Functions – for better code organization and code reuse

Functions

C++ has many built in functions. For example, in the math library the following functions are offered:

https://cplusplus.com/reference/cmath/

A short down list of functions used frequently is:

```
// Returns base raised to the power exponent.
double pow(double base, double exponent);

// Returns the square root of x.
double sqrt(double x);

// Returns the natural logarithm of x.
double log(double x);

// Returns the absolute value of x.
double fabs(double x);
```

In a competitive programming competition, competitors always have access to the language documentation and should look up functions to use. Here is a short program that reads in a, b and c from the quadratic equation and prints out the roots (in sorted order) if they are real, or "no real roots" otherwise. Note that we include cmath so we have access to the pow, sqrt functions.

```
using namespace std;
#include <iostream>
#include <cmath>

int main() {
    double a, b, c;
    cin >> a >> b >> c;

    double disc = pow(b,2) - 4*a*c;

    if (disc < 0)
        cout << "no real roots" << endl;
    else {
        double r1 = (-b-sqrt(disc))/(2*a);
        double r2 = (-b+sqrt(disc))/(2*a);
        cout << r1 << " " << r2 << endl;
    }

    return 0;
}
```

As we can see here, just like in other programming languages, to call a function, use its name, and for each of the parameters, put expressions that are the correct type, but don't include the type information in the function call. If the function returns something, then typically the function call will be a portion of a line of code, where, an expression of the return type would be appropriate to see. In the case of this program, we call pow as a portion of an expression in an assignment statement and sqrt in the same context. Once these function calls return values, those values are then used to evaluate the greater expression, followed by assigning the appropriate variable (because of the assignment statement).

Much of what allows us to write programs quickly is using functions that others have already written, so that we can simply **call** those functions and use them without fully thinking through and coding the logic of those functions on our own. In addition, not only can we call C's built in functions, but if C doesn't have a function built in, we can write our own!

Using C's functions AND writing our own give us several advantages:

1. Code Reuse – Imagine if we had to find the square root of 5 separate expressions during the course of our program, and that 10 lines of code were involved with finding a square root. Then, we would have to write 50 lines of code! But, since someone has written the 10 lines for us, we just write 1 line, the call to the sqrt function which invokes those 10 lines to be written. This shortens the overall code.

2. Limiting Scope of what's active – when programming, we have to keep track, simultaneously in our head, of each variable at each point in time while a function is running. The longer our functions the more difficult is it to keep track of everything we need to know at a given point in time. In order to limit function length but still achieve complicated tasks, we must write multiple functions! This reduces the probability that we'll "lose track of some variables" and make an error coding.

3. Aiding Program Design and Organization – functions symbolically represent fixed specific tasks that can be reused. Breaking our program conceptually into different subtasks will help us solve the problem in a way where we can mix and match the pieces as needed. Forcing ourselves to think about how problems subdivide into smaller discrete problems helps us solve problems and helps us plan what functions we will need and what those functions should do. Once we have this plan, then it becomes fairly straightforward to fill each of the functions and write main to call those functions as necessary.

4. Simplifying Debugging – if we carefully test each function and know that it works, then we can be confident when we call that function, it does what we want. If for some reason it has an error, then we just have to go one place to fix it. If, on the other hand, we rewrote the same buggy code in 5 places in main, then we have to go to 5 separate places to fix it. This is decidedly harder and more error prone!!!

Let's look at a simple example where we design a program with one function to solve a problem on Kattis. This problem can be solved without extra functions, but this is to illustrate the syntax needed to define your own function:

Function Example: GCVWR

Here is the problem statement:

https://open.kattis.com/problems/gcvwr

A subtask necessary to solve the problem is to add up ***n*** numbers from standard input. We can write a function that takes in one formal parameter, ***n***, and then proceeds to read in ***n*** values from standard input, add these up and return this value. Here is the function:

```
int sum(int n) {
    int res = 0, x;
    for (int i=0; i<n; i++) {
        cin >> x;
        res += x;
    }
    return res;
}
```

Notice that when we write a function, we give it a name, put the return type on the left (void means returning nothing), and then create a formal parameter list, which is a list of information the function needs to do its task. Each thing on this list is a type followed by a variable name. If there is more than one thing, then we separate the items by commas. Once we're in the function, we assume that n already has a value that was given to us by whoever called the function. Our job is to use that value to complete our subtask. In this case, that means reading in n integers from standard input and adding them, followed by returning that value. The return statement is just like you might expect it: the keyword return followed by the expression you want to return, followed by a semicolon.

Now, let’s take a look at the whole program and how it's laid out on the next page:

```
using namespace std;
#include <iostream>

// We must always put all function prototypes up here.
int sum(int n);

int main() {

    // Read input.
    int cap, carW, numItems;
    cin >> cap >> carW >> numItems;

    // Calculate free weight.
    int res = (int)((cap - carW)*.9 - sum(numItems));

    // Output result.
    cout << res << endl;

    return 0;
}

// Reads in n values from standard input, adds them and returns
// the sum.
int sum(int n) {

    // Loop and add into our accumulator.
    int res = 0, x;
    for (int i=0; i<n; i++) {
        cin >> x;
        res += x;
    }

    // Ta da!
    return res;
}
```

Notice that we call the function exactly the same as we call a pre-defined library function. But, we have to add a function prototype (first line of the function, roughly) to the top of the program. This is so when the compiler sees the function call, it can access if the function call is syntactically correct. We can place our functions, in any order, after main. Also note that our actual parameter, what we put in the slot when we **call** the function need not be the same as the formal parameter. In this case, the actual parameter is "numItems" but the formal parameter is "n". In fact, while formal parameters must be variables, actual parameters can be any expression of the appropriate type.

Let's look at another example with trapezoid area as our key function:

https://open.kattis.com/problems/taisformula

For our solution, we have to calculate and add the area of several trapezoids. If we write a function to calculate the area of one trapezoid, then we can call that for each trapezoid we'd like to add into our area sum. Notice, that in order to get the necessary precision to get a correct solution on Kattis, we had to add digits via the setprecision function in the iomanip library:

```
using namespace std;
#include <iostream>
#include <iomanip>

// Returns the area of a trapezoid with height h, and two bases, b1 and b2.
double trapArea(double h, double b1, double b2);

int main() {

    // Read # of points.
    int n;
    cin >> n;

    // Get first reading.
    double oldT, oldG;
    cin >> oldT >> oldG;

    double res = 0;

    // Loop through rest.
    for (int i=0; i<n-1; i++) {

        // Get next reading.
        double newT, newG;
        cin >> newT >> newG;

        // Add area.
        res += trapArea( (newT-oldT)/1000.0, oldG, newG);

        // For the next iteration, these become the old readings.
        oldT = newT;
        oldG = newG;
    }

    // Ta da!
    cout << setprecision(9) << res << endl;

    return 0;
}

// Returns the area of a trapezoid with height h, and two bases, b1 and b2.
double trapArea(double h, double b1, double b2) {
    return (b1+b2)*h/2;
}
```

Solve Function Idea

In competitive programming problems with multiple independent cases, one possible error one can make is to forget to reset variables to their initial values, so that a program that works on the first case, may give an incorrect answer to future cases. A simple way to avoid this error is to keep main very, very simple, where it just has one loop to process the cases, and each case is solved by a call to a function (which, if you want, you can call solve.)

Here is an example problem on Kattis that has multiple cases to read in, Speed Limit:

https://open.kattis.com/problems/speedlimit

In this problem, the first line of each test case is the number of segments on a drive. Then, the number of lines follow describing the segments. The input can have more than one test case. The end of input is designated with a value of -1 for the number of segments.

Thus, we can design our code to have a solve function that takes in the number of segments and returns the result of the test case:

```
int solve(int n) {
    // Solution to one case here.
}
```

and we can design our main as follows:

```
int main() {
    int n;
    cin >> n;

    while (n != -1) {
        cout << solve(n) << " miles" << endl;
        cin >> n;
    }

    return 0;
}
```

Note that the problem description requires us to output the string " miles" after the answer to each test case.

In this problem, all we have to do is calculate the length of each segment. The input is given in a slightly funny way, in that the hours given for driving are the total driving hours and not just the driving hours for that particular segment. Thus, we must subtract successive times to get the actual segment driving time. Here is the full solution to the problem, with a solve function:

```
using namespace std;
#include <iostream>

int solve(int n);

int main() {
    int n;
    cin >> n;

    while (n != -1) {
        cout << solve(n) << " miles" << endl;
        cin >> n;
    }

    return 0;
}

int solve(int n) {

    int curT = 0, res = 0;
    for (int i=0; i<n; i++) {

        int mph, t;
        cin >> mph >> t;
        res = res + (t-curT)*mph;
        curT = t;
    }

    return res;
}
```