UFUG 2601
C++ Programming

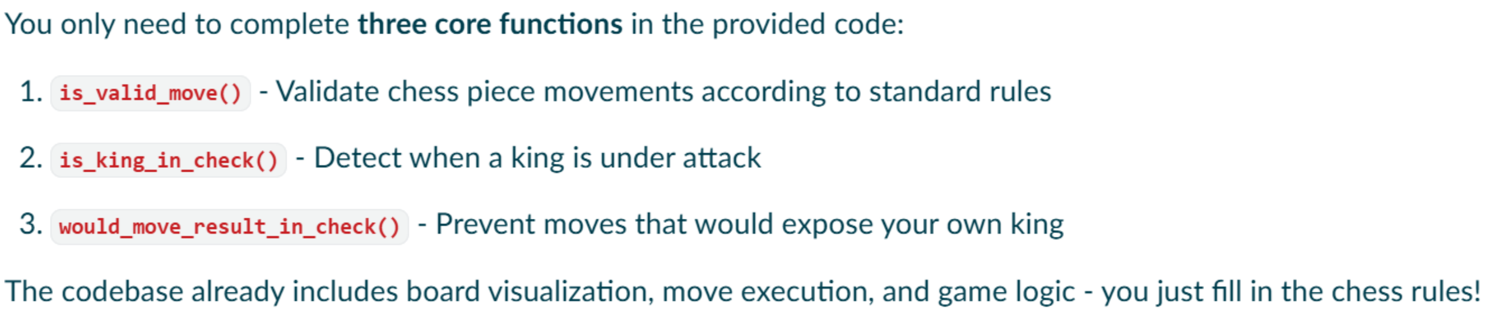
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Feedback

- vscode断点调试怎么用？可以用中文讲一遍吗

Chess Quiz (Optional)

- An optional chess programming quiz

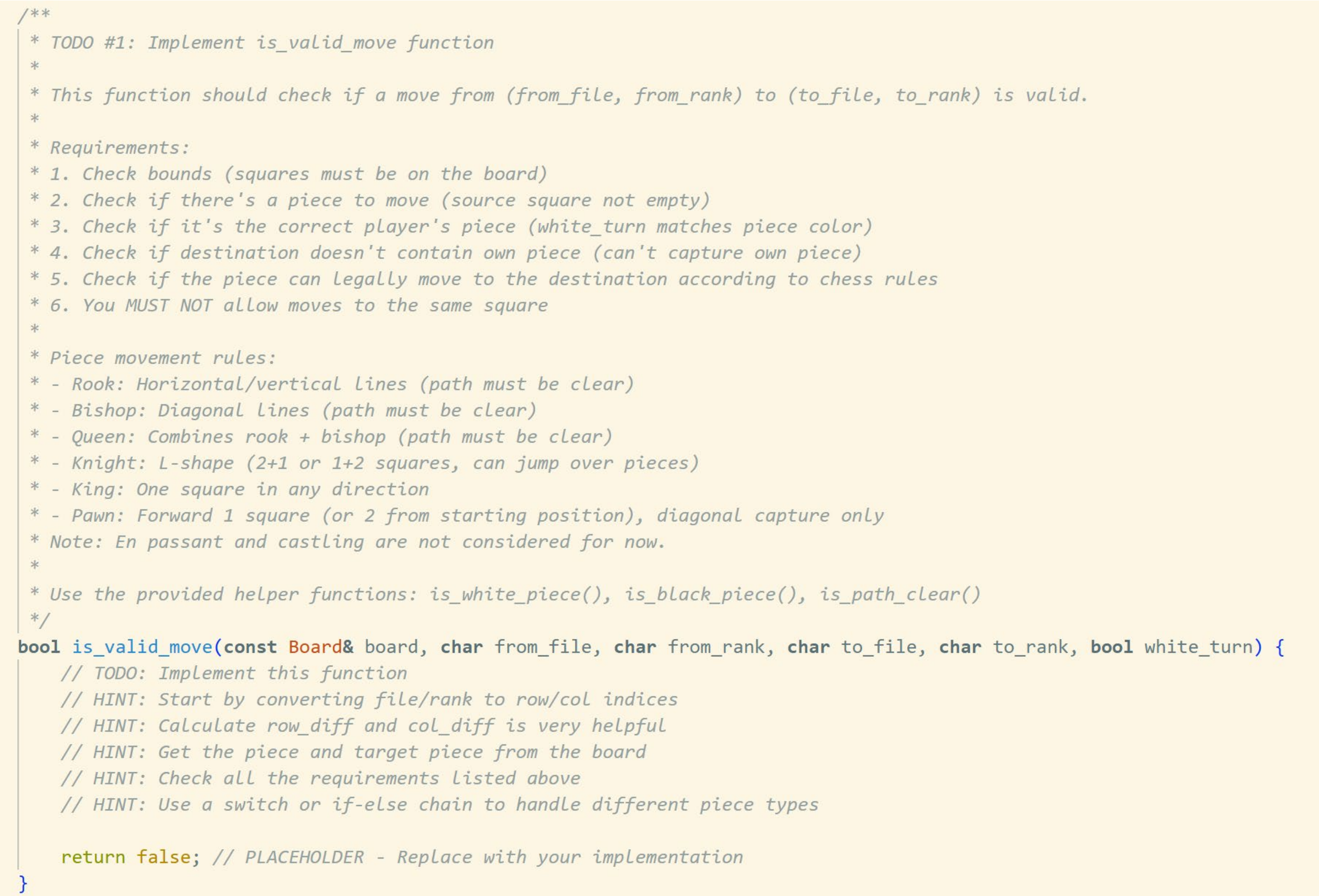


You only need to complete **three core functions** in the provided code:

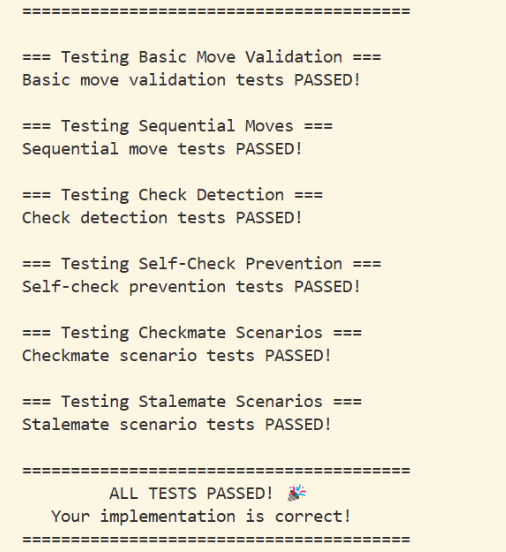
1. `is_valid_move()` - Validate chess piece movements according to standard rules
2. `is_king_in_check()` - Detect when a king is under attack
3. `would_move_result_in_check()` - Prevent moves that would expose your own king

The codebase already includes board visualization, move execution, and game logic - you just fill in the chess rules!

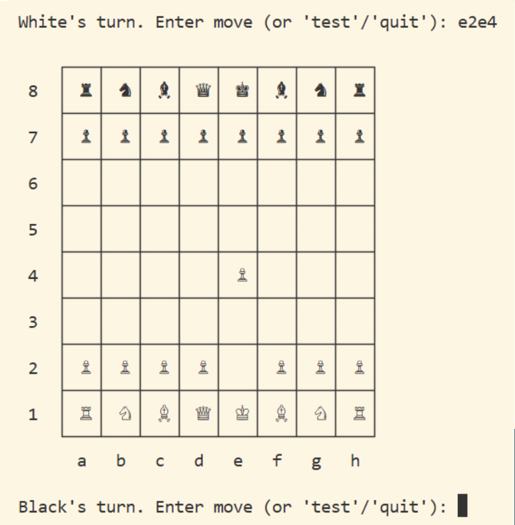
Chess Quiz (Optional)



```cpp
/**
 * TODO #1: Implement is_valid_move function
 *
 * This function should check if a move from (from_file, from_rank) to (to_file, to_rank) is valid.
 *
 * Requirements:
 * 1. Check bounds (squares must be on the board)
 * 2. Check if there's a piece to move (source square not empty)
 * 3. Check if it's the correct player's piece (white_turn matches piece color)
 * 4. Check if destination doesn't contain own piece (can't capture own piece)
 * 5. Check if the piece can legally move to the destination according to chess rules
 * 6. You MUST NOT allow moves to the same square
 *
 * Piece movement rules:
 * - Rook: Horizontal/vertical lines (path must be clear)
 * - Bishop: Diagonal lines (path must be clear)
 * - Queen: Combines rook + bishop (path must be clear)
 * - Knight: L-shape (2+1 or 1+2 squares, can jump over pieces)
 * - King: One square in any direction
 * - Pawn: Forward 1 square (or 2 from starting position), diagonal capture only
 * Note: En passant and castling are not considered for now.
 *
 * Use the provided helper functions: is_white_piece(), is_black_piece(), is_path_clear()
 */
bool is_valid_move(const Board& board, char from_file, char from_rank, char to_file, char to_rank, bool white_turn) {
    // TODO: Implement this function
    // HINT: Start by converting file/rank to row/col indices
    // HINT: Calculate row_diff and col_diff is very helpful
    // HINT: Get the piece and target piece from the board
    // HINT: Check all the requirements listed above
    // HINT: Use a switch or if-else chain to handle different piece types

    return false; // PLACEHOLDER - Replace with your implementation
}
```



```
========================================

=== Testing Basic Move Validation ===
Basic move validation tests PASSED!

=== Testing Sequential Moves ===
Sequential move tests PASSED!

=== Testing Check Detection ===
Check detection tests PASSED!

=== Testing Self-Check Prevention ===
Self-check prevention tests PASSED!

=== Testing Checkmate Scenarios ===
Checkmate scenario tests PASSED!

=== Testing Stalemate Scenarios ===
Stalemate scenario tests PASSED!

========================================
        ALL TESTS PASSED! 🎉
   Your implementation is correct!
========================================
```



```
White's turn. Enter move (or 'test'/'quit'): e2e4

8 | ♜ | ♞ | ♝ | ♛ | ♚ | ♝ | ♞ | ♜ |
7 | ♟ | ♟ | ♟ | ♟ | ♟ | ♟ | ♟ | ♟ |
6 |   |   |   |   |   |   |   |   |
5 |   |   |   |   |   |   |   |   |
4 |   |   |   |   | ♙ |   |   |   |
3 |   |   |   |   |   |   |   |   |
2 | ♙ | ♙ | ♙ | ♙ |   | ♙ | ♙ | ♙ |
1 | ♖ | ♘ | ♗ | ♕ | ♔ | ♗ | ♘ | ♖ |

    a   b   c   d   e   f   g   h

Black's turn. Enter move (or 'test'/'quit'): █
```

Recap

	Vector	Array
Declaration	`vector<int> vec;`	`int arr[10];`
Initialization	`vector<int> vec{1, 3, 4};`	`int arr[] {1, 3, 4};`
Access Elements	`vec.at(3)` or `vec[3]`	`arr[3]`
Get Size	`vec.size()`	N/A
Dynamic Size	Yes	No

Recap

	Reference	Pointer
Declaration	N/A	`int* ptr;`
Initialization	`int& ref = val;`	`int* ptr = &val;`
Access Value	`ref = 5`	`*ptr = 5`
Use as Parameters	Can change the original value Avoid copy an object like vector	Can change the original value An array will decay to a pointer

Recap

	Array	Pointer
Syntax	`int arr[10];`	`int* ptr;`
Memory	Fixed, contiguous	Reassignable
`sizeof`	Length * `sizeof(type)`	Size of the pointer (8/4 for 64/32-bit program)
Get value	`arr[i];`	`*(ptr + i)`

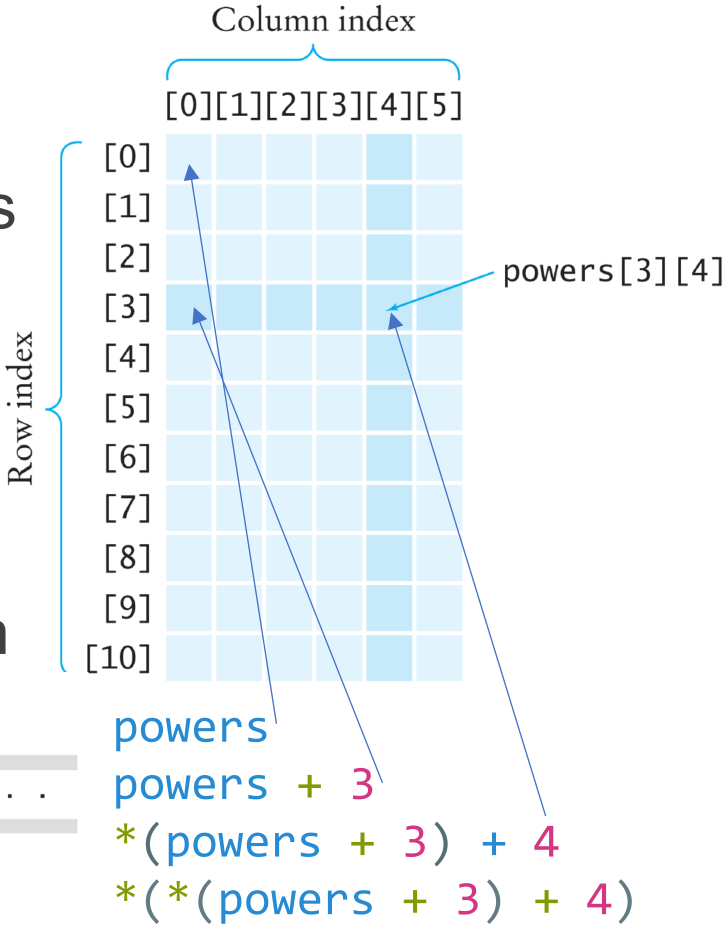
Arrays decay to pointers when passed as a parameter to functions, or in many arithmetic expression

Two-Dimensional Array

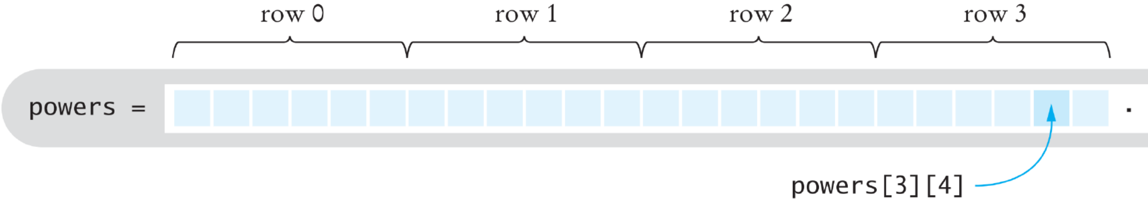
- C++ uses an array with two subscripts to store a two-dimensional array

```
constexpr int ROWS = 11;
constexpr int COLS = 6;
double powers[ROWS][COLS];
```

- Specify two subscripts in separate brackets to select the row and column



```
powers
powers + 3
*(powers + 3) + 4
*(*(powers + 3) + 4)
```



Two-Dimensional Array as Parameter

- When passing a multi-dimensional array to a function, you must have bounds for all dimensions **except the first**.

```
void print(const double table[][10], int rows) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < 10; j++) // Try j < 11
        cout << table[i][j] << '\t';
        cout << "\n";
    }
}
```

Just like `int a[10]` will decay to `int a[]` or `int* a`
`double a[5][10]` will decay to `double a[][10]` or `double* a[10]`

```
void print(const double table[][], int rows, int cols)
```

Recursion

Goal of This Lecture

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

A Simple Example

- We can use a loop to computer the value of $n!$

```
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

- But there is also a recursive solution $n! = (n-1)! \times n$

A Simple Example

- How to compute $n!$
  - If n is 0 or 1, then the factorial is 1
  - Otherwise, it is $(n - 1)! \times n$

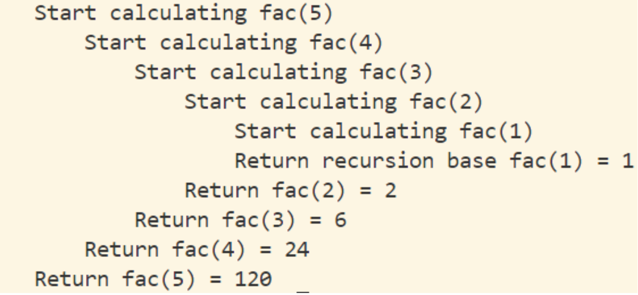
```
int factorial(int n) {
    if (n == 0 || n == 1) return 1;
    return factorial(n - 1) * n;
}
```

base case

How the Value is Calculated

```
void print_space(int n) {
    for (int i = 0; i < n; ++i) cout << " ";
}
```

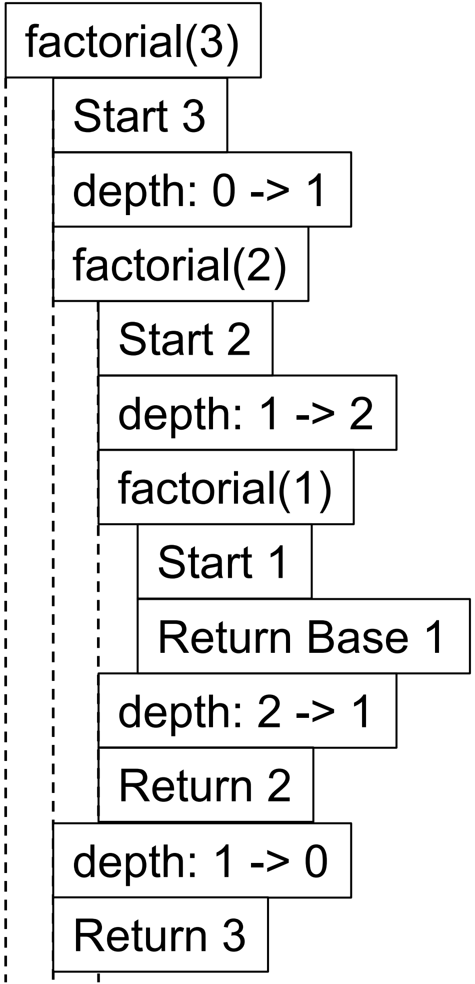
```
int depth = 0;
int factorial(int n) {
    print_space(depth);
    cout << "Start calculating fac(" << n << ")" << endl;
    if (n == 0 || n == 1) {
        print_space(depth);
        cout << "Return recursion base fac(" << n << ") = " << 1 << endl;
        return 1;
    }
    depth += 1;
    int temp = factorial(n - 1);
    depth -= 1;
    print_space(depth);
    cout << "Return fac(" << n << ") = " << temp * n << endl;
    return temp * n;
}
```



```
Start calculating fac(5)
 Start calculating fac(4)
  Start calculating fac(3)
   Start calculating fac(2)
    Start calculating fac(1)
    Return recursion base fac(1) = 1
   Return fac(2) = 2
  Return fac(3) = 6
 Return fac(4) = 24
Return fac(5) = 120 _
```

How the Value is Calculated

```
int depth = 0;
int factorial(int n) {
    print_space(depth);
    cout << "Start " << n << endl;
    if (n == 0 || n == 1) {
        print_space(depth);
        cout << "Return Base " << n << endl;
        return 1;
    }
    depth += 1;
    int temp = factorial(n - 1);
    depth -= 1;
    print_space(depth);
    cout << "Return " << n << endl;
    return temp * n;
}
```



factorial(3)

Start 3

depth: 0 -> 1

factorial(2)

Start 2

depth: 1 -> 2

factorial(1)

Start 1

Return Base 1

depth: 2 -> 1

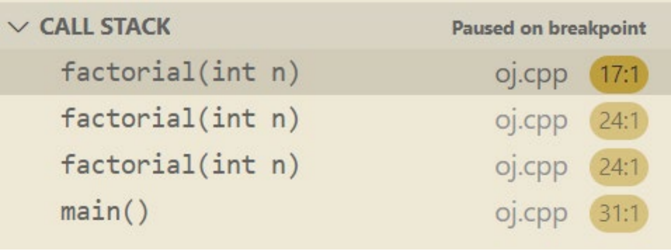
Return 2

depth: 1 -> 0

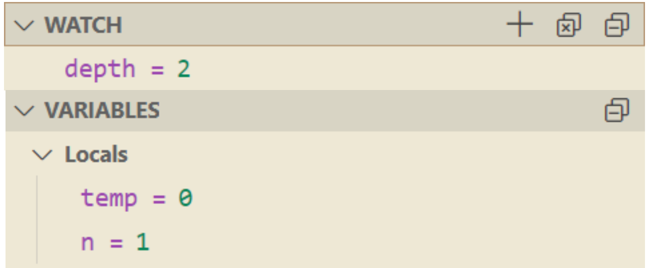
Return 3

Use Debugger to Track the Flow

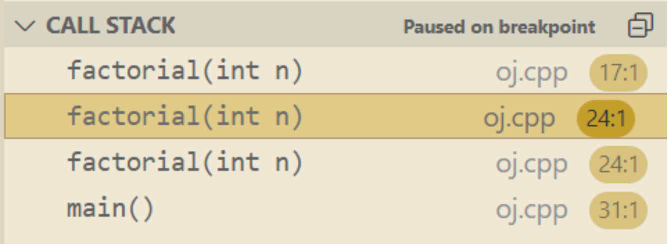
- Step into: go into a deeper level
- Step out: go out of current level
- Step over: go through current level



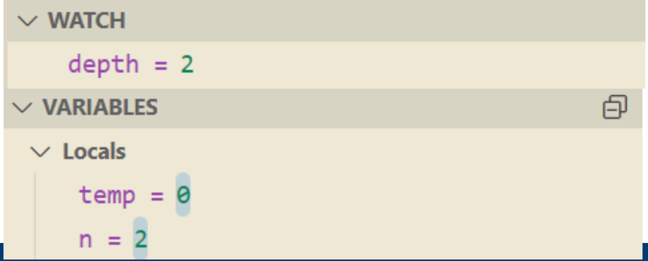
```
CALL STACK                Paused on breakpoint
  factorial(int n)        oj.cpp   17:1
  factorial(int n)        oj.cpp   24:1
  factorial(int n)        oj.cpp   24:1
  main()                  oj.cpp   31:1
```



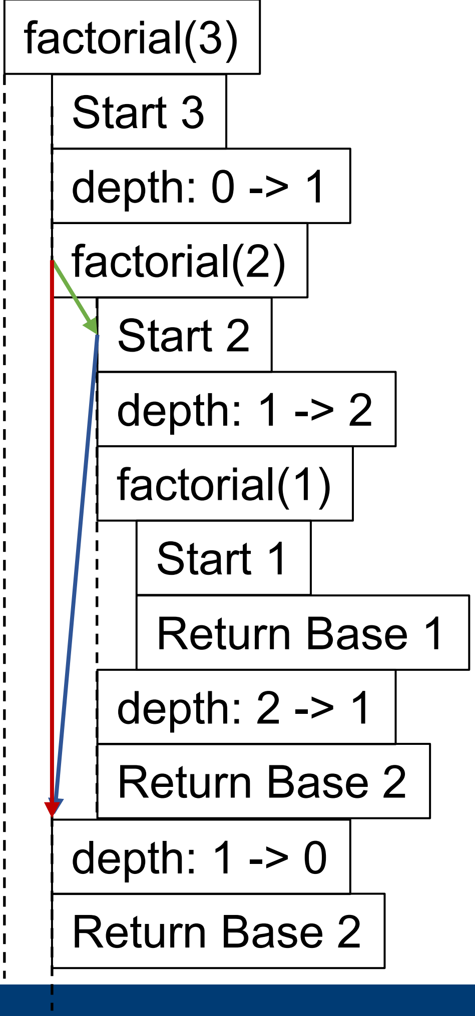
```
WATCH
  depth = 2
VARIABLES
  Locals
    temp = 0
    n = 1
```



```
CALL STACK                Paused on breakpoint
  factorial(int n)        oj.cpp   17:1
  factorial(int n)        oj.cpp   24:1
  factorial(int n)        oj.cpp   24:1
  main()                  oj.cpp   31:1
```



```
WATCH
  depth = 2
VARIABLES
  Locals
    temp = 0
    n = 2
```



factorial(3)
Start 3
depth: 0 -> 1
factorial(2)
Start 2
depth: 1 -> 2
factorial(1)
Start 1
Return Base 1
depth: 2 -> 1
Return Base 2
depth: 1 -> 0
Return Base 2

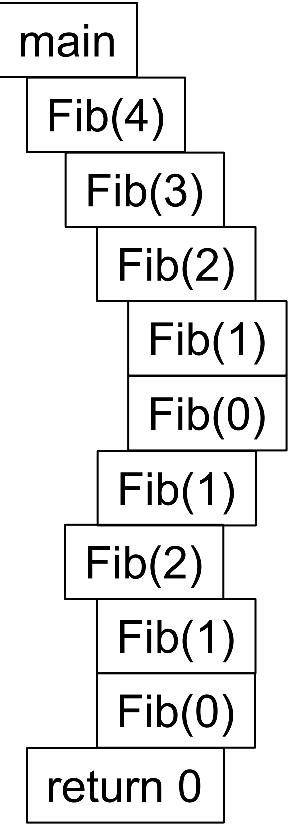
Recursion

- The technique of expressing a solution to a problem in terms of solution to a smaller version of the same problem is called recursion
- Two key requirements to make a successful recursion
  - Base cases that handle the simplest computations directly
  - Every recursive call must simplify the computation in some way
- Sometimes, recursion is not needed, a simple loop is better

Exercise

- Write a recursive function to compute Fibonacci(n)
    - Fib(0)=0, Fib(1)=1, Fib(n)=Fib(n-1)+Fib(n-2)
- How to write a simple loop to compute it?
- Will revisit this example when we talk about efficiency

Exercise



```
int Fib(unsigned int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return Fib(n - 1) + Fib(n - 2);
}
```

```
int Fib(unsigned int n) {
    int x = 0, y = 1;
    for (int i = 0; i < n; ++i) {
        int temp = x + y;
        x = y;
        y = temp;
    }
    return x;
}
```

Think Recursively

- To illustrate the steps, we will test whether a sentence is a palindrome - a string that is equal to itself when you reverse all characters
    - Examples: level, madam, rotor
- Our goal is to implement a predicate function

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

Think Recursively

- Step 1: Consider various ways for simplifying inputs
  - How can you simplify the inputs in such a way that the same problem can be applied to simpler input
  - Here are several possibilities for the palindrome test problem
    - Remove the first character
    - Remove the last character
    - Remove both the first and last character
    - Remove a character from the middle
    - Cut the string into two halves

Think Recursively

- Step 2: Combine solutions with simpler inputs to a solution of the original problem
  - Don’t worry how those solutions are obtained. These are simpler inputs, so someone else will solve the problem for you
  - Removing the first and last characters seems promising
    - “level” -> “eve”
  - A word is a palindrome if
    - the first and last letters match, and
    - the word obtained by removing the first and last letters is still a palindrome

Think Recursively

- Step 3: Find solutions to the simplest inputs
  - To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately
  - Sometimes you get into philosophical questions dealing with degenerate inputs: empty strings, shapes with no area, and so on
  - You may want to investigate a slightly larger input that gets reduces to a degenerate input and see what value you should attach to the degenerate input yields the correct answer

Infinite Recursion: A common programming error that a function calling itself over and over with no end in sight.

Think Recursively

- Step 3 (continued): Find solutions to the simplest inputs
  - The simplest strings for the palindrome test may contain:
    - strings with two characters
    - strings with a single character
    - the empty string
  - A single character string is a palindrome
  - An empty string is a palindrome
- Step 4: Implement the solution by combining the simple cases and the reduction step

Solution

```
bool is_palindrome_v1(string s) {
    if (s.length() <= 1) return true; // base cases
    if (s[0] != s[s.length() - 1]) return false;
    return is_palindrome_v1(s.substr(1, s.length() - 2));
}
```

- Question: can it be more efficient?

Avoid copying string `const string& s`

Avoid generating substring

```
bool is_substring_palindrome(const string& s, int start, int end)
```

More Solutions

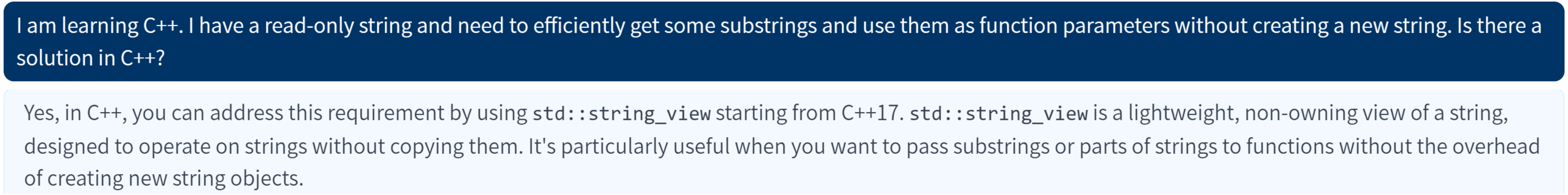
```
bool is_substring_palindrome(const string& s, int start, int end) {
    // check the substring s[start,...,end)
    if (end - start < 2) return true; // base cases
    if (s[start] != s[end - 1]) return false;
    return is_substring_palindrome(s, start + 1, end - 1);
}
bool is_palindrome_v2(const string& s) {
    return is_substring_palindrome(s, 0, s.length());
}
```

`is_substring_palindrome` is called recursive helper functions

`is_substring_palindrome` is the wrapper function for ease of use

Is there other way to avoid copying a string when passing as parameter?

More Solutions



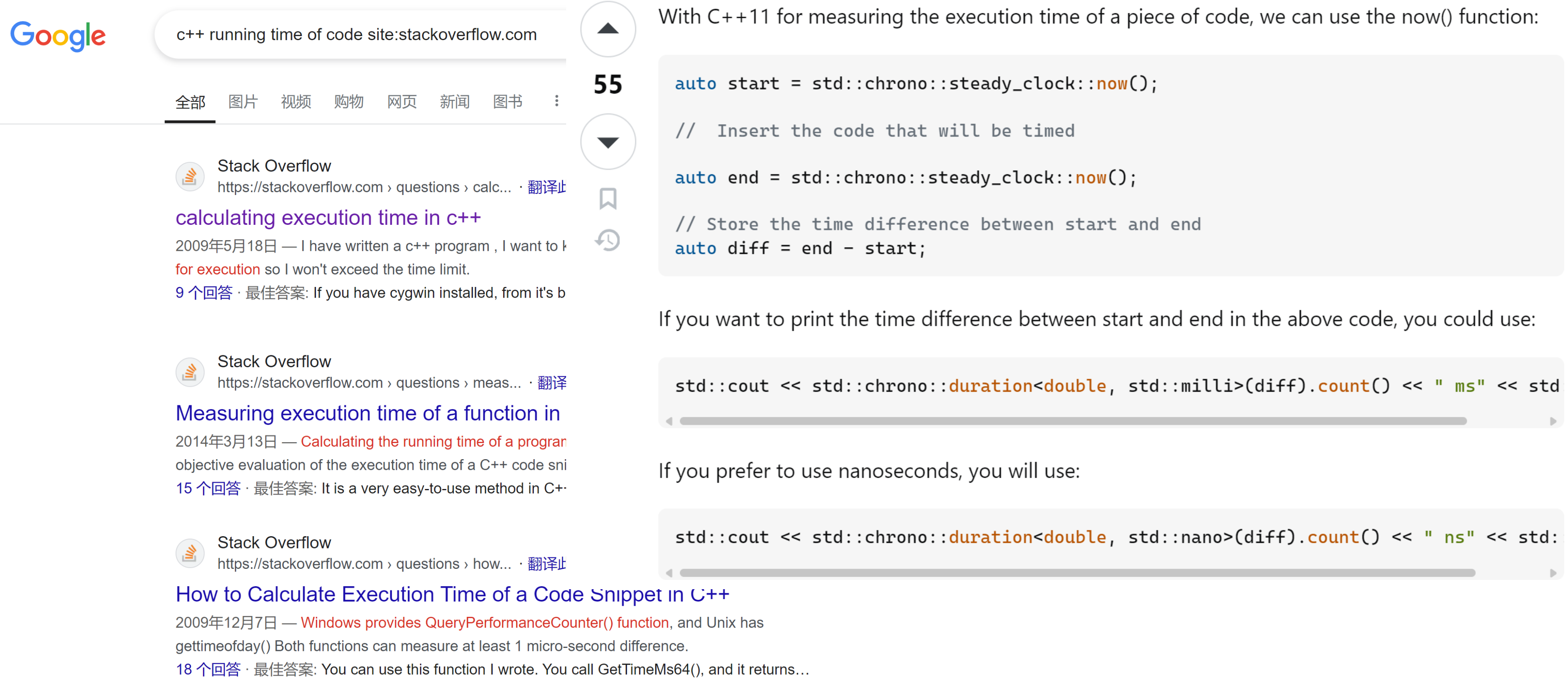
I am learning C++. I have a read-only string and need to efficiently get some substrings and use them as function parameters without creating a new string. Is there a solution in C++?

Yes, in C++, you can address this requirement by using `std::string_view` starting from C++17. `std::string_view` is a lightweight, non-owning view of a string, designed to operate on strings without copying them. It's particularly useful when you want to pass substrings or parts of strings to functions without the overhead of creating new string objects.

```
using std::string_view;
bool is_palindrome_v3(string_view s) {
    if (s.length() <= 1) return true; // base cases
    if (s[0] != s[s.length() - 1]) return false;
    return is_palindrome(s.substr(1, s.length() - 2));
}
```

Much efficient to get a read-only substring!

Compare the Efficiency



Stack Overflow
https://stackoverflow.com › questions › calc... · 翻译此

calculating execution time in c++

2009年5月18日 — I have written a c++ program , I want to
for execution so I won't exceed the time limit.
9 个回答 · 最佳答案: If you have cygwin installed, from it's

Stack Overflow
https://stackoverflow.com › questions › meas... · 翻译

Measuring execution time of a function in

2014年3月13日 — Calculating the running time of a progra
objective evaluation of the execution time of a C++ code sni
15 个回答 · 最佳答案: It is a very easy-to-use method in C+

Stack Overflow
https://stackoverflow.com › questions › how... · 翻译此

How to Calculate Execution Time of a Code Snippet in C++

2009年12月7日 — Windows provides QueryPerformanceCounter() function, and Unix has gettimeofday() Both functions can measure at least 1 micro-second difference.
18 个回答 · 最佳答案: You can use this function I wrote. You call GetTimeMs64(), and it returns…

With C++11 for measuring the execution time of a piece of code, we can use the now() function:

```
auto start = std::chrono::steady_clock::now();

//  Insert the code that will be timed

auto end = std::chrono::steady_clock::now();

// Store the time difference between start and end
auto diff = end - start;
```

If you want to print the time difference between start and end in the above code, you could use:

```
std::cout << std::chrono::duration<double, std::milli>(diff).count() << " ms" << std
```

If you prefer to use nanoseconds, you will use:

```
std::cout << std::chrono::duration<double, std::nano>(diff).count() << " ns" << std:
```

Compare the Efficiency

```
#include <chrono>
int main() {
    std::string s(100000, 'a'); // construct a corner case
    auto start = std::chrono::steady_clock::now();
    is_palindrome_v1(s);
    auto end = std::chrono::steady_clock::now();
    std::cout << "V1: " << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << "ms" << endl;

    start = std::chrono::steady_clock::now();
    is_palindrome_v2(s);
    end = std::chrono::steady_clock::now();
    std::cout << "V2: " << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << "ms" << endl;
}
```

```
V1: 736ms
V2: 0ms
```

It is not required to master the usage of `string_view` or `chrono`

But it is important to learn how to use tools to help you practice your C++ skills

And… Be curious

Exercise: Permutations

- We will design a recursive function that lists all permutations of a vector of distinct integers
- A permutation is simply a rearrangement of the integers:
  - [1,2,4], [1,4,2], [2,1,4], [2,4,1], [4,1,2], [4,2,1]
- If a vector has n distinct numbers, what's the total number of permutations?
- What are the input and output types of the function?
  - For simplicity, we print all the permutations here

```
void permute(vector<int>& v)
```

- How to design the helper function?

Think Recursively

[1,2,4], [1,4,2], [2,1,4], [2,4,1], [4,1,2], [4,2,1]

Step 1: Consider various ways for simplifying inputs
Step 2: Combine solutions with simpler inputs to a solution of the original problem
Step 3: Find solutions to the simplest inputs
Step 4: Implement the solution by combining the simple cases and the reduction step

- There are $n!$ permutations
- After we fix the first element, there are $(n-1)!$ permutations

```
void permute(vector<int>& v, int idx) // determine v.at(idx)
    if (idx == v.size()) // just print the result
    else // fix the first element and call permute(v, idx+1)
```

Think Recursively

```
void permute(vector<int>& v, int idx) {
    if (idx == v.size()) {
        for (auto val : v) cout << val << " ";
        cout << endl;
    }
    for (int j = idx; j < v.size(); ++j) {
        std::swap(v.at(idx), v.at(j));
        permute(v, idx + 1);
        std::swap(v.at(j), v.at(idx)); // backtrack
    }
}
void permute_wrapper(vector<int>& v) {
    permute(v, 0);
}
```

Use debugger to track this program

Permutations

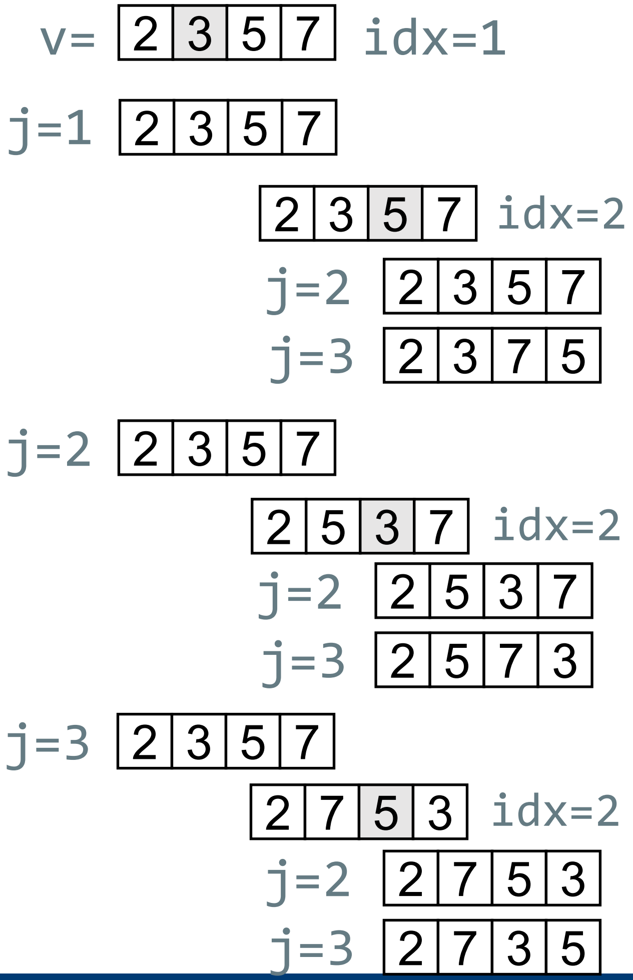
- Assume we have determined the elements before `idx`
- Now I will try every possible values in `v.at(idx)`
- And let the recursive function to handle the remaining part

```
void permute(vector<int>& v, int idx) {
    if (idx == v.size()) { // base case
        for (auto val : v) cout << val << " ";
        cout << endl;
    }
    for (int j = idx; j < v.size(); ++j) {
        std::swap(v.at(idx), v.at(j));
        permute(v, idx + 1);
        std::swap(v.at(j), v.at(idx)); // backtrack
    }
}
```

Ensure that v is unchanged after each loop

Permutations

```
void permute(vector<int>& v, int idx) {
    if (idx == v.size()) {
        for (auto val : v) cout << val << " ";
        cout << endl;
    }
    for (int j = idx; j < v.size(); ++j) {
        std::swap(v.at(idx), v.at(j));
        permute(v, idx + 1);
        std::swap(v.at(j), v.at(idx));
    }
}
```



v= 2 3 5 7 idx=1

j=1 2 3 5 7

2 3 5 7 idx=2

j=2 2 3 5 7

j=3 2 3 7 5

j=2 2 3 5 7

2 5 3 7 idx=2

j=2 2 5 3 7

j=3 2 5 7 3

j=3 2 3 5 7

2 7 5 3 idx=2

j=2 2 7 5 3

j=3 2 7 3 5

Mutual Recursion

- Sometimes you might have multiple functions calls each other in a recursive fashion
- For our example, we will develop a program that can compute the values of arithmetic expressions such as

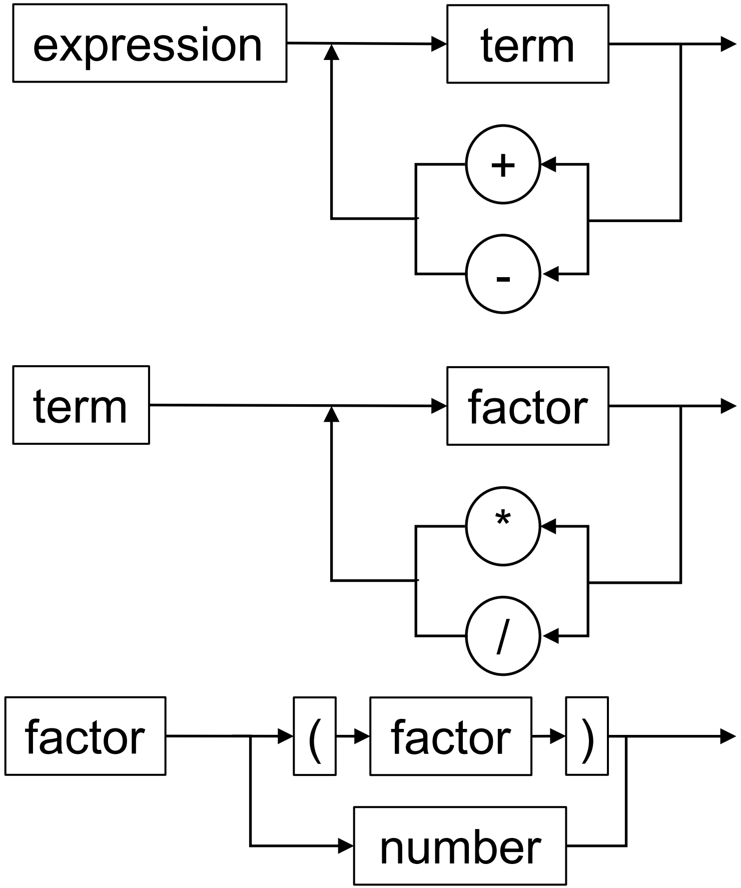
$$3 + 4 * 5$$

$$(3 + 4) * 5$$

$$1 - (2 - (3 - (4 - 5)))$$

Mutual Recursion

- The follow syntax diagrams describe the syntax of these expressions
  - An expressions is either a term, or a sum of different terms
  - A term is either a factor, or a product or quotient of factors
  - A factor is either a number or an expression closed in parentheses



Mutual Recursion

Expression: term or sum of terms
Terms: factor or product of factors
Factor: number or (expression)

- Examples
  - $3 + 4 * 5$ is an expression
  - It contains two terms: 3 and $4 * 5$
  - 3 is a factor, and then a number
  - $4 * 5$ is a product of two factors 4 and 5, and they are numbers

  - $(3 + 4) * 5$ is a term
  - It contains two factors: $(3 + 4)$ and 5
  - $(3 + 4)$ is an (expression), which is a sum of two terms
  - Each term is a factor and then a number
  - 5 is a factor, and then a number

Mutual Recursion

Expression: term or sum of terms
Terms: factor or product of factors
Factor: number or (expression)

- To compute the value of an expression, we implement three functions. Each of them will read input and return a number

```
int expression_value();
int term_value();
int factor_value();
```

- For `expression_value`, it first calls `term_value`, and if meets + or -, calls `term_value` again and calculate the result
- For `term_value`, it first calls `factor_value`, and if meets * or /, calls `factor_value` again and calculate the result
- For `factor_value`, check the next input. If it is a digit then read the number, else “eats” ‘(’ and calls `expression_value`

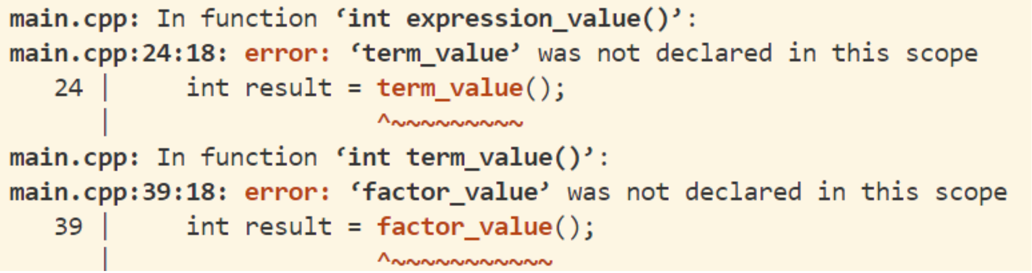
Solution

```
int expression_value() {
    int result = term_value();
    while (true) {
        char op = cin.peek(); // peek but dont eat!
        if (op == '+' || op == '-') {
            cin.get(); // eat the op, read term, and calculate
            int value = term_value();
            if (op == '+') result += value;
            else result -= value;
        } else break;
    }
    return result;
}
```

Solution

```
int term_value() {
    int result = factor_value();
    while (true) {
        char op = cin.peek(); // peek but dont eat!
        if (op == '*' || op == '/') {
            cin.get(); // eat the op, read term, and calculate
            int value = factor_value();
            if (op == '*') result *= value;
            else result /= value;
        } else break;
    }
    return result;
}
```

Solution



```
main.cpp: In function ‘int expression_value()’:
main.cpp:24:18: error: ‘term_value’ was not declared in this scope
   24 |     int result = term_value();
      |                  ^~~~~~~~~~
main.cpp: In function ‘int term_value()’:
main.cpp:39:18: error: ‘factor_value’ was not declared in this scope
   39 |     int result = factor_value();
      |                  ^~~~~~~~~~~~
```

```
int factor_value() {
    char c = cin.peek();
    if (c == '(') {
        cin.get(); // eat '('
        int result = expression_value();
        cin.get(); // eat ')'
        return result;
    } else {
        int result;
        cin >> result;
        return result;
    }
}
```

Make it more robust to the input?

(3+4)   (3   + 4)* 6

```
// tell the compiler that you will
// implement these functions later
int expression_value();
int term_value();
int factor_value();
int main() {
    std::cout << expression_value();
}
// implementation here
```

The Efficiency of Recursion

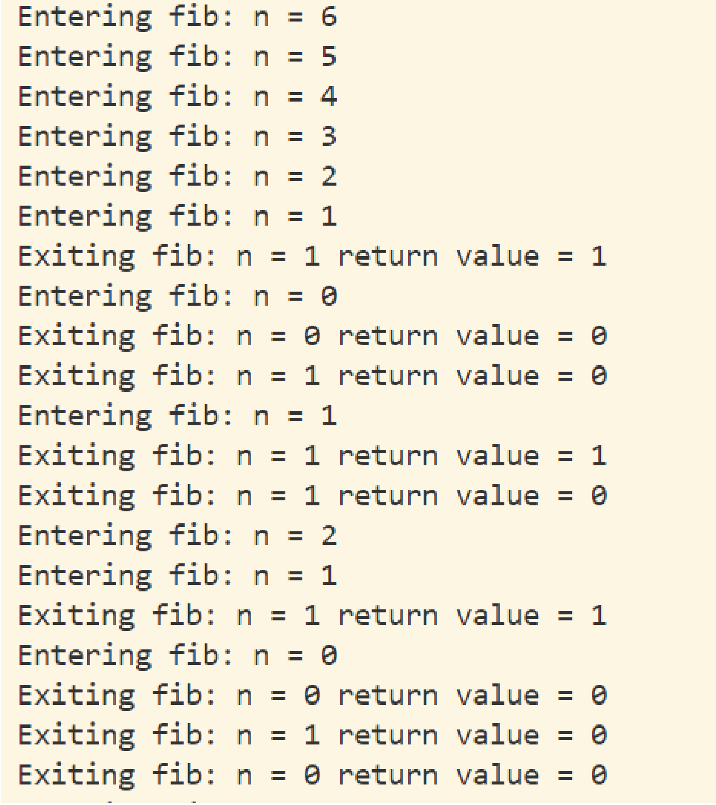
- Although recursion can be a powerful tool to implement complex algorithms, it can lead to algorithms that perform poorly
- We will analyze the question of when recursion is beneficial and when it is inefficient
- For our study we will examine the Fibonacci sequence of numbers

The Efficiency of Recursion

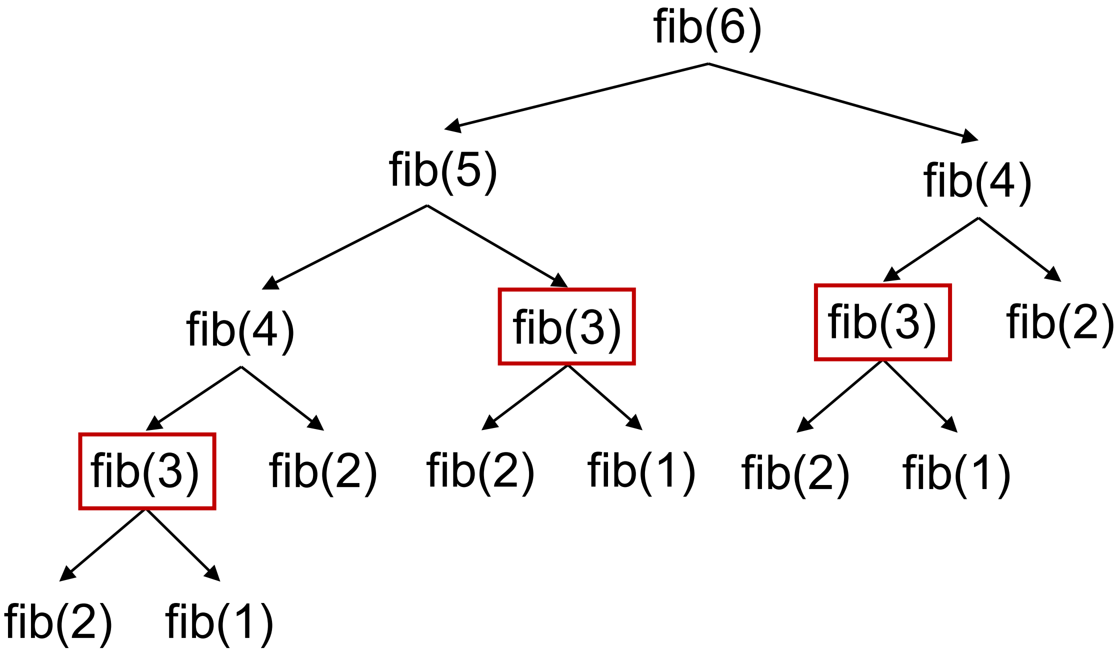
- To determine the problem, we insert trace messages into the function

```
int fib(int n) {
    int ret = 0;
    cout << "Entering fib: n = " << n << endl;
    if (n == 0) ret = 0;
    else if (n == 1) ret = 1;
    else ret = fib(n - 1) + fib(n - 2);
    cout << "Exiting fib: n = " << n
    << " return value = " << ret << endl;
    return ret;
}
```

The Efficiency of Recursion



```
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Entering fib: n = 0
Exiting fib: n = 0 return value = 0
Exiting fib: n = 1 return value = 0
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 1 return value = 0
Entering fib: n = 2
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Entering fib: n = 0
Exiting fib: n = 0 return value = 0
Exiting fib: n = 1 return value = 0
Exiting fib: n = 0 return value = 0
```



How to avoid repeat calculation?

Memoization

- Memoization stores the results of expensive function calls and return the cached result

```
vector<int> FIB(10000, -1);
int fib(int n) {
    if (FIB.at(n) != -1) return FIB.at(n);
    int ret = 0;
    if (n == 0) ret = 0;
    else if (n == 1) ret = 1;
    else ret = fib(n - 1) + fib(n - 2);
    FIB.at(n) = ret;
    return ret;
}
```

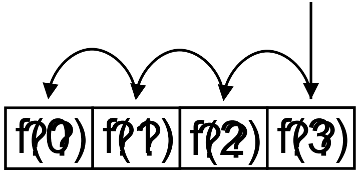
⬅ Each slot will be calculated only once

The Efficiency of Recursion

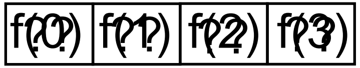
- A person would just write down the values as they were computed and add up the last two to get the next one; no sequence value would ever be computed twice
- You can user a loop to imitate this process

```
int Fib(unsigned int n) {
    int x = 0, y = 1;
    for (int i = 0; i < n; ++i) {
        int temp = x + y;
        x = y;
        y = temp;
    }
    return x;
}
```

Recursive vs Iterative



- Recursive: solve the problem “top down”
  - Can use *memoization* to accelerate the result



- Iterative: solve the problem “bottom up”
  - It is also called *tabulation* in some context
- Typically, the iterative solution is faster than recursive solution, because each function call will create a stack that need to store parameter values
- Recursive solution is usually easier to implement, and has advantages if not all the subproblems need to be solved

Tail Recursion

- Compiler can optimize the stack creation if the recursion is tail recursion, i.e., nothing left after recursive call

```cpp
long long sum(int n) {
    if (n == 0) return 0;
    int temp = sum(n - 1);
    int result = temp + n;
    return result;
}
```

Segmentation fault!

```cpp
long long sum_helper(int n, long long ret) {
    if (n == 0) return ret;
    return sum_helper(n - 1, ret + n);
}
long long sum_v2(int n) {
    return sum_helper(n, 0);
}
```

```
g++ -O0 test.cpp -o test; ./test
```

Segmentation fault!

```
g++ -O2 test.cpp -o test; ./test
```

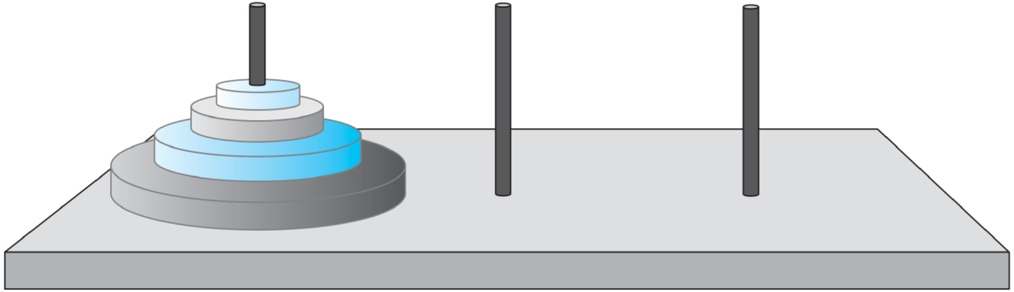
500000500000

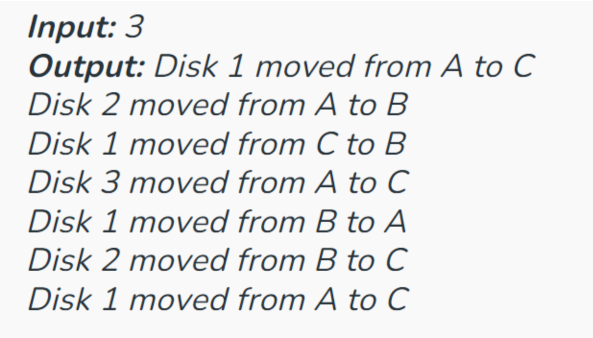
Recursive vs Iterative

- There are quite a few problems that are dramatically easier to solve recursively that iteratively
- Sometimes, code simplicity and readability is more important than small performance gain

Exercise: Towers of Hanoi

- This is a well-known puzzle. A stack of disks of decreasing size is to be transported from the left-most rod to the right-most rod. You can place smaller disks only on top of larger ones, not the other way around.





Input: *3*
Output: *Disk 1 moved from A to C*
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C

Solution

- How to move n disk from A to B?
  - Move $n-1$ disk from A to C
  - Move (the last) disk from A to B
  - Move $n-1$ disk from C to B

```
void towers_of_hanoi(int n) {
    // discuss base cases
    towers_of_hanoi(n - 1);
    // print the message about the move
    towers_of_hanoi(n - 1);
}
```

Solution

```
void towers_of_hanoi(int n, char from_rod, char to_rod, char aux_rod) {
    if (n == 0)  return;
    towers_of_hanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk " << n << " from rod " << from_rod
         << " to rod " << to_rod << endl;
    towers_of_hanoi(n - 1, aux_rod, to_rod, from_rod);
}
```

Exercise: Merge Sort

2	3	5	7

1	4	6	8

- The basic idea is
  - Merge sort the first half
  - Merge sort the second half
  - Merge left and right (how?)

```
void merge_sort(vector<int>& vec)

void merge(const vector<int>&left, const vector<int>& right,
vector<int>& vec)
```

Solution

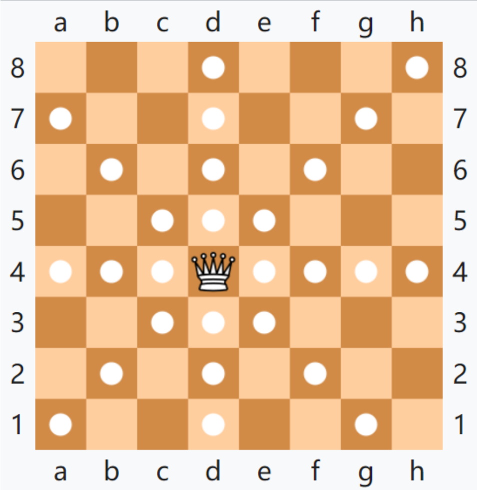
```
void merge_sort(vector<int>& vec) {
    if (vec.size() <= 1) return;
    int mid = vec.size() / 2;
    vector<int> left;
    vector<int> right;
    for (auto i = 0; i < mid; i++)
        left.push_back(vec[i]);
    for (auto i = mid; i < vec.size(); i++)
        right.push_back(vec[i]);
    merge_sort(left);
    merge_sort(right);
    merge(left, right, vec);
}
```

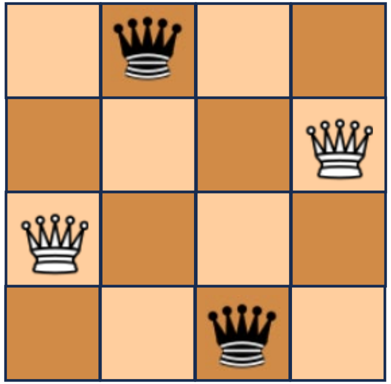
Solution

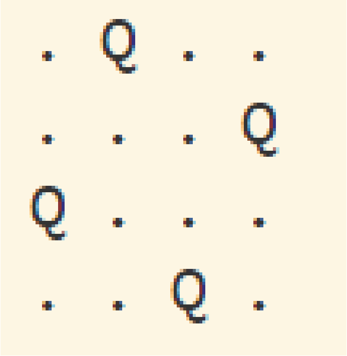
```
void merge(const vector<int>& left, const vector<int>& right,
vector<int>& vec) {
    int i = 0, j = 0, k = 0;;
    while (i < left.size() && j < right.size()) {
        if (left.at(i) < right.at(j)) {
            vec.at(k) = left.at(i);
            ++i; ++k;
        } else {
            vec.at(k) = right.at(j);
            ++j; ++k;
        }
    }
    while (i < left.size()) {vec.at(k) = left.at(i); ++i; ++k;}
    while (j < right.size()) {vec.at(k) = right.at(j); ++j; ++k;}
}
```

Exercise: Backtracking in Recursion

- N-queen problem
  - The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other
  - For example, the following is a solution for the 4 Queen problem







```
. Q . .
. . . Q
Q . . .
. . Q .
```

Solution

- We can use a `vector<vector<int>>&` to store the board

```
bool is_safe(const vector<vector<int>>& board, int row, int col)
// Is it safe to place a queen on board.at(row).at(col)?
```

```
bool solve_NQ(vector<vector<int>>& board, int col)
// we have finished col columns, now check the col-th column
```

Solution

```
void solve_NQ(vector<vector<int>>& board, int col) {
    if (col == board.size()) {
        print(board);
        return;
    }
    for (int i = 0; i < board.size(); i++) {
        if (is_safe(board, i, col)) {
            board[i][col] = 1;
            solve_NQ(board, col + 1);
            board[i][col] = 0; // backtrack
        }
    }
}
```

Solution

```
                    void print(vector<vector<int>>& board) {
                        int n = board.size();
                        for (int i = 0; i < n; i++) {
                            for (int j = 0; j < n; j++)
                                if(board[i][j]) cout << "Q ";
                                else cout << ". ";
                            cout << endl;
                        }
                        cout << "--------------------" << endl;}
bool is_safe(vector<vector<int>>& board, int row, int col) {
    for (int i = 0; i < col; i++) if (board[row][i]) return false;
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j]) return false;
    for (int i = row, j = col; j >= 0 && i < board.size(); i++, j--)
        if (board[i][j]) return false;
    return true;
}
```

Next Week

- Class and Structure
- **Happy National Day Holiday!**

If you have any question or feedback…

Anonymous questionnaire

https://www.wjx.cn/vm/OPiwiXj.aspx

Thanks