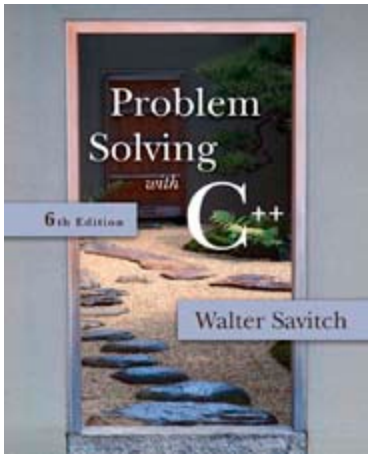
APS105: Lecture 26

Wael Aboelsaadat

wael@cs.toronto.edu

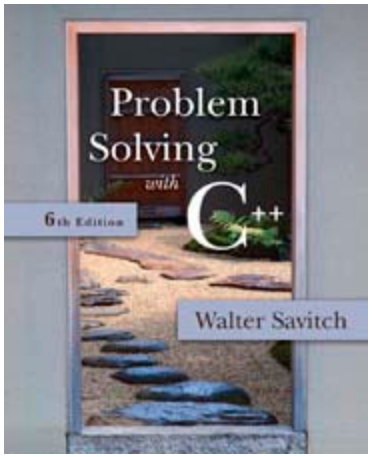
http://ccnet3.utoronto.ca/20079/aps105h1f/

Acknowledgement: These slides are a modified version of the text book slides as supplied by Addison Wesley



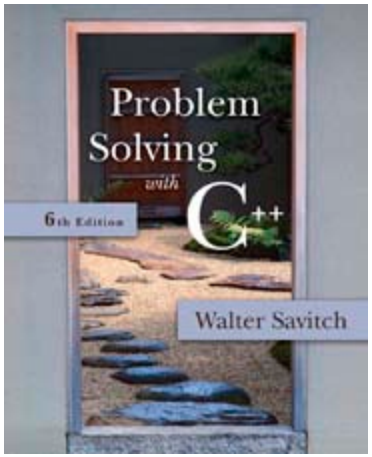
Chapter 14

Recursion



14.1

Recursive Functions for Tasks

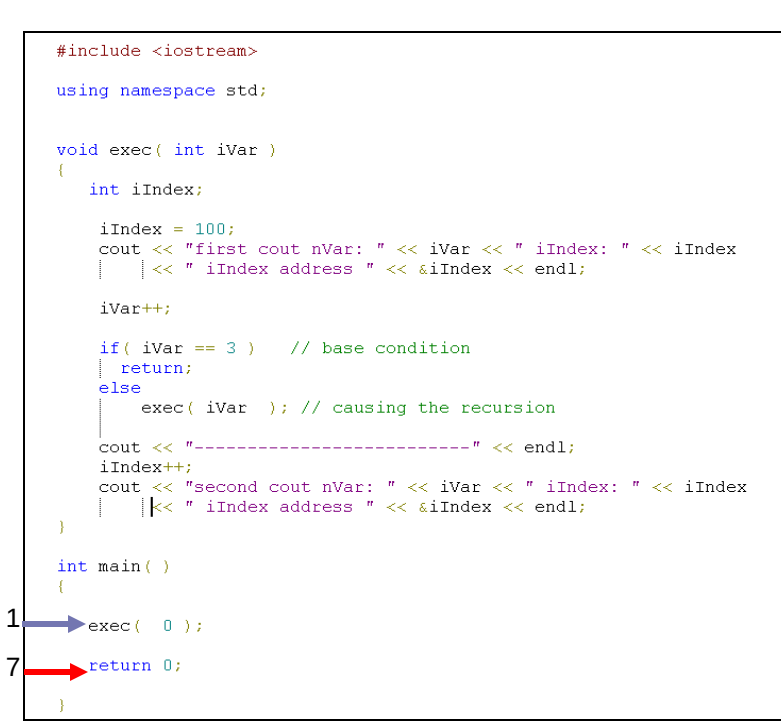


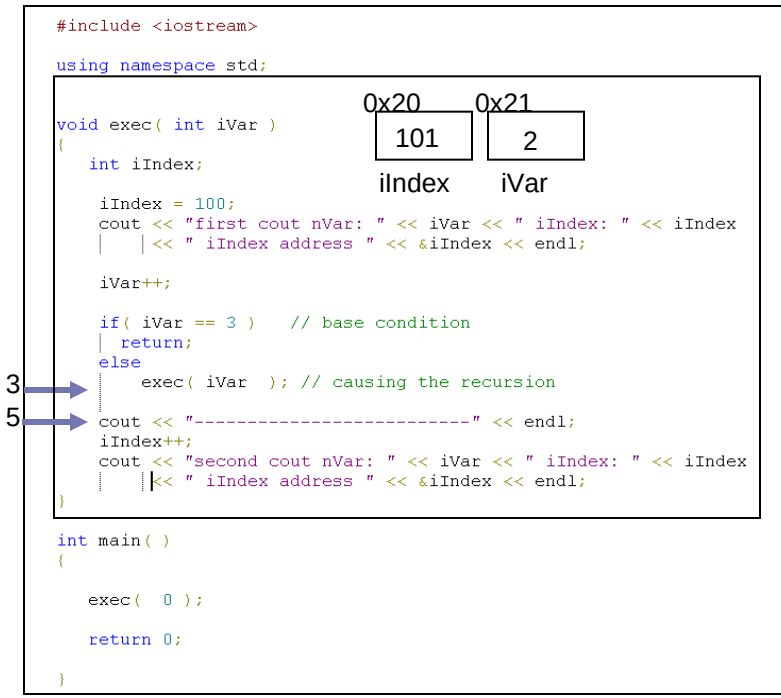
```cpp
#include <iostream>

using namespace std;


void exec( int iVar )
{
   int iIndex;

    iIndex = 100;
    cout << "first cout nVar: " << iVar << " iIndex: " << iIndex
    |    | << " iIndex address " << &iIndex << endl;

    iVar++;

    if( iVar == 3 )    // base condition
    |  return;
    else
    |    exec( iVar  ); // causing the recursion
    |
    cout << "--------------------------" << endl;
    iIndex++;
    cout << "second cout nVar: " << iVar << " iIndex: " << iIndex
    |    |<< " iIndex address " << &iIndex << endl;
}

int main( )
{

   exec(  0 );

   return 0;

}
```

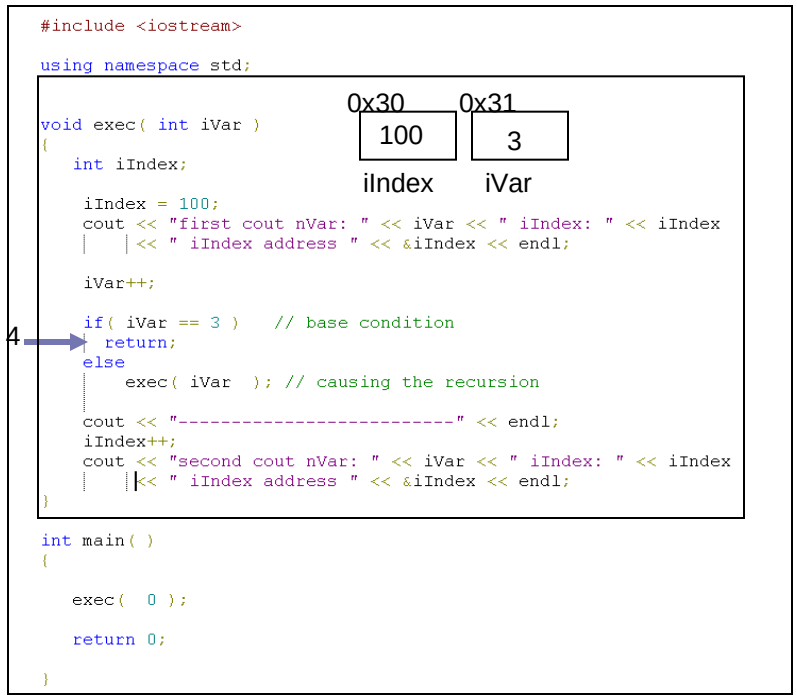
```cpp
#include <iostream>

using namespace std;


void exec( int iVar )
{
   int iIndex;

    iIndex = 100;
    cout << "first cout nVar: " << iVar << " iIndex: " << iIndex
         << " iIndex address " << &iIndex << endl;

    iVar++;

    if( iVar == 3 )   // base condition
      return;
    else
       exec( iVar  ); // causing the recursion

    cout << "--------------------------" << endl;
    iIndex++;
    cout << "second cout nVar: " << iVar << " iIndex: " << iIndex
         << " iIndex address " << &iIndex << endl;
}

int main( )
{

   exec(  0 );

   return 0;

}
```

1

7







Recursive Functions for Tasks

- A recursive function contains a call to itself
- When breaking a task into subtasks, it may be that the subtask is a smaller example of the same task
  - Searching an array could be divided into searching the first and second halves of the array
  - Searching each half is a smaller version of searching the whole array
  - Tasks like this can be solved with recursive functions

A Closer Look at Recursion

- Recursive calls are tracked by
  - Temporarily stopping execution at the recursive call
    - The result of the call is needed before proceeding
  - Saving information to continue execution later
  - Evaluating the recursive call
  - Resuming the stopped execution

How Recursion Ends

- Eventually one of the recursive calls must not depend on another recursive call
- Recursive functions are defined as
  - One or more cases where the task is accomplished by using recursive calls to do a smaller version of the task
  - One or more cases where the task is accomplished without the use of any recursive calls
    - These are called base cases or stopping cases

"Infinite" Recursion

- A function that never reaches a base case, in theory, will run forever
  - In practice, the computer will run out of resources and the program will terminate abnormally

Example: Infinite Recursion

- Function write_vertical, without the base case

```
void new_write_vertical(int n)
  {
      new_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), …

Stacks for Recursion

- Computers use a structure called a stack to keep track of recursion
  - A stack is a memory structure analogous to a stack of paper
    - To place information on the stack, write it on a piece of paper and place it on top of the stack
    - To place more information on the stack, use a clean sheet of paper, write the information, and place it on the top of the stack
    - To retrieve information, only the top sheet of paper can be read, and thrown away when it is no longer needed

Last-in / First-out

- A stack is a last-in/first-out memory structure
  - The last item placed is the first that can be removed
- Whenever a function is called, the computer uses a "clean sheet of paper"
  - The function definition is copied to the paper
  - The arguments are plugged in for the parameters
  - The computer starts to execute the function body

Stacks and The Recursive Call

- When execution of a function definition reaches a recursive call
  - Execution stops
  - Information is saved on a "clean sheet of paper" to enable resumption of execution later
  - This sheet of paper is placed on top of the stack
  - A new sheet is used for the recursive call
    - A new function definition is written, and arguments are plugged into parameters
    - Execution of the recursive call begins

The Stack and Ending Recursive Calls

- When a recursive function call is able to complete its computation with no recursive calls
  - The computer retrieves the top "sheet of paper" from the stack and resumes computation based on the information on the sheet
  - When that computation ends, that sheet of paper is discarded and the next sheet of paper on the stack is retrieved so that processing can resume
  - The process continues until no sheets remain in the stack

Activation Frames

- The computer does not use paper
- Portions of memory are used
  - The contents of these portions of memory is called an activation frame
    - The activation frame does not actually contain a copy of the function definition, but references a single copy of the function

Stack Overflow

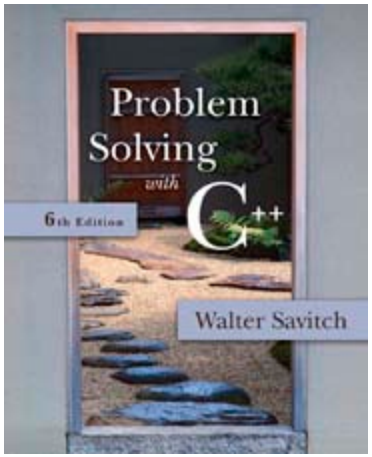
- Because each recursive call causes an activation frame to be placed on the stack
  - infinite recursion can force the stack to grow beyond its limits to accommodate all the activation frames required
  - The result is a stack overflow
  - A stack overflow causes abnormal termination of the program

Recursion versus Iteration

- Any task that can be accomplished using recursion can also be done without recursion
  - A nonrecursive version of a function typically contains a loop or loops
  - A non-recursive version of a function is usually called an iterative-version
  - A recursive version of a function
    - Usually runs slower
    - Uses more storage
    - May use code that is easier to write and understand

14.2

Recursive Functions for Values



Recursive Functions for Values

- Recursive functions can also return values
- The technique to design a recursive function that returns a value is basically the same as what you have already seen
  - One or more cases in which the value returned is computed in terms of calls to the same function with (usually) smaller arguments
  - One or more cases in which the value returned is computed without any recursive calls (base case)

Program Example:
A Powers Function

$2^3 = 8$

$2 * 2 * 2$

$9^2 = 81$

Program Example: A Powers Function

- To define a new power function that returns an int, such that

  int y = power(2,3);

  places 23 in y
  - Use this definition:

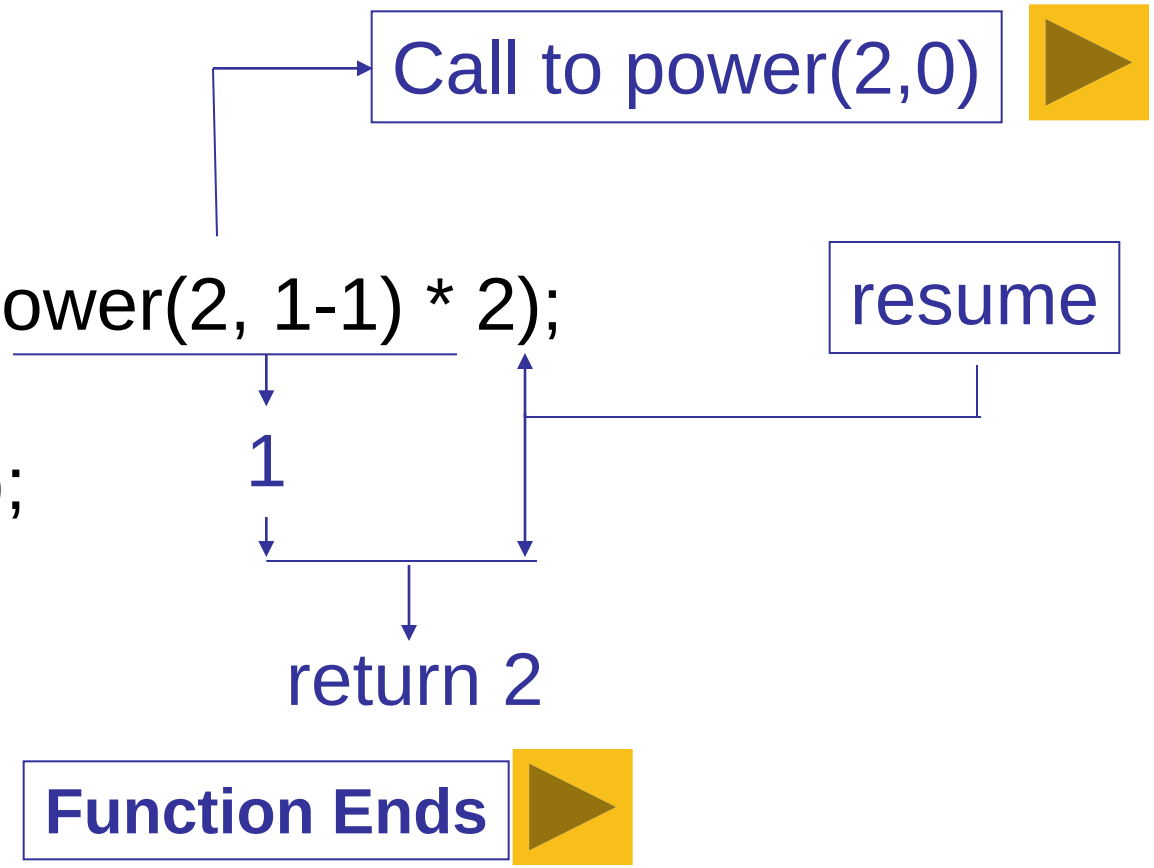
    xn = xn-1 * x

    Display 14.3
  - Translating the right side to C++ gives:

    power(x, n-1) * x
  - The base case: n = = 0 and power should return 1

Tracing power(2,1)

- int power(2, 1)
  {
  …
  if (n > 0)
  return (power(2, 1-1) * 2);
  else
  return (1);
  }



Tracing power(2,0)

- int power(2, 0)
  {
  …
  if (n > 0)
  return (power(2, 0-1) * 2);
  else
  return (1);
  }

Tracing power(2, 3)

- Power(2, 3) results in more recursive calls:
  - power(2, 3) is power(2, 2) * 2
  - Power(2, 2) is power(2, 1) * 2
  - Power(2, 1) is power(2, 0) * 2
  - Power (2, 0) is 1 (stopping case)

- See details in **Display 14.4**

Section 14.2 Conclusion

- Can you
  - Determine the output of this function if called with rose(4)?

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

Display 14.3

The Recursive Function power

```
//Program to demonstrate the recursive function power.
#include <iostream>
#include <cstdlib>
using namespace std;

int power(int x, int n);
//Precondition: n >= 0.
//Returns x to the power n.

int main( )
{
    for (int n = 0; n < 4; n++)
        cout << "3 to the power " << n
             << " is " << power(3, n) << endl;

    return 0;
}

//uses iostream and cstdlib:
int power(int x, int n)
{
    if (n < 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }

    if (n > 0)
        return ( power(x, n - 1)*x );
    else // n == 0
        return (1);
}
```

Sample Dialogue

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

Display 14.4

Back Next

Evaluating the Recursive Function Call `power(2, 3)`