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

comes in the form of an *identifier*. An identifier is a series of characters and digits, starting with a character. The underscore symbol _ counts as a character and is frequently used to separate parts of names, as we will see in many examples

- If an object is not intended to change its state, define it constant with the `const` keyword
- Do not operate on raw numbers – give them names by defining constant objects
- Choose the proper types for objects
- Properly *initialize* each object
- Do not mix variables of different types

Mixing types can lead to nasty errors. Let's consider the simple conversion from degrees Fahrenheit to Celsius, which is given by the following formula

$$T_C = \left(T_F - 32\right) \cdot \frac{5}{9} \tag{3.1}$$

To convert $T_F = 100$ to T_C, we can translate the previous formula into the following code:

```
1  double TF { 100.0 };
2  double TC {};
3
4  TC = ( TF - 32 ) * ( 5 / 9 );  // Compiles but does not compute what is expected...
5  cout << TF << " Fahrenheit = " << TC << " Celsius" << endl;
```

However, after compiling and running, the following message appears in the terminal window

```
100 Fahrenheit = 0 Celsius
```

which obviously is not correct. So, what happened? The problem is in the wrong data type on line [4]. Division of the integer constant 5 by another integer constant 9 results in 0. On the other hand, the variables `TF` and `TC` are correctly declared with the `double` type. So the subtraction `TF - 32` on line [4] is not a problem since the integer constant 32 will be promoted to `double`, and the result will also be `double`. But this result multiplied by 0 promoted to `double` will produce 0, as shown. The correct version is as follows:

```
4  TC = ( TF - 32.0 ) * ( 5.0 / 9.0 );
```

Here we do not mix data types, and we do not count on behind-the-scenes promotions. On line [4], all variables and constants are of the same `double` type. Now the division is correct, and the correct answer is displayed:

```
100 Fahrenheit = 37.7778 Celsius
```

Table 3.1 presents some of the basic built-in types along with their properties, such as length in bytes and value representation. This knowledge is necessary when deciding what type to choose to represent quantities in a program. The value representation specifies whether we can store positive or negatives values, as well as whether they can be fractions. A variable's length affects the

precision of the stored representation. These issues are further discussed in Chapter 7. The length (in bytes) occupied by a given type can be determined using the `sizeof` operator (see the description of the *GROUP 3* operators in Table 3.15). As shown in Table 3.1, only `char` has a length equal to exactly 1 byte.[2] The other types require more than 1 byte. Their exact lengths are system dependent and can change from system to system. However, the following is guaranteed to hold:

```
sizeof( char ) == 1   <= sizeof( short )
                      <= sizeof( int )
                      <= sizeof( long )
                      <= sizeof( long long )
```

In C++, when an object is created, two things happen:

- A sufficient amount of memory is allocated to store the object
- The object is initialized (if a proper initialization function, called *a constructor*, is available)

As alluded to previously, all objects need to be initialized during the creation process. It is very important for a variable to have a predictable value at all times. Usually, this is easily achieved in more complex objects, since they usually have special initialization functions called *constructors* (see Table 3.2, as well as Section 3.15). However, for performance reasons, the basic types – such as those in Table 3.1 – do not initialize automatically. Their initialization can be performed with *literals*, i.e. a fixed value or another object that has already been created. Table 3.1 contains examples showing the creation of variables and their initialization with literals. There are many versions of them, though (Stroustrup 2013). Interestingly, there is no problem with constant objects – the compiler always enforces their initialization.

For historical reasons, there are many ways to initialize objects (i.e. variables and constants). Table 3.2 shows three such mechanisms. Currently, the most popular and versatile approach is initialization with the *initializer list*, as shown in the first row of Table 3.2. The second row of Table 3.2 shows initialization by a constructor call.

On the other hand, assigning a value with the assignment operator (`=`) looks natural, especially when coding mathematical formulas. However, in this case, we have to be aware that such assignment means first creating an object with a default or undefined value, and then assigning to it a value on the right side of the `=`. This doesn't seem like an issue when assigning to, for example, `int` or `double`, but assigning large objects like matrices this way may incur excessive operation time.

As already mentioned, for a given type *T*, the operator `sizeof(`*T*`)` returns the number of bytes necessary to store an object of type *T* (see also Table 3.15). This works fine with basic types, but calling `sizeof` on more complex objects usually does not return what we expect, since data can be spread throughout many memory locations. In such cases, we should call a specific member function, such as `size`, which is usually provided by the object.

2 The primary role of `char` is to store characters in the one-byte ASCII code representation (*www.ascii.com*). However, for international symbols, multiple-byte Unicode is used.

Table 3.1 Built-in basic data types and their properties and literals.

Type	Bytes	Description	Examples

`char`	1	Stores signed integer values that fit in 1 byte, i.e. in the range −128...+127. Used to store the ASCII codes for characters.	`// Create and init c`
			`char c { 'Y' };`
			
			`std::cout << "Press N to exit" << endl;`
			`std::cin >> c;`
			
			`// Operator == to compare, || is logical OR`
			`bool exit_flag = c == 'N' || c == 'n';`
			`if( exit_flag )`
			`  return;  // Exit now`
			
			`int x { 100 };  // Create x and set 100`
			
			`x = 0x64;  // in hex (leading 0x)`
			`x = 0144;  // in octal (leading 0)`
			`x = 0b01100100;  // as binary (0b, C++14)`
			
			`// Set to 1111111...111`
			`// Operator ~ negates all bits`
			`// 0UL is an unsigned long version of 0`
			`unsigned long light_register { ~ 0UL };`

`short`	≥2	Signed integer values (an optional signed keyword can be put in front).	
`int`	≥2		
`long`	≥4		
`long long`	≥8		
`unsigned [ char, short, int, long]`	As for signed types	Unsigned integer values.	
`bool`	≥1	Boolean value: true or false.	Various formats for literals (constants) are possible (dec, hex, oct, bin). The type of a literal should comply with the type of a variable, as in the last line: `0` is of type `int`, `0L` is `long`, and `0UL` is `unsigned long`.

`float`	≥4	Single-precision floating-point (IEEE-754).	`std::cout << "Enter value \n";`
			`// Create double and init with 0.0 (0 int)`
			`double val { 0.0 };`
			
			`std::cin >> val;`
			`if( val >= 0.0 )`
			`  cout << "sqrt( " << val << " ) = "`
			`  << sqrt( val ) << endl;`
			`else`
			`  cout << "Negative value" << endl;`

`double`	≥8	Double-precision floating-point (IEEE-754).	
`long double`	≥8	Extended-precision floating-point.	A literal to initialize `float` and `double` should have a dot, such as `0.0`, `.3`, `-13`, etc. Pure `0` is of `int` type. The `sqrt` function (from *cmath*) computes a square root. Constant text is provided in quotation marks `""`.

`wchar_t`	≥2; 4 for Unicode	Wide character representation.	`// Unicode letter`
			`wchar_t w_letter { L'A' };`
			
			`// Its memory size in bytes`
			`size_t w_letter_bytes = sizeof( w_letter );`
			
			`// wcout is necessary to display Unicode`
			`std::wcout << w_letter << L" bytes = "`
			`  << w_letter_bytes << std::endl;`

`size_t`	≥4	Unsigned integer type; the result of the sizeof operator (*GROUP 4*, Table 3.15).	To enter constant wide letters, use `L'A'`. Constant wide text is provided as `L""`: in quotation marks with the prefix `L`.

Table 3.2 Syntax for defining and initializing variables and constants (objects) in C++.

Operation	Syntax	Examples

Definition of a variable or constant (object) named by *identifier* and with features set by *type*.
The initial *value* is set with the initializer list { }.
If `const` is added, then the object cannot be changed (no new values are allowed).
This is the *preferable* way to initialize objects in C++.
{ } is called zero-initialization since an object is guaranteed to have its defult constructor called, which for built-in types sets its value to 0.
		`// Vars and const initialized with {}`
		`int x {};  // Create x and set to 0`
		`x += 1;  // Can increase by 1`
		
		`// Create const y and set to 100 - cannot change`
		`const int y { 100 };`
		`//y += 1;  // Error, cannot change constant`
		
		`// Create c and set to ASCII code of 'A'`
		`char c { 'A' };`
		`// Can display on the screen`
		`std::cout << "c = " << c << std::endl;`
		
		`// A variable of floating arithmetic`
		`double radius { 2.54 };`
		
		`// Create const kPi of floating type`
		`const double kPi { 3.141592653589 };`
		`// Compute and display area`
		`std::cout << "Area = " << kPi * radius * radius;`
		
		`bool flag { true }; // A logical var init to true`

Operation	Syntax	Examples

As in the previous item, but initialization is in the form of a call to a constructor (i.e. an initialization function defined for *type*).		`// Vars and const initialized with ()`
		`//int x(); // x is int and set to 0?? No !!`
		`int x ( 0 );  // Create x and set to 0`
		`x += 1;  // Can increase by 1`
		
		`int y( 100 );  // Ok, var x with 100`
		`y += 1;`
		
		`// A variable of floating-point arithmetic`
		`double radius ( 2.54 );`
		
		`// Create const kPi of floating type`
		`const double kPi ( 3.141592653589 );`
		`// Compute and display area`
		`std::cout << "Area = " << kPi * radius * radius;`
		
		`bool flag ( true );  // A logical var init to true`
		
		Using `()` for initialization looks like calling a function, and indeed it is. This function is called a *constructor*, which performs an action defined by the object's type. If no arguments are provided, then the *default constructor* is called. With parameters, this is the *parametric constructor*. These are discussed in Section (3.15).
		Writing `int x();` can be confusing. It does not create a variable x with a call to its default constructor. Instead, the compiler treats it as a declaration of a function named x that takes no arguments and returns `int`.

(Continued)

Table 3.2 (continued)

Operation	Syntax	Examples

As in the previous item, but initialization is in the form of a copy.
A value can also be provided in the initializer list {}. (This is how initialization is done in C.)
		`// Vars and const initialized with assignment =`
		`int x = 0;  // Create x and set to 0`
		`x += 1;  // Can increase by 1`
		
		`// A variable of floating arithmetic initialized`
		`// from an initialization list`
		`double radius = { 2.54 };`
		
		`// Create const kPi of floating type`
		`const double kPi = 3.141592653589;`
		`// Compute and display area`
		`std::cout << "Area = " << kPi * radius * radius;`
		
		`bool flag = true;  // A logical var set to true`
		
		Initialization by assignment works fine but may not be efficient in some cases. This is the case because two steps are executed: first an object is created with its default value, and then it is initialized with a value on the right side of the assignment operator `=`.

Other parameters of basic types, such as values of `lowest`, `min`, `max`, etc. can be determined using the `numeric_limits<` *T* `>` class (from the *limits*), as shown in Listing 3.1.

Listing 3.1 Displaying values and measuring the size in bytes of basic C++ built-in data types (from *CppBookCode, Variables.cpp*).

```
1   #include <limits>    // for basic types' limits
2
3   using std::cout, std::endl, std::numeric_limit;
4
5   // Print basic facts about char
6   cout << "bytes of char = " << sizeof( char )    << endl;
7   cout << "char lowest = "   << +numeric_limits< char >::lowest() << endl;
8   cout << "char min = "      << +numeric_limits< char >::min()    << endl;
9   cout << "char max = "      << +numeric_limits< char >::max()    << endl;
10
11  // Print basic facts about int
12  cout << "bytes of int = "  << sizeof( int )     << endl;
13  cout << "int lowest = "    << numeric_limits< int >::lowest()   << endl;
14  cout << "int min = "       << numeric_limits< int >::min()      << endl;
15  cout << "int max = "       << numeric_limits< int >::max()      << endl;
```

To use `numeric_limits`, we need to include the `limits` header, as shown on line [1]. On lines [7–9], we print the `lowest`, `min`, and `max` value of the type `char`. However, to properly display values, we use a small trick here: adding a `+` sign in front of the `numeric_limits` objects. This causes the values to be treated as integers and displayed properly.

To store a value, we need to check whether the chosen representation has a sufficient number of bits. In the following example, we simply ask if type `long` is sufficient to store the number containing up to 11 digits. The computations are simple, but they require calling the `log` function to return a value of the natural logarithm, as shown on line [18] of Listing 3.2. The answer to the question of whether `long` is long enough is found using the comparison (`<`) on lines [18–19]. This `true` or `false` result goes to the `bool` variable `longFitFlag`.

Listing 3.2 Computing the number of bits for a value representation (from *CppBookCode, Variables.cpp*).

```
16  // Check whether long is long enough to store 11-digit values
17  const int bits_in_byte = 8;
18  bool longFitFlag = ( std::log( 99999999999.0 ) / std::log( 2.0 ) + 1 )
19                 < sizeof( long ) * bits_in_byte - 1;
20
21  cout  << "We " << ( longFitFlag ? "can" : "cannot" )
22     << " store 11 digits in long in this system" << endl;
```

On line [21], a message is displayed depending on the value of the flag `longFitFlag`. To check it, the conditional `? :` operator is used, the details of which are presented in *GROUP 15* (Table 3.15). Finally, the following is shown in the terminal window:

```
bytes of char = 1
char lowest = -128
char min = -128
char max = 127
bytes of int = 4
int lowest = -2147483648
int min = -2147483648
int max = 2147483647
We cannot store 11 digits in long in this system
```

We see that in this system, `long` is not long enough to store a value composed of 11 digits. Also, we need to be very careful here, since `long` denotes a signed type. That is, it can store both positive and negative values. This means one bit needs to be used for sign representation. Hence, 1 is subtracted on line [19]. Further differences between signed and unsigned types are discussed in Section 7.1.4. As a rule of thumb, if no specific requirements are set, signed integers should be chosen, and types should not be mixed.

`wchar_t` can be used to store all kinds of international characters. The following code snippet shows how to create and initialize a wide character, as well as how to assign text composed of wide characters using the `std::wstring` object.

```
23  wchar_t pl_letter = L'Ą';
24  wcout << pl_letter << L" needs " << sizeof( pl_letter ) << L" bytes." << endl;
25
26  std::wstring holiday_pl( L"Święto" );
27
28  // We cannot use sizeof here - we need to call size()
29  size_t holiday_pl_len = holiday_pl.size();
30
31  std::wcout << holiday_pl << L" has " << holiday_pl_len << L" letters\n";
```

Also note that to display wide characters properly, we need to use `std::wcout` rather than `std::cout`. In addition, on some platforms, we need to call a platform-specific function to display the national letters properly. The results of running the previous code are as follows:

```
Ą needs 2 bytes.
Święto has 6 letters
```

Finally, note that the wide text literals begin with the prefix `L`. To conclude, if we write applications that can be used in international markets, using `wchar_t`, `wstring`, `wcout`, etc. from the start is a good option. We will see examples in later chapters of this book.

Things to Remember

- Choose meaningful and readable names (identifiers) for objects (variables, constants, functions, classes, etc.)
- Choose the proper data type for computations
- Prefer signed types
- Always initialize variables. For zero-initialization, it is sufficient to add { }

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.

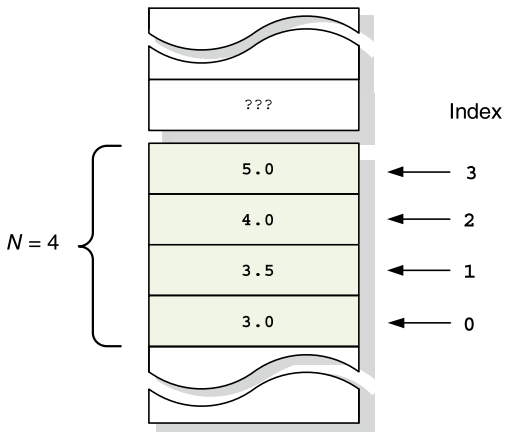

Figure 3.2 A vector is composed of a number of objects occupying contiguous space in memory. Here we have $N = 4$ objects that can store floating-point data. Each cell of a vector can be accessed by providing its index. The first element is at index 0, and the last is at index $N - 1$.

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`.