Formal DSA in C++

Nathan Warner

Computer Science
Northern Illinois University
United States

Contents

Recursion

1.1 Elementary recursion

A recursive definition consists of two parts. In the first part, called the anchor or the ground case, the basic elements that are the building blocks of all other elements of the set are listed. In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed. These rules are applied again and again to generate new objects. For example, to construct the set of natural numbers, one basic element, 0, is singled out, and the operation of incrementing by 1 is given as:

1. $0 \in \mathbb{N}$
2. If $n \in \mathbb{N},\ then (n+1) \in \mathbb{N}$
3. There are no other objects in the set $\mathbb{N}$

It is more convenient to use the following definition, which encompasses the whole range of Arabic numeric heritage:

1. $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \in \mathbb{N}$
2. If $n \in \mathbb{N}$, then $n0, n1, n2, n3, n4, n5, n6, n7, n8, n9 \in \mathbb{N}$
3. These are the only natural numbers

Recursive definitions serve two purposes: generating new elements, as already indicated, and testing whether an element belongs to a set. In the case of testing, the problem is solved by reducing it to a simpler problem, and if the simpler problem is still too complex it is reduced to an even simpler problem, and so on, until it is reduced to a problem indicated in the anchor

1.1.1 Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n \neq 0 \end{cases}.$$

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

    // Expands to
    // n * n-1 * n-2 * ... * 1
}
```

1.1.2 Powers

Consider the recursive definition for a power of x

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}.$$

```cpp
constexpr int power(int x, int n) {
    if (n == 0) return 1;
    return x * power(x,n-1);
}
```

The function power() can be implemented differently, without using any recursion, as in the following loop:

```cpp
int power2(int x, int n) {
    int res = 1;

    for (res = x; n > 1; --n) {
        res*=x;
    }
    return res;
}
```

Do we gain anything by using recursion instead of a loop? The recursive version seems to be more intuitive because it is similar to the original definition of the power function. The definition is simply expressed in C++ without losing the original structure of the definition. The recursive version increases program readability, improves self-documentation, and simplifies coding. In our example, the code of the nonrecursive version is not substantially larger than in the recursive version, but for most recursive implementations, the code is shorter than it is in the nonrecursive implementations

1.2 Tail recursion

Tail recursion is a type of recursion where the recursive call is the last thing the function does before returning a result. This means there are no more computations or operations to perform after the recursive call.

Because of this, tail recursion can be optimized by some compilers or interpreters to avoid adding new frames to the call stack, making it more memory-efficient than regular recursion.

In simple terms, if a recursive function calls itself, and after that call there's nothing left to do, it's tail recursion. This allows the function to reuse the same memory space, preventing stack overflow in cases with deep recursion.

the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect. For example, the function tail() defined as

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

Is an example of a function with tail recursion, whereas the function nonTail() defined as

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

Is not. Tail recursion is simply a glorified loop and can be easily replaced by one. In this example, it is replaced by substituting a loop for the if statement and decrementing the variable i in accordance with the level of recursive call. In this way, tail() can be expressed by an iterative function:

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

Is there any advantage in using tail recursion over iteration? For languages such as C++, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight. In languages endowed with a loop or its equivalents, such as an if statement combined with a goto statement, tail recursion should not be used.

Another problem that can be implemented in recursion is printing an input line in reverse order. Here is a simple recursive implementation:

```
void reverse() {
    char ch;
    cin.get(ch);
    if (ch != '\n') {
        reverse();
        cout.put(ch);
    }
}
```

Compare the recursive implementation with a nonrecursive version of the same function:

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

functions like strlen() and getline() from the standard C++ library can be used. If we are not supplied with such functions, then our iterative function has to be implemented differently:

```
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--]));
}
```

1.3 Indirect Recursion

The preceding sections discussed only direct recursion, where a function $f()$ called itself. However, $f()$ can call itself indirectly via a chain of other calls. For example, $f()$ can call $g()$, and $g()$ can call $f()$. This is the simplest case of indirect recursion. The chain of intermediate calls can be of an arbitrary length, as in:

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

There is also the situation when $f()$ can call itself indirectly through different chains. Thus, in addition to the chain just given, another chain might also be possible. For instance

$$f() \rightarrow g_1() \rightarrow g_2() \rightarrow ... \rightarrow g_m() \rightarrow f().$$

This situation can be exemplified by three functions used for decoding information. receive() stores the incoming information in a buffer, decode() converts it into legible form, and store() stores it in a file. receive() fills the buffer and calls decode(), which in turn, after finishing its job, submits the buffer with decoded information to store(). After store() accomplishes its tasks, it calls receive() to intercept more encoded information using the same buffer. Therefore, we have the chain of calls

$$\text{recieve()} \rightarrow \text{decode()} \rightarrow \text{store()} \rightarrow \text{recieve()} \rightarrow \text{decode()} \rightarrow ....$$

1.4 Nested Recursion

A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting

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

1.5 Excessive Recursion

Logical simplicity and readability are used as an argument supporting the use of recursion. The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a nonrecursive approach. If recursion is too deep (for example, computing $5.6^{100,000}$), then we can run out of space on the stack and our program crashes. But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent

However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases

Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

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

The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence.

How can this definition be implemented in C++? It takes almost term-by-term translation to have a recursive version, which is

```cpp
constexpr unsigned long fib(int n) {
    if (n < 2) return n;
    return fib(n-2) + fib(n-1);
}
```

The function is simple and easy to understand but extremely inefficient. To see it, compute Fib(6), the seventh number of the sequence, which is 8. Based on the definition, the computation runs as follows:

$$\begin{aligned} Fib(6) &= Fib(4) + Fib(5) \\ &= Fib(2) + Fib(3) + Fib(5) \\ &= Fib(0) + Fib(1) + Fib(3) + Fib(5) \\ &= 0 + 1 + Fib(3) + Fib(5) \\ &= 1 + Fib(1) + Fib(2) + Fib(5) \\ &= 1 + Fib(1) + Fib(0) + Fib(1) + Fib(5). \end{aligned}$$

Etc... The source of this inefficiency is the repetition of the same calculations because the system forgets what has already been calculated. For example, Fib() is called eight times with parameter n = 1 to decide that 1 can be returned. For each number of the sequence, the function computes all its predecessors without taking into account that it suffices to do this only once.

It takes almost a quarter of a million calls to find the twenty-sixth Fibonacci number, and nearly 3 million calls to determine the thirty-first! This is too heavy a price for the simplicity of the recursive algorithm. As the number of calls and the run time grow exponentially with n, the algorithm has to be abandoned except for very small numbers

An iterative algorithm may be produced rather easily as follows:

```
unsigned long iterativeFib(unsigned long n) {
    if (n < 2)
    return n;
    else {
        register long i = 2, tmp, current = 1, last = 0;
        for ( ; i <= n; ++i) {
            tmp = current;
            current += last;
            last = tmp;
        }
        return current;
    }
}
```

However, there is another, numerical method for computing Fib(n), using a formula discovered by Abraham de Moivre:

$$\mathrm{Fib}(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}.$$

Where $\phi = \frac{1}{2}(1+\sqrt{5})$, and $\hat{\phi} = 1 - \phi = \frac{1}{2}(1-\sqrt{5})$. $\hat{\phi}$ becomes very small when n grows, thus it can be omitted.

$$\mathrm{Fib}(n) = \frac{\phi^n}{\sqrt{5}}.$$

Approximated to the nearest integer

```
unsigned long deMoivreFib(unsigned long n) {
    return ceil(exp(n*log(1.6180339897) - log(2.2360679775)) -
↪   .5);
}
```

1.6 Backtracking

In solving some problems, a situation arises where there are different ways leading from a given position, none of them known to lead to a solution. After trying one path unsuccessfully, we return to this crossroads and try to find a solution using another path. However, we must ensure that such a return is possible and that all paths can be tried. This technique is called backtracking, and it allows us to systematically try all available avenues from a certain point after some of them lead to nowhere. Using backtracking, we can always return to a position that offers other possibilities for successfully solving the problem. This technique is used in artificial intelligence, and one of the problems in which backtracking is very useful is the eight queens problem.

The eight queens problem attempts to place eight queens on a chessboard in such a way that no queen is attacking any other To solve this problem, we try to put the first queen on the board, then the second so that it cannot take the first, then the third so that it is not in conflict with the two already placed, and so on, until all of the queens are placed. What happens if, for instance, the sixth queen cannot be placed in a nonconflicting position? We choose another position for the fifth queen and try again with the sixth. If this does not work, the fifth queen is moved again. If all the possible positions for the fifth queen have been tried, the fourth queen is moved and then the process restarts. This process requires a great deal of effort, most of which is spent backtracking to the first crossroads offering some untried avenues. In terms of code, however, the process is rather simple due to the power of recursion, which is a natural implementation of backtracking

```
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 finds all possible solutions without regard to the fact that some of them are symmetrical.