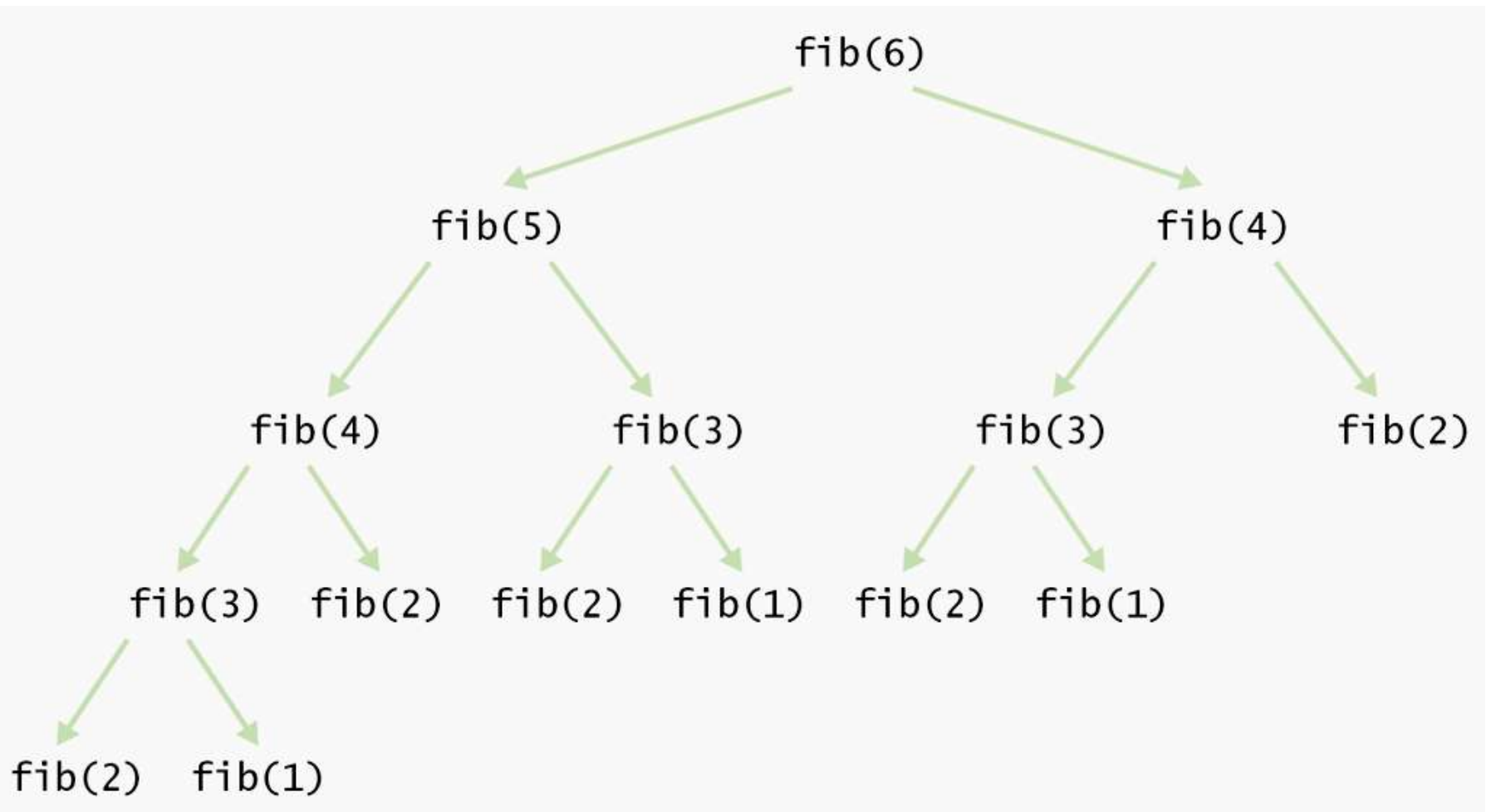
```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
```

# The Efficiency of Recursion

```
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

# The Efficiency of Recursion

This can be shown more clearly as a *call tree*:



# The Efficiency of Recursion

---

Notice that this shows that the same values,  
for example, `fib(2)`, are computed

over  
and over  
and over  
and over  
and...

This is why this recursion is inefficient.



# The Efficiency of Recursion

---

The *iterative* solution is best for Fibonacci numbers:

# The Efficiency of Recursion

ch11/fibloop.cpp

```
#include <iostream>
using namespace std;
/**
    Computes a Fibonacci number.
    @param n an integer
    @return the nth Fibonacci number
*/
int fib(int n)
{
    if (n <= 2) { return 1; }
    int fold = 1;
    int fold2 = 1;
    int fnew;
    for (int i = 3; i <= n; i++)
    {
        fnew = fold + fold2;
        fold2 = fold;
        fold = fnew;
    }
    return fnew;
}
```

# The Efficiency of Recursion

ch11/fibloop.cpp

```
int main()
{
    cout << "Enter n: ";
    int n;
    cin >> n;
    int f = fib(n);
    cout << "fib(" << n << ") = " << f << endl;
    return 0;
}
```

# The Efficiency of Recursion

---

So is the iterative solution *always*  
better than the recursive solution?

# The Efficiency of Recursion

Look at the iterative palindrome solution:

```
bool is_palindrome(string s)
{
    int start = 0;
    int end = s.length() - 1;
    while (start < end)
    {
        if (s[start] != s[end]) { return false; }
        start++;
        end--;
    }
    return true;
}
```

# The Efficiency of Recursion

Both the iteration and the recursion  
run at about the same speed.

If a palindrome has  $n$  characters,  
the iteration *executes* the loop  $n/2$  times.  
and  
the recursive solution *calls itself*  $n/2$  times,  
because two characters are removed in each step.

# The Efficiency of Recursion

---

In such a situation, the iterative solution tends to be a bit faster, because every function call takes a certain amount of processor time (and a recursive function call is a function call).

In principle, it is possible for a smart compiler to avoid recursive function calls altogether if they follow simple patterns, but most compilers don't do that.

From that point of view,  
an iterative solution is preferable.

# The Efficiency of Recursion

---

However,  
recursive solutions are  
easier to understand  
and implement correctly  
than their iterative counterparts.

There is a certain *elegance* and *economy of thought* to recursive solutions that makes them more appealing.



# The Efficiency of Recursion

---

As L. Peter Deutsch

(computer scientist and creator of the Ghost Script  
interpreter for the PostScript graphics description language)

puts it:

*To iterate is human,  
to recurse, divine.*

# Permutations

---

Now for a problem that would be  
difficult to program with an iterative solution:

permutations.

# Permutations

A permutation is simply a rearrangement of the letters.

For example, the string "eat" has six permutations  
(including the original string itself):

"eat"

"eta"

"aet"

"ate"

"tea"

"tae"


(no one said they have to be real words!)

# Permutations

We would like to write a function that generates all permutations of a string.

For example here we use it for the string “eat”:

```
vector<string> v = generate_permutations("eat");  
  
for (int i = 0; i < v.size(); i++)  
{  
    cout << v[i] << endl;  
}
```



# Permutations

---

Now you need a way to generate the permutations recursively.

Consider the string "eat" and simplify the problem.

First, generate all permutations  
that start with the letter 'e',  
then those that start with 'a',  
and finally those that start with 't'.

# Permutations

---

How do you generate the permutations that start with 'e'?

You need to know the permutations of the substring "at".

But that's the same problem

—to generate all permutations—

with a simpler input, namely the shorter string "at".

# Permutations

---

Using recursion generates  
the permutations of the substring "at".

You will get the strings

"at"

"ta"

# Permutations

---

For each result of the simpler problem,  
add the letter 'e' in front.

Now you have all permutations of  
"eat" that start with 'e', namely

"eat"

"eta"



# Permutations

Next, turn your attention to the permutations of "eat" that start with 'a'.

You must create the permutations of the remaining letters, "et", namely

"et"

"te"

Add the letter 'a' to the front of the strings and obtain

"aet"

"ate"

Generate the permutations that start with 't' in the same way.

# Permutations

Notice that you will use a loop to iterate through the character positions of the word  
Each loop iteration creates a shorter word that omits the current position:

```
vector<string> generate_permutations(string word)
{
    vector<string> result;
    ...
    for (int i = 0; i < word.length(); i++)
    {
        string shorter_word = word.substr(0, i) + word.substr(i + 1)
        ...
    }
    return result;
}
```

# Permutations

The next step is to compute the permutations of the shorter word.

```
vector<string> shorter_permutations =  
    generate_permutations(shorter_word);
```

For each of the shorter permutations, add the omitted letter:

```
for (int j = 0; j < shorter_permutations.size(); j++)  
{  
    string longer_word = word[i] + shorter_permutations[j];  
    result.push_back(longer_word);  
}
```

# Permutations

When does the recursion stop?  
The simplest possible string is the empty string,  
which has a single permutation—itself.

```
vector<string> generate_permutations(string word)
{
    vector<string> result;
    if (word.length() == 0)
    {
        result.push_back(word) ;
        return result;
    }
    ...
}
```

# Permutations

---

Could you generate the permutations without recursion?

There is no obvious way of writing a loop that iterates through all permutations.

For generating permutations,  
it is much easier to use  
recursion than  
iteration.

The code:

# Permutations

ch11/permute.cpp

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
/**
    Generates all permutations of the characters in a string.
    @param word a string
    @return a vector that is filled with all permutations of the word
*/
vector<string> generate_permutations(string word)
{
    vector<string> result;
    if (word.length() == 0)
    {
        result.push_back(word);
        return result;
    }
    for (int i = 0; i < word.length(); i++)
    {
        string shorter_word = word.substr(0, i) + word.substr(i + 1);
        vector<string> shorter_permutations =
            generate_permutations(shorter_word);
```

# Permutations

ch11/permute.cpp

```
    for (int j = 0; j < shorter_permutations.size(); j++)
    {
        string longer_word = word[i] + shorter_permutations[j];
        result.push_back(longer_word);
    }
}
return result;
}

int main()
{
    cout << "Enter a string: ";
    string input;
    getline(cin, input);
    vector<string> v = generate_permutations(input);
    for (int i = 0; i < v.size(); i++)
    {
        cout << v[i] << endl;
    }
    return 0;
}
```



# CHAPTER SUMMARY

---

## **Understand the control flow in a recursive computation.**

---

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest inputs.

## **Design a recursive solution to a problem.**

---

- The key step in finding a recursive solution is reducing the input to a simpler input for the same problem.
- When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

## **Identify recursive helper methods for solving a problem.**

---

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

# CHAPTER SUMMARY

---

## **Contrast the efficiency of recursive and non-recursive algorithms.**

---

- Occasionally, a recursive solution runs much more slowly than its iterative counterpart. However, in most cases, the recursive solution runs at about the same speed.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

## **Review a complex recursion example that cannot be solved with a simple loop.**

---

- For generating permutations, it is much easier to use recursion than iteration.

## **Recognize the phenomenon of mutual recursion in an expression evaluating application.**

---

- In a mutual recursion, a set of cooperating functions calls each other repeatedly.



## End Chapter Eleven