Recursion

Tan Cher Wah (isstcw@nus.edu.sg)

Recursion

- Recursion is a programming technique by which a function calls itself repeatedly to solve a smaller version of the problem
- A recursive function terminates when a terminating or base condition is met
- To understand a recursive function better, we need to take a look at a Call Stack

Call Stack

- A running program uses a **Call Stack** to track function calls
- When Function A is called, information such as its parameter values, are stored in a Call Stack
- Now, if Function A calls Function B, information about Function B is now stacked on top of Function A in the Call Stack
- The Call Stack follows a Last-In-First-Out (LIFO) order
  - When Function B exits, it is popped out of the call stack and control now returns to Function A
  - When Function A exits, it is popped out of the call stand and control now returns to the Main program

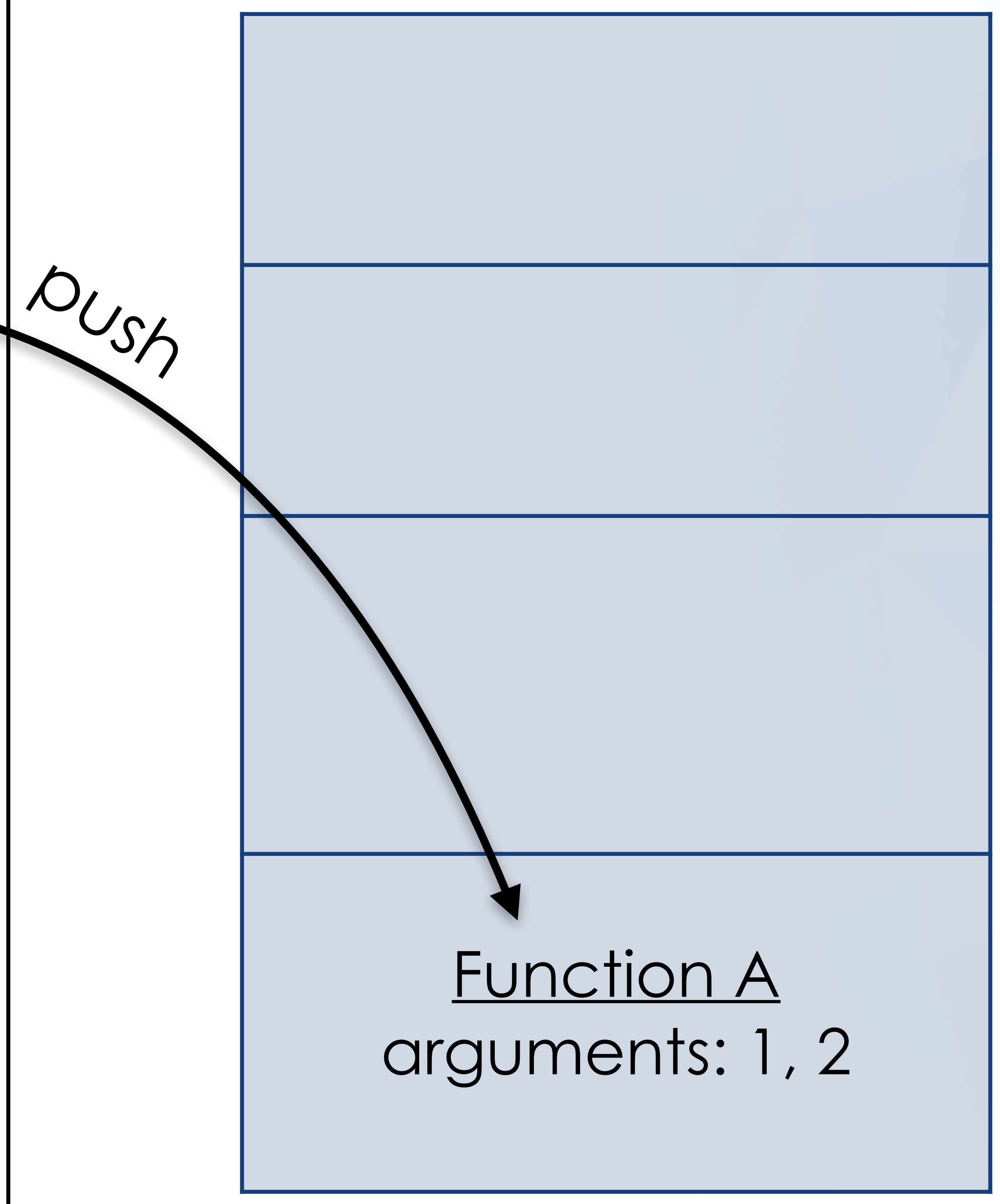
Call Stack

Arguments of Function A are pushed onto Call Stack when Function A is invoked

```
class Program
{
    static void Main(string[] args)
    {
        Function_A(1, 2);
    }

    static void Function_A(int x, int y)
    {
        Function_B("hello");
    }

    static void Function_B(string str)
    {
    }
}
```



Call Stack

Call Stack

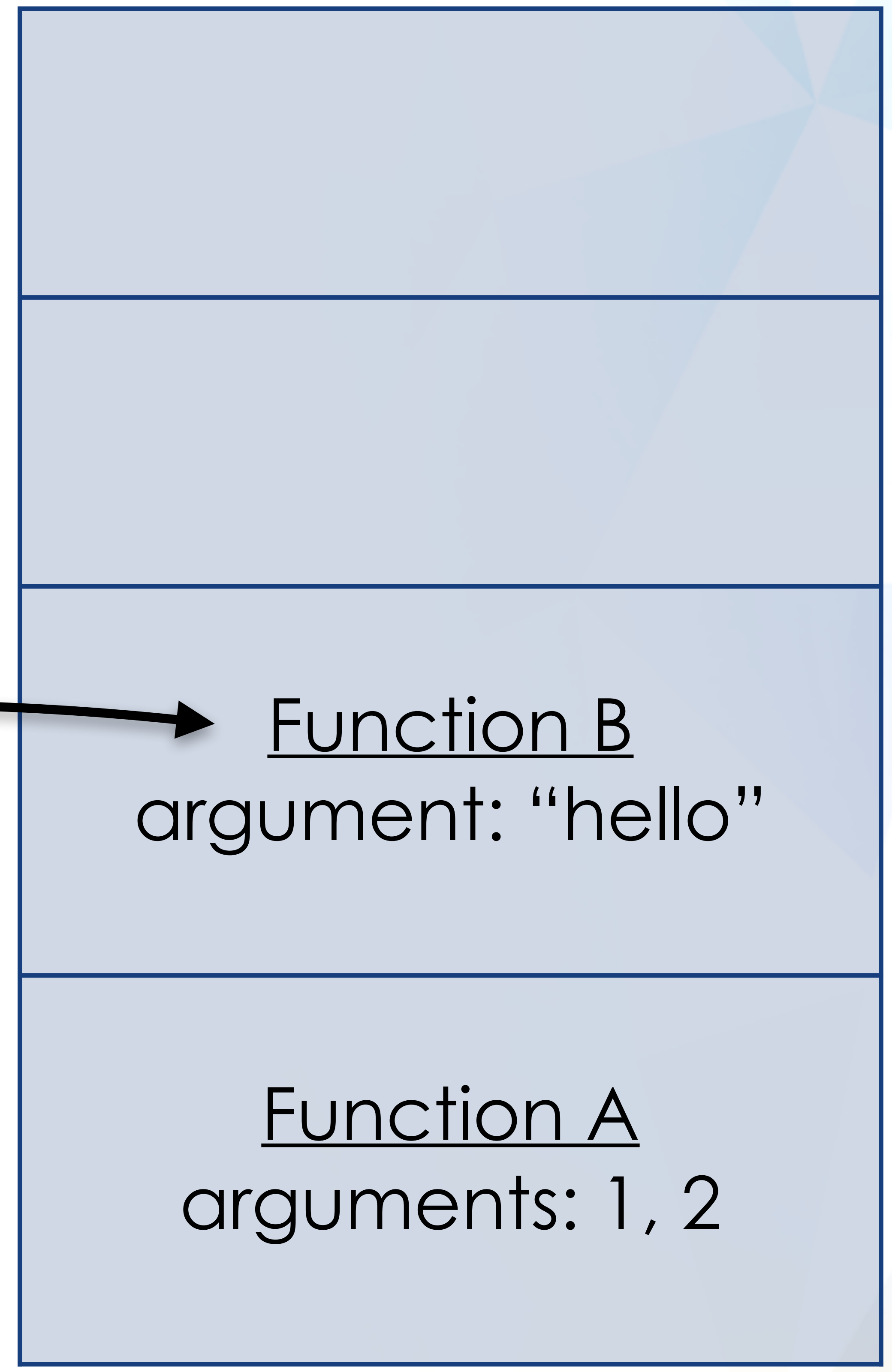
Argument of Function B is pushed onto the Call Stack when Function B is invoked

```
class Program
{
    static void Main(string[] args)
    {
        Function_A(1, 2);
    }

    static void Function_A(int x, int y)
    {
        Function_B("hello");
    }

    static void Function_B(string str)
    {
    }
}
```

push



Call Stack

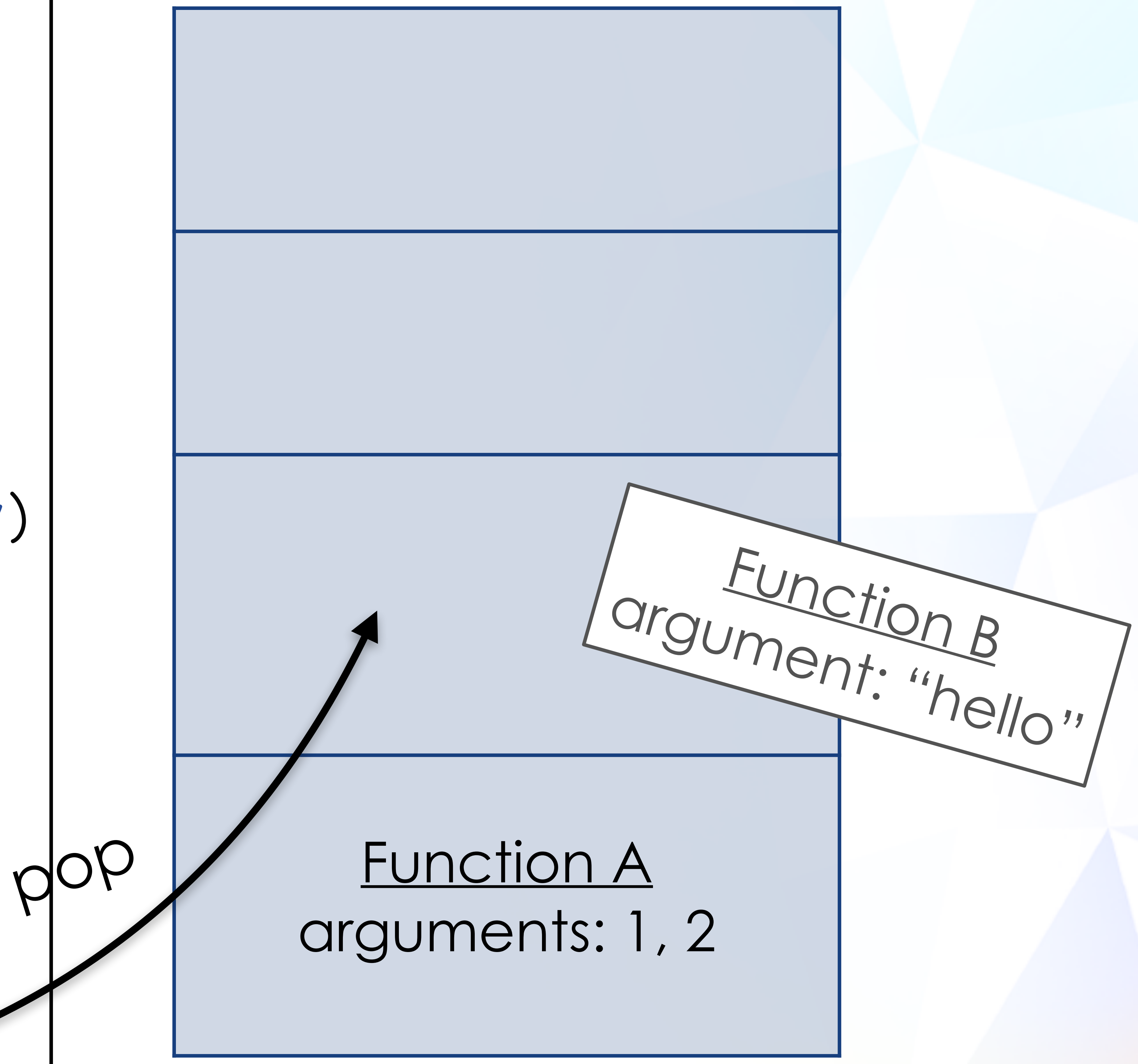
Call Stack

Argument of Function B is popped out of Call Stack when Function B exits

```
class Program
{
    static void Main(string[] args)
    {
        Function_A(1, 2);
    }

    static void Function_A(int x, int y)
    {
        Function_B("hello");
    }

    static void Function_B(string str)
    {
    }
}
```



Call Stack

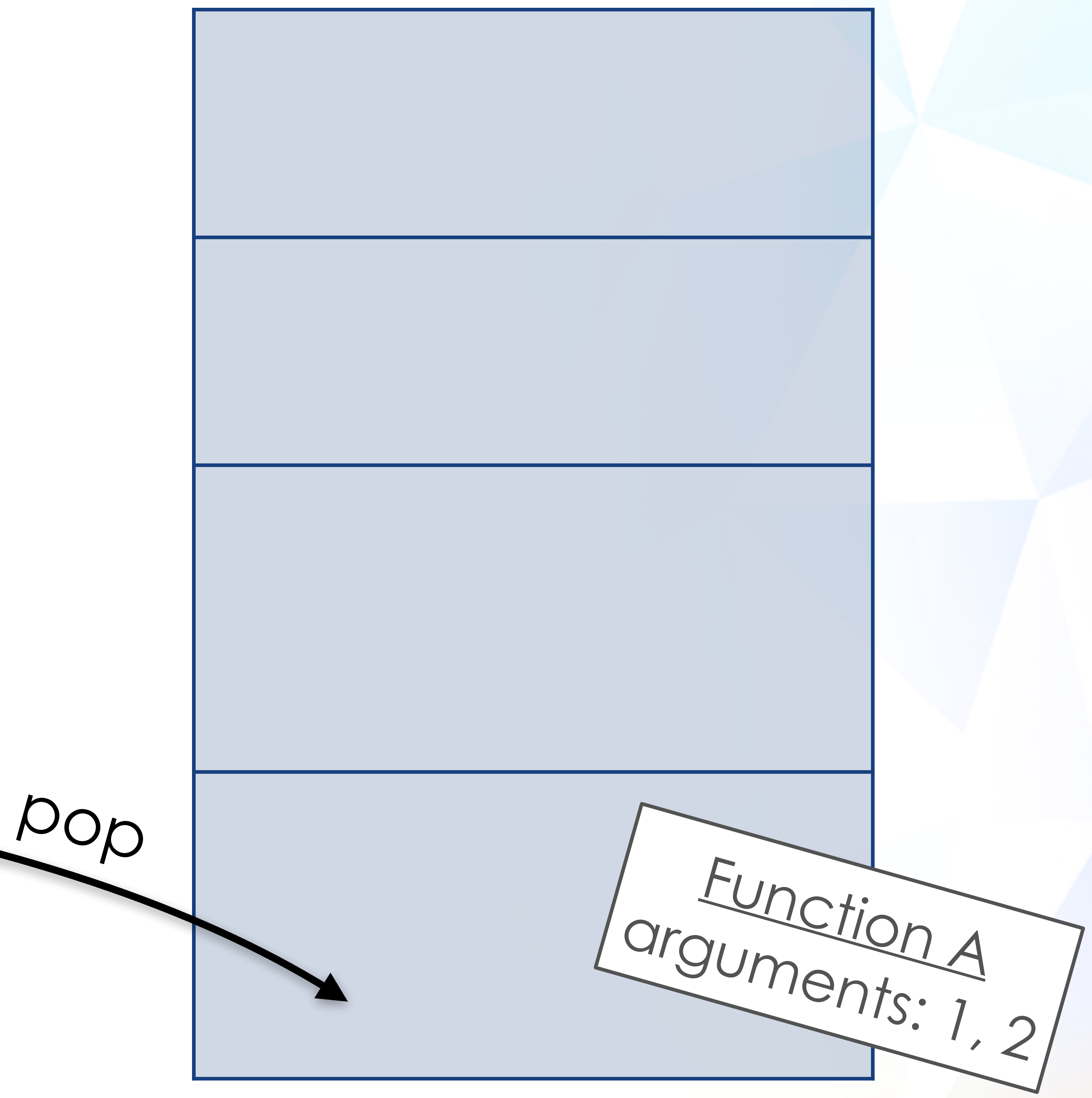
Call Stack

Arguments of Function A are popped out of Call Stack when Function A exits

```
class Program
{
    static void Main(string[] args)
    {
        Function_A(1, 2);
    }

    static void Function_A(int x, int y)
    {
        Function_B("hello");
    }

    static void Function_B(string str)
    {
    }
}
```



Call Stack

Stack Overflow

- A recursive function calls itself repeatedly - each time with slightly different parameter values
- The Call Stack only unwinds when the **base or terminating condition** of the recursive function is reached
- As the size of a Call Stack is finite, a recursive function that never reaches its base condition result in a **stack overflow**

Stack Overflow

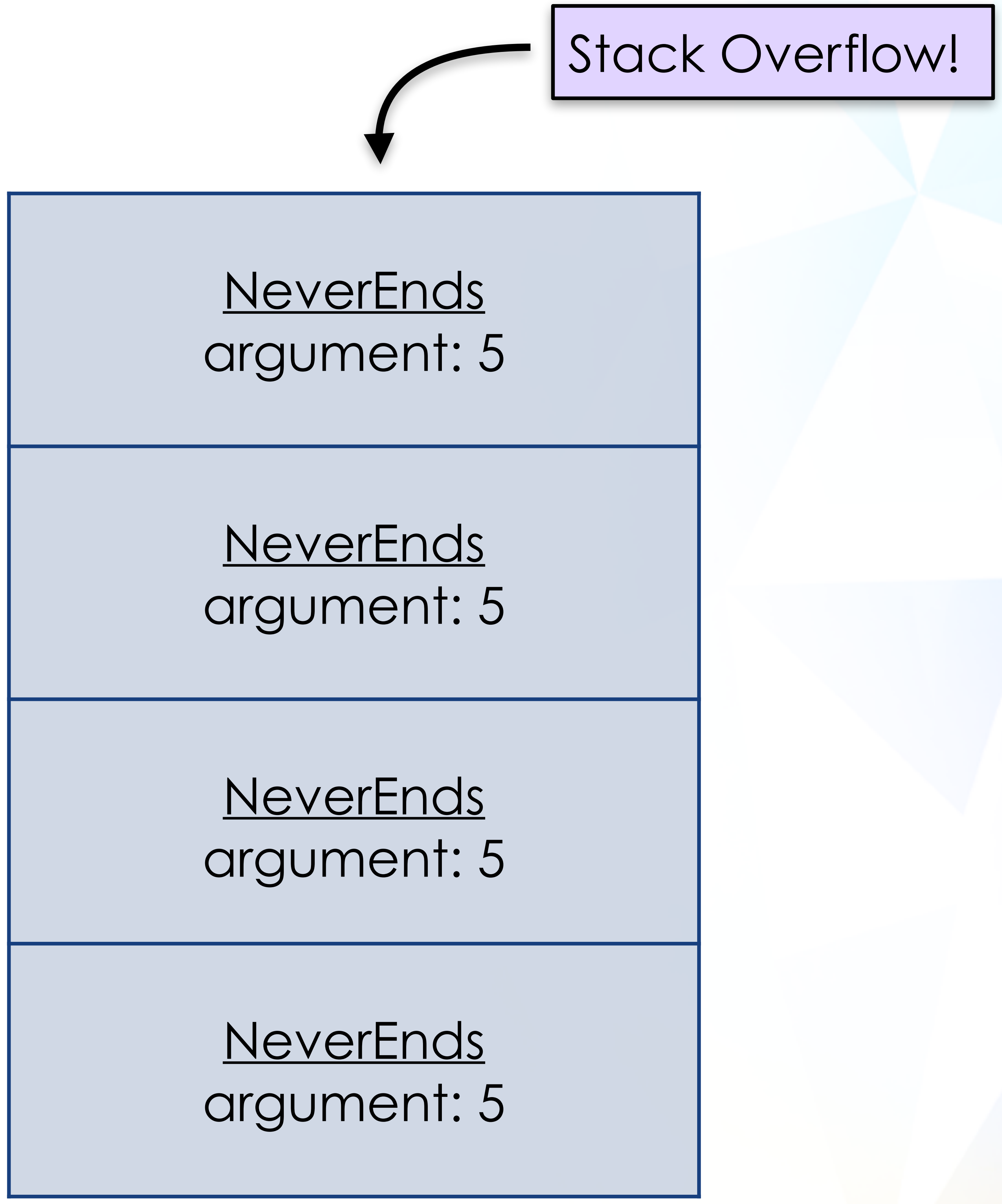
Bug: Terminating Condition for NeverEnds(...) can never be reached

```
class Program
{
    static void Main(string[] args)
    {
        NeverEnds(5);
    }

    static int NeverEnds(int n)
    {
        if (n == 0) {
            return 1;
        }

        return NeverEnds(n);
    }
}
```

Terminating Condition



Call Stack

Factorial

- Factorial n, denoted by n!, is to multiply all positive integers from n down to 1

- That is, $n! = n(n-1)! = n(n-1)(n-2)\ldots(2)(1)$

- Special case: $0! = 1$

Factorial (iterative)

Iterative Approach: Given a number n (zero-based), return the Factorial of n

```
int Factorial(int n)
{
    int fact = 1;

    for (int i = n; i > 0; i--) {
        fact *= i;
    }

    return fact;
}
```

if n = 4:

$4 * 3 * 2 * 1 = 24$

Factorial (recursive)

Recursive Approach: Given a number n (zero-based), return the Factorial of n

```
int Factorial(int n)
{
    if (n == 0) {
        return 1;
    }

    return n * Factorial(n - 1);
}
```

Terminating Condition

Sub-problem: Factorial of (n-1)

Factorial (recursive)

Breaking down the recursive approach of Factorial

```
int Factorial(int n)
{
    if (n == 0) {
        return 1;
    }

    return n * Factorial(n - 1);
}
```

Factorial(4)

↓

4 * Factorial(3)

↓

4 * 3 * Factorial(2)

↓

4 * 3 * 2 * Factorial(1)

↓

4 * 3 * 2 * 1 * Factorial(0)

↓

4 * 3 * 2 * 1 * 1 = 24

Fibonacci

- The Fibonacci Numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …
- Fibonacci Sequence $F_n = F_{n-1} + F_{n-2}$
- Seed Values
  - $F_0 = 0$
  - $F_1 = 1$

Fibonacci (iterative)

Iterative Approach: Given a number n (zero-based), return the n^{th} Fibonacci number

```
int Fibonacci(int n)
{
    int n_minus_1 = 1;
    int n_minus_2 = 0;
    int fib = 0;

    if (n == 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
    }
```

Terminating Conditions

Track F_{n-2}

```
    for (int i=2; i<=n; i++) {
        fib = n_minus_1 + n_minus_2;
        n_minus_2 = n_minus_1;
        n_minus_1 = fib;
    }

    return fib;
}
```

Track F_{n-1}

Fibonacci (recursive)

Recursive Approach: Given a number n (zero-based), return the n^{th} Fibonacci number

```
int Fibonacci(int n)
{
    if (n == 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
    }

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Terminating Conditions

$F_n = F_{n-1} + F_{n-2}$

Fibonacci (recursive)

Breaking down the recursive approach of Fibonacci

```
int Fibonacci(int n)
{
    if (n == 0) {
        return 0;
    }

    if (n == 1) {
        return 1;
    }

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Fib(3)

Fib(2) + Fib(1)

Fib(1) + Fib(0) + Fib(1)

1 + 0 + Fib(1)

1 + 0 + 1

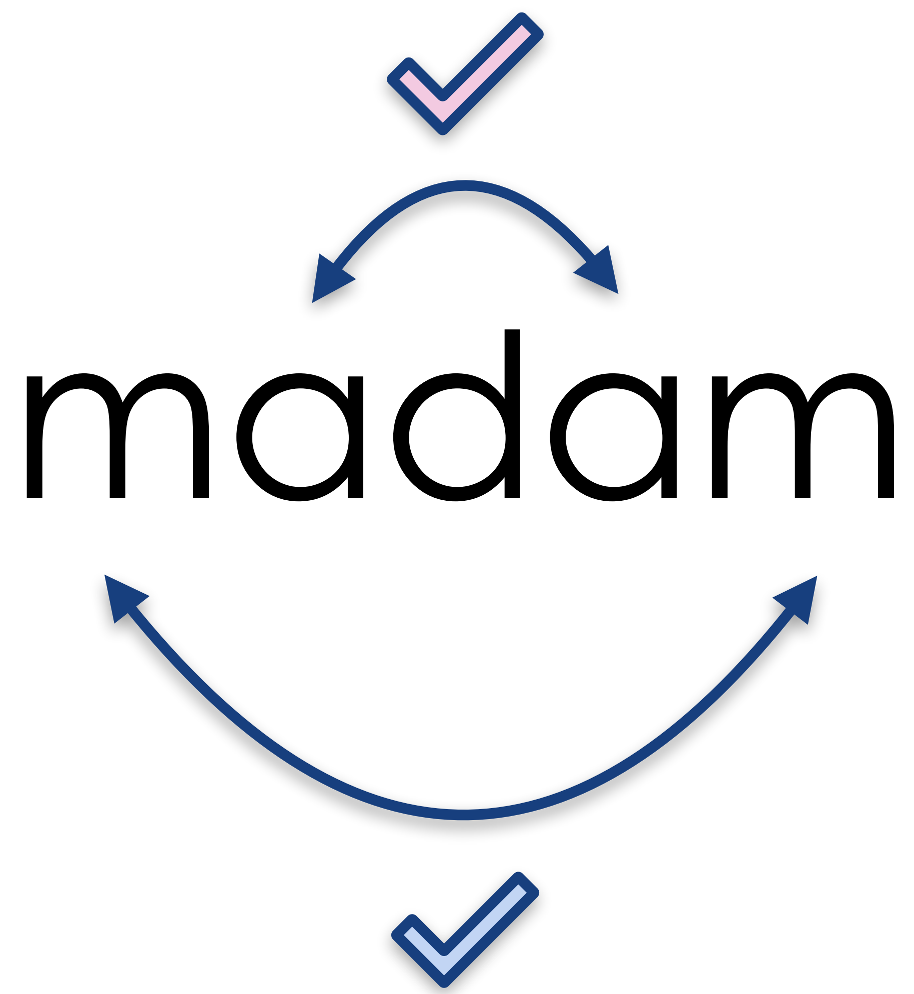
= 2

Palindrome

- A Palindrome is a word or number which reads the same backward as forward
- Examples: radar, level, rotor, madam, refer, wow
- Special Case:
  - An empty string (e.g. “”) is a Palindrome
  - A string with a single character (e.g. “a”) is a Palindrome

Palindrome

- To determine if a string is a Palindrome, keep checking that its Front and End characters are the same



Palindrome (iterative)

Strategy: Move pointers, from both ends, towards each other; check for matches

```
bool isPalindrome(string str)
{
    int i = 0;
    int j = str.Length - 1;

    while (i < j) {
        if (str[i] != str[j])
            return false;

        i++;
        j--;
    }

    return true;
}
```

Point to the Front → `int i = 0;`

Point to the End → `int j = str.Length - 1;`

Front and End characters did not match → `return false;`

Move front pointer right → `i++;`

Move end pointer left → `j--;`

All pairs match → `return true;`

Palindrome (recursive)

Recursively check if Front and End characters match

```
bool isPalindrome(string str)
{
    if (str.Length <= 1) {
        return true;
    }


    if (str[0] == str[str.Length-1]) {
        return isPalindrome(str.Substring(1, str.Length-2));
    }

    return false;
}
```

Terminating Condition:
Empty or Single character string

Front and End characters match?

Pass in a new string by removing the Front and End characters

THE END