Recursive Functions

CS 16: Solving Problems with Computers I
Lecture #17 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 1 of 3

Lecture Outline

- Examples using recursion
- How does the Stack work with recursion?

Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion
Recursion

Case Study: Vertical Numbers

- Problem Definition:
  Write a recursive function that takes an integer number and prints it out one digit at a time vertically :

```
write_vertical(3):
3
write_vertical(12):
1
2
write_vertical(123):
1
2
3
```

```
void write_vertical( int n );
//Precondition:  n >= 0
//Postcondition: n is written to the screen vertically
//          with each digit on a separate line
```

Case Study: Vertical Numbers

Analysis:

- Take a decimal number, like 543.
- How do I separate the digits from each other?
  - So that I can print out **5**, then **4**, then **3**?

- <u>Hint</u>: Note that 543 = 500 + 40 + 3

Case Study: Vertical Numbers

Algorithm design

- *Simplest case* *(what do we call that again???)*

  If **n** is 1 digit long, just write the number

- *More typical case*:

  1) Output all but the last digit vertically (recursion!)

  2) Write the last (least significant) digit (base case!)

  – *Step 1 is a smaller version of the original task - The recursive case*

  – *Step 2 is the simplest case - The base case*

Case Study: Vertical Numbers

*The **write_vertical** algorithm:*

```
void write_vertical( int n )
{
    if (n < 10)  {
        cout << n << endl;
    }  // NOTE: n < 10 means n is only one digit
    else { // n is two or more digits long
        write_vertical(n-with-the-least-significant-digit-removed);
        cout << the least-significant digit of n << endl;
    }
}
```

Case Study: Vertical Numbers

```
void write_vertical( int n )
{
     if (n < 10) cout << n << endl;
     else
     {
        write_vertical
          (n-without-last-digit);
        cout << LSD << endl;
     }
}
```

- Note that: **n / 10 (integer division)** returns **n** with ***just*** *the least-significant digit removed*
  - So, for example, **85 / 10 = 8** *or* **124 / 10** = **12**
- Whereas: **n % 10** returns the *least-significant digit of **n***
  - In this example, **124 % 10 = 4**
- **How might we combine these in the previous function?**

Case Study: Vertical Numbers

*The **write_vertical** function in C++*

```
void write_vertical( int n )
{
    if (n < 10) {
        cout << n << endl;
    }
    else {
        write_vertical(n / 10);
        cout << (n % 10) << endl;
    }
}
```

Example Run

```
void write_vertical( int n )
{
     if (n < 10) cout << n << endl;
     else
     {
         write_vertical(n / 10);
         cout << n % 10 << endl;
     }
}
```

```
write_vertical(543)  --5-->  cout << 3 << endl;
   | 1
write_vertical(54)   --4-->  cout << 4 << endl;
   | 2
write_vertical(5)    --3-->  cout << 5 << endl;
```

```
stdout:
5
4
3
```

"Infinite" Recursion

- A function that never reaches a base case, in theory, *will run forever*
  - Why "in theory"?

- What if we wrote the function **write_vertical**, *without the base case*

```
void write_vertical(int n)
   {
        write_vertical (n / 10);
        cout << n % 10 << endl;
   }
```

- Will *eventually* call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

which will call **write_vertical(0)**,

…etc…

“Infinite” Recursion

- In **practice**, the computer will often run out of *resources* (i.e. memory usually) and the program will *terminate abnormally*
  - This can happen even in non-infinite recursion situations!
    (can you think of a case where this could happen?)

- So… remember that computers are machines, not Math Gods and design your (recursive) functions with that in mind!

END OF PART 1

Recursive Functions

CS 16: Solving Problems with Computers I
Lecture #17 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 2 of 3

Stacks for Recursion

- Computers use a memory structure called a ***stack*** to keep track of recursion
- **Stack**: a **computer memory structure** analogous to a ***stack of paper***
  - Start at zero: no papers, just knowledge of where to start (via a “stack pointer”)

  - To place data on the stack: write it on a piece of paper and place it on **top** of the stack
  - To **insert *more*** information on the stack: use a new sheet of paper, write the information, and place it on the **top** of the stack
  - Keep going… until you don’t need to anymore…

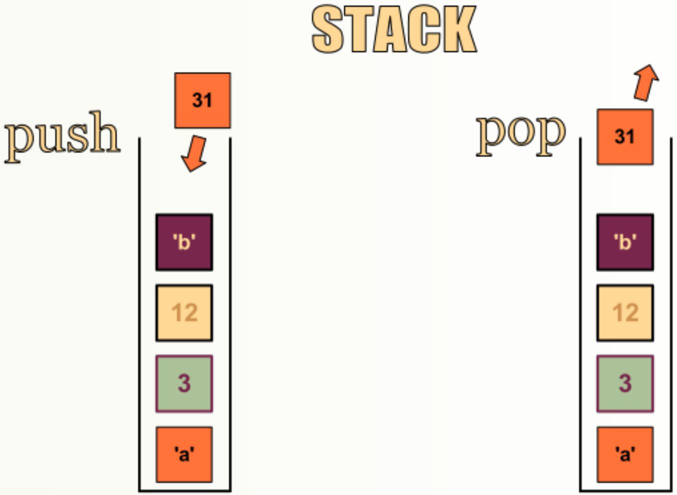
  - To **retrieve** information: you can only take the top sheet of paper
    - Then throw it away when it you’re done “reading” it
  - If you want access to any paper farther down, go thru the stack to get to it

LIFO

- This scheme of handling sequential data in a stack is called:

  Last In-First Out (LIFO)

- When we put data in a LIFO, we call it a **push**
- When we pull data out of a LIFO, we call it a **pop**

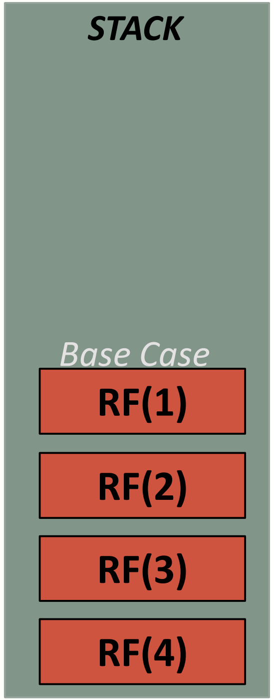


- The other common scheme in data organization is FIFO (First In-First Out) aka *queue*

Stacks & Making the Recursive Call

When execution of a function definition reaches a **recursive call**…

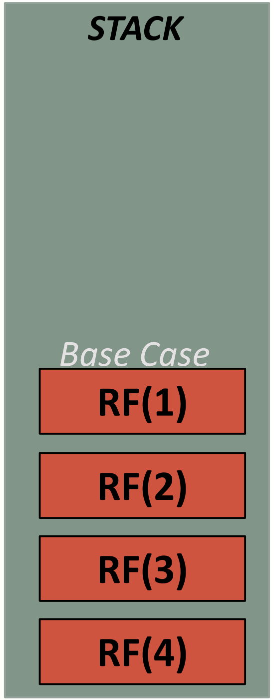
1. Execution is paused
2. Data is then saved in a new place in the stack on top
   - Remember, this is part of **computer memory**
3. Then, a *new* place in memory is "prepared" for the recursive call
   a) A new function definition is written, arguments are plugged into parameters
   b) Execution of the recursive call begins
4. New data is saved on top of the stack
5. Repeat until you get to the base case



Stacks & Ending Recursive Calls

When a recursive function call gets to the **base case**…

1. The computer retrieves the top memory unit of the stack
2. It resumes computation based on the information on the sheet
3. When that computation ends, that memory unit is “discarded”
4. The mem. unit on the stack is retrieved so that processing can resume
5. The process continues until the stack is back to it original status



Stack Overflow

- Stacks are finite things…
- Infinite recursions can force the stack to grow **beyond** its physical limits
- The result of this erroneous operation is called a ***stack overflow***
  - This causes abnormal termination of the program

Recursive Functions for *Values*

- Recursive functions don't have to be **void** types
  - They can also return values

- The technique to design a recursive function that returns a value is basically the same as what we described earlier…

Program Example: A Powers Function

Example: Define a new **power** function (not the one in <cmath>)

- Let it return an integer, **2^3** ,when we call the function as: **int y = power(2,3);**
- Use the following definition: $x^n = x^{n-1} * x$ *i.e.* $2^3 = 2^2 * 2$
  - Note that this only works if n is a positive number
- Translating the right side of that equation into C++ gives: **power(x, n-1) * x**
  - What is the base/stopping case?

    *It's when **n = 0***
  - What should happen then?

    *power() should return **1***

Tracing ***power(2, 3)***

1

power(2, 0) * 2

power(2, 1) * 2

power(2, 2) * 2

power(2, 3)

pushing into the stack

1

1 * 2

2 * 2

4 * 2

= 8

popping out of the stack

```
int power(int x, int n)
{
   // Before you do a base-case, it’s always a good idea to
   // take care of “illegal” operations…
   if (n < 0)
   {
      cout << “Cannot use negative powers in this function!\n”;
      exit(1);
   }

   if (n > 0)
      return ( power(x, n – 1) * x );

   // if n == 0
   return (1);
}
```

Stopping or base case

Recursion versus Iteration

- **Any task that can be accomplished using recursion *can also be done* without recursion (using loops)**

- A **non-recursive** version of a repeating function is called an ***iterative-version***

Recursion versus Iteration

```
int power(int x, int n)
{
   if (n == 0) {
      return 1;
   return( power(x, n - 1) * x );
}
```

Recursive Version

```
int power(int x, int n)
{
   int p = 1;
   for (int k = 1; k <= n; k ++) {
        p *= x;
   }
   return(p);
}
```

Iterative Version

- A **recursive** version of a function…
  - Usually runs a little slower, takes up more memory
  - BUT it uses code that is *easier to write and understand*

END OF PART 2

Recursive Functions

CS 16: Solving Problems with Computers I

Lecture #17 PRE-RECORDED

Ziad Matni

Dept. of Computer Science, UCSB

Part 3 of 3

Case Study: Fibonacci Series

- The Fibonacci Series is a numerical series that looks like this:

  0 1 1 2 3 5 8 13 21 34 … etc…

- That is, **Fibonacci**[n] = **Fibonacci**[n-1] + **Fibonacci**[n-2]

- Can we write this as a recursive function?
  – As a function that returns the n^{th} position in the series?

```cpp
#include <iostream>
using namespace std;

int fibo(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    }

    return fibo(n-1) + fibo(n-2);
}

int main() {
    int N;
    cout << "What position in the Fibonacci series do you want? ";
    cin >> N;
    cout << fibo(N) << endl;
}
```

Case Study: Reversing a String

- We all know that this routine:

```
string s = “Hello”;
for(int k = s.size() - 1; k >= 0; k--) {
     cout << s[k];
}
```

- Will end up printing this: **`olleH`**

- Can we do this as a recursive function?

Case Study: Reversing a String

1. Print the last letter of the string (“o”)

2. Repeat (recurse) but with a modified string that is missing the last letter
   - So, “Hello” becomes “Hell”

3. Stop the recursion once the string becomes of size = 1
   - So, we’re down to “H”
   - Print that and return

```
#include <iostream>
#include <string>
using namespace std;

void reverseString(string s){
    if (s.size() == 1) {
        cout << s;
        return;
    }
    cout << s[s.size() - 1];
    reverseString( s.substr(0, s.size() - 1) );
}

int main() {
    string MyString;
    cout << "Enter 1 word: ";
    cin >> MyString;

    reverseString(MyString);
    cout << endl;
}
```

Case Study: Summing an Integer Array

- We all know that this routine:

```
int array[3] = {10, 20, 30}, size = 3, sum = 0;
for(int k = 0; k < size; k++) {
      sum += array[k];
}
```

- Will end up having the sum of all elements in the array.

- Can we do this as a recursive function?

Case Study: Summing an Integer Array

1. The recursive function needs to have 2 arguments: the array, the size

2. Repeat (recurse) taking the array, with size – 1, PLUS (add) array[size – 1];
   - Keep pushing the values in the stack
   - i.e. *return f(array, size – 1) + array[size – 1]*

3. Stop when size is 0
   - In which case return 0 (i.e. nothing more to add)
   - i.e. *if size == 0, return 0*

- *So, in the end the pop operations will add: 0 + array[0] + array[1] + … array[size]*

```
#include <iostream>
using namespace std;

int sumArray(int a[], int size) {
    if (size == 0) {
        return 0;
    }
    return sumArray(a, size - 1) + a[size - 1];
}

int main() {
    int array[10] = {1,2,3,4,5,6,7,8,9,10};
    int size = 10;

    cout << sumArray(array, size) << endl;
}
```

</LECTURE>