

Introduction to Programming Recursively in C++

Gladys Monagan
Department of Computer Science
Langara College

January 13, 2023

Abstract


Writing *recursive programs* takes practice. This document is an attempt to help a student become proficient in *programming recursively in C++*.

Prior experience in programming in C++ is required.

There are solved examples and exercises.

Many of the examples and exercises are intrinsically *not* recursive but since the main purpose of this document is to give a student *practice* in coding recursively, we had to borrow problems that should have been coded iteratively, especially for efficiency reasons.

None of the examples nor exercises are graphical nor pictorial examples:

so much for the beautiful Koch snowflake and plasma clouds .

Contents

1	Introduction	4
2	Definition of Recursion	4
3	Working with integers	4
3.1	Exercise: factorial	5
3.2	Exercise: sum	6
3.3	Exercise: number of multiples of 4	6
3.3.1	Slow Version	6
3.3.2	Better Version	6
3.4	Exercise: number of multiples of 7 in a range	7
3.5	Exercise: number of multiples of x in a range	7
3.6	Exercise: harmonic sum	7
3.7	Exercise: sum up digits	7
3.8	Exercise: print the minimum transformation	8
3.9	Exercise: digital root	8
3.9.1	Loop Version	9
3.9.2	With a Helper Recursive Call	9
4	Work Before or After?	9
4.1	Examples: writing up and down	9
4.2	Exercise: write backwards and then forwards	11
5	Working with Arrays	11
5.1	Example: linear search	12
5.2	Exercise: addition	12
5.3	Exercise: multiplication	13
5.4	Exercise: less than x?	13
5.5	Exercise: maximum	13
5.6	Exercise: minimum positive number	13
5.7	Exercise: is ascending?	13
5.8	Exercise: is strictly descending?	14
5.9	Example: sum	14
5.10	Example: another sum	14
5.11	Example: binary search	15
5.12	Exercise: swap pairs of elements in an array	16
5.13	Exercise: swap pairs of elements in an array from last to first	17
5.14	Exercise: move elements around in an array	17

6	Processing Two Arguments Simultaneously	18
6.1	One Argument Used As Reference	18
6.1.1	Exercise: occurrences	18
6.2	Processing in Lockstep	18
6.2.1	Example: ridiculous equal	18
6.2.2	Exercise: dot product	19
6.3	Consuming at Different Rates	19
6.3.1	Example: atLeast	19
6.3.2	Example: subset	19
7	In Binary and Octal and Hexadecimal	20
7.1	Example: Non recursive example of output in binary	20
7.2	About the Following Exercises	21
7.2.1	Exercise: output as a binary number	21
7.2.2	Exercise: output as an octal number	22
7.2.3	Exercise: output as a hexadecimal number	22
7.2.4	Exercise: output as a binary number with bit shifting	23
7.2.5	Exercise: output as an octal number with bit shifting	23
7.2.6	Exercise: output as a hexadecimal number with bit shifting	24
8	Working with Strings	24
8.1	Example: reversing a string allocating extra memory	24
8.2	Example: reversing a string in place	25
8.3	Exercise: convert a string into an integer	25
8.4	Exercise: determine if in alphabetical order in a dictionary	26
8.5	Exercise: determine if commas are properly placed	27
8.6	Exercise: determine if a pattern is in a string	28
8.7	Exercise: same number and position of a character	29
8.8	Exercise: string matching	29
8.9	Exercise: equals ignoring vowels	30
9	Recursion with a Collector Variable	31
10	Epilogue	31

1 Introduction

The purpose of this document is to give a student practice in programming recursively. Since this document was written for students in a first course in Data Structures and Algorithms (CPSC 1160 at Langara College), and since this document is meant to be used early in the course, there are no functions that use linked lists nor are there recursive data structures like trees 🌲.

For students in a second Data Structures and Algorithms course (CPSC 2150 at Langara College), we recommend that the student reading this document solve some of the problems using linked lists instead of arrays. The base case would be typically the empty list (the null pointer) as opposed to an index or the number of elements in the array, and obviously, the recursive call would be a pointer to the next node in a singly linked list.

Note that none of our examples (nor exercises) have *mutual recursion* nor *nested recursion* [Dro13]. Why? We don't want to make this document overly long.

2 Definition of Recursion

A C++ function is *recursive* if it has at least one call to itself [Sav15]. If the recursive function has an argument, the argument of the recursive call is a “smaller” version of the current function's argument. With each recursive call, we make our way towards the *base case*. Continuing with the “smaller” analogy, the base case would be the smallest version.

Recursive functions may have more than one argument.

There can be more than one recursive call in a function (linear recursion) so the results from the various recursive calls may need to be combined inside the function. And, there can be more than one base case.

3 Working with integers

A name for the positive integers with 0 is the *natural numbers* and as Niklaus Wirth mentions on page 87 of [Wir85], the definition of natural numbers in Mathematics is recursive. Here are some definitions:

1. Natural numbers:

- (a) 0 is a natural number.
- (b) the successor of a natural number is a natural number.

2. Tree structures

- (a) 0 is a tree (called the empty tree).
- (b) If t_1 and t_2 are (binary trees, then the structures consisting of a node with two descendants t_1 and t_2 are also binary trees.

3. The factorial function $f(n)$:

```
f(0) = 1
f(n) = n * f(n-1)    for n > 0
```

Suppose that we want to write a C++ function to sum up the positive integers up to and including a positive integer n

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + (n-2) + (n-1) + n \quad (1)$$

Because addition is commutative, I can change the order of the addition of the integers (to simplify coding) as in equation (2)

$$\sum_{i=1}^n i = n + (n-1) + (n-2) + \dots + 3 + 2 + 1 \quad (2)$$

i.e. the sum is started with n instead of with 1.

We can now code equation (2) iteratively with a loop as follows

```
unsigned int addIterative(unsigned int n) {
    int sum = 0u;
    for (auto i = n; i > 0u; i--) {
        sum += i;
    }
    return sum;
}
```

Let's modify equation (2) to

$$\begin{aligned} \sum_{i=1}^n i &= n + \sum_{i=1}^{n-1} i \\ &= n + (n-1) + (n-2) + \dots + 3 + 2 + 1 \end{aligned} \quad (3)$$

Listing 1: the function add implements equation (3)

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

3.1 Exercise: factorial

Write a recursive function to calculate the factorial of a non-negative integer n , for $n \geq 0$. The factorial is defined as

$$0! = 1 \text{ because of the empty product}$$

$$\begin{aligned}
1! &= 1 \\
2! &= 2 \cdot 1 = 2 \cdot 1! \\
3! &= 3 \cdot 2 \cdot 1 = 3 \cdot 2! \\
&\vdots \\
n! &= n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1 = n \cdot (n-1)!
\end{aligned}$$

3.2 Exercise: sum

Write a recursive function to compute the sum

$$\sum_{i=m}^n i$$

i.e. add all the integers from m up to and including n . Deal appropriately with the “border line cases” when m is equal to n or when there are no integers in the range m to n (hint: note that the range $[m \dots n]$ is different from the range $[n \dots m]$.)

Document the function with the proper function description (i.e. what does the function do), plus the precondition(s) and/or the postcondition(s).

3.3 Exercise: number of multiples of 4

Write a recursive function to count the number of positive multiples of 4 up to, and possibly including, n , for $n > 0$.

Use **addition** (or subtraction) to give you practice writing recursive functions even though there is a more natural and efficient way of coming up with the multiples of 4 in general.

Examples

- `numberOfPosMults4(15)` returns 3
because there are 3 multiples of 4 namely the integers 4, 8, 12
- `numberOfPosMults4(20)` returns 5
because there are 5 multiples of 4 namely the integers 4, 8, 12, 16, 20

3.3.1 Slow Version

Implement the recursive function to count the positive multiples of 4 by making n recursive calls.

3.3.2 Better Version

Implement the recursive function to count the positive multiples of 4 by making at most $n/4 + 1$ recursive calls.

3.4 Exercise: number of multiples of 7 in a range

Write a function to count the number of multiples of 7, in the range $[m \dots n]$, where m and n are integers.

Use **addition** (or subtraction) to give you practice writing recursive functions. Use the function prototype

```
int numberOfMults7(int m, int n);
```

Examples

- `numberOfMults7(-22, 7)` returns 5: there are 5 multiples of 7 in the range $[-22 \dots 7]$ namely the numbers -21, -14, -7, 0, 7.
- `numberOfMults7(14, 14)` returns 1: there is only one multiple of 7, namely 14, in the range $[14 \dots 14]$.

... it is somewhat funny to say that 0 is a “multiple of 7” but for the purpose of the exercise, we’ll say that it is.

3.5 Exercise: number of multiples of x in a range

Write a recursive function to count the number of multiples of x , for $x > 0$, in the range $[m \dots n]$, where m and n are integers. We are **not** assuming that $m > 0$ nor $m > 0$.

Use **addition** (or subtraction) to give you practice writing recursive function and again, as Exercise 3.4, in this context the number 0 is considered a “multiple of x ”.

3.6 Exercise: harmonic sum

Write a recursive function to compute the sum of the harmonic numbers from 1 for $n \geq 1$.

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

Use the function prototype

```
double hSum(unsigned int n);
```

3.7 Exercise: sum up digits

Write a recursive function to sum up the digits of a non-negative integer. Use the function prototype

```
int sumDigits(unsigned int n);
```

Example

`sumDigits(2019)` returns 12 because $2 + 0 + 1 + 9 = 12$.

You can use the operator `% 10` (for mod 10) to obtain the rightmost digit of a number. For instance, `2019%10` gives the digit 9.

You can use integer division `/` by 10 to drop the right most digit of the number. For instance, `2019/10` yields 201.

3.8 Exercise: print the minimum transformation

Given two positive integers x and y , with $y \geq x$, write a recursive function to output the steps of the minimum transformations needed to transform x into y given the only two possible operations of multiplication by 2 and addition by 1.

Use the function prototype

```
void outMinTransformation(int x, int y, std::ostream& out = std::cout);
```

Example

`outMinTransformation(6, 19)` outputs `19 = (((6 + 1) + 1) + 1) * 2 + 1)`

`outMinTransformation(7, 58)` outputs `58 = (7 * 2 * 2 + 1) * 2`

We suggest that you output “`y =` ” inside of `outMinTransformation` first and then call a recursive function to output the transformations. Note that there is no end-of-line output at the end of the expression.

Blanks are not significant.

Superfluous brackets are allowed. This exercise was taken from Sedgewick and Wayne in [SW13].

3.9 Exercise: digital root

The digital root of a non-negative integer n is computed by summing the digits of n . The digits of the resulting number are then added, and this process is continued until a single digit number is obtained. Compute the digital root of a non-negative number with a recursive function that has the function prototype

```
int digitalRoot(int n);
```

Examples

`digitalRoot(2019)` returns 3 because $2 + 0 + 1 + 9 = 12$ and $1 + 2 = 3$

`digitalRoot(3)` returns 3

`digitalRoot(55555)` returns 7 because $5 + 5 + 5 + 5 + 5 = 25$ and $2 + 5 = 7$

This exercise was taken from the CEMC from the University of Waterloo [[Wat14]].

Go to <http://cscircles.cemc.uwaterloo.ca/16-recursion/> to see many examples though they are in Python.

3.9.1 Loop Version

Write a version of `digitalRoot` that is recursive but also has a loop in it.

3.9.2 With a Helper Recursive Call

Write a version of `digitalRoot` that calls the helper, recursive function, `sumDigits` from Exercise 3.7.

4 Work Before or After?

4.1 Examples: writing up and down

Savitch gives two Self-Test Exercises on page 582 of *Absolute C++* [Sav13]

4. Write a recursive `void` function that takes a single `int` argument `n` and writes the integers
 5. Write a recursive `void` function that takes a single `int` argument `n` and writes the integers `n, n - 1, ..., 3, 2, 1`.
- (Notice that Exercise 4. and 5. can vary by only two lines.)

Listing 2: function to solve Savitch's Self-Test Exercise 4

```
// Write onto standard output the integers
// 1, 2, 3, ..., n-1, n
// precondition:
//   n >= 0
void writeUp(int n) {
    if (n == 0) {
        return;
    }
    writeUp(n - 1);
    std::cout << n << " ";
}
```

Listing 3: function to solve Savitch's Self-Test Exercise 5

```
// Write onto standard output the integers
// n, n-1, ..., 3, 2, 1
// precondition:
//   n >= 0
void writeDown(int n) {
    if (n == 0) {
        return;
    }
    std::cout << n << " ";
    writeDown(n - 1);
}
```

Savitch gives as solutions the following two functions

```
void writeUp1(int n) {
    if (n > 0) {
        writeUp1(n - 1);
        cout << n << " ";
    }
}

void writeDown1(int n) {
    if (n > 0) {
        cout << n << " ";
        writeDown1(n - 1);
    }
}
```

writeUp and writeUp1 output the numbers the same way; however, I don't like the writeUp1 as much as I feel that it's harder to identify the base case of the recursive function.

What is the difference between the writeUp of Listing 2 and the writeDown of Listing 3? In writeUp of Listing 2 the “work”, the writing of the number, happens *after* the recursive call. For instance,

- writeUp(3) cannot be completed and the number 3 cannot be written until writeUp(2) is completed
- writeUp(2) cannot be completed and the number 2 cannot be written until writeUp(1) is completed
- writeUp(1) cannot be completed and the number 1 cannot be written until writeUp(0) is completed
- writeup(0) is called and in this base case no work is done and control is returned to the calling function
- back to writeUp(1), since writeUp(0) is completed, the number 1 is written onto standard output: the “work” of writeUp(1) is done
- back to writeUp(2), since writeUp(1) is completed, the number 2 is written: the “work” of writeUp(2) is done
- back to writeUp(3), since writeUp(2) is completed, the number 3 is written: the “work” of writeUp(3) is done

Contrast with the function writeDown of Listing 3 where the “work”, the writing of the number, happens *before* the recursive call. For instance,

- writeDown(3) writes onto standard output the number 3, i.e. it does its “work” and then it calls writeDown(2). Note that though there is no “work” left to do, writeDown(3) is not completed until writeDown(2) is finished
- writeDown(2) does its “work” of writing the number 2 and then it calls writeDown(1)

- `writeDown(1)` writes the number 1 and then calls `writeDown(0)`
- `writeDown(0)` is called and in this base case no work is done and control is returned to the calling function
- back to `writeDown(1)` nothing to do except return
- back to `writeDown(2)` nothing to do except return
- back to `writeDown(3)` nothing to do except return

In these two examples we have not had to combine the results of the recursive functions because the procedures are “C++ `void`”, i.e., they are not value-returning functions. But we used these functions to illustrate how sometimes the “work” is done before the recursive call and sometimes after. Note that sometimes there are several recursive calls within one function.

At this point, it begs for me to talk about *tail recursion* and the *stack*. Since I won’t, I suggest that you read up on both topics on your own.

4.2 Exercise: write backwards and then forwards

Given a value `n > 0`, write the decimal number `n` backwards (digit-wise), then a new line i.e. “`\n`”, and then write the same number forwards. Write to the output stream `out` which has as default value `std::cout`. Use the function prototype

```
void writeBackwardForward(unsigned int n, std::ostream& out = std::cout);
```

The call `writeBackwardForward(123456)` outputs

```
654321
123456
```

Note that there is no new line after the last 6.

The call `writeBackwardForward(10000)` outputs

```
00001
10000
```

Note that there is no new line after the last 0.

5 Working with Arrays

The declaration in C++

```
int A[5];
```

allocates space for 5 integers that are contiguous in memory. Typically we declare an array using a named constant

```
const int MAX_SIZE = 5;
int A[MAX_SIZE];
```

There are programming languages like Java, which allow for a field of the “object array” to be queried so that the length of the array can be obtained. It’s different in C++ (it’s more like the programming language C). In C++ , “we as programmers” keep track of the number of elements that are being used in the array¹.

I will be using the variable n to denote the number of elements used in the array. In the program fragment below,

```
const int MAX_SIZE = 5;
int A[MAX_SIZE];
int n;
n = 3;
for (int i=0; i < n; i++)
    A[i] = -i;
```

we assume that $n \leq \text{MAX_SIZE}$.

5.1 Example: linear search

Write a recursive function to search for the value x in an array of integers. If the value x is found, return **true** and return **false** otherwise.

```
// Do a linear search for x in the first n integers of A
// precondition:
//   n <= declared size of A
// postcondition:
//   return true if x is in A, false otherwise
bool found(int x, const int A[], int n) {
    if (n == 0) { // no elements in A
        return false;
    }
    else if (A[n-1] == x) {
        return true;
    }
    else {
        return found(x, A, n - 1);
    }
}
```

5.2 Exercise: addition

Write a recursive function to add the first n elements of an array of integers

$$\sum_{i=0}^{n-1} A_i = A_0 + A_1 + \dots + A_{n-2} + A_{n-1} \quad (4)$$

Assume that $n \geq 0$. Use the function prototype

¹we *could* get the declared size of an array using *sizeof* but we usually don’t

```
int sum(const int A[], int n);
```

5.3 Exercise: multiplication

Write a recursive function to multiply the first n elements of an array of doubles

$$\prod_{i=0}^{n-1} A_i = A_0 \times A_1 \times \dots \times A_{n-2} \times A_{n-1}$$

Assume that $n > 0$. Use the function prototype

```
double product(const double A[], int n);
```

You would never program this function recursively. And note that it is very easy for multiplication to overflow!

5.4 Exercise: less than x?

Determine if all the elements of A (i.e. the first n elements of A) are less than x . Assume that $n > 0$. Use the function prototype

```
bool smaller(int x, const int A[], int n);
```

5.5 Exercise: maximum

Find the maximum value in the array. Assume that $n \geq 1$. Use the function prototype

```
int maximum(const int A[], int n);
```

5.6 Exercise: minimum positive number

Find the minimum positive value that is in the array A . Assume that $n \geq 0$. If the array A does not have any positive integers, return 0. Use the function prototype

```
int minPosNum(const int A[], int n);
```

Do not use any library function e.g. do not use `std::min`.

5.7 Exercise: is ascending?

Determine if an array A of n integers is sorted in ascending order, for $n \geq 1$. Determine the validity of

$$A_0 \leq A_1 \leq A_2 \leq \dots \leq A_{n-1}$$

Use the function prototype

```
bool isAscending(const int A[], int n);
```

5.8 Exercise: is strictly descending?

Determine if an array A of n integers is sorted in *strictly* descending order. Determine the validity of

$$A_0 > A_1 > A_2 > \dots > A_{n-1}$$

An empty array *is* in strictly descending order.

```
bool isStrictlyDescending(const int A[], int n);
```

5.9 Example: sum

One way of implementing equation (4) of Exercise 5.2 is with the following code.

```
int sum1(const int A[], int n) {
    if (n == 0) {
        return 0;
    }
    return A[n-1] + sum1(A, n - 1);
}
```

We could have written `sum1`, by “changing the beginning” of the array A as in `sum2`:

```
int sum2(const int A[], int n) {
    if (n == 0) {
        return 0;
    }
    return A[0] + sum2(A + 1, n - 1);
}
```

5.10 Example: another sum

Instead of adding one array element $A[n-1]$ to the sum of the first $n-1$ elements as in `sum1`, (or to the sum of the last $n-1$ elements as in `sum2`), we can split the array into two “halves” and continue splitting the “resulting array” into two “halves” again, and so on, until we have one element left in the array. And then we can pop out of the recursion and add that element to the result of adding the other “half”.

```
// Add the first n elements of an array of integers.
// precondition:
//    0 <= n <= declared size of A
// postcondition:
//    sum of A[0] + A[1] + ... A[n-1]
int sum(const int A[], int n);

// Add the elements from A[left] up to and including A[right]
// postcondition:
//    return A[left] + ... + A[right-1] + A[right] for left <= right
//    and 0 otherwise (for left > right)
int sum(const int A[], int left, int right);
```

```

int sum(const int A[], int n) {
    return sum(A, 0, n-1);
}
int sum(const int A[], int left, int right) {
    if (left > right) {
        return 0;
    }
    else if (left == right) {
        return A[left];
    }
    else {
        int m = (left + right) / 2;
        return sum(A, left, m) + sum(A, m + 1, right);
    }
}

```

Note that every element $A[i]$ is added exactly once. We have not put any preconditions for the indices `left` and `right` so we need to check that if `left` is greater than `right`, then the function will return 0 to prevent an “infinite” loop (no more stack space). If there are no elements in the array, the call would be `sum(A, 0)` and the call to the overloaded function would be `sum(A, 0, -1)` needing the base case `left > right`.

Up to now we have not talked about the “run-time stack”, the “stack frame” nor about “blowing your stack” when programming recursively. These topics are very vital and you should comprehend the physical limitations of the “stack memory”. In this writeup, we are concentrating on learning how to program recursively; therefore, we have left out the “mechanics” of recursion and the optimization done by the compilers when the function is “tail recursive”.

Having too many recursive calls that “do not do much” can slow down the computations, so we add below another base case to save on having so many recursive calls.

```

int sum(const int A[], int left, int right) {
    if (left > right) {
        return 0;
    }
    else if (left == right) {
        return A[left];
    }
    else if (right - left == 1) {
        return A[left] + A[right];
    }
    else {
        int m = (left + right) / 2;
        return sum(A, left, m) + sum(A, m + 1, right);
    }
}

```

5.11 Example: binary search

If the array is ordered (sorted already) do a *binary search*.

```

// Search for x in the first n elements of A and return the index i
// such that A[i] == x (for duplicates of x in A, return any of the indices)
// precondition:
//    0 <= n <= declared size of A, A is SORTED in ascending order
// postcondition:
//    return the index in A of x and if x is not in A, return -1
int search(int x, const int A[], int n);
// return index if x in A[left] up to and including A[right], -1 if not found
int search(int x, const int A[], int left, int right);

int search(int x, const int A[], int n) {
    return search(x, A, 0, n - 1);
}
int search(int x, const int A[], int left, int right) {
    if (right < left) {
        return -1; // not found
    }
    int mid = (left + right) / 2;
    if (A[mid] == x) {
        return mid;
    }
    else if (x < A[mid]) {
        return search(x, A, left, mid - 1);
    }
    else { // x > A[mid]
        return search(x, A, mid + 1, right);
    }
}

```

5.12 Exercise: swap pairs of elements in an array

Write a recursive function to overwrite an array A of n integers so that the first and second element are swapped, the third and the fourth, and so on. The swapping in pairs happens from “left to right”. Assume that $n > 0$. Use the function prototype

```
void swapPairsLeftToRight(int A[], int n);
```

You may overload the function but the overloaded function must be called from inside swapPairsLeftToRight with the prototype given above.

Examples

- if the array A has the values 11 22 33 44 55 66 and $n = 6$ then after the call to swapPairsLeftToRight, A has the values 22 11 44 33 66 55
- if the array A has the values 11 22 33 44 55 and $n = 5$ then after the call to swapPairsLeftToRight, A has the values 22 11 44 33 55

5.13 Exercise: swap pairs of elements in an array from last to first

Write a recursive function to overwrite an array A of n integers so that the last and “second to last” element are swapped, the “third to last” and the “fourth to last”, and so on. The swapping in pairs happens from “right to left”. Assume that $n > 0$. Use the function prototype

```
void swapPairsRightToLeft(int A[] , int n);
```

You may overload the function but the overloaded recursive function must be called from inside swapPairsRightToLeft with the prototype given above.

Examples

if the array A has the values 11 22 33 44 55 66 and $n = 6$ then after the call to swapPairsRightToLeft, A has the values 22 11 44 33 66 55

if the array A has the values 11 22 33 44 55 and $n = 5$ then after the call to swapPairsRightToLeft, A has the values 11 33 22 55 44

- You can overload the required function and make the overloaded function the one that is recursive.

5.14 Exercise: move elements around in an array

Based on an exercise in [GTM11].

Write a recursive function to move the elements of an array A so that the negative integers go first before all the positive integers and zero.

Use the function prototype

```
// precondition:  
// A has been declared to have room for at least n integers , n >= 0  
void moveNegativesFirst(int A[] , int n);
```

- Do not allocate any space for new arrays nor strings: move the elements of the array overwriting the original elements of the array as needed.
- Do not make your solution quadratic.
- Ideally we would want to make at most \log_2 calls but, a linear number of calls might be needed.
- Do not use any loops.
- You can overload the required function and make the overloaded function the one that is recursive.

6 Processing Two Arguments Simultaneously

It is very nice to program linked lists recursively; however, since you might not have seen linked lists when you get this write up I will work with arrays and hope that this material can be useful when you get to learn about linked lists.

In the class notes from the University of Waterloo [Wat14], the functions that “consume two lists”, or some two arguments, can be categorized into three types of functions.

1. A function where one of the two arguments “goes along for the ride”, in other words, one of the arguments is not “made smaller” with each recursive call.
2. A function with two arguments, say two lists, which are processed in lockstep.
3. A function with two arguments, say two lists, which are of different length and the arguments are consumed at different rates.

6.1 One Argument Used As Reference

6.1.1 Exercise: occurrences

Write a function that counts the number of occurrences of x in the first n elements of an array A . Use the function prototype

```
// precondition:  
// A has been declared to have room for at least n integers, n >= 0  
int occurrences(int x, const int A[], int n);
```

6.2 Processing in Lockstep

6.2.1 Example: ridiculous equal

We want to test whether the elements of two arrays are equal. The two arrays can be processed in lockstep. Now, it is ridiculous, ugly, and inefficient to compare two arrays of the same length using recursion. There are no indices in linked lists, so one cannot access the i^{th} element directly. Contrast with arrays where the i^{th} element is accessed with $A[i]$. So, when comparing two arrays to see if they are equal, one would typically use a **for** loop. Now, since this writeup is on recursion and since we have not talked about linked lists, the following example determines *recursively* if two arrays are equal.

```
bool equal(const int A[], int n, const int B[], int m) {  
    if (n != m) { // different number of elements  
        return false;  
    }  
    if (n == 0) { // no (more) elements in A and B  
        return true;  
    }  
    return (A[n - 1] == B[m - 1]) && equal(A, n - 1, B, m - 1);  
}
```

6.2.2 Exercise: dot product

Exercise based on the dot product of two lists from [Wat14].

Write a function to calculate the dot product of two arrays of n elements

$$A_0 \cdot B_0 + A_1 \cdot B_1 + \dots + A_{n-2} \cdot B_{n-2} + A_{n-1} \cdot B_{n-1}$$

Exercise: string matching in 8.8

6.3 Consuming at Different Rates

6.3.1 Example: atLeast

Determine if the value x appears at least m times in an array of n integers.

```
// precondition:
//   m >= 0
//   0 <= n <= declared size of A
// postcondition:
//   returns true if there are at least m x's in A
bool atLeast(int x, int m, const int A[], int n) {
    if (m == 0) { // no occurrence of m in A means true
        return true;
    }
    if (n == 0) { // m > 0 and no elements in A
        return false;
    }
    // we decrement the number of occurrences m if there is a match
    if (x == A[n-1]) {
        m = m - 1; // found one occurrence
    }
    return atLeast(x, m, A, n - 1);
}
```

6.3.2 Example: subset

Given an array A with n unique elements (i.e. without repeats) *sorted in ascending order* and given an array B with m unique elements *sorted in ascending order*, determine if every element of A appears in B . In other words, determine if

$$A \subset B$$

```
// Determine if A with n elements is a subset of B with m elements
// precondition:
//   0 <= n <= declared size of A
//   0 <= m <= declared size of B
//   A is sorted in ascending order without repeats A[i] != A[i+1]
//   B is sorted in ascending order without repeats B[i] != B[i+1]
// postcondition:
```

```

//      return true if every element of A is in B, false otherwise
bool subset(const int A[], int n, const int B[], int m) {
    if (n == 0) { // A is the empty subset
        return true;
    }
    if (m == 0) { // B is the empty subset
        return false;
    }
    if (A[n-1] > B[m-1]) { // A[n-1] is not in B
        return false;
    }
    else if (A[n-1] == B[m-1]) {
        return subset(A, n - 1, B, m - 1);
    }
    else { // A[n-1] < B[m-1]
        return subset(A, n, B, m - 1);
    }
}

```

Exercise: equals in [8.9](#)

7 In Binary and Octal and Hexadecimal

In this section we work with integers yet instead of displaying the numbers base 10, we want to show the bits of those decimal integers. And we want to practice bit shifting and bit operators without using the C++ libraries that would facilitate this.

7.1 Example: Non recursive example of output in binary

Using the function prototype below, that assigns the default value `std::cout` to the stream out

```
#include <iostream>
```

```
void nonRecursiveoutAllBits(unsigned int n, std::ostream& out=std::cout);
```

write a function that outputs a decimal number `n` in binary. Pad left with leading (bit) zeros so that the total number of bits matches the internal representation of an int.

Listing 4: this function does contain loops

```
#include <climits> // CHAR_BIT
#include <iostream>
```

```

void nonRecursiveoutAllBits(unsigned int n, std::ostream& out) {
    std::size_t sizeInBytes = sizeof(n); // in bytes
    unsigned int numBits = sizeInBytes * CHAR_BIT;
    unsigned int mask = 0x08;
    for (unsigned i = 1; i < numBits / 4; i++) mask = mask << 4;
    for (unsigned i = 0; i < numBits; i++) {
        char ch;

```

```

        if ((mask & n) == 0)
            ch = '0';
        else
            ch = '1';
        out << ch;
        mask = mask >> 1;
    }
}

```

The function in Listing 4 below is not recursive, uses loops, and is not that efficient. And yes, there are extra variables declared.

7.2 About the Following Exercises

The exercises in this subsection 7.2 are not “recursive problems” in nature but they illustrate how the system stack, which recursion uses, can be utilized. For these exercises, you must code as follows.

- Do not use `bitset`.
- Do not use any explicit loops (use recursion).
- Use standard C++ and not the gcc extensions.
- Do not use any math function like `pow`.
- Do not use arrays.
- Do not use strings (neither C strings nor `std::string` nor string concatenation).
- Do not use the STL.
- Do not use `std::hex`, `std::oct` nor `std::dec`.
- You can overload the required function and make the overloaded function the one that is recursive. You might want to first calculate the number of digits when left padding with zeros if required in the exercise.

7.2.1 Exercise: output as a binary number

Write a recursive function to output the decimal number `n` as a binary number. Use the function prototype

```
void outputAsBinary(unsigned int n, std::ostream& out = std::cout);
```

- You *can* use the integer arithmetic operators `%` for mod and `/` for integer division.
- Do not output any leading zeros.

Example

`outputAsBinary(12)` outputs 1100

7.2.2 Exercise: output as an octal number

Write a recursive function to output the decimal number n as an octal number, i.e. base 8. Use the function prototype

```
void outputAsOctal(unsigned int n, std::ostream& out = std::cout);
```

- You *can* use the integer arithmetic operators $\%$ for mod and $/$ for integer division.
- Do not output any leading zeros.

Example

`outputAsOctal(32)` outputs 40

7.2.3 Exercise: output as a hexadecimal number

Write a recursive function to output the decimal number n as an hexadecimal number. Use the standard hexadecimal digits 0, 1, ... 9 and then

$$\begin{aligned} 10_{10} &= A_{16} \\ 11_{10} &= B_{16} \\ 12_{10} &= C_{16} \\ 13_{10} &= D_{16} \\ 14_{10} &= E_{16} \\ 15_{10} &= F_{16} \end{aligned}$$

Use the function prototype

```
void outputAsHex(unsigned int n, std::ostream& out = std::cout);
```

- You *can* use the integer arithmetic operators $\%$ for mod and $/$ for integer division.
- Do not output any leading zeros.
- We suggest that you write a short function that returns the right representation of a hex digit to prevent clobbering your recursive function.

Example

`outputAsHex(1529)` outputs 5F9

7.2.4 Exercise: output as a binary number with bit shifting

Write a recursive function to output the decimal number `n` as a binary number.
Use the function prototype

```
void outInBinary(unsigned int n, std::ostream& out = std::cout);
```

- You *cannot* use the integer arithmetic operators `%` for mod and `/` for integer division.
- Do bit shifting with the operators `>>` and `<<` and use the bitwise operators `&` and `|` and `~` as needed.
- Pad left with leading, non significant (bit) zeros so that the total number of bits matches the internal representation of an `int`.
- You can use literals like 16 in your code (you don't need to use a named constant).

Example when `CHAR_BIT` is 8 and `sizeof(unsigned)` is 4

`outInBinary(12)` outputs 00000000000000000000000000001100

`outInBinary(1025)` outputs 00000000000000000000000010000000001

7.2.5 Exercise: output as an octal number with bit shifting

Write a recursive function to output the decimal number `n` as an octal number.
Use the function prototype

```
void outInOctal(unsigned int n, std::ostream& out = std::cout);
```

- You *cannot* use the integer arithmetic operators `%` for mod and `/` for integer division.
- Do bit shifting with the operators `>>` and `<<` and use the bitwise operators `&` and `|` and `~` as needed.
- Pad left with leading, non significant (octal) zeros so that the total number of bits matches the internal representation of an `int`.
- You can use literals like 3 in your code (you don't need to use a named constant).

Example when `CHAR_BIT` is 8 and `sizeof(unsigned)` is 4

`outInOctal(32)` outputs 00000000040

`outInOctal(2147483647)` outputs 17777777777

7.2.6 Exercise: output as a hexadecimal number with bit shifting

Write a recursive function to output the decimal number `n` as a hexadecimal number. Use the function prototype

```
void outInHex(unsigned int n, std::ostream& out = std::cout);
```

- You *cannot* use the integer arithmetic operators `%` for mod and `/` for integer division.
- Do bit shifting with the operators `>>` and `<<` and use the bitwise operators `&` and `|` and `~` as needed.
- Pad left with leading, non significant (hex) zeros so that the total number of bits matches the internal representation of an `int`.

Example when `CHAR_BIT` is 8 and `sizeof(unsigned)` is 4

```
outInHex(1529) outputs 000005F9
```

```
outInHex(2147483647) outputs 7FFFFFFF
```

8 Working with Strings

The STL `std::string`, as opposed to C string, is not part of the core C++ language. The STL `std::string` is a class and there are methods to manipulate the objects.

Use `#include <string>`.

One can access the i^{th} element of the string directly using the square brackets operator `[]` in the same way that we use the operator `[]` in arrays. Or by using the member function `at`. There is range checking when using the member function `at` and an *out of range* exception is thrown if the index i is out of bounds.

We can also get the number of characters in the string by calling the member method `length` or the member method `size`.

8.1 Example: reversing a string allocating extra memory

If you are coming from Java, you know that Java Strings are immutable and there is garbage collection in Java. So, one could write a function to reverse a string, definitely *not in place*, creating lots of substrings. Extra memory is allocated because of the substrings allocated.

```
string reverse(const string& str) {  
    int n = str.length();  
    if ( (n == 0) || (n == 1) ) {  
        // the empty string and a string with a single character  
        // are reversed already  
        return str;  
    }  
    else if (n == 2) { // swap the two characters
```



```

        // create a null string for use of string concatenate "+"
        string emptyStr = "";
        return str[1] + emptyStr + str[0];
    }
    // n >= 3
    string midStr = reverse(str.substr(1, n - 2));
    return str[n - 1] + midStr + str[0];
}

```

8.2 Example: reversing a string in place

But we can reverse a string *in place* as well. The initial string `str` is overwritten. `str` is both an input parameter and an output parameter. In the version below we are using indices to traverse the string.

```

// reverse the chars from str[low] up to and including str[high]
void reverseInPlace(int low, int high, string& str);

// reverse the string str in place
// postcondition:
//   str has the characters of the input str in reverse order
void reverseInPlace(string& str) {
    reverseInPlace(0, str.length() - 1, str);
}
void reverseInPlace(int low, int high, string& str) {
    if (low >= high) {
        return; // nothing to do
    }
    // swap str[low] with str[high]
    char tmp = str[low];
    str[low] = str[high];
    str[high] = tmp;
    reverseInPlace(low + 1, high - 1, str);
}

```

8.3 Exercise: convert a string into an integer

Write a recursive function that converts an STL `std::string` into an integer. Assume that the argument, which is a string, has characters that are digits exclusively (there is no unary `+` sign) and that the resulting value corresponds to a positive integer or to zero. Use the function prototype

```
int toInteger(const string& str);
```

- Do not use string library functions nor C string functions except for the function `length` <http://www.cppreference.com/wiki/string/length>, `at` and the overloaded operator `[]`. Do not use `to_string`.

- Do not use any math library functions (like `pow`)
- You may write your own helper function(s).
- You may overload `toInteger` if you need extra parameters
- Do not allocate any memory in the function `toInteger` (nor when calling it) except for allocating a few integer local variables: so do not use `substr` nor concatenation. And by memory here we do *not* mean “stack memory” as in the “system stack memory” because it’s a recursive function.

Examples

`toInteger("753")` returns the value 753

`toInteger("1")` returns the value 1

`toInteger("005")` returns the value 5

8.4 Exercise: determine if in alphabetical order in a dictionary

Write a recursive function to determine if `strA` goes before `strB` in a dictionary. Deciding whether to go first depends on the alphabetical order and the length of the strings.

Make your function *case insensitive* i.e. upper and lower case letters are treated as being the same: the case is ignored.

Do not make your solution quadratic.

Use the function prototype

```
// precondition:
//   strA and strB contain letters exclusively, or are the empty string
short goesFirst(const std::string& strA, const std::string& strB);
```

`goesFirst` returns

0 if `strA` is equal to `strB`

a positive value if `strA` goes first in a dictionary i.e. `strA` is listed before `strB`

a negative value if `strB` goes first in a dictionary i.e. `strB` is listed before `strA`

- Do not use string library functions nor C string functions except for the function `length` <http://www.cppreference.com/wiki/string/length>, `at`, and the overloaded operator `[]`.
- You may write your own helper function(s).
- You may overload `goesFirst` if you need extra parameters.
- Do not use magic numbers e.g. numeric ASCII codes.
- Do not allocate any memory in the function `goesFirst` (nor when calling it) except for allocating a few `int`/`short`/`char` local variables: so do not use `substr` nor concatenation.

- ‘loops.

Examples

`goesFirst("fun", "funny")` returns a positive value

`goesFirst("", "empty")` returns a positive value

`goesFirst("laughing", "Laughing")` returns 0

`goesFirst("BeEr", "Bear")` returns a negative value

`goesFirst("Bear", "BeEr")` returns a positive value

8.5 Exercise: determine if commas are properly placed

Write a recursive function to determine if `str` represents a proper natural number or zero. The commas are placed every 3 positions.

Return **true** if `str` is a proper natural number or zero as described below and **false** otherwise. Use the function prototype

```
bool isProper(const std::string& str);
```

`str` represents a “proper” natural number or zero if all the characters are digits and there is a comma every 3 positions when counting from right to left. There cannot be a leading comma nor a trailing comma. A decimal point is not allowed, i.e. a number with a point is not a “proper number”. The empty string is “proper”. There is no unary operator for the number (i.e. no +).

- Do not use string library functions nor C string functions except for the function `length` https://en.cppreference.com/w/cpp/string/basic_string/size, `at` and the operator `[]`. Do not use `stoi`.
- You may write your own helper function(s) e.g. you could have a function to determine that part of `str` consists of digits exclusively: make this function also recursive (not that making it recursive is particularly clever but this requirement is for sheer practice).
- Do not use any loops.
- You may overload `isProper` if you need extra parameters.
- Do not use magic numbers e.g. numeric ASCII codes.
- Do not allocate any memory in the function `isProper` (nor when calling it) except for allocating a few `int`/`short`/`char` local variables: so do not use `substr` nor concatenation.

Examples

`isProper("10,000")` returns **true**

`isProper("0")` returns **true**

`isProper("5,444,333,222,111")` returns **true**

`isProper("aaa,123")` returns **false**

`isProper(",")` returns **false**

`isProper("123,")` returns **false**

8.6 Exercise: determine if a pattern is in a string

Write a recursive function to determine if a pattern appears *at least* `m` times in a string `str`. Use the function prototype

```
bool insideInOrder(const string& pattern, int m, const string& str);
```

- Do not use string library functions nor C string functions except for the function `length` https://en.cppreference.com/w/cpp/string/basic_string/size, `at` and the operator `[]`. Do not use `stoi`.
- You may write your own helper function(s) e.g. to determine if a single pattern is in a part of the `str`.
- Do not use any loops.
- Do not allocate any memory in the function `insideInOrder` (nor when calling it) except for allocating a few `int`/`short`/`char` local variables: so do not use `substr`, `npos` nor concatenation.

Examples

`insideInOrder("dog", 0, "dog")` returns **true**

`insideInOrder("dog", 1, "dog")` returns **true**

`insideInOrder("dog", 1, "dogs")` returns **true**

`insideInOrder("dog", 1, "ogd")` returns **false**

`insideInOrder("dog", 1, "dXoXg")` returns **false**

`insideInOrder("dog", 1, "a_doggie")` returns **true**

`insideInOrder("dog", 2, "doggie_dxg")` returns **false**

`insideInOrder("dog", 2, "two_doggie_doggie")` returns **true**

8.7 Exercise: same number and position of a character

Bradley Kjell's Exercise 4 [Kje].

The function `samePosChar` returns `true` for two strings

if both strings have the same number of occurrences of a character *ch*

and if that character *ch* is in the same position in each of the two strings.

It returns `false` otherwise.

Strings of different lengths are allowed as the longer string does not have a *ch* in its extra characters at the end. Use the function prototype

```
bool samePosChar(const std::string& strA, const std::string& strB, char ch);
```

- You may use helper functions e.g. as shown in 8.1 for the function `reverseInPlace` which is overloaded.
- Do not allocate any extra memory, e.g. do not use the function `substr`.
- Do work with indices as you process the strings.
- No loops.
- The only method (function) of the class `std::string` that you can use is `length`.

Examples

```
samePosChar("M", "M", 'M') returns true
```

```
samePosChar("aaaMaaaM", "abcMcbaM", 'M') returns true
```

```
samePosChar("MaMbMcM", "MtMoMpMdef", 'M') returns true
```

```
samePosChar("MaMbMcM", "MaMbMcMdMe", 'M') returns false
```

```
samePosChar("MMMM", "MM", 'M') returns false
```

```
samePosChar("aMbMcMdM", "MaMbMcMd", 'M') returns false
```

8.8 Exercise: string matching

<http://chortle.ccsu.edu/java5/Notes/chap75/progExercises75.html>

Kjell's Exercise 3 is given here.

Compare every character of `strA` against every character of `strB` and if the characters in the same positions are the same, return `true`.

In addition, the wild character in either string “matches” any single character at the same position in the other string. If both strings have this wild character in the same position, the characters match.

Write a recursive function to implement this string matching using the function prototype

```
bool matches(const std::string& strA, const std::string& strB,
             char wildCharacter);
```

- You may use helper functions e.g. as shown in 8.1 for the function `reverseInPlace` which is overloaded.
- Do not allocate any extra memory, e.g. do not use the function `substr`.
- Do work with indices as you process the strings.
- No loops.
- The only method (function) of the class `std::string` that you can use is `length`.

Examples

```
matches( "L/NGARA", "LANGAR/", '/') returns true
```

```
matches( "!!!", "!!!", '!' ) returns true
```

```
matches( "##O###cake", "##A###cake", '#') returns false
```

```
matches( "#a#b#c#d", "m#n#o#p#", '#') returns true
```

8.9 Exercise: equals ignoring vowels

<http://chortle.ccsu.edu/java5/Notes/chap75/progExercises75.html>

Kjell's Exercise 6 is given here.

Change the Java definition of `equals` for Java Strings so that vowels are ignored. (A vowel could be an upper case vowel and it could be a lower case vowel.) Now "kangaroo" equals "kongeroo", both of them equal "kaangaro", and also equal "kngr".

The easy way to program this would be to first strip out all vowels and then compare what is left. But for this exercise, write a recursive function that does the comparison on the unaltered strings. You may overload the function prototype given below to add the lengths of the strings as extra parameters: see helper functions if you need them. Do not create any substrings nor allocate any extra memory (as in extra strings or extra arrays – you may certainly have local variables that are counters).

Implement the `equals` function in C++ ignoring vowels using the function prototype

```
bool equalsNoVowels(const std::string& strA, const std::string& strB);
```

Note that `strA` and `strB` need not have the same number of characters and they may not even have characters at all (think, of the possibility of an empty string).

9 Recursion with a Collector Variable

The recommended recursive function for implementing the equation (2) is in Listing 1, repeated here

```
int add(int n) {
    if (n == 0) {
        return 0;
    }
    return n + add(n - 1);
}
```

I would loosely categorize `add` as *functional* in the sense of the *functional programming paradigm*. The sum is computed and returned and the parameters are not modified: the parameters are passed-by-value and not passed-by-reference. (Sure, I’m simplifying matters a bit but that’s OK).

Contrast with the logic programming language Prolog that also uses recursion but needs a parameter to *collect* the result in. To illustrate this form of recursive programming, let’s implement equation (2) with a collector variable.

```
// Add the positive integers up to and including n
// n + n-1 + n-2 + ... 3 + 2 + 1
// precondition: n >= 0
int add(int n);
int addCollector(int n, int sum); // helper function

int add(int n) {
    return addCollector(n, 0);
}
int addCollector(int n, int result) {
    if (n == 0) {
        return result;
    }
    return addCollector(n - 1, result + n);
}
```

We could pass parameters-by-reference when computing recursively in C++. I find it ugly to return the result in a parameter that has been modified (a parameter that was passed-by-reference). However, there are cases where we need a parameter that is passed-by-reference. I only use that when I have to (best to avoid in general).

10 Epilogue

If you have gotten this far and if you are a computer scientist, you might be still be looking for those examples that are recursive by nature and that can be solved more simply with recursion: the “Towers of Hanoi” and the “Fibonacci numbers” (true, the Fibonacci numbers are obscenely inefficient when programmed recursively but at least, they provide a nice example to point out problems with recursion).

And backtracking....aren't some solutions so much neater when programmed recursively as opposed to using an explicit stack? The answer is yes. The Standard Template Library STL in C++ provides a stack so using it is easy (and writing one from scratch is simple too) but, I find that the solution to a problem that is being solved with backtracking often looks more elegant if programmed recursively than with an explicit stack.

And what about recursive data structures? Surely they deserve at least a section or two? Sorry to disappoint. The purpose of this write up is to give students lots of practice in writing (and thinking?) recursively. Some of the juicy recursive examples and data structures are covered in class at Langara.

Sedgewick and Wayne in [SW13] have a beautiful write up on recursion with examples and exercises. Do work through that online document if you can.

I would like to acknowledge my colleagues Dr. Kim Lam and Bryan Green for input on exercises.

References

- [Dro13] Adam Drozdek. *Data Structures and algorithms in C++*. Cengage Learning, fourth edition, 2013.
- [GTM11] Michael T. Goodrich, Roberto Tamassia, and David Mount. *Data Structures and Algorithms in C++, 2nd Edition*. John Wiley & Sons, Incorporated, 2011.
- [Kje] Bradley Kjell. *Part 12 Recursion*, chapter <http://chortle.ccsu.edu/java5/index.html>. Central Connecticut State University.
- [Sav13] Walter Savitch. *Absolute C++*. Pearson Education, Inc, 5th edition edition, 2013.
- [Sav15] Walter Savitch. *Problem solving with C++*. Pearson Education, Inc, 2015.
- [SW13] Robert Sedgewick and Kevin Wayne. *2.3 Recursion*, chapter <http://introcs.cs.princeton.edu/java/23recursion/>. Addison-Wesley, 2013.
- [Wat14] Waterloo. Working with recursion. In *CS135: Designing Functional Programs*, chapter <https://www.student.cs.uwaterloo.ca/~cs135/handouts/06-recursion-post3up.pdf>. Computer Science at the University of Waterloo, 2014.
- [Wir85] Niklaus Wirth. *Algorithms and Data Structures*. <http://content.yudu.com/Library/A1bb2l/AlgorithmsandDataStr/resources/82.htm>, 1985.