Tutorial 3

MATH10017

27-28 Oct 2022

Contents

Instructions. Implement all of the sample code and do the exercises.

- Make sure your code compiles.
- Don't copy and paste from the .pdf file, rewrite the code.
- Try to complete the first two sections and one problem from Section 3. Do more if you can.

Reminder CW1 was just announced. We will talk more about it next week.

1 More about `return` statements

One way functions can communicate is through *arguments* and *return values*. (The other way is through *external variables*.)

Remember, a function definition has the form:

```
return-type function-name(argument declarations)
{
  declarations and statements
}
```

Some parts may be absent: the function

```
void dummy(){}
```

does nothing and returns nothing.

If return-type is not void (e.g. `int`, `double`, we'll see more types later), then there must be a `return` statement in the function:

```
return expression;
```

When a function is used in your code, we say it is **called**, and the function that calls it is the **caller**. For instance, in

```
double square(double x)
{
  return x * x;
}

int main()
{
  square(2.0)
}
```

the function `square` is called by the function `main`, so `main` is the caller of `square`.

The `return` statement does two things:

1. if there is an expression after `return`, it return the value of that expression to the caller

2. it returns **control** to the caller, which means a function stops as soon as a return statement is reached.

A return statement can occur anywhere in a function definition, though for simple functions it is at the end. In the next section, we will examples of code with one return statement at the end, and equivalent code with multiple return statements.

2 Recursion

A recursive function is a function that calls itself in its definition.

2.1 Factorial

An example of a recursive function is *factorial*: for a non-negative integer n, we define $0! = 1$ and $n! = n \cdot (n-1)!$ if $n > 0$. Thus:

$$4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 4 \cdot 3 \cdot 2 \cdot 1.$$

We can convert this into C code as follows:

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

Exercise 1. Create a file called "factorial.c" and put the previous code in it. Write a loop in `main()` to print the first 10 values of $n!$.

So you can see another way of writing recursive code, here is an alternate version of the code using multiple return statements:

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

If we call `factorial(3)`, it terminates by returning `3*factorial(2)`. But before control is returned to the caller, the expression `3*factorial(2)` must be evaluated. Similarly, before `factorial(2)` can return control to `factorial(3)`, it must wait for `factorial(1)` to return control. So we have a growing stack of functions that shrinks once we get to $n = 0$:

```
factorial(3)
3 * factorial(2)
3 * (2 * factorial(1))
3 * (2 * (1 * factorial(0)))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * (2)
6
```

2.2 Fibonacci numbers

The nth Fibonnaci number is defined recusively as follows:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1.$$

Exercise 2. Create a file called fibonacci.c.

- Write a function that computes the nth Fibonacci number using recursion.
- Write a loop in `main()` to print the first 10 Fibonacci numbers.

```
int fibonacci(int n)
{
  int fibn;
  if (n == 0){
    fibn =  0;
  }
  else if (n == 1){
    fibn = 1;
  }
  else {
    //fibn = YOUR CODE HERE
  }
  return fibn;
}
```

Again, here is a version using multiple return statements:

```
int fibonacci(int n)
{
  if (n == 0){
    return 0;
  }
  else if (n == 1){
    return 1;
  }
  else {
    //return YOUR CODE HERE
  }
}
```

3 Replacing recursion with loops

Sometimes, it is possible to replace a recursive algorithm with a loop. This can reduce memory usage and the number of function calls.

3.1 Factorial

The code for computing $n!$ with a loop is similar to the code for computing a sum:

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

Exercise 3. Enter the code above in "factorial.c" and check that it computes the same values as the recursive version.

3.2 Fibonnaci

To compute a Fibonacci number, we need to know the previous two numbers. Storing "state" values often allows you to solve problems with loops; we will see this when we find maximum and minimum values.

```
// returns the n-th Fibonacci number
int fib_loop(int n)
{
  int a = 0, int b = 1;
  int temp;

  for(int i = 0; i < n; i++){
    temp = a;
    a = b;
    b = temp + b;
  }
  return a;
}
```

Exercise 4. Put the above code in "fibonacci.c". Check that it computes the same values as the recursive version.

What happens if we write

```
a = b;
b = a + b;
```

instead of using the temporary variable?

(Note, some languages like Python let you do this without a temporary variable, e.g. `a, b = b, a+b`.)

4 Exercises

4.1 Pascal's triangle

The numbers below are rows 0 to 4 of *Pascal's triangle*.

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.

Let n denote the row of the triangle and let $0 \leq k \leq n$ denote the position in row n. (Note: n starts at 0.)

Exercise 5. Create a file called "pascal.c"

- Write a recursive function of to compute the kth term of the nth row of Pascal's triangle.
- Use a double loop to print the first 7 rows of Pascal's triangle, each row on a different line, with the numbers separated by spaces.
- Bonus: make a function that prints out the first N rows of Pascal's triangle, and make it look like a triangle. (Hint use `%3d` to print all integers as if they had at least three digits and adjust the spaces. You'll need a loop to get the blank space in front of the 1's to look right.)

4.2 Another linear recursion

Define a function f by

1. $f(n) = n$ if $n < 3$.
2. $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$.

This function is similar to the Fibonacci numbers: it defines a sequence of numbers recursively as a linear combination of previous terms of the sequence.

Exercise 6. Create a file called "linear_recurrance.c".

1. Write a function `int f(int n)` that computes the function f using recursion.
2. Write a function `int f_loop(int n)` that computes the function f using a loop.

Hint: for part 2, if $n < 3$ return n, otherwise, use the ideas from the loop for computing Fibonacci numbers. Keep track of state variables `f0, f1, f2`, with initial values $f(0) = 0, f(1) = 1, f(2) = 2$.

4.3 Approximating $\sin(x)$ using recursion

For small values of x, the value of $\sin(x)$ is approximately x: $\sin(x) \approx x$. The following identity allows us to compute $\sin(x)$ in terms of $\sin(x/3)$:

$$\sin(x) = 3\sin(x/3) - 4\sin^3(x/3).$$

Each time we substitute this formula, we replace x by $x/3$; after n substitutions, we just need to compute $\sin(x/3^n)$ and for n large, the approximation $\sin(x/3^n) \approx x/3^n$ is very good.

Exercise 7. Create a file "sine.c" and copy in the following code.

- Complete the definition of `sine` using recursion
  - If $x < 0.0001$, return x.
  - Otherwise, use the identity for $\sin(x)$ in terms of $\sin(x/3)$.
- How many times does `sine` get called to compute $\sin(12.5)$?

```
#include <stdio.h>
#include <math.h>

double cube(double x)
{
  return x * x * x;
}

double sine(double x)
{
// YOUR CODE HERE
}

int main()
{
  double x = 1.4;
  printf("We compute sin(%f) = %f.\n", x, sine(x));
  printf("C computes sin(%f) = %f.\n", x, sin(x));
}
```

Note: we have called our function "sine" instead of "sin" so that it will not be mixed up with the built in "sin" function in the math.h library.

To compile this code, you may have to add -lm to the end of your compile command:

```
gcc sine.c -o sine.exe -lm
```

4.4 Counting change

How many ways can you make change for 1 pound (100p) using 50p, 20p, 10p, 5p, and 1p coins? We can answer this recursively: it is

1. the number of ways to change 1 pound using 20p, 10p, 5p, and 1p coins, plus

2. the number of ways to change 50p using 50p, 20p, 10p, 5p, and 1p coins.

More generally, the number of ways to change an amount a using n kinds of coins is

1. the number of ways to change a using all but the first kind of coin, plus

2. the number of ways to change $a - d$ using all of the coins, where d is the value of the first kinds of coin.

This step can be applied recursively. There are three degenerate cases we must cover:

1. If a is 0, there is 1 way to make change.

2. If a is less than 0, there are 0 ways to make change.

3. If the number of coins is 0, there are 0 ways to make change.

Exercise 8. Create a file called "count_change.c". Copy in the following code and complete the function `cc` using the recursive algorithm described above.

```
#include <stdio.h>

/*
 * "highest_value"
 * Return the highest value when you have n kinds of coins.
 */
int highest_value(int n){
  if (n == 5)
    return 50;
  else if (n == 4)
    return 20;
  else if (n == 3)
    return 10;
  else if (n == 2)
    return 5;
```

```
  else if (n == 1)
    return 1;
  else
    return 0; // this case shouldn't happen!
}

/*
 * "cc"
 * helper function for count_change
 * PUT RECURSIVE CODE IN THIS FUNCTION
 */
int cc(int amount, int kinds_of_coins){
  // YOUR CODE HERE
}

/*
 * "count_change"
 * Returns the number of ways to change "amount" with 5 coins.
 */
int count_change(int amount){
  return cc(amount, 5);
}

int main()
{
  printf("The number of ways to change 1 pound is %d.\n", count_change(100));
}
```