INFO0004-2
Object-Oriented Programming Projects
in C++

Laurent Mathy

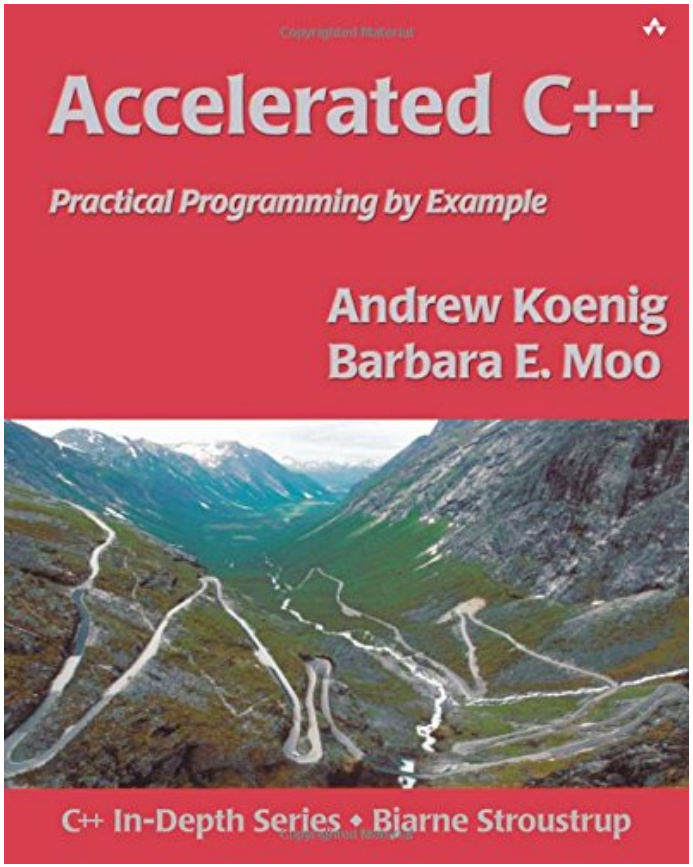
February 1, 2021

Outline

Reference book

C++ is a complex language, so we only see the most useful subset.

Accelerated C++
by Andrew Koenig and Barbara Moo
ISBN 0-201-70353-X



Beware! C++11/14 is not covered in the book.

Prerequisites

We assume you have knowledge of:

- programming in C;
- object-oriented programming.

Outline

First C++ program

```
1  // A small C++ program
2  #include <iostream>
3
4  int main()
5  {
6      std::cout << "Hello, world!" << std::endl;
7      return 0;
8  }
```

Java programmers beware:
Not everything in C++ is a class/object!

Comments

`//` begins a **comment** which extends to the end of the line.

```
1 // A small C++ program
```

Other (multi-line) comment style:

```
1 /* I am a comment. */
2 /* I am a comment
3 which spans
4 multiple lines. */
```

`/* ... */` comments **don't nest** in C++.

```
1 /* Comment start /* inner comment */
2 not a comment anymore, but a syntax error */
```

Includes

Programs ask for external facilities with **include directives**, *e.g.*

```
1 #include <iostream>
```

`#include <...>` indicates a **standard header** (from the C++ standard library, or another system library).

To include your own headers, use quotes:

```
1 #include "my_header.hpp"
```

`main` function

Like in C, every C++ program must contain a `main` function.

```
1  int main()
2  {    // Left brace
3       // Statements
4  }    // Right brace
```

`main` is required to yield an integer as a result:

- 0 means success.
- Any other value indicates there was a problem.

Standard output

We use the standard library's **output stream operator**, `<<`, to print to *standard output*.

```
1  std::cout << "Hello, world!" << std::endl;
```

Preceding a name by `std::` indicates that the name is part of a namespace called `std`:

- A **namespace** is a collection of related names.
- The standard library uses `std` to contain all the names it defines.

`::` is the **scope operator**.
`scp::name` is a **qualified name**, where the name `name` is defined in the scope `scp`.

`std::cout` refers to the *standard output stream*.
`std::endl` ends current line of output and *flushes* output buffer.

Wait . . . there is something funny going on

An **expression** is made out of **operators** and **operands** (each operand has a **type**).

The effect of an operator depends on the type of its operands.

`<<` is a binary operator: it takes 2 operands.

But we have written an expression with 2 `<<` and 3 operands!

How can this work?

```
1  std::cout << "Hello, world!" << std::endl;
```

Wait . . . there is something funny going on

An **expression** is made out of **operators** and **operands** (each operand has a **type**).

The effect of an operator depends on the type of its operands.

`<<` is a binary operator: it takes 2 operands.

But we have written an expression with 2 `<<` and 3 operands!

How can this work?

```
1 std::cout << "Hello, world!" << std::endl;
```

Answer: operator `<<`:

- is **left-associative**, *i.e.* takes as much as it can from the expression to its left, and as little as it can from its right;
- returns as result its left operand (in our case `std::cout` of type `std::ostream`).

$\Rightarrow$ the expression is equivalent to:

```
1 (std::cout << "Hello, world!") << std::endl;
```

Standard input

```cpp
// Ask for a person's name, and greet the person

#include <iostream>
#include <string>

int main() {
    // Ask for the person's name
    std::cout << "Please enter your first name: ";

    // Read the name
    std::string name; // Define `name`
    std::cin >> name; // Read into `name`

    // Write a greeting
    std::cout << "Hello, " << name  << "!" << std::endl;

    return 0; // 0 means success
}
```

Standard input (2)

We are using the standard input and standard string facilities:

```
3  #include <iostream>
4  #include <string>
```

The statement

```
11 std::string name; // Define `name`
```

defines a variable `name` of type `std::string`.

The STL says that a `std::string` variable always contains a value, which defaults to the *empty* `string` if not provided.

`name` is a **local variable**, which:

- only exists while execution is within the pair of braces `{}` where variable was defined;
- is created and destroyed automatically.

Java programmers beware: this is the only automatic memory management in C++.

Standard input (3)

```
12  std::cin >> name; // Read into `name`
```

- flushes standard output buffer;
- discards *white spaces* from standard input stream;
- reads characters from standard input stream into `name`;
- stops when encounters either white-space character or end-of-line.

Framing the greeting

```
Please enter your first name: Me
**************
*            *
* Hello, Me! *
*            *
**************
```

Framing the greeting: code

```
5   std::cout << "Please enter your first name: ";
6   std::string name;
7   std::cin >> name;
8
9   // Build the message that we intend to write
10  const std::string greeting = "Hello, " + name + "!";
11  // Build the second and fourth lines of the output
12  const std::string spaces(greeting.size(), ' ');
13  const std::string second = "* " + spaces + " *";
14  // Build the first and fifth lines of the output
15  const std::string first(second.size(), '*');
16
17  // Write it all
18  std::cout << first << std::endl;
19  std::cout << second << std::endl;
20  std::cout << "* " << greeting << " *" << std::endl;
21  std::cout << second << std::endl;
22  std::cout << first << std::endl;
```

Initialising a `string`

Saying explicitly what value we want for a string:

```
10  const std::string greeting = "Hello, " + name + "!";
```

- Variable `greeting` is initialised when defined.
- **String literals** are automatically converted to `std::string`.
- `+` *concatenates* two `std::strings`.
- Keyword `const` promises that value of variable will not change after initialisation (which must happen at definition time).

Constructing a `string`

Computing the value of a string:

```
12  const std::string spaces(greeting.size(), ' ');
```

- This actually calls one of the `std::string` **constructors**. Constructors depend on arguments **types**.
- `string(size_t n, char c)` builds a `std::string` that contains `n` copies of character `c`.
- `size()` is a **member function** (a.k.a. *method*) of `std::string`, that returns the size of the `string`.
- `' '` is a **character literal**. Do not confuse them with **string literals** (`" "`).

C++ expressions and statements

C++ inherits a rich set of operators from C.

C++ also inherits statement syntax from C (loops, conditionals, *etc.*).

Question: What's the difference between these two loops?

```
1  int c;
2  for (c = 0; c < 10; c++) {
3    // Do something
4  }
```

```
for (int c = 0; c < 10; c++) {
  // Do something
}
```

C++ expressions and statements

C++ inherits a rich set of operators from C.

C++ also inherits statement syntax from C (loops, conditionals, *etc.*).

Question: What's the difference between these two loops?

```
1  int c;
2  for (c = 0; c < 10; c++) {
3    // Do something
4  }
5  // c still in scope here
```

```
for (int c = 0; c < 10; c++) {
  // Do something
}
// c undefined here
```

Answer: the **scope** of c!

Outline

Computing student grades

Student's final grade is 40% of final exam, 20% of midterm exam, and 40% of average homework grade.

```
1   #include <iomanip>
2   #include <iostream>
3   #include <string>
4
5   using std::cin; using std::cout; using std::endl;
6   using std::setprecision; using std::streamsize;
7   using std::string;
8
9   int main() {
10      // Ask for and read the student's name
11      cout << "Please enter your first name: ";
12      string name;
13      cin >> name;
14      cout << "Hello, " << name << "!" << endl;
15
16      // Ask for and read the midterm and final grades
17      cout << "Please enter your midterm and final exam grades: ";
18      double midterm, final;
19      cin >> midterm >> final;
```

Computing student grades (2)

```
21        // Ask for the homework grades
22        cout << "Enter all your homework grades, "
23                "followed by end-of-file: ";
24
25        int count = 0;  // Number of grades read so far
26        double sum = 0; // Sum of grades read so far
27        double x;       // A variable into which to read
28
29        // Invariant: we have read `count` grades so far,
30        // and `sum` is the sum of the first `count` grades
31        while (cin >> x) {
32            ++count;
33            sum += x;
34        }
35
36        // Compute and write the final grade
37        double final_grade = 0.2 * midterm + 0.4 * final + 0.4 * sum / count;
38        streamsize prec = cout.precision(); // Save initial precision
39        cout << "Your final grade is "
40             << setprecision(3) << final_grade << endl;
41        cout.precision(prec); // Restore initial precision
42
43        return 0;
44    }
```

`using` and more STL facilities

A **using-declaration** binds a name to its qualified version:

```
7  using std::string;
```

allows to use `string` when meaning `std::string`.

`streamsize` is the type used to represent sizes in I/O library.

```
39 cout << "Your final grade is "
40      << setprecision(3) << final_grade << endl;
```

sets floating-point precision to 3 significant digits (*e.g.* 3.14) before printing `final_grade`.

`setprecision` modifies the output stream, so it is a good idea to save and restore original precision.

Wait... there is something funny going on

Look carefully at the following statement:

```
22  cout << "Enter all your homework grades, "
23          "followed by end-of-file: ";
```

Wait... there is something funny going on

Look carefully at the following statement:

```
22  cout << "Enter all your homework grades, "
23          "followed by end-of-file: ";
```

How can we write two string literals with a single << operator?

Wait... there is something funny going on

Look carefully at the following statement:

```
22  cout << "Enter all your homework grades, "
23          "followed by end-of-file: ";
```

How can we write two string literals with a single << operator?

Answer:

Two (or more) string literals separated only by white-space, are automatically concatenated.

Default initialisation

Recall that when we defined a `std::string` but did not provide and initial value, it was implicitly initialised by default (to the empty string).

- **Default-initialisation** depends on the type.
- Implicit initialisation does not exist for built-in types, and thus un-initialised variables of built-in type will contain garbage.

```
25 int count = 0;  // Number of grades read so far
26 double sum = 0; // Sum of grades read so far
```

Note that the initial value for `sum` is of type `int`, which gets implicitly converted into a `double`. To avoid this conversion, use `double sum = 0.0;`

Reading multiple input

```
31  while (cin >> x) {
32      ++count;
33      sum += x;
34  }
```

Recall that the operator >> returns its left operand (of type `std::istream`) as a result.

However, this type is used in a condition!
$\Rightarrow$ it must be converted into a `bool`.

Conversion to `bool`

Arithmetic value:

- Zero converts to `false`.
- Non-zero values convert to `true`.

Similarly, `std::istream` provides a conversion from `cin` to `bool`. `std::cin` is `true` if last attempt to read was successful.

Ways for reading to be unsuccessful:

- reached end-of-file;
- encountered input incompatible with type read;
- system detected hardware failure on input device.

Using medians instead of averages

What if we want to take the *median* of homeworks, instead of their average?

Now, we must read and store values:

- read a number of values, not knowing this number;
- into a container;
- sort values;
- get median.

Using medians: read and store multiple values

```
26  vector<double> homeworks;
27  double x;
28  // Invariant: `homeworks` contains all the
29  // homework grades read so far
30  while (cin >> x)
31      homeworks.push_back(x);
```

`vector` is a **template** class defined in `<vector>` header.

- C++ *templates* are similar to Java *generics*.
- All values in a `vector` have the same type.
- Different `vectors` can hold different types.

`push_back` appends a new element at the end of the `vector`.

Using medians: container size

```
33  // Check the student entered some homework grades
34  typedef vector<double>::size_type vec_sz;
35  vec_sz size = homeworks.size();
36  if (size == 0) {
37      cout << endl << "You must enter your grades.  "
38                   "Please try again." << endl;
39      return 1;
40  }
```

`vector` defines type `vector<double>::size_type` as **`unsigned`** type guaranteed to hold size of largest possible `vector`.

`size()` is a method of `vector` class; returns the number of elements.

C++11 `auto`

Using types such as `std::vector<double>::size_type` can be cumbersome and hinder legibility.

C++ 2011 supports a limited form of **type-inference**.

When a variable is defined with an initializer, one can use `auto` to have the compiler automatically *deduce* the correct type from the right-hand side.

```
34  auto size = homeworks.size();
```

would automatically give variable `size` the type `std::vector<double>::size_type`, since it is the type of `homeworks.size()`.

Only use `auto` where it *improves* legibility!

Using medians: sorting

```
41  // Sort the grades
42  sort(homeworks.begin(), homeworks.end());
```

`sort` is defined in `<algorithm>` header.

`begin()` is a `vector` method denoting first element.
`end()` is a `vector` method denoting **one past** last element.

All ranges in the STL are given as `[begin, end)`.

Using medians: compute and print final grade

```
44  // Compute the median homework grade
45  auto mid = size / 2;
46  double median = (size % 2 == 0)
47      ? (homeworks[mid] + homeworks[mid - 1]) / 2
48      : homeworks[mid];
49
50  // Compute and write the final grade
51  double final_grade =
52      0.2 * midterm + 0.4 * final + 0.4 * median;
53  streamsize prec = cout.precision(3); // Set precision
54  cout << "Your final grade is " << final_grade << endl;
55  cout.precision(prec); // Restore original precision
```

Complete median program

```
1   #include <algorithm>
2   #include <iostream>
3   #include <string>
4   #include <vector>
5
6   using std::cin; using std::cout; using std::endl;
7   using std::sort; using std::streamsize;
8   using std::string; using std::vector;
9
10  int main() {
11      // Ask for and read the student's name
12      cout << "Please enter your first name: ";
13      string name;
14      cin >> name;
15      cout << "Hello, " << name << "!" << endl;
16
17      // Ask for and read the midterm and final grades
18      cout << "Please enter your midterm and final exam grades: ";
19      double midterm, final;
20      cin >> midterm >> final;
```

Complete median program (2)

```
22  // Ask for and read the homework grades
23  cout << "Enter all your homework grades, "
24          "followed by end-of-file: ";
25
26  vector<double> homeworks;
27  double x;
28  // Invariant: `homeworks` contains all the
29  // homework grades read so far
30  while (cin >> x)
31      homeworks.push_back(x);
32
33  // Check the student entered some homework grades
34  auto size = homeworks.size();
35  if (size == 0) {
36      cout << endl << "You must enter your grades.  "
37                      "Please try again." << endl;
38      return 1;
39  }
```

Complete median program (3)

```
41      // Sort the grades
42      sort(homeworks.begin(), homeworks.end());
43
44      // Compute the median homework grade
45      auto mid = size / 2;
46      double median = (size % 2 == 0)
47          ? (homeworks[mid] + homeworks[mid - 1]) / 2
48          : homeworks[mid];
49
50      // Compute and write the final grade
51      double final_grade =
52          0.2 * midterm + 0.4 * final + 0.4 * median;
53      streamsize prec = cout.precision(3); // Set precision
54      cout << "Your final grade is " << final_grade << endl;
55      cout.precision(prec); // Restore original precision
56
57      return 0;
58  }
```