Programming Practicum Report: Meeting #3

R. Ethan Halim

September 10th, 2024

1 Quadratic Equation Solver

The entire source file is hosted on a GitHub repository **here**.

1.1 Explanation

As the program is to solve the value of x given a quadratic equation, firstly, the program must receive input from the user about the coefficients, which are stored in variables `a`, `b`, and `c`.

```
int program(std::istream& cin, std::ostream& cout) {
    double a, b, c;
    cout << "Insert `a`: ";
    cin >> a;
    cout << "Insert `b`: ";
    cin >> b;
    cout << "Insert `c`: ";
    cin >> c;

    ...
}
```

$$ax^2 + bx + c = 0$$

$$\text{discriminant} = D \equiv b^2 - 4ac$$

$$x = \frac{-b \pm \sqrt{D}}{2a}$$

The rest of the program calculates the value of x by the quadratic formula, as detailed above. However, as the program needs to inform the user regarding whether the value(s) of x may be two in amount, solely one, or complex, the intermediate discriminant value is calculated, which is used to check for all of the three cases, as the comments in the code below entail.

```
int program(std::istream& cin, std::ostream& cout) {
    ...

    double discriminant = std::pow(b, 2.0) - 4.0 * a * c;
    // If D > 0, then both roots are distinct and real.
    if (discriminant > 0) {
        double x1 = (-b + std::sqrt(discriminant)) / (2.0 * a);
        double x2 = (-b - std::sqrt(discriminant)) / (2.0 * a);

        cout << "\nThe roots are distinct: " << x1 << " and "
        ↪  << x2 << ".\n";
    }
    // If D == 0, then both roots are the same.
    else if (discriminant == 0) {
        double x = (-b + std::sqrt(discriminant)) / (2.0 * a);

        cout << "\nThere is only one real root: " << x <<
        ↪  ".\n";
    }
    // If D < 0, then no real roots exist.
    else {
        cout << "\nThe roots are imaginary.\n";
    }

    return 0;
}
```

1.2 Manual Testing

Below is the compilation and the testing of the source code.

```
avaxar@AvaxarTUF:~/Repos/uni-practica-1/week_3/01_quadratic$ make
g++ -Wall quadratic.cpp -o quadratic
./quadratic
Insert `a`: -1
Insert `b`: 6
Insert `c`: 3

The roots are distinct: -0.464102 and 6.4641.
```

1.3 Test Cases

1.3.1 Tests

Below is copied directly from the `tests.txt` file. The first two test cases test for the scenario where there are two distinct real roots; the second two test cases test for the scenario where there is only one single real root; and the third two test cases test for the scenario where the roots are complex.

```
%INPUT
-1
0
1
%OUTPUT
Insert `a`: Insert `b`: Insert `c`:
The roots are distinct: -1 and 1.
%END

%INPUT
-2
4
6
%OUTPUT
Insert `a`: Insert `b`: Insert `c`:
The roots are distinct: -1 and 3.
%END

%INPUT
1
2
1
%OUTPUT
Insert `a`: Insert `b`: Insert `c`:
```

```
There is only one real root: -1.
%END

%INPUT
-4
-4
-1
%OUTPUT
Insert `a`: Insert `b`: Insert `c`:
There is only one real root: -0.5.
%END

%INPUT
100
200
400
%OUTPUT
Insert `a`: Insert `b`: Insert `c`:
The roots are imaginary.
%END

%INPUT
-400
-200
-100,
%OUTPUT
Insert `a`: Insert `b`: Insert `c`:
The roots are imaginary.
%END
```

1.3.2 Execution

Below are the results of the test cases. No test cases failed.

```
avaxar@AvaxarTUF:~/Repos/uni-practica-1/week_3/01_quadratic$ make clean
rm -f quadratic quadratic_test
avaxar@AvaxarTUF:~/Repos/uni-practica-1/week_3/01_quadratic$ make test
g++ -Wall -g quadratic.cpp -o quadratic_test -DTEST
./quadratic_test
[*] The program is currently in test mode!

[*] Running test #1 with the input...
-1
0
1
[*] Test ran successfully.

[*] Running test #2 with the input...
-2
4
6
[*] Test ran successfully.

[*] Running test #3 with the input...
1
2
1
[*] Test ran successfully.

[*] Running test #4 with the input...
-4
-4
-1
[*] Test ran successfully.

[*] Running test #5 with the input...
100
200
400
[*] Test ran successfully.

[*] Running test #6 with the input...
-400
-200
-100,
[*] Test ran successfully.

[*] All tests passed.
```

2 Maximum Number Finder

The entire source file is hosted on a GitHub repository **here**.

2.1 Explanation

The program is to figure out the maximum value of three given numbers. Firstly, it requests input from the user of those three values, stored as `n1`, `n2`, and `n3`.

```
int program(std::istream& cin, std::ostream& cout) {
    double n1, n2, n3;
    cout << "Input the first number: ";
    cin >> n1;
    cout << "Input the second number: ";
    cin >> n2;
    cout << "Input the third number: ";
    cin >> n3;

    ...
}
```

It then goes through a series of comparisons between the values to see and reassign the greatest values among them. The variable `max` is initially assigned to `n1`, which is then subsequently checked to the value of `n2` and `n3` to see which one is greater; figuratively, it is alike to a staging match among each pair of two values. The greatest of the three numbers is then printed out.

```
int program(std::istream& cin, std::ostream& cout) {
    ...

    // Generally, using a loop to iterate over all of the
    ↪  remaining elements within an array to check the
    ↪  maximum would be better, in order to reduce code
    ↪  duplication. However, since there are only three
    ↪  elements in total, stored in separate variables,
    ↪  unwound if-statements are used instead.
    double max = n1;
    if (n2 > max) {
        max = n2;
    }
    if (n3 > max) {
        max = n3;
    }
```

```
    cout << "\nThe maximum is " << max << ".\n";
    return 0;
}
```

2.2 Manual Testing

Below is the compilation and the testing of the source code.

```
avaxar@AvaxarTUF:~/Repos/uni-practica-1/week_3/02_max_number$ make
g++ -Wall max_number.cpp -o max_number
./max_number
Input the first number: 3
Input the second number: 5
Input the third number: 4

The maximum is 5.
```

2.3 Test Cases

2.3.1 Tests

Below is copied directly from the `tests.txt` file.

```
%INPUT
4
5
6
%OUTPUT
Input the first number: Input the second number: Input the
↪  third number:
The maximum is 6.
%END

%INPUT
324
32478
54
%OUTPUT
Input the first number: Input the second number: Input the
↪  third number:
The maximum is 32478.
%END

%INPUT
3273
```

```
1982
983
%OUTPUT
Input the first number: Input the second number: Input the
↪  third number:
The maximum is 3273.
%END

%INPUT
53434
-23234
21245
%OUTPUT
Input the first number: Input the second number: Input the
↪  third number:
The maximum is 53434.
%END

%INPUT
-34232
24244
24243
%OUTPUT
Input the first number: Input the second number: Input the
↪  third number:
The maximum is 24244.
%END

%INPUT
-111111
-333333
-222222
%OUTPUT
Input the first number: Input the second number: Input the
↪  third number:
The maximum is -111111.
%END
```

2.3.2 Execution

Below are the results of the test cases. No test cases failed.

```
avaxar@AvaxarTUF:~/Repos/uni-practica-1/week_3/02_max_number$ make clean
rm -f max_number max_number_test
avaxar@AvaxarTUF:~/Repos/uni-practica-1/week_3/02_max_number$ make test
g++ -Wall -g max_number.cpp -o max_number_test -DTEST
./max_number_test
[*] The program is currently in test mode!

[*] Running test #1 with the input...
4
5
6
[*] Test ran successfully.

[*] Running test #2 with the input...
324
32478
54
[*] Test ran successfully.

[*] Running test #3 with the input...
3273
1982
983
[*] Test ran successfully.

[*] Running test #4 with the input...
53434
-23234
21245
[*] Test ran successfully.

[*] Running test #5 with the input...
-34232
24244
24243
[*] Test ran successfully.

[*] Running test #6 with the input...
-111111
-333333
-222222
[*] Test ran successfully.

[*] All tests passed.
```

3 Explanation of Branching in C/C++

In C/C++, like any other conventional programming languages, there are statements which divert the flow of code into separate "branches" depending on the satisfaction of certain conditions. This allows the program to handle multiple possibilities/paths that the user may take, or handle certain conditional cases within any algorithms therein. Akin to a tree, these multiple paths are considered branches; there may be branches within branches; and branches may loop around itself.

3.1 Goto Statements

Goto-statements are the barest form of branching, as they enable the programmer to directly jump into certain parts of a function; they perform **unconditional jumps**. In terms of their usage, a function is to have at least one label, onto which the goto-statement can jump. Goto-statements are the closest that C/C++ offers to Assembly's `jmp` instruction.

```
goto my_label;
a();
my_label:
b();
```

In the example above, `goto my_label;` will jump onto the part preceding `b();`. Thus, `a();` will never get run. In the example below, `goto my_label;` will jump onto `my_label`, which is placed before the goto-jump. Therefore, this will produce an infinite loop, constantly executing `a();`.

```
my_label:
a();
goto my_label;
```

In most cases however, the usage of goto-statements is discouraged, because it is known to be hard to optimize by compilers, which would produce slower binaries. Moreover, there are most likely alternatives which utilize existing constructions provided by the language instead, the usage of which aids readability for the programmers and ease of optimization for the compiler.

3.2 Conditional Statements

In this category, statements that perform conditional branching are included. They check for certain satisfied or unsatisfied conditions, and act accordingly as instructed by the programmer. This is the most common form of branching.

3.2.1 If-Else Statements

If statements are the most commonly-used forms of branching, as they take an expression which either returns `true` or `false`, both of which determine whether the enclosed code within the statement is to be executed or not. Below is the boilerplate for an if statement.

```
if (condition) {
    code();
}

rest();
```

If the given expression (represented by the variable `condition`) returns `true`, then it will execute the code enclosed within the brackets after it; in this case, it will call the function named `code`. Regardless of the truth value of `condition` (i.e. whether the if-statement is satisfied), the code following the conclusion of the if-statement will be run, that is the function named `rest` in this case, which resumes back onto the main branch of the code.

In order to handle both cases where `condition` returns either `true` or `false`, with the focus being on the latter, an else-statement may be put following the if-statement. This splits the main branch into two subbranches that the code execution may take.

```
if (condition) {
    planA();
}
else {
    planB();
}
```

In the code above, if the `condition` is satisfied, then `planA` will be run; if it isn't, then `planB` will be run instead. If `planA` and `planB` were return statements within a value-returning function, then the said function would be mathematically similar to the formula below.

$$f(...) = \begin{cases} a() & \text{if } condition \\ b() & \text{if } \neg condition \end{cases}$$

Additionally, multiple if-statements can be chained together to check for and handle multiple conditions, as written below for instance.

```
if (conditionA) {
    planA();
}
else if (conditionB) {
    planB();
}
else if (conditionC) {
    planC();
}
else {
    planD();
}
```

However, if the conditions above involved the equality of the same expression with different integers or characters, a switch-statement would be more appropriate for such use case.

3.2.2 Switch Statements

Switch-statements are similar to a chain of if-statements which check for multiple conditions when it comes to checking for and handling multiple equality conditions between a given expression and the provided cases. Thus, this snippet of code involving the switch-statement below ...

```
switch (x) {
    case 1:
        planA();
        break;
       case 2:
        planB()
        break;
       case 3:
        planC();
        break;
       default:
        planD();
}
```

... is equivalent to the chain of if-statements below ...

```
if (x == 1) {
    planA();
}
```

```
else if (x == 2) {
    planB();
}
else if (x == 3) {
    planC();
}
else {
    planD();
}
```

... and is similar to the mathematical formula below.

$$f(...) = \begin{cases} a() & \text{if } x = 1 \\ b() & \text{if } x = 2 \\ c() & \text{if } x = 3 \\ d() & \text{if neither} \end{cases}$$

In order to add more cases, one can add more `case` labels followed by the integer or character constant they please and a colon. `break;` statements following the conclusion of each case are necessary as to quit the switch-statement upon case-completion. Otherwise, code execution would run into other cases, which may not be desirable. This code snippet below ...

```
switch (x) {
    case 1:
        planA();
    case 2:
        planB()
}
```

... would be equivalent to the code below, as the execution from `planA();` would spill over downwards, with nothing halting it preceding the `case 2:` label.

```
if (x == 1) {
    planA();
    planB();
}
else if (x == 2) {
    planB();
}
```

The reason why one may need to prefer switch-statements over if statement is because switch-statement offer more performance, as the compiler is able to recognize and perform optimizations in a way which does not involve having to check for every single given condition, as how an if-statement would be handled. The resulted binary would receive gains in performance.

3.3 Looping

Loops are parts of code that are executed repeatedly (multiple times) in a sequential manner, given a condition which defines how many times the given code should be run, or until when the looping should be stopped. On each iteration, the code may receive different inputs or interact with the changes made by its previous iterations.

3.3.1 While Loops

A while-loop is the most basic form of loops. Given the satisfaction of a condition, the code within will always be run over and over until the said condition is no longer satisfied; from which, code execution will resume out of the loop.

```
while (condition) {
    code();

    // Upon the end of the code, `condition` is reevaluated of
    ↪  its truth value, in order to determine whether the
    ↪  loop should continue.
}
```

In the example above, the value of `condition` will be determined of its truth value (either `true` or `false`). If it is `true`, then `code` will get executed. Upon completion, the `condition` will be checked again and the cycle repeats. If `condition` were set to always be `true`, then it would be an infinite loop with no foreseeable end.

In order to illustrate how while-loops work beneath the abstractions of the language, if the snippet were rewritten using `goto` and `if`, it would look as follows. `start:` and `end` are jump labels representing the start and end of the loop respectively.

```
start:
// If `condition` is no longer satisfied, it will end the
↪  loop.
if (!condition) {
    goto end;
```

```
}

// What is contained in the loop starts here.
code();

goto start; // Loops back for condition-rechecking
end:
```

Midway inside a while-statement, loop-halting and iteration-skipping can be controlled directly using `break;` and `continue;`. `break;` stops and ends the loop immediately regardless of the value of `condition`. `continue;` stops and ends the current iteration of the loop. These two statements are available in all of the loop types in the language and apply similarly.

```
while (condition1) {
    if (condition2) {
        break;       }
    else {
        continue;
    }
}
```

The boilerplate above is equivalent to the code snippet below which illustrates the destination of the jumps made by the two statements using goto-statements.

```
while (condition1) {
    if (condition2) {
        goto loop_end; // formerly `break;`
    }
    else {
        goto iteration_end; // formerly `continue;`
    }

iteration_end:
    // From here, `condition1` will be reevaluated of its
    ↪  truth value, in order to determine whether the loop
    ↪  should continue.
}
loop_end:
```

3.3.2 Do-While Loops

Do-while-loops are similar to while-loops, however the code within is executed on the first iteration regardless of the given condition. They have the syntax as follows ...

```
do {
    code();
} while (condition);
```

... which is equivalent to the code snippet below using a regular while-loop.

```
code();
while (condition) {
    code();
}
```

3.3.3 For Loops

For-loops are loops which act as "condensed" while-loops. They accept an initializing statement, a condition expression, and a post-iteration statement. Their syntax is as follows ...

```
for (INITIAL; condition; POST) {
    code();
}
```

... which is equivalent to the code snippet below using a regular while-loop.

```
INITIAL;
while (condition) {
    code();
    POST;
}
```

Typically, for-loops are used to iterate over a range of numbers. For that reason, the initializing statement tends to be a variable declaration which declares and defines the iterating variable (a.k.a *iterator*), commonly named `i`. The condition after it defines whether the variable is part of the given range. The post-iteration statement increments or decrements the iterator, in order to loop onto the next number in the range. Below is a code example which iterates and prints numbers from 1 to 5.

```
for (int i = 1; i <= 5; i++) {
```

```
    std::cout << i << '\n';
}
```

The iterator starts out at `1` and gets printed to the terminal. The post-iteration statement increments the iterator by 1, and thus the iterator becomes `2`. This repeats until the end of the iteration where the iterator is `5`, whereby the incrementation of the iterator invalidates the loop condition (i.e. `i <= 5` is no longer satisfied), thereby stopping it before `6` would be printed.

3.3.4 For-Each Loops

For-each-loops (alternatively called *ranged-for-loops*) are a loop type which was added in C++11, and is not present in C. They are syntactic sugar which aids in iterating over the elements of a collection (typically of a container type implemented by the C++ STL [standard template library]). For instance, when used with a `std::vector` instance, it will loop over each element starting with the lowest index (from 0). The boilerplate in doing so is as follows.

```
std::vector<int> vec = {9, 7, 5, 3, 1};
for (int e : vec) {
    // Prints "9 7 5 3 1" in sequence
    std::cout << e << ' ';
}
```

In this case, the "iterator [variable]" refers to the variable which is declared to funnel every element of the vector (that is, `e`). In the context of iterating over a `std::vector`, the example usage above offers better readability and general beauty than the alternate implementation below, which solely uses a regular for-loop.

```
std::vector<int> vec = {9, 7, 5, 3, 1};
for (int i = 0; i < vec.size(); i++) {
    int e = vec[i];

    // Prints "9 7 5 3 1" in sequence
    std::cout << e << ' ';
}
```

3.4 Function Calling

```
return_type function([argument_type1 argument1, [argument_type2
↪  argument2, ...]]) {
    code();
```

```
}
```

Functions are pieces of code which may call other functions or themselves, in a way that lets them to be reusable. They enclose technicalities and pack them into more abstract "tools", such that the end-programmer may use them imperatively in the main code without worrying how it works below the hood or rewriting them over and over on demand. They work similarly to functions in mathematics, where they can accept arguments and return a value. The following mathematical function which does a dot product from its given arguments ...

$$dot(ax, ay, bx, by) = ax * bx + ay * by$$

... is equivalent to the code below, where the type of the numeric arguments and return value is expected to be a real number, using `double`.

```
double dot(double ax, double ay, double bx, double by) {
    return ax * bx + ay * by;
}
```

However, functions in programming can do much more than what they can do in math. They do not necessarily have to include a return value (for such case, the return type would be `void`). They enclose programming statements/instructions; they are more akin to procedures. In fact, the `main` function is itself a function, where statements live and are executed. Function calling acts more similarly to code-substitution, as illustrated below.

```
void func() {
    std::cout << "Hello, world!\n";
    std::cout << "I am `func`.\n";
}

int main() {
    func()
}
```

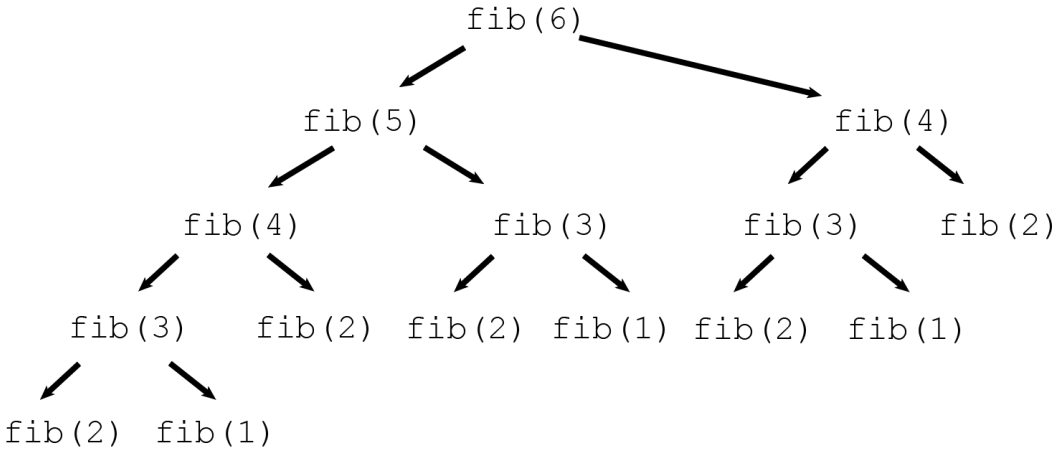
```
int main() {
    // Calling `func` is similar to writing directly below.
    std::cout << "Hello, world!\n";
    std::cout << "I am `func`.\n";
}
```

In practical use, they help uphold the DRY ("do not repeat yourself") rule, as repetitive similar blocks of code cluttered around a codebase can be refactored in a way such that the said code is only written once. As they can call themselves, *recursion* can be utilized to make code more compact when dealing with tree-related data structures or certain algorithms, such as generating the Fibonacci sequence.

```
int fib(int n) {
    // Base cases, where the recursion will hit an end
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }

    // Calls itself, making two branches
    return fib(n - 1) + fib(n - 2);
}
```

If `fib` is called with its argument as `6` by `fib(6)`, then multiple branching recursions will entail, as illustrated in the tree diagram below, courtesy of **this resource**.



3.5 Exceptions

Exceptions are sudden jumps in code, possibly across functions, made due to unexpected errors that are to be handled. They are a feature exclusive to C++ and does not exist in C. While they are not conventional branchings, they still cause diversion in control flow upon the throwing of an exception. The said throwing is a programmer-written indication that something went wrong within the code, and the code above must handle it, in order to not let the program crash.

3.5.1 Throw Expressions

Take an example where a programmer is writing a function which may fail upon certain circumstances. They would need to be able to indicate and panic out of the function for the end-programmer, in order to let the code that is using the function know. Throw expressions in C++ lets the programmer throw an object, conventionally an instance of the base class `std::exception`, which entails the underlying error. Below is an example function which encapsulates division with a check to detect for any zero-divisions.

```
throw EXPRESSION;
```

```
int divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Zero division occurred");
    }

    return a / b;
```

```
}
```

If `divide` were to be called with the argument `b` set as zero, an exception object—an `std::runtime_error` instance (a subclass inheriting the STL base class `std::exception`), which contains the given error message (`"Zero division occurred"`), would be thrown and the function would immediately be exited along with any other functions that had called it until the exception is caught to be handled, the process of which is explained in the next section.

3.5.2 Catching Exceptions

Following from before, after the exception has been thrown, it needs to be caught; otherwise, if the exception unwound upwards reaching the `main` function, the program would crash. In order to catch exceptions, a try-block must enclose the potential exception-throwing code and a catch-block must be present, which entails the code to deal with the thrown exception object and act appropriately upon the faulty case.

```
try {
    code();
}
catch (VARIABLE_DECLARATION) {
    handler();
}
```

Resuming from the `divide` function implemented before, in order to catch for any errors given out by the function, the function call must be enclosed inside a try-block. The `VARIABLE_DECLARATION` following the `catch` keyword holds a reference to the thrown exception object. In the example below, although what was thrown is an `std::runtime_error` object, since it inherits `std::exception` as its base class, polymorphism lets the object be cast up, exposing the `.what()` method to yield its error message.

```
int result;
try {
    result = divide(1, 0);
}
catch (const std::exception& exc) {
    std::cerr << "Exception was thrown, saying: " << exc.what()
    ↪  << '\n';
}
```

If the code in the catch-block cannot proceed with resolving the issue, then the exception can be rethrown, so that the exception may be passed upwards

to the calling function until it is again handled or the program crashes.

```
catch (const std::exception& exc) {
    // The block cannot handle the exception, thus
    // the exception is rethrown.
    throw;
}
```