1

Introduction

Success is not final, failure is not fatal: it is the courage to continue that counts.

Winston Churchill

This book arose as a result of my fascination with computers and programming with the C++ language. It is also a result of my over 20 years of teaching the basics of computer science and particularly the C++ language to the students of the faculties of electrical engineering as well as mechanical engineering and robotics at AGH University of Science and Technology in Krakow, Poland. I have also worked as a programmer and consultant to several companies, becoming a senior software engineer and software designer, and have led groups of programmers and served as a teacher for younger colleagues.

Learning programming with a computer language is and should be fun, but learning it well can be difficult. Teaching C++ is also much more challenging than it was a decade ago. The language has grown up significantly and provided new exciting features, which we would like to understand and use to increase our productivity. As of the time of writing, C++20 will be released soon. In the book, we use many features of C++17, as well as show some of C++20. On the other hand, in many cases, it is also good to know at least some of the old features as well, since these are ubiquitous in many software projects, libraries, frameworks, etc. For example, once I was working on a C++ project for video processing. While adjusting one of the versions of the JPEG IO libraries, I discovered memory leaks. Although the whole project was in modern C++, I had to chase a bug in the old C code. It took me a while, but then I was able to fix the problem quickly.

The next problem is that the code we encounter in our daily work is different than what we learn from our courses. Why? There are many reasons. One is *legacy* code, which is just a different way of saying that the process of writing code usually is long and carries on for years. Moreover, even small projects tend to become large, and they can become huge after years of development. Also, the code is written by different programmers having different levels of understanding, as well as different levels of experience and senses of humor. For example, one of my programmer colleagues started each of his new sources with a poem. As a result, programmers must not only understand, maintain, and debug software as it is, but sometimes also read poems. This is what creates a discrepancy between the nice, polished code snippets presented in classes as compared to "real stuff." What skills are necessary to become a successful programmer, then?

Why did I write this book, when there are so many programming Internet sites, discussion lists, special interest groups, code examples, and online books devoted to software development?

Introduction to Programming with C++ for Engineers, First Edition. Bogusław Cyganek.

Companion website: http://home.agh.edu.pl/~cyganek/BookCpp.htm

3.2 Example Project – Collecting Student Grades

So far, we have seen how to define and use single variables and constants of different types. However, quite frequently, we need to store many such objects one after the other. Sometimes we do not know how many will be needed. To do this, we can use a data structure called *a vector* (or an *array*).[3] A vector containing four numerical objects is depicted in Figure 3.2.

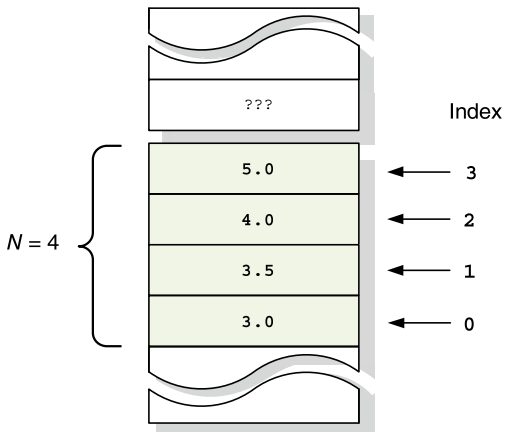

Figure 3.2 A vector is composed of a number of objects occupying contiguous space in memory. Here we have $N = 4$ objects that can store floating-point data. Each cell of a vector can be accessed by providing its index. The first element is at index 0, and the last is at index $N - 1$.

To begin with, a vector is a data structure with the following characteristics:

- It contains objects of the same type
- The objects are densely placed in memory, i.e. one after another (no holes)
- Each object in a vector can be accessed by providing its *index* to the *subscript operator* `[]`. The valid indices are 0 to $N-1$, where N denotes the number of elements in the vector

In C++, there are many ways of creating vectors. The most flexible is to use the `std::vector` object from the SL library, which

- Can be empty at the start
- Can be attached to new objects once they become available
- Contains many useful functions

Before we go into detail about what `std::vector` can do for us, let's start with a simple example program in Listing 3.3. Its purpose is to collect students' grades and compute their rounded average.

3 The term *array* is usually reserved for fixed-sized structures (Section 3.10); *vector* is used for objects with dynamically changing size.

Listing 3.3 A program to collect and compute average student grades (in *StudentGrades*, *main.cpp*).

```
1   #include <vector>       // A header to use std::vector
2   #include <iostream>     // Headers for input and output
3   #include <iomanip>      // and output formatting
4   #include <cmath>        // For math functions
5
6   // Introduce these, to write vector instead of std::vector
7   using std::cout, std::cin, std::endl, std::vector;
8
9
10  int main()
11  {
12
13    cout << "Enter your grades" << endl;
14
15    vector< double >    studentGradeVec; // An empty vector of doubles
16
17    // Collect students' grades
18    for( ;; )
19    {
20      double grade {};
21
22      cin >> grade;
23
24      // If ok, push new grade at the end of the vector
25      if( grade >= 2.0 && grade <= 5.0 )
26        studentGradeVec.push_back( grade );
27
28
29      cout << "Enter more? [y/n] ";
30      char ans {};
31      cin >> ans;
32
33      if( ans == 'n' || ans == 'N' )
34        break;    // the way to exit the loop
35    }
```

To use `std::vector` and to be able to read and write values, a number of standard headers need to be included, as on lines [1–3]. Then, to avoid repeating the `std::` prefix over and over again, on line [7], common names from the `std` namespace are introduced into our scope with the help of the `using` directive. We already know the role of the `main` function, which starts on line [10]: it defines our C++ executable. Then, the central part is the definition of the `vector` object on line [15]. Notice that the type of objects to store needs to be given in `<>` brackets. To be able to store objects, the `vector` class is written in a generic fashion with the help of the template mechanism, which will be discussed in Section 4.7.

Then, on line [18], the `for` loop starts (see Section 3.13.2.2). Its role is to collect all grades entered by the user. However, we do not know how many will be entered. Therefore, there is no condition in `for`: it iterates until the `break` statement is reached on line [34] when the user presses `'n'`, as checked for on line [33].

During each iteration, on line [20], a new 0-valued `grade` is created, which is then overwritten on line [22] with a value entered by the user. If its value is correct, i.e. between 2.0 and 5.0, then on line [26] it is pushed to the end of the vector object `studentGradeVec` by a call of its `push_back` member function. The object and its member function are separated by a dot operator (`.`) – its syntax is presented in Section 3.9, as well as in the description of the *GROUP 2* operators (Table 3.15).

In Polish school systems, we have grades from 2.0 to 5.0, with 3.0 being the first "passing" grade. The valid grades are { 2.0, 3.0, 3.5, 4.0, 4.5, 5.0 }.

The rest of this program is devoted to computing the final grade, which should be the properly rounded average of the grades that have been entered:

```
36    // Ok, if there are any grades compute the average
37    if( studentGradeVec.size() > 0 )
38    {
39      double sum { 0.0 };
40      // Add all the grades
41      for( auto g : studentGradeVec )
42        sum += g;
43
44      double av = sum / studentGradeVec.size();    // Type will be promoted to double
45
46      double finalGrade {};
47
48      // Let it adjust
49      if( av < 3.0 )
50      {
51        finalGrade = 2.0;
52      }
53      else
54      {
55        double near_int  = std::floor( av );    // get integer part
56        double frac      = av - near_int;       // get only the fraction
57
58        double adjust { 0.5 };                  // new adjustment value
59
60        if( frac < 0.25 )
61          adjust = 0.0;
62        else if( frac > 0.75 )
63          adjust = 1.0;
64
65        finalGrade = near_int + adjust;
66      }
67
68      cout << "Final grade: "
69           << std::fixed << std::setw( 3 ) << std::setprecision( 1 )
70           << finalGrade << endl;
71
72    }
73
74
75    return 0;
76  }
```

We start on line [37] by checking whether `studentGradeVec` contains at least one grade. To see this, the `size` data member is called. It returns the number of objects contained by the vector (but not their size in bytes!).

On line [39], a 0-valued object `sum` is created. Then, on lines [41–42], a `for` loop is created, which accesses each element stored in `studentGradeVec`. For this purpose, `for` has a special syntax with the colon operator `:` in the middle, as presented in Section 2.6.3. To the left of the colon, the `auto g` defines a variable `g` that will be copied to successive elements of `studentGradeVec`. Thanks to the `auto` keyword, we do not need to explicitly provide the type of `g` – to make our life easier, it will be automatically deduced by the compiler. Such constructions with `auto` are very common in contemporary C++, as will be discussed later (Section 3.7). After the loop sums up all the elements, on line [44] the average grade is computed by dividing `sum` by the number of elements

returned by `size`. We can do this with no stress here since we have already checked that there are elements in the vector, so `size` will return a value greater than 0. Also, since `sum` is of `double` type, due to the type promotion of whatever type of value is returned by `size`, the result will also be `double`. To be explicit, `av` is declared `double`, rather than with `auto`.

We are almost done, but to compute `finalGrade`, defined on line [46], we need to be sure it is one of the values in the set of allowable grades in our system. Thus, on line [49], we check whether the average `av` it is at least 3.0. If not, then 2.0 is assigned. Otherwise, `av` needs to be rounded ± 0.25. In other words, if the entered grades were 3.5, 3.5, and 4.0, their average would be 3.666(6). This would need to be rounded to the 3.5, since $(3.5 - 0.25) \leq 3.666(6) < (3.5 + 0.25)$.

For proper rounding, `av` is split on lines [55] and [56] into its integer `near_int` and fraction `frac` parts. The former is obtained by calling the `std::floor` function, which returns the nearest integer not greater than its argument.[4] On the other hand, `frac` is computed by subtracting `near_int` from `av`. The variable `adjust` on line [58] is used to hold one of the possible adjustments: 0.0, 0.5, or 1.0. Depending on `frac`, the exact value of `adjust` is computed in the conditional statement on lines [60–63]. That is, if `frac` is less than 0.25, then `adjust` is 0, and `near_int` represents the final grade. If `frac` is greater than 0.75, then `adjust` takes on 1, and `near_int` increased by 1 will be the final grade. Otherwise, the final grade is `near_int` + 0.5. The adjustment takes place on line [65].

Finally, the final grade is displayed on lines [68–70]. Since we want the output to have a width of exactly three digits, and only one after the dot, the `std::fixed << std::setw( 3 ) << std::setprecision( 1 )` *stream manipulators* are added to the expression (see Table 3.14).

As with the previous projects, we can run this code in one of the online compiler platforms. However, a better idea is to build the complete project with help of the *CMake* tool, as mentioned in Section 2.5.3. In this case, we will be able to take full advantage of a very important tool – the debugger.

3.3 Our Friend the Debugger

So far, we have written simple code examples and, if they were built successfully, immediately executed them to see their results. However, such a strategy will not work in any serious software development cycle. One of the key steps after the software building process is testing and debugging, as shown in the software development diagram in Figure 2.3. This would not be possible without one of the most important tools – *the debugger*. Running freshly baked software in an uncontrolled way can even be dangerous. A debugger gives us necessary control over the execution of such a software component. For example, it allows us to step through our programs line-by-line and see if the values of objects are correct, or to terminate the entire process if things do not go in the right direction. However, most important, a debugger allows us to really comprehend the operation of our software and make the software behave as intended. In this section, we will trace the basic functionality of debuggers, illustrating with examples.

Before we start debugging, the program has to be built with the debug option turned on. This is either preset in the *CMakeLists.txt* file or set in an IDE. The debug version of a program is fully operational software but without any code optimization, and it includes debugging information. This allows for the exact matching of the current place of execution with concrete lines

4 The nearest higher integer is returned by `std::ceil`.