CMPE 180-92

Data Structures and Algorithms in C++

October 26 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

Assignment #9: Suggested Solution

- C++ is a very powerful but complicated language.
- It performs operations “behind your back”.
  - Example: Calls to constructors, copy constructors, and destructors.
- If you are not careful, vectors operations can generate much overhead.
- Small tweaks to vector code can significantly reduce the overhead.

Function Parameters

- You can pass a function as a parameter to another function.
- The function that you called (the one that received a function parameter) can then call the function that it was passed.
  - The formal parameter for the passed function must have the function signature.
  - Example:

```
long timed_test(SortedVector& sv, const int size,
                void f(SortedVector& sv, const int size));
```

Function Parameters, *cont'd*

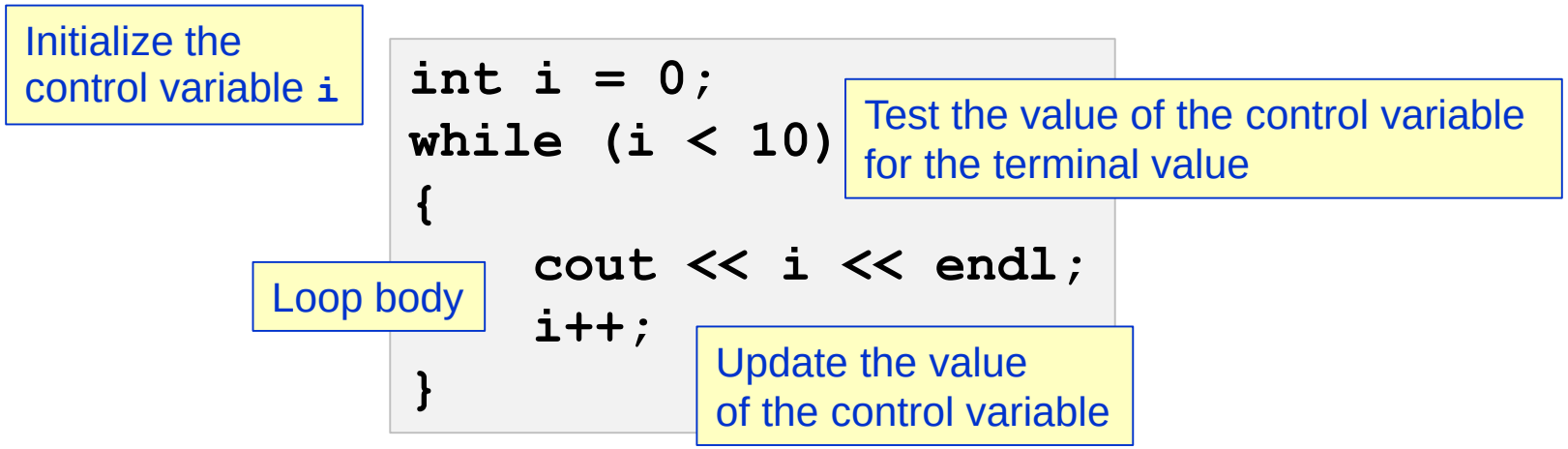
```
long timed_test(SortedVector& sv, const int size,
                void f(SortedVector& sv, const int size));
```

- In this example, function **`timed_test`** can call the function that it was passed:

```
f(sv, size);
```

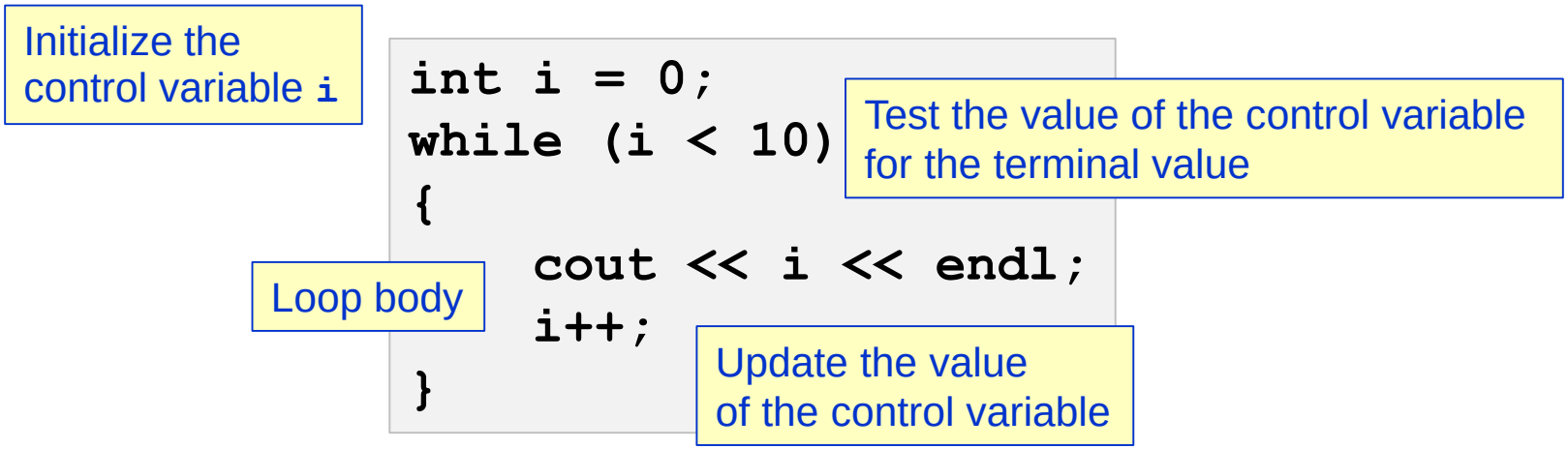
A Review of Iteration (Looping)

- A basic loop example:



- The loop body must update the value of the control variable.
- Each update brings the control variable's value closer to the terminal value.

A Review of Iteration (Looping), *cont'd*



- This review of iteration is a way to introduce the concept of recursion.

Recursion

A new way to think???

- Recursion requires a whole new way of thinking.
- Recursion is a required skill for all programmers.

How to Think Recursively

- Does this problem contain a simpler but similar case of the problem?
- Can I solve the overall problem if I can solve the simpler case?
- Is there a simplest case that has an immediate and obvious solution?
  - The simplest case is called the base case.
  - There may be more than one base case.

Some Initial Examples of Recursion

- We'll start with some examples of simple problems that we'll solve using recursion.
- Recursion is not the best way to solve these problems.
- But these simple problems help us to understand how to use recursion.

Factorials: The Classic Recursion Problem

- 5! = 5 x 4 x 3 x 2 x 1
  = 5 x 4!
- Therefore, we can solve 5! if we can solve 4!
  - 4! is a simpler but similar case of the problem.
- We can solve 4! = 4 x 3! if we can solve 3!
- We can solve 3! = 3 x 2! if we can solve 2!
- We can solve 2! = 2 x 1! if we can solve 1!
- But by definition, 1! = 1

Factorials, *cont'd*

- But by definition, 1! = 1
  - That's the simplest case (base case) with an immediate and obvious solution.
- Therefore, 2! = 2 x 1! = 2 x 1 = 2
- Therefore, 3! = 3 x 2! = 3 x 2 = 6
- Therefore, 4! = 4 x 3! = 4 x 6 = 24
- Therefore, 5! = 5 x 4! = 5 x 24 = 120

Factorials, *cont'd*

- Solve $n!$ recursively:
- What's the base case?
  - 1! = 1
- What's the simpler but similar case?
  - $(n\text{-}1)!$
  - Note that n-1 is closer to the base case of 1.

Factorial.cpp

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

Reaching the base case stops the recursion and returns an immediate value.

Recursive Multiplication

- Solve i x j recursively.
- Base cases:
  - i equals 0: product = 0
  - i equals 1: product = j
- Simpler but similar case:
  - If we can solve the problem for i-1 (which is closer to 0 and 1), then i x j is j + [(i-1) x j]

Recursive Multiplication, *cont'd*

Multiply.cpp

```
long multiply(int i, int j)
{
    switch (i)
    {
        case 0:  return 0;
        case 1:  return j;
        default: return j + multiply(i-1, j);
    }
}
```

Iterative Fibonacci

- Fibonacci sequence: 1 1 2 3 5 8 13 21 34 55
  - $f_n = f_{n-2} + f_{n-1}$
    $f_1 = 1$
    $f_2 = 1$
- An iterative solution:

Fibonacci1.cpp

```
long fibonacci(int n)
{
    if (n <= 2) return 1;
    else {
        long older = 1;
        long old   = 1;
        long next  = 1;

        for (int i = 3; i <= n; i++) {
            next = older + old;
            older = old;
            old   = next;
        }

        return next;
    }
}
```

Recursive Fibonacci

- According to the definition:
  - $f_n = f_{n-2} + f_{n-1}$
    $f_1 = 1$
    $f_2 = 1$

Fibonacci2.cpp

```
long fibonacci(int n)
{
    if (n <= 2) return 1;
    else        return fibonacci(n-2) + fibonacci(n-1);
}
```

Recursive Fibonacci, *cont'd*

- Why does the recursive solution take a long time when n is large?
- Let's trace the recursive calls:

Fibonacci3.cpp

```
long fibonacci(int n)
{
    cout << "Called fibonacci(" << n << ")" << endl;

    long f;
    if (n <= 2) f = 1;
    else        f = fibonacci(n-2) + fibonacci(n-1);

    cout << "Returning fibonacci(" << n << ") = " << f << endl;
    return f;
}
```

Recursive Fibonacci, *cont'd*

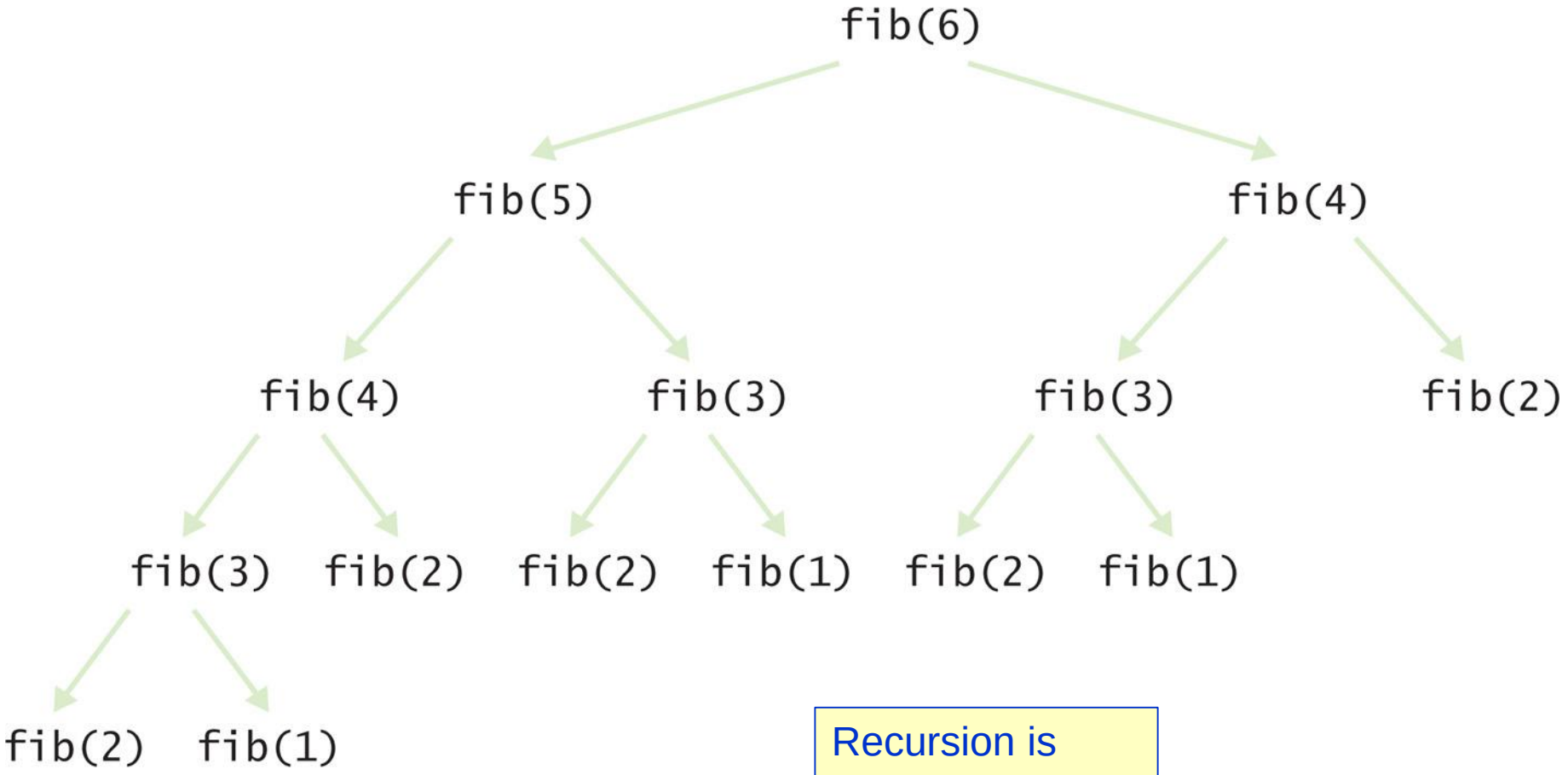


Figure 13-2

Recursion is not always good!

Member of

- Given a linked list of n integers, is x in the list?
- Base case
  - The list is empty: x is not in the list.
- Simpler but similar case:
  - Either x is equal to the first element in the list, or x is in the rest of the list.
  - The rest of the list is one element shorter, so it's closer to the base case.

Member of, *cont'd*

```
bool member_of(const int value, const list<int>& alist,
               list<int>::const_iterator it)
{
    if (it == alist.end()) return false;

    return (*it == value) || member_of(value, alist, ++it);
}
```

Member.java

Unique

- Given a list of n integers in a list, remove all the duplicate values so that what remains is a list of unique values.
- Base case
  - The list is empty or it contains only one value: Just return the list (it's empty or it has a single unique value).
- Simpler but similar case:
  - Take out the first value. Make the rest of the list unique. Then if the value we took out is not in the rest of the list, put it back. Otherwise, leave it out.

Unique, *cont'd*

Unique.cpp

```
void unique(list<int>& alist)
{
    if (alist.size() <= 1) return;

    int first = alist.front();
    alist.erase(alist.begin());  // remove the first element

    unique(alist);  // make the rest of the list unique

    if (!member_of(first, alist, alist.begin()))
    {
        alist.push_front(first);  // put back the first element
    }
}
```

Reverse

- Reverse the values of a list of n integers.
- Base case
  - The list is empty or it contains only one value: Just return the list.
- Simpler but similar case:
  - Take out the first value of the list. Reverse the rest of the list. Append the removed value to the end of the reversed rest of the list.

Reverse, *cont'd*

Reverse.java

```
void reverse(list<int>& alist)
{
    if (alist.size() <= 1) return;

    int first = alist.front();
    alist.erase(alist.begin());  // remove the first element

    reverse(alist);              // reverse the rest of the list
    alist.push_back(first);  // append the first element at the end
}
```

Quiz

- Quiz 6 – 2017 Apr 6

Break

Better Recursion Problems

- Some problems are a natural fit for recursion.
- They are much more easily solved with recursion than without.

Word Permutations

- Given a word, generate all the permutations of the letters of the word.
  - Example: “eat”: eat eta aet ate tea tae
- Base cases
  - Empty word: No permutations.
  - Single-letter word: The only permutation is the letter.
- Simpler but similar case:
  - Remove a letter and generate the permutations of the word that’s one letter shorter.
  - Prepend the removed letter to each permutation.

Word Permutations, *cont'd*

```
vector<string> generate_permutations(string word)
{
    vector<string> permutations;

    // Base case: Return an empty vector.
    if (word.length() == 0) return permutations;

    // Base case: Return a vector with a one-letter word.
    if (word.length() == 1)
    {
        permutations.push_back(word);
        return permutations;
    }
```

Word Permutations, *cont'd*

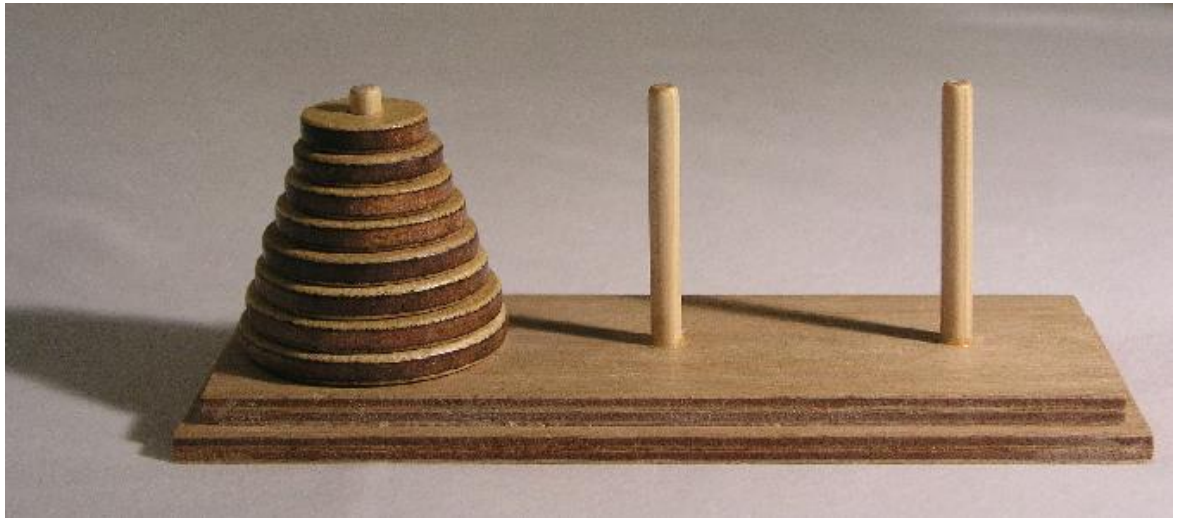
```
    // Simpler but similar case.
    else
    {
        for (int i = 0; i < word.length(); i++)
        {
            char removed_letter = word[i];
            string shorter_word = word.substr(0, i) + word.substr(i + 1);

            vector<string> shorter_permutations =
                                generate_permutations(shorter_word);

            for (string shorter_perm : shorter_permutations)
            {
                permutations.push_back(removed_letter + shorter_perm);
            }
        }

        return permutations;
    }
}
```

Towers of Hanoi



- **Goal:** Move the stack of disks from the source pin to the destination pin.
  - You can move only one disk at a time.
  - You cannot put a larger disk on top of a smaller disk.
  - Use the third pin for temporary disk storage.

Animation: http://towersofhanoi.info/Animate.aspx

Towers of Hanoi, *cont'd*

- Label the pins A, B, and C. Initial roles:
  - A: source
  - B: destination
  - C: temporary

During recursive calls, the pins will assume different roles.

- Base case: n = 1 disk
  - Move disk from A to B *(source ➔ destination)*
- Simpler but similar case: n-1 disks
  - Solve for n-1 disks: A to C *(source ➔ temp)*
  - Move 1 disk from A to B *(source ➔ destination)*
  - Solve for n-1 disks: C to B *(temp ➔ destination)*

Towers of Hanoi, *cont'd*

Hanoi.cpp

```
enum Pin { A = 'A', B = 'B', C = 'C' };

void solve(const int n, const Pin source, const Pin dest, const Pin temp);
void move(const Pin from, const Pin to);

int main()
{
    int n;
    cout << "Number of disks? ";
    cin >> n;

    cout << "Solve for " << n << " disks:" << endl << endl;
    solve(n, Pin::A, Pin::B, Pin::C);
}

void move(Pin from, Pin to)
{
    cout << "Move disk from " << static_cast<char>(from)
         << " to " << static_cast<char>(to) << endl;
}
```

Towers of Hanoi, *cont'd*

- Solve n disks (source = A, destination = B)
  - Solve for n-1 disks: A to C *(source ➔ temp)*
  - Move 1 disk from A to B *(source ➔ destination)*
  - Solve for n-1 disks: C to B *(temp ➔ destination)*

```
void solve(const int n, const Pin source, const Pin dest, const Pin temp)
{
    if (n > 0)
    {
        solve(n-1, source, temp, dest);  // solve source ==> temp
        move(source, dest);              // move 1 disk source ==> dest
        solve(n-1, temp, dest, source);  // solve temp ==> dest
    }
}
```

Hanoi.cpp

Linear Search

- Search for a target value in an array of *n* elements.
  - The array is not sorted in any way.
- What choices do we have?
  - Look at all the elements one at a time.
- On average, you have to examine half of the array.

Binary Search

- Now assume the array is sorted.
  - Smallest value to largest value.
- First check the middle element.
- Is the target value smaller than the middle element?
  - If so, search the first half of the array.
- Is the target value larger than the middle element?
  - If so, search the second half of the array.

Binary Search, *cont'd*

- The binary search keeps cutting in half the part of the array it's searching.
  - Next search either the first half or the second half.
  - Eventually, you'll either find the target value, or conclude that the value is not in the array.
- The order of growth of the number of steps in a binary search is expressed $O(\log_2 n)$ Big-O notation
  - To search 1000 elements, it takes < 10 steps.
  - Computer science logarithms are by default base 2.

Iterative Binary Search

- It's easy to write an iterative binary search:

```
int search(int value, vector<int> v, int low, int high)
{
    while (low <= high) {
        int mid = (low + high)/2;

        if (value == v[mid]) {
            return mid;
        }
        else if (value < v[mid]) {
            high = mid-1;
        }
        else {
            low = mid+1;
        }
    }

    return -1;
}
```

Get the midpoint of the subrange.

Found the target value?

Search the first half next.

Search the second half next.

The target value is not in the array.

BinarySearchIterative.cpp

Elegant Recursion

- Al though an iterative solution may be more efficient, sometimes a recursive solution is more elegant.

Recursive Binary Search

- A binary search can be done recursively:

```
int search(int value, vector<int> v, int low, int high)
{
    if (low <= high) {
        int mid = (low + high)/2;

        if (value == v[mid]) {
            return mid;
        }
        else if (value < v[mid]) {
            return search(value, v, low, mid-1);
        }
        else {
            return search(value, v, mid+1, high);
        }
    }

    return -1;
}
```

Get the midpoint of the subrange.

Found the target value?

Search the first half.

Search the second half.

The target value is not in the subrange.

BinarySearchRecursive.cpp

Mutual Recursion

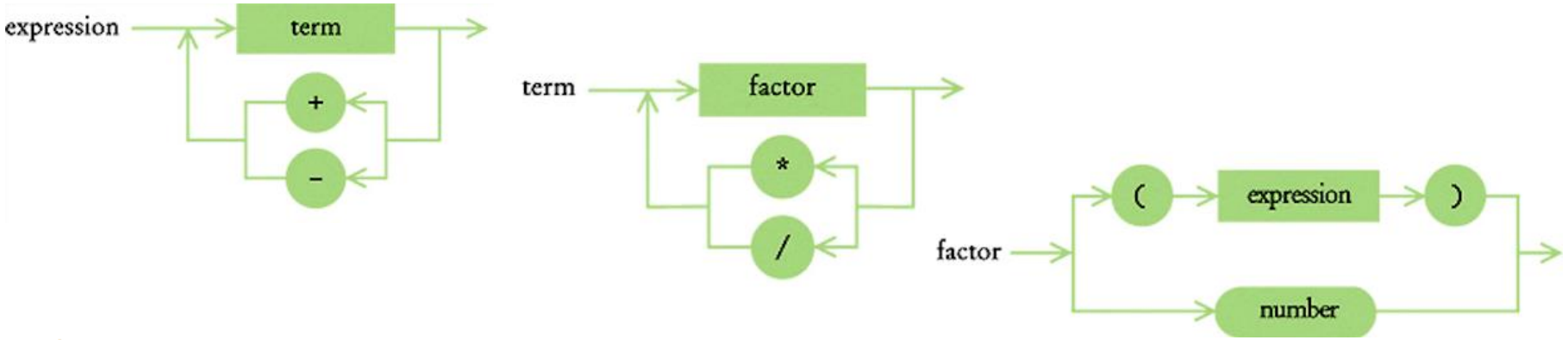
- Mutual recursion occurs when a set of cooperating functions call each other.
  - Example:
    `func_a` → `func_b` → `func_c` → `func_a`

- Suppose we want to write a program that can read and evaluate arithmetic expressions:
  - Example:

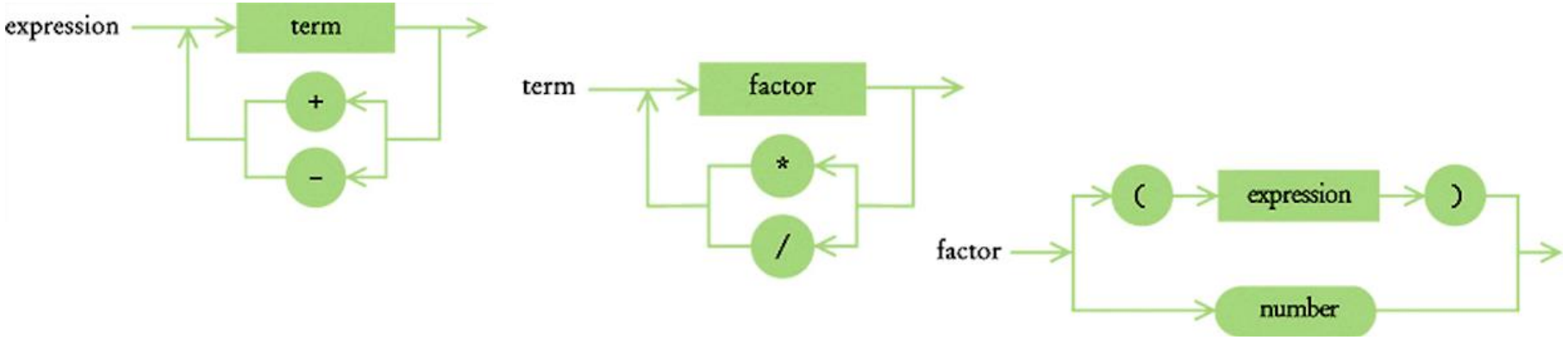
```
Expression? (4*(8 - (2*3 - 1)))/6 =
2
```

Mutual Recursion, *cont'd*

- The following syntax diagrams specify that
  - An expression is composed of one or more terms separated by the operators + or –
  - A term is composed of one or more factors separated by the operator * or /
  - A factor is either a number or a parenthesized expression.



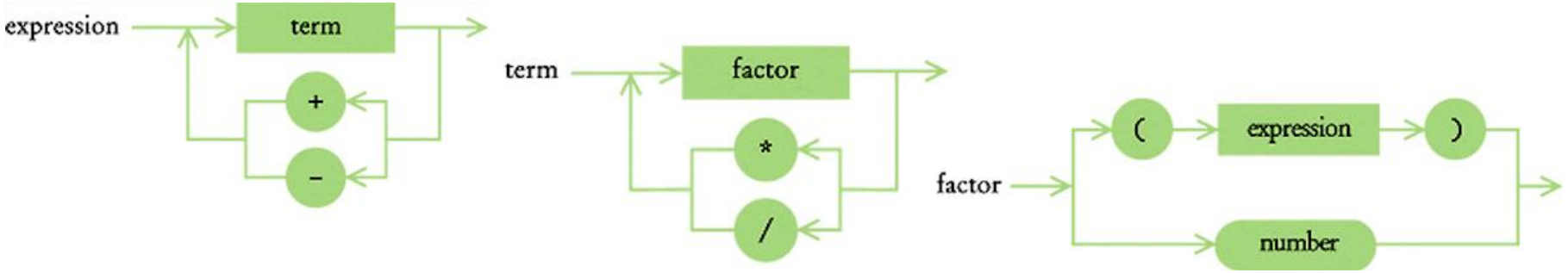
Mutual Recursion, *cont'd*



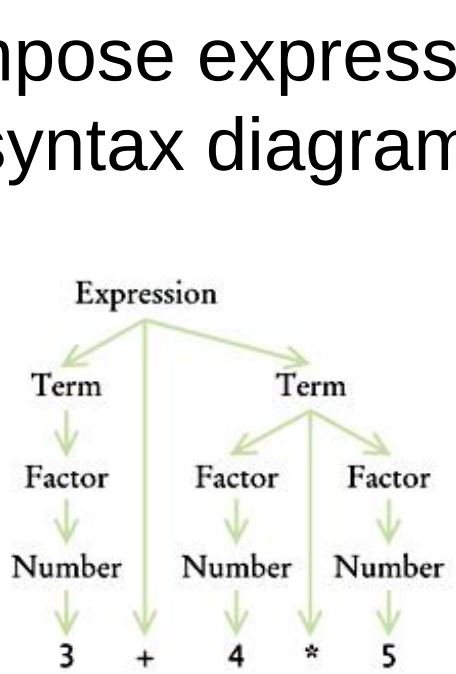
- If we write functions **`expression`**, **`term`**, and **`factor`**, they will be mutually recursive:

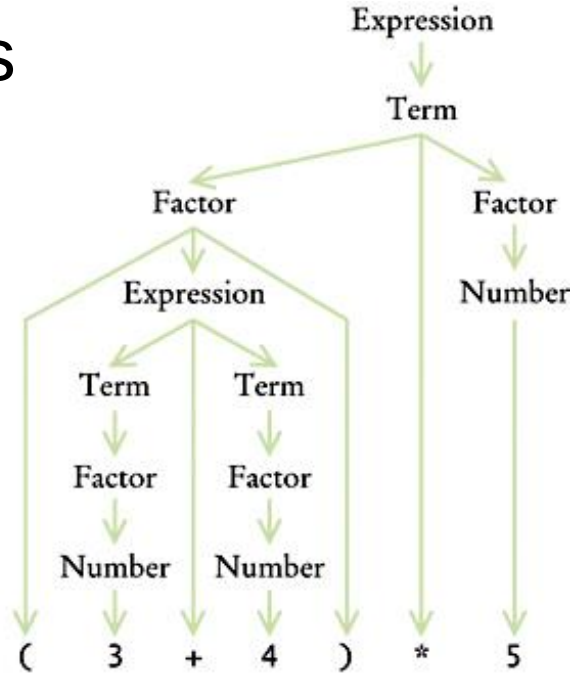
`expression` → **`term`** → **`factor`** → **`expression`**

Mutual Recursion, *cont'd*



- We can decompose expressions based on the syntax diagrams:





Note how the syntax diagrams determine the operator precedence.

Assignment #10. Calculator

- Write a program that repeatedly prompts you to type an arithmetic expression.
  - Use the diagrams to determine the syntax.
  - A number should be a double in any valid form.
  - End each expression with a equal sign.
  - Ignore all blanks.
  - Evaluate and print the value of each expression.
- An input line containing only a period should terminate the program.

Assignment #10. Calculator, *cont'd*

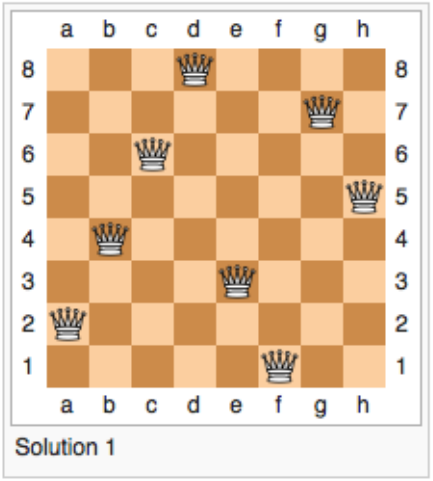
- Perform basic syntax error checking.
  - Unexpected input characters.
  - Unbalanced parentheses.
  - Missing = at the end of an expression.
- And runtime error checking.
  - Division by zero.
- Stop evaluating upon encountering an error.
  - Output an appropriate error message.
  - Prompt for the next expression.

Assignment #10. Calculator, *cont'd*

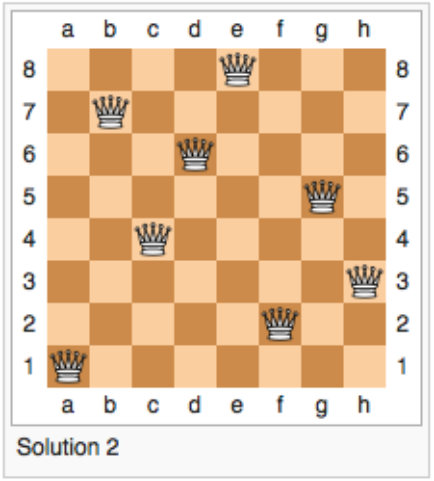
- The assignment write-up and sample output will be available shortly.
- Due Thursday, April 13 at 5:30 PM.

The Queens Problem

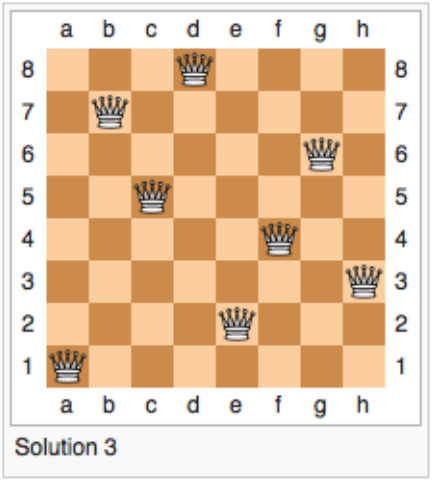
- Place queens on a chessboard such that no queen can attack another.
- Solve with recursion and backtracking.



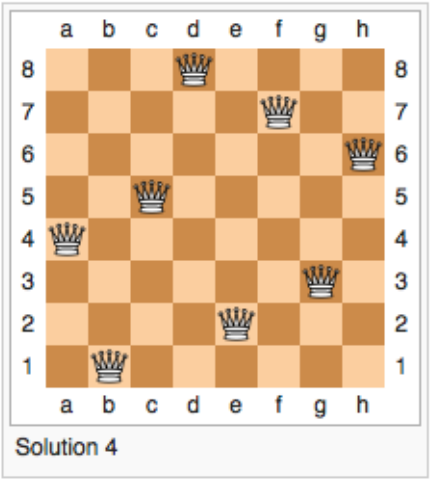
Solution 1



Solution 2



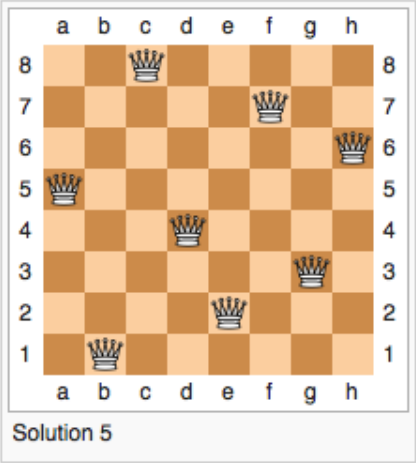
Solution 3

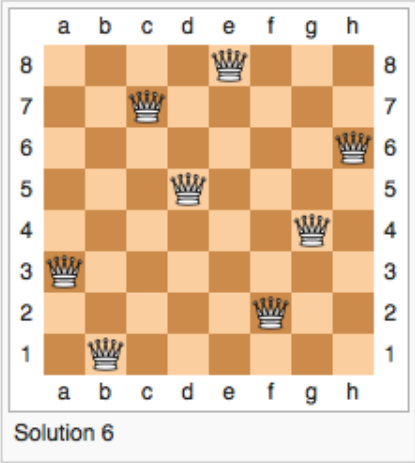


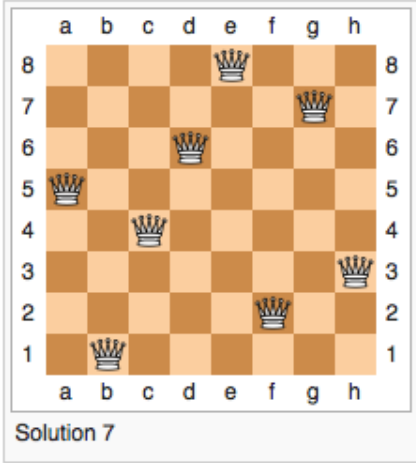
Solution 4

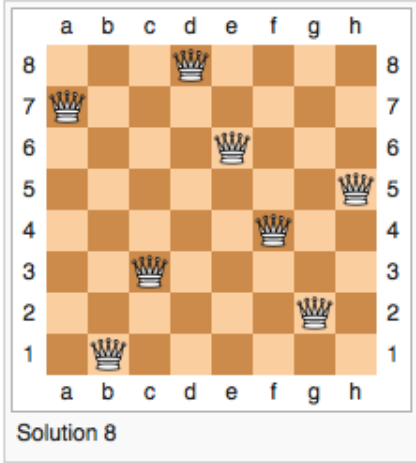
https://en.wikipedia.org/wiki/Eight_queens_puzzle

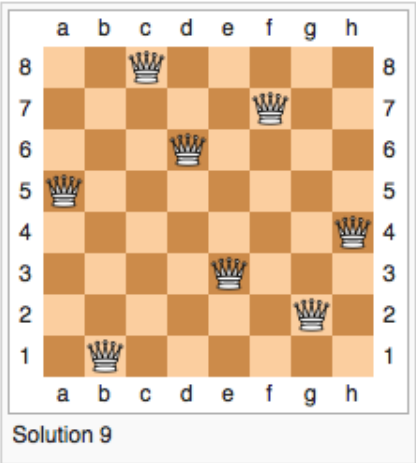
The Queens Problem, *cont'd*

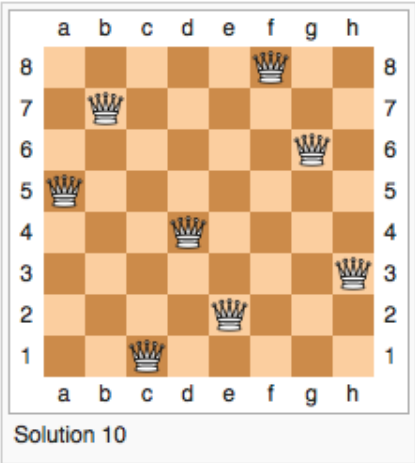

Solution 5

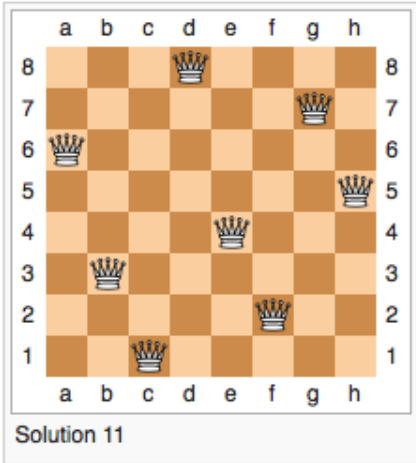

Solution 6

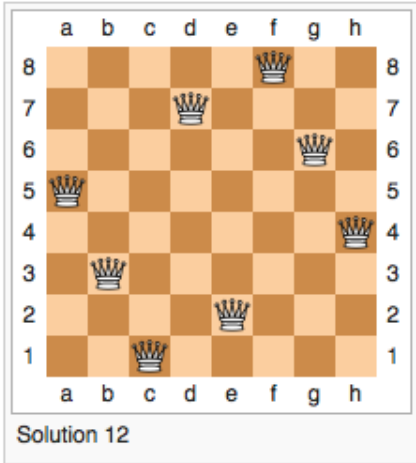

Solution 7


Solution 8


Solution 9


Solution 10

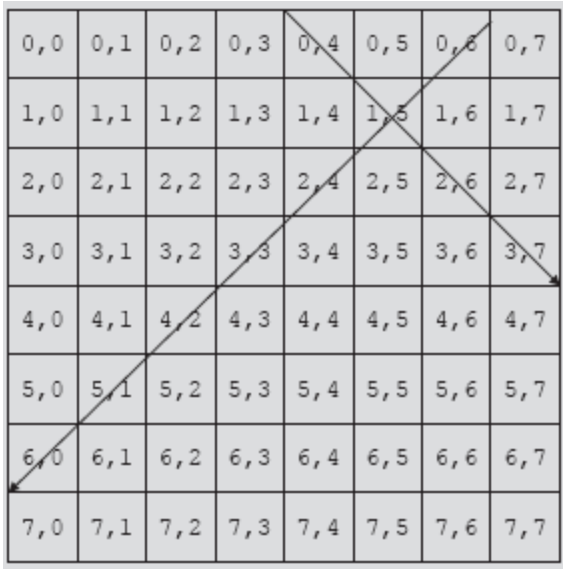

Solution 11


Solution 12

https://en.wikipedia.org/wiki/Eight_queens_puzzle

Safe Queen Positions

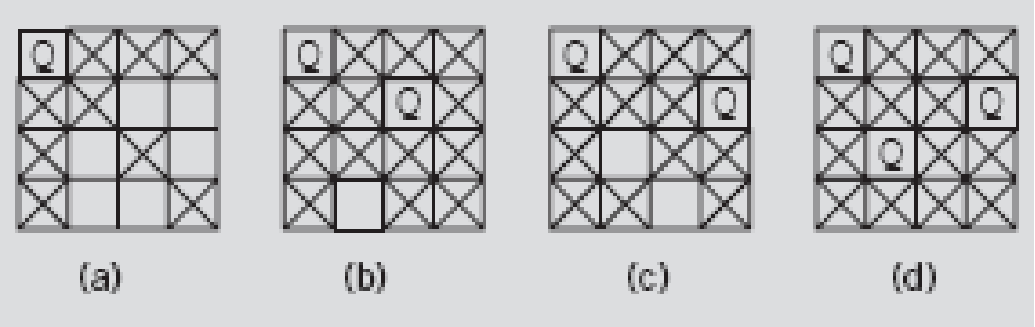
- For each queen, check that it isn't in the same row, column, or diagonal as another queen.



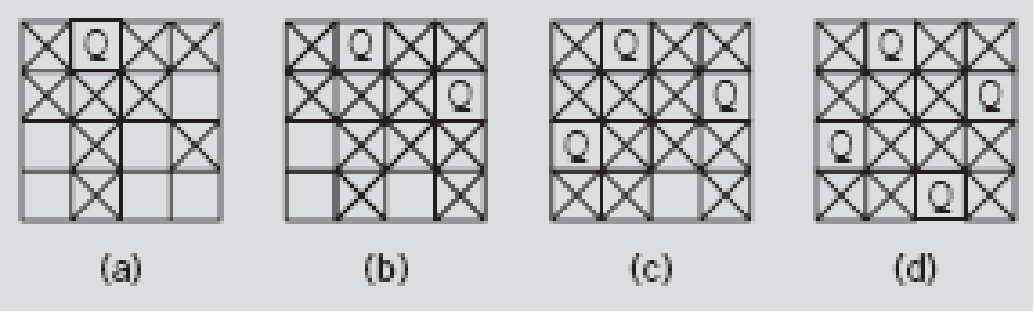
Data Structures Using C++, 2nd ed.
by D.S. Malik
Cengage Learning, 2010

Backtracking

- Backtrack when reaching a dead end.
  - Example dead end with 4 queens:



  - Example solution with 4 queens:



(1,3,0,2)

Data Structures Using C++, 2nd ed.
by D.S. Malik
Cengage Learning, 2010

Backtracking Algorithm

- Find problem solutions by constructing partial solutions using recursion.
- Ensure that a partial solution does not violate requirements.
- Extend a partial solution toward completion.
- If a partial solution does not lead to a solution (i.e., it's a dead end):
  - Back up
  - Remove most recently added part
  - Try other possibilities

See Malik, pp. 376-383.