DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 5: Recursion

Objectives

Looking ahead – in this chapter, we'll consider

- Recursive Definitions
- Function Calls and Recursive Implementation
- Anatomy of a Recursive Call
- Tail Recursion
- Nontail Recursion
- Indirect Recursion
- Nested Recursion
- Excessive Recursion
- Backtracking

Recursive Definitions

- It is a basic rule in defining new ideas that they not be defined circularly
- However, it turns out that many programming constructs are defined in terms of themselves
- Fortunately, the formal basis for these definitions is such that no violations of the rules occurs
- These definitions are called ***recursive definitions*** and are often used to define infinite sets
- This is because an exhaustive enumeration of such a set is impossible, so some others means to define it is needed

Recursive Definitions (continued)

- There are two parts to a recursive definition
  - The ***anchor*** or ***ground case*** (also sometimes called the ***base case***) which establishes the basis for all the other elements of the set
  - The ***inductive clause*** which establishes rules for the creation of new elements in the set
- Skipping book example of factorial

Function Calls and Recursive Implementation

- What kind of information must we keep track of when a function is called?
- If the function has parameters, they need to be initialized to their corresponding arguments
- In addition, we need to know where to resume the calling function once the called function is complete
- Since functions can be called from other functions, we also need to keep track of local variables for scope purposes
- Because we may not know in advance how many calls will occur, a stack is a more efficient location to save information

Function Calls and Recursive Implementation (continued)

- So we can characterize the state of a function by a set of information, called an ***activation record*** or ***stack frame***
- This is stored on the runtime stack, and contains the following information:
  - Values of the function's parameters, addresses of reference variables (including arrays)
  - Copies of local variables
  - The return address of the calling function
  - A dynamic link to the calling function's activation record
  - The function's return value if it is not `void`
- Every time a function is called, its activation record is created and placed on the runtime stack

Function Calls and Recursive Implementation (continued)

- So the runtime stack always contains the current state of the function
- As an illustration, consider a function `f1()` called from `main()`
- It in turn calls function `f2()`, which calls function `f3()`
- The current state of the stack, with function `f3()` executing, is shown in Figure 5.1
- Once `f3()` completes, its record is popped, and function `f2()` can resume and access information in its record
- If `f3()` calls another function, the new function has its activation record pushed onto the stack as `f3()` is suspended

Function Calls and Recursive Implementation (continued)

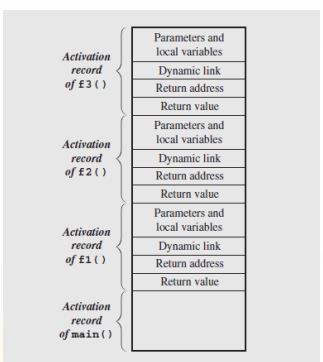


Fig. 5.1 Contents of the run-time stack when `main()` calls function `f1()`, `f1()` calls `f2()`, and `f2()` calls `f3()`

Function Calls and Recursive Implementation (continued)

- The use of activation records on the runtime stack allows recursion to be implemented and handled correctly
- Essentially, when a function calls itself recursively, it pushes a new activation record of itself on the stack
- This suspends the calling instance of the function, and allows the new activation to carry on the process
- Thus, a recursive call creates a series of activation records for different instances of the **same** function

- Skipping example from book of recursive x^n

Tail Recursion

- The nature of a recursive definition is that the function contains a reference to itself
- This reference can take on a number of different forms
- We're going to consider a few of these, starting with the simplest, ***tail recursion***
- The characteristic implementation of tail recursion is that a single recursive call occurs at the end of the function
- No other statements follow the recursive call, and there are no other recursive calls prior to the call at the end of the function

Tail Recursion (continued)

- An example of a tail recursive function is the code:

```
void tail(int i) {
   if (i > 0) {
      cout << i << '';
      tail(i-1);
   }
}
```

- Essentially, tail recursion is a loop; it can be replaced by an iterative algorithm to accomplish the same task
- In fact, in most cases, languages supporting loops should use this construct rather than recursion

Tail Recursion (continued)

- An example of an iterative form of the function is shown below:

```
void iterativeEquivalentOfTail(int i) {
   for ( ; i > 0; i--)
      cout << i << '';
}
```

- Another example from the book is a Koch snowflake https://www.openprocessing.org/sketch/147104

Nontail Recursion

- As an example of another type of recursion, consider the following code from page 178 of the text:

```
/* 200 */   void reverse() {
                char ch;
/* 201 */       cin.get(ch);
/* 202 */       if (ch != '\n') {
/* 203 */          reverse();
/* 204 */          cout.put(ch);
                }
            }
```

- The function displays a line of input in reverse order

Nontail Recursion (continued)

- To accomplish this, the function `reverse()` uses recursion to repeatedly call itself for each character in the input line
- Assuming the input "ABC", the first time `reverse()` is called an activation record is created to store the local variable `ch` and the return address of the call in `main()`
- This is shown in Figure 5.3a
- The `get()` function (line 201) reads in the character "A" from the line and compares it with the end-of-line character
- Since they aren't equal, the function calls itself in line 203, creating a new activation record shown in Figure 5.3b

Nontail Recursion (continued)

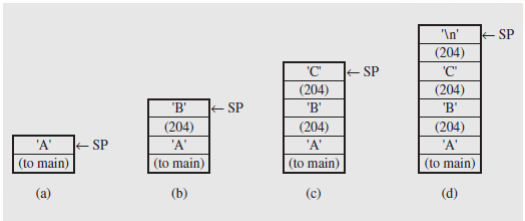


Fig. 5.3 Changes on the run-time stack during the execution of `reverse()`

- This process continues until the end of line character is read and the stack appears as in Figure 5.3d
- At this point, the current call terminates, popping the last activation record off the stack, and resuming the previous call at line 204

Nontail Recursion (continued)

- This line outputs the current value of `ch`, which is contained in the current activation record, and is the value “C”
- The current call then ends, causing a repeat of the pop and return action, and once again the output is executed displaying the character “B”
- Finally, the original call to `reverse()` is reached, which will output the character “A”
- At that point, control is returned to `main()`, and the string “CBA” will be displayed
- This type of recursion, where the recursive call precedes other code in the function, is called ***nontail recursion***

Nontail Recursion (continued)

- Now consider a non-recursive version of the same algorithm:

```
void simpleIterativeReverse() {
   char stack[80];
   register int top = 0;
   cin.getline(stack,80);
   for(top = strlen(stack)-1; top >= 0;
       cout.put(stack[top--]));
}
```

- This version utilizes functions from the standard C++ library to handle the input and string reversal
- So the details of the process are hidden by the functions

Nontail Recursion (continued)

- If these functions weren't available, we'd have to make the processing more explicit, as in the following code:

```
void iterativeReverse() {
   char stack[80];
   register int top = 0;
   cin.get(stack[top]);
   while(stack[top]!='\n')
      cin.get(stack[++top]);
   for (top -= 2; top >= 0;
        cout.put(stack[top--]));
}
```

Nontail Recursion (continued)

- These iterative versions illustrate several important points
- First, when implementing nontail recursion iteratively, the stack must be explicitly implemented and handled
- Second, the clarity of the algorithm, and often its brevity, are sacrificed as a consequence of the conversion
- Therefore, unless compelling reasons are evident during the algorithm design, we typically use the recursive versions

Indirect Recursion

- The previous discussions have focused on situations where a function, `f()`, invokes itself recursively (***direct recursion***)
- However, in some situations, the function may be called not by itself, but by a function that it calls, forming a chain:

$$f() \rightarrow g() \rightarrow f()$$

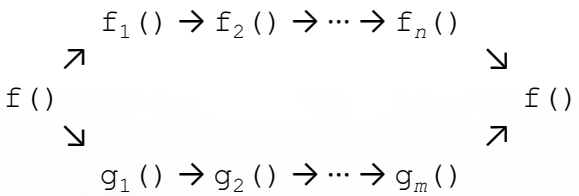
- This is known as ***indirect recursion***
- The chains of calls may be of arbitrary length, such as:

$$f() \rightarrow f_1() \rightarrow f_2() \rightarrow \cdots \rightarrow f_n() \rightarrow f()$$

- It is also possible that a given function may be a part of multiple chains, based on different calls

Indirect Recursion (continued)

- So our previous chain might look like:



Nested Recursion

- Another interesting type of recursion occurs when a function calls itself and is also one of the parameters of the call
- Consider the example shown on page 186 of the text:

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ n & \text{if } n > 4 \\ h(2 + h(n)) & \text{if } n \leq 4 \end{cases}$$

- From this definition, it is given that the function has solutions for $n = 0$ and $n > 4$
- However, for $n = 1$, 2, 3, and 4, the function determines a value based on a recursive call that requires evaluating itself
- This is called ***nested recursion***

Nested Recursion (continued)

- Another example is the famous (or perhaps infamous) *Ackermann function*, defined as:

$$A(m,n)=\begin{cases} m+1 & \text{if } n=0 \\ A(n-1,1) & \text{if } n>0, m=0 \\ A(n-1,A(n,m-1)) & \text{otherwise} \end{cases}$$

- First suggested by Wilhelm Ackermann in 1928 and later modified by Rosza Peter, it is characterized by its rapid growth
- To get a sense of this growth, consider that $A(3,m) = 2^{m+3} - 3$, and

$$A(4,m)=2^{2^{2^{\cdot^{\cdot^{\cdot^{2^{16}}}}}}}-3$$

Nested Recursion (continued)

- There are $(m - 1)$ 2s in the exponent, so $A(4,1) = 2^{16} - 3$, which is 65533
- However, changing m to 2 has a dramatic impact as the value of $A(4,2)=2^{2^{16}}-3=2^{65536}-3$ has 19,729 digits in its expansion
- As can be inferred from this, the function is elegantly expressed recursively, but quite a problem to define iteratively

Excessive Recursion

- Recursive algorithms tend to exhibit simplicity in their implementation and are typically easy to read and follow
- However, this straightforwardness does have some drawbacks
- Generally, as the number of function calls increases, a program suffers from some performance decrease
- Also, the amount of stack space required increases dramatically with the amount of recursion that occurs
- This can lead to program crashes if the stack runs out of memory
- More frequently, though, is the increased execution time leading to poor program performance

Excessive Recursion (continued)

- As an example of this, consider the Fibonacci numbers
- They are first mentioned in connection with Sanskrit poetry as far back as 200 BCE
- Leonardo Pisano Bigollo (also known as Fibonacci), introduced them to the western world in his book *Liber Abaci* in 1202 CE
- The first few terms of the sequence are 0, 1, 1, 2, 3, 5, 8, … and can be generated using the function:

$$Fib(n) = \begin{cases} n & \text{if } n < 2 \\ Fib(n-2) + Fib(n-1) & \text{otherwise} \end{cases}$$

Excessive Recursion (continued)

- This tells us that that any Fibonacci number after the first two (0 and 1) is defined as the sum of the two previous numbers
- However, as we move further on in the sequence, the amount of calculation necessary to generate successive terms becomes excessive
- This is because every calculation ultimately has to rely on the base case for computing the values, since no intermediate values are remembered
- The following algorithm implements this definition; again, notice the simplicity of the code that belies the underlying inefficiency

Excessive Recursion (continued)

```
unsigned long Fib(unsigned long n) {
   if (n < 2)
      return n;
// else
      return Fib(n-2) + Fib(n-1);
}
```

- If we use this to compute `Fib(6)` (which is 8), the algorithm starts by calculating `Fib(4)` + `Fib(5)`
- The algorithm then needs to calculate `Fib(4)` = `Fib(2)` + `Fib(3)`, and finally the first term of that is `Fib(2)` = `Fib(0)` + `Fib(1)` = 0 + 1 = 1

Excessive Recursion (continued)

- The entire process can be represented using a tree to show the calculations:

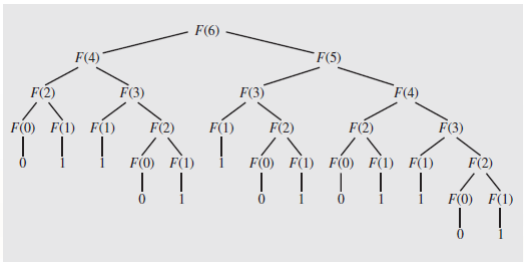


Fig. 5.8 The tree of calls for Fib(6).

- Counting the branches, it takes 25 calls to `Fib()` to calculate `Fib(6)`

Excessive Recursion (continued)

- The total number of additions required to calculate the n^{th} number can be shown to be Fib(n + 1) – 1
- With two calls per addition, and the first call taken into account, the total number of calls is 2 · Fib(n + 1) – 1
- Values of this are shown in the following table:

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

Fig. 5.9 Number of addition operations and number of recursive calls to calculate Fibonacci numbers.

Excessive Recursion (continued)

- Notice that it takes almost 3 million calls to determine the 31st Fibonacci number
- This exponential growth makes the algorithm unsuitable for anything but small values of n
- Fortunately there are acceptable iterative algorithms that can be used far more effectively

Backtracking

- ***Backtracking*** is an approach to problem solving that uses a systematic search among possible pathways to a solution
- As each path is examined, if it is determined the pathway isn't viable, it is discarded and the algorithm returns to the prior branch so that a different path can be explored
- Thus, the algorithm must be able to return to the previous position, and ensure that all pathways are examined
- Backtracking is used in a number of applications, including artificial intelligence, compiling, and optimization problems
- One classic application of this technique is known as ***The Eight Queens Problem***

Backtracking (continued)

- In this problem, we try to place eight queens on a chessboard in such a way that no two queens attack each other

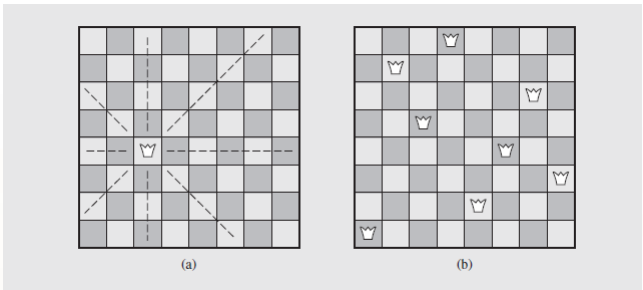


Fig. 5.11 The eight queens problem

Backtracking (continued)

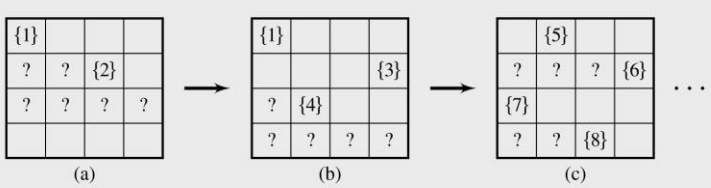
- The approach to solving this is to place one queen at a time, trying to make sure that the queens do not check each other
- If at any point a queen cannot be successfully placed, the algorithm backtracks to the placement of the previous queen
- This is then moved and the next queen is tried again
- If no successful arrangement is found, the algorithm backtracks further, adjusting the previous queen's predecessor, etc.
- A pseudocode representation of the backtracking algorithm is shown in the next slide; the process is described in detail on pages 192 – 197, along with a C++ implementation

Backtracking (continued)

```
putQueen(row)
    for every position col on the same row
      if position col is available
         place the next queen in position col;
         if (row < 8)
            putQueen(row+1);
         else success;
         remove the queen from position col;
```

- This algorithm will find all solutions, although some are symmetrical

Example on 4x4 Board



Concluding Remarks

- The foregoing discussion has provided us with some insight into the use of recursion as a programming tool
- While there are no specific rules that require we use or avoid recursion in any particular situation, we can develop some general guidelines
- For example, recursion is generally less efficient than the iterative equivalent
- However, if the difference in execution times is fairly small, other factors such as clarity, simplicity, and readability may be taken into account
- Recursion often is more faithful to the algorithm's logic

Concluding Remarks (continued)

- The task of converting recursive algorithms into their iterative equivalents can often be difficult to perform
- As we saw with nontail recursion, we frequently have to explicitly implement stack handling to handle the runtime stack processing incorporated into the recursive form
- Again, this may require analysis and judgment by the programmer to determine the best course of action
- The text suggests a couple of situations where iterative versions are preferable to recursive ones
- First, real-time systems, with their stringent time requirements, benefit by the faster response of iterative code

Concluding Remarks (continued)

- Another situation occurs in programs that are repeatedly executed, such as compilers
- However, even these cases may be changed if the hardware or operating environment of the algorithm supports processing that speeds the recursive algorithm (consider a hardware supported stack)
- Sometimes the best way to decide which version to use relies simply on coding both forms and testing them
- This is especially true in cases involving tail recursion, where the recursive version may be faster, and with nontail recursion where use a stack cannot be eliminated

Concluding Remarks (continued)

- One place where recursion must be examined carefully is when excessive, repeated calculations occur to obtain results
- The discussion of the Fibonacci sequence illustrated this concern
- Often, drawing a call tree such as figure 5.8 can be helpful
- Trees with a large number of levels can threaten stack overflow problems
- On the other hand, shallow, “bushy” trees may indicate a suitable recursive candidate, provided the number of repetitions is reasonable