University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

Recursion Examples

Writing the Fibonacci Sequence

First, let's go back to Fibonacci.

We had: F(0) = 1
F(1) = 1
F(N) = F(N – 2) + F(N – 1)

Let's write Fibonacci as a **C** function:

```
int32_t fib (int32_t N);
```


Fibonacci as a Recursive Function

```
int32_t fib (int32_t N)
{
    if (0 == N || 1 == N) {
        return 1;
    }
    return (fib (N - 1) +
            fib (N - 2));
}
```

stopping conditions

handle children

handle one node

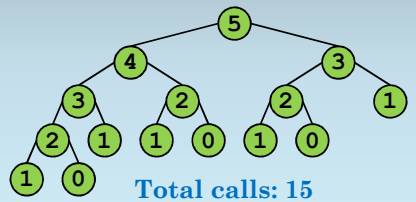
Pitfall: Recursion May Not Be the Answer

Being able to write a function recursively does not imply that doing so is a good idea.

Consider calling `fib (5)`, for example.

How many times is `fib` called as a result?

Recursive Fibonacci Does Too Much Work



Total calls: 15

There is a closed form solution…

And iteration is not hard.

A Brief Iterative Implementation of Fibonacci

```
int32_t fib (int32_t N)
{
    int32_t i, j, k, t;
    for (i = j = k = 1; N > k;
         t = j, j = j + i, i = t,
         k++) { }
    return j;
}
```


Linear Recursive Fibonacci Using a `static` Variable

```
int32_t fib (int32_t N)
{
    static int32_t fibval;
    int32_t pred, result;
    if (2 > N) {
        fibval = 1;
        return 1;
    }
    pred = fib (N - 1);
    result = pred + fibval;
    fibval = pred;
    return result;
}
```


Binary Search Can Also Be Written Recursively

We can also write binary search recursively…

```
int32_t binary_search
    (int32_t array[],
     int32_t low,
     int32_t high,
     int32_t value)
{
```

the array to search

the part of the array in which to look

the value to find

Recursive Version is Slightly Simpler

(The code is slightly simpler.)

```
{
    int32_t mid;

    if (high < low) { return -1; }

    mid = low + (high - low) / 2;
```

stopping condition: nowhere to look

same expression as before

Recurse with Modified Bounds When Not Found

```
if (value == array[mid]) {
    return mid;  // Found!
}
if (value < array[mid]) {
    return binary_search
        (array, low, mid - 1, value);
}
return binary_search
    (array, mid + 1, high, value);
} // end of function
```

recurse with modified **high**

recurse with modified **low**

Some Types of Recursion Can Be Compiled Away

When recursion
- happens **only at the end of a function**,
- in other words: return <recursive call>,
- it is called **tail recursion**.

Binary search is an example of tail recursion.

A **good optimizing compiler**
- **can transform tail recursion**
- **into an iterative version,**
- avoiding use of extra stack frames.

Let's Do an Example Together

Help me solve this problem recursively…

Task:
- print a string backwards and
- return its length (not counting NUL).

Let's call the function **`print_reverse`**.

What arguments should be passed?

a (constant) string

What should the return type be? `int32_t`

What Comes First in a Recursive Function?

```
int32_t print_reverse
    (const char* s)
{

    if ('\0' == *s) {
        return 0;
    }
```

What comes first?

stopping conditions

How long is **s** in that case?

0

When do we stop?

at end of string (NUL)

What About This "Node?" What About Children?

If the string isn't empty, we need to

- **print one character**, and
- **call `print_reverse` with the rest of the string**.

In what order?

Call first, then print.

We also need a local variable to store the return value from the call.

How Do We Make the Recursive Call?

```
int32_t print_reverse
    (const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
```

What should be passed to the recursive call?

What Do We Print?

```
int32_t print_reverse
    (const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
```

What character should be printed?

What Do We Return?

```
int32_t print_reverse
    (const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

What should be returned?

Reference Version of `print_reverse`

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Code is available on the web page.

Let's See How a Recursive Function Works in Detail

Let's execute a call:

print_reverse ("Now")

With … stack frames!

This visualization will probably be the last time that we see stack frames in class.

Feel free to get nostalgic.

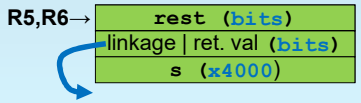
Stack Frame for `print_reverse` (Call Depth 1)

call depth 1 ☐
`p_r ("Now")`

Abbreviated as `p_r` here.

Say that the `N` is at x4000.

What are the values when the function starts?



What Happens First?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Check for NUL.

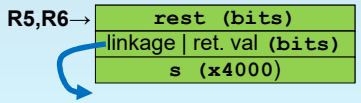
Have We Found a NUL?

call depth 1

`p_r ("Now")`

What is **s**? **x4000**

What is stored at x4000? **'N'**



What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

'N' is not NUL, so don't return yet.

Call **print_reverse** again with (**s + 1**).

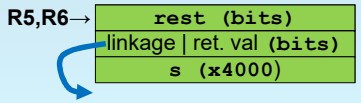
What is Passed to the Recursive Call?

call depth 1

`p_r ("Now")`

What is **s**? **x4000**

Adding 1, we obtain x4001.



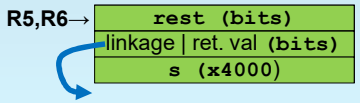
Calling `print_reverse` (Call Depth 2)

call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

What is the argument?

The `o` is at x4001.

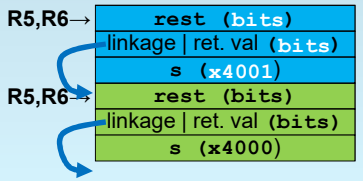


Stack Frame for `print_reverse` (Call Depth 2)

call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

What are the values when the function starts?



What Happens First?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Check for NUL.

Have We Found a NUL?

call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

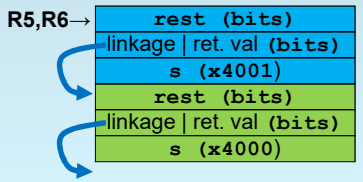
What is `s`? **x4001**

Why not x4000?

s is relative to **R5**.

What is stored at x4001?

'o'



What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

'o' is not NUL, so don't return yet.

Call **print_reverse** again with (**s + 1**).

Calling print_reverse (Call Depth 3)

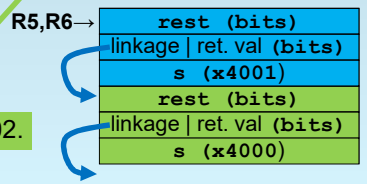
call depth 1
p_r ("Now")

call depth 2
p_r ("ow")

call depth 3
p_r ("w")

What is the argument?

The w is at x4002.



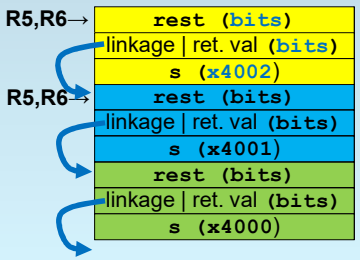
Stack Frame for print_reverse (Call Depth 3)

call depth 1
p_r ("Now")

call depth 2
p_r ("ow")

call depth 3
p_r ("w")

What are the values when the function starts?



What Happens First?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

… same as before, of course!

Check for NUL.

Then call **print_reverse** again.

Calling `print_reverse` (Call Depth 4)

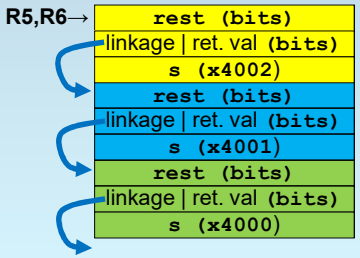
call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

call depth 3
`p_r ("w")`

call depth 4
`p_r ("")`

The argument is the empty string at x4003.



Stack Frame for `print_reverse` (Call Depth 4)

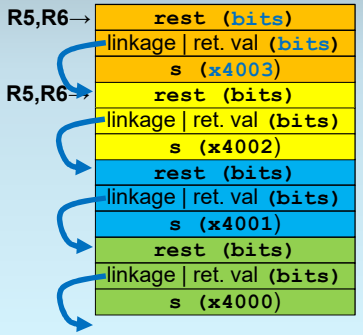
call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

call depth 3
`p_r ("w")`

call depth 4
`p_r ("")`

Values?



What Happens First Now?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Found NUL, so return 0.

Write 0 into Return Value Slot

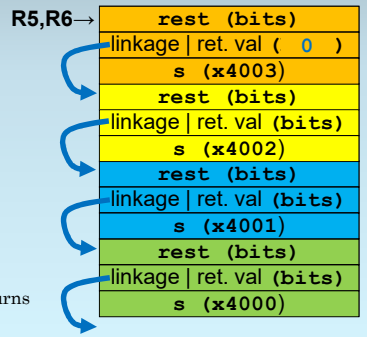
call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

call depth 3
`p_r ("w")`

call depth 4
`p_r ("")`

returns 0

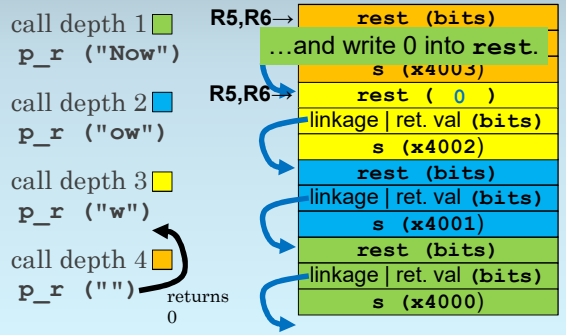


What Happens When `print_reverse` Returns?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

On return, return value is written into **`rest`**.

Tear Down Stack Frame

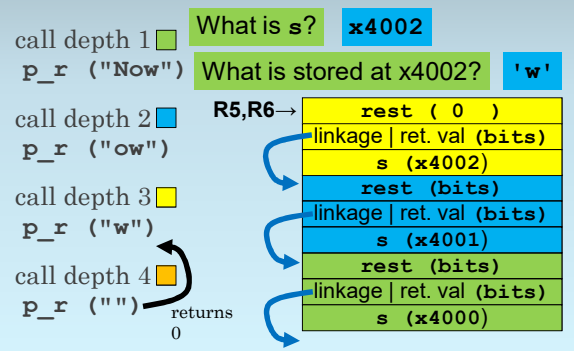


What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Print *s.

Print Character at `s`



What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Output is now "w".

Return (rest + 1).

Return (rest + 1)

call depth 1 p_r ("Now")

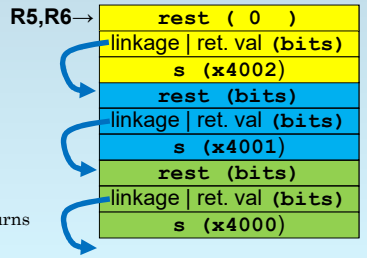
call depth 2 p_r ("ow")

call depth 3 p_r ("w")

call depth 4 p_r ("") returns 0

What is rest? 0

Adding 1, we obtain 1.



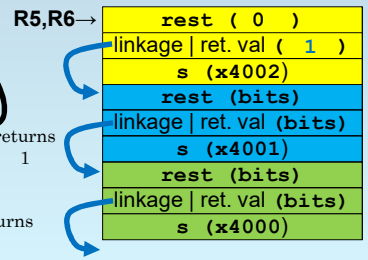
Write 1 into Return Value Slot

call depth 1 p_r ("Now")

call depth 2 p_r ("ow")

call depth 3 p_r ("w") returns 1

call depth 4 p_r ("") returns 0

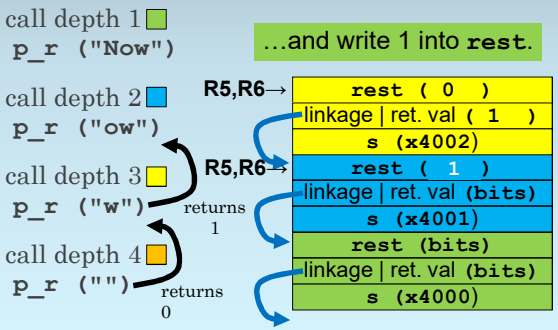


What Happens When print_reverse Returns?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

On return, return value is written into rest.

Tear Down Stack Frame

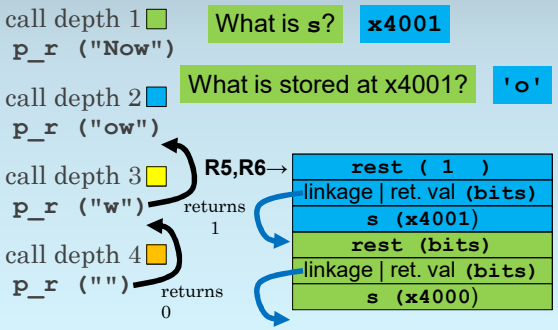


What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Print *s.

Print Character at s



What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Output is now "wo".

Return (rest + 1).

Return (`rest + 1`)

call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

call depth 3
`p_r ("w")`

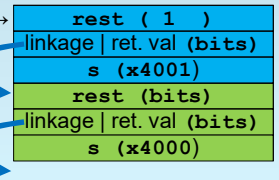
call depth 4
`p_r ("")`

returns 0

returns 1

What is `rest`? **1**

Adding 1, we obtain 2.



Write 2 into Return Value Slot

call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

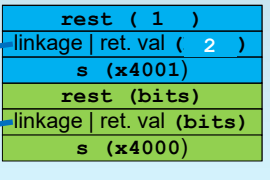
call depth 3
`p_r ("w")`

call depth 4
`p_r ("")`

returns 2

returns 1

returns 0



What Happens When `print_reverse` Returns?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

On return, return value is written into `rest`.

Tear Down Stack Frame

call depth 1
`p_r ("Now")`

call depth 2
`p_r ("ow")`

call depth 3
`p_r ("w")`

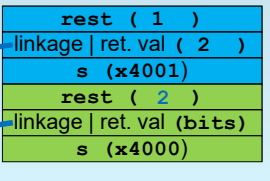
call depth 4
`p_r ("")`

returns 2

returns 1

returns 0

…and write 2 into `rest`.

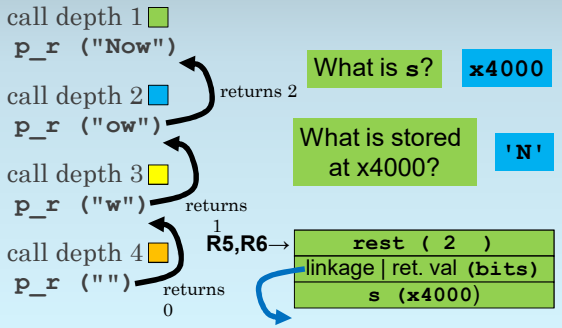


What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

Print *s.

Print Character at `s`



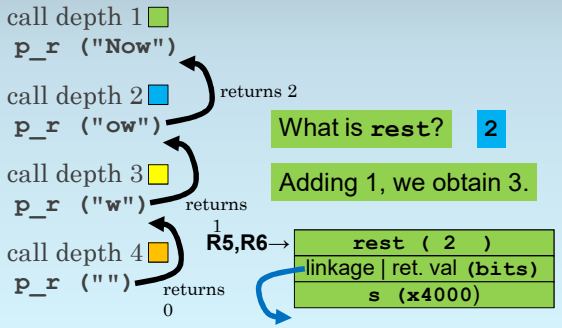
What Happens Next?

```
int32_t print_reverse(const char* s)
{
    int32_t rest;
    if ('\0' == *s) {
        return 0;
    }
    rest = print_reverse (s + 1);
    printf ("%c", *s);
    return (rest + 1);
}
```

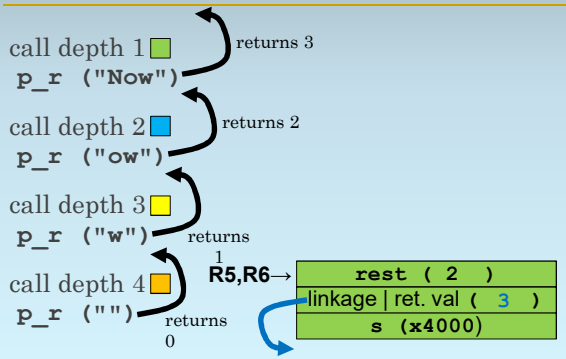
Output is now "woN".

Return (`rest + 1`).

Return (`rest + 1`)



Write 3 into Return Value Slot



Tear Down Stack Frame