Program Organisation & Sequential Containers

Laurent Mathy

Object-Oriented Programming Projects

February 25, 2019

Outline

Functions

```
9  double grade(double midterm, double final, double homework)
10 {
11     return 0.2 * midterm + 0.4 * final + 0.4 * homework;
12 }
```

- `midterm`, `final`, `homework` are **parameters**; behave like local variables.
- When we call the function, we supply **arguments** which are used to initialise the parameters.
- Semantics of the call is *call by value*: parameters take on a **copy** of the value of the arguments.
- Returns a `double` value.

Function name and parameter types define the function **signature**.

Functions (2)

```
8  double median(vector<double> vec) {
9      if (vec.empty())
10         throw domain_error("median of an empty vector");
11
12     sort(vec.begin(), vec.end());
13
14     auto mid = vec.size() / 2;
15
16     return (vec.size() % 2 == 0) ? (vec[mid] + vec[mid - 1]) / 2
17                                  : vec[mid];
18 }
```

- Call copies the entire argument `vector`:
  - may be slow;
  - is safe: taking median should not change vector.
- General way of complaining: **`throw` exception**
  - `domain_error` defined in `<stdexcept>` header.
  - Argument describes what went wrong.

Functions: `const` reference and overloading

```
14  double grade(double midterm, double final,
15               const vector<double>& homeworks)
16  {
17      if (homeworks.empty())
18          throw domain_error("student has done no homework");
19      return grade(midterm, final, median(homeworks));
20  }
```

- Third parameter is a **reference**.
  - A reference is an *alias*: reference and original are the same thing.
  - Reference to reference is same thing as reference to original.
  - Function gets direct access to argument: **no copying**.
  - `const` reference: the function promises not to change original vector.
- `grade` function is now **overloaded**.
  - We defined two different versions of `grade`.
  - No ambiguity: the two functions have different signatures.

Functions: returning several values

There is no direct way to return more than one value.

Indirect way: give function a parameter that is a reference to an object where to place one result.

```
1  istream& read_hws(istream& in, vector<double>& hws) {
2      // ...
3      return in;
4  }
```

- Non-`const` reference parameter:
  - usually signals intention to modify the object;
  - must be an **lvalue**: a non-temporary object.
- Both parameters are refs as function changes state of both.
- Return value is a reference: we are returning the stream we were given *as is* without copying.

Reading values within function

How difficult can it be?

```
1  istream& read_hws(istream& in, vector<double>& hws) {
2      double grade;
3      while (in >> grade)
4          hws.push_back(grade);
5      return in;
6  }
```

Reading values within function

How difficult can it be?

```
1  istream& read_hws(istream& in, vector<double>& hws) {
2      double grade;
3      while (in >> grade)
4          hws.push_back(grade);
5      return in;
6  }
```

- We do not know what's in `hws` ⇒ we should clear it.
- Loop reads until failure: either *end-of-file*, or encountered a *non-number*:
  - How will the user know the difference?
  - Difference between "we have just read last record" vs "sorry, no more record"?
  - Must only fail when function can read nothing more ⇒ must clear it.
  - On entry in function, if stream already in error, must leave it. alone.

Reading values within function (2)

```
22  istream& read_hws(istream& in, vector<double>& hws) {
23      if (in) {
24          // Get rid of previous contents
25          hws.clear();
26
27          // Read homework grades
28          double grade;
29          while (in >> grade)
30              hws.push_back(grade);
31
32          // Clear the stream so that input will work
33          // for the next student
34          in.clear();
35      }
36      return in;
37  }
```

Calculating one student's grade

```
 9  int main() {
10      // Ask for and read 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 midterm and final grades
17      cout << "Please enter your midterm and final exam grades: ";
18      double midterm, final;
19      cin >> midterm >> final;
20
21      // Ask for and read homework grades
22      cout << "Enter all your homework grades, "
23              "followed by end-of-file: ";
24      vector<double> homeworks;
25      read_hws(cin, homeworks);
```

Calculating one student's grade (2)

```
27      // Compute and generate final grade, if possible
28      try {
29          double final_grade = grade(midterm, final, homeworks);
30          streamsize prec = cout.precision();
31          cout << "Your final grade is " << setprecision(3)
32               << final_grade << setprecision(prec) << endl;
33      } catch (domain_error) {
34          cerr << endl << "You must enter your grades.  "
35                       "Please try again." << endl;
36          return 1;
37      }
38
39      return 0;
40  }
```

- `try` statement:
  - tries to execute statements in { };
  - pass control to `catch` *clause* if `domain_error` occurs anywhere in these statements.
- `cerr` is the standard error stream.

Organising Data

Students data all in a file:

```
1 Zorglub 93 91 47 90 92 73 100 87
2 Aaron 75 90 87 92 93 60 0 98
3 ...
```

Want final results, alphabetically:

```
1 Aaron          86.8
2 ...
3 Zorglub        90.4
```

Keeping related things together

```
7  struct Student_info {
8      std::string name;
9      double midterm, final;
10     std::vector<double> homeworks;
11 }; // Semicolon in REQUIRED
```

We can then use a `vector<Student_info>` to hold information about an arbitrary number of students.

Managing student records

```
9   istream& read(istream& in, Student_info& s) {
10      // Read and store student's name, midterm and final grades
11      in >> s.name >> s.midterm >> s.final;
12      // Read and store student's homework grades
13      read_hws(in, s.homeworks);
14      return in;
15  }
```

- `read` is overloaded (if other `read` function(s) already exist).
- Input stream can fail at anytime:
  - OK, as subsequent input attempts will do nothing.
  - Relies on `read_hws` leaving stream in error.

```
17  double grade(const Student_info& s) {
18      return grade(s.midterm, s.final, s.homeworks);
19  }
```

`grade` is not catching exceptions: they will be passed back to its caller.

Sorting student records

`sort` function relies on `<` operator being defined for type being sorted. But `<` is not defined for `Student_info` type.

But we can use version of `sort` that takes a *predicate* as third argument.

```
21  bool compare(const Student_info& x, const Student_info& y)
22  {
23      return x.name < y.name;
24  }
```

```
20  sort(students.begin(), students.end(), compare);
```

Generating the report

```
10  // Read all the records, and find the length of the longest name
11  Student_info record;
12  vector<Student_info> students;
13  string::size_type maxlen = 0;
14  while (read(cin, record)) {
15      maxlen = max(maxlen, record.name.size());
16      students.push_back(record);
17  }
18
19  // Alphabetize the records
20  sort(students.begin(), students.end(), compare);
21
22  auto prec = cout.precision(3);
```

- `max` in `<algorithm>`.
- `cout.precision(3)` sets `cout`'s number of significant floating-point digits to 3, and returns its previous precision.

Generating the report (2)

```
23  for (vector<Student_info>::size_type i = 0;
24           i != students.size(); ++i) {
25      // Write the name, padded on the right
26      cout << students[i].name
27           << string(maxlen + 1 - students[i].name.size(), ' ');
28      // Compute and write the grade
29      try {
30          double final_grade = grade(students[i]);
31          cout << final_grade << endl;
32      } catch (domain_error e) {
33          cerr << e.what() << endl;
34      }
35  }
36  cout.precision(prec); // Restore precision
```

- `string(n, ' ')` creates a `string` of `n` blanks.
  - No name: `string(...)` is a valid expression.

Managing complex code

Like in C, group abstractions into separate header and source files.

Support for **separate compilation**, and **information hiding**.

Header file must include:

- all headers *strictly* needed for its declarations;
- **declarations** of implemented **public** functions;
- declarations or definitions of required types.

Source file must include:

- all headers needed for implementation of functions (including corresponding header);
- **definitions** of functions;
- definitions of types that are only declared in the header.

Managing complex code (2)

Always protect your header files against double inclusion:

```
1  #ifndef MEDIAN_HH
2  #define MEDIAN_HH
3
4  #include <vector>
5
6  // Return the median of the given values.
7  double median(std::vector<double> values);
8
9  #endif
```

- Avoid proprietary *#pragma*, use standard **include guards**.
- Avoid polluting the namespace with **using** directives in headers.
- Parameter names are optional in declarations.
  - Use them to document your code.

Outline

Sequential containers

```
14  bool fgrade(const Student_info& s) {
15      return grade(s) < 60;
16  }
17
18  vector<Student_info> extract_fails_1(vector<Student_info>& students) {
19      vector<Student_info> passes, fails;
20
21      for (vector<Student_info>::size_type i = 0;
22              i != students.size(); ++i)
23          if (fgrade(students[i]))
24              fails.push_back(students[i]);
25          else
26              passes.push_back(students[i]);
27
28      students = passes;
29      return fails;
30  }
```

`students = passes;` results in original contents to be replaced by the content in `passes`. This is so because of the way the `=` operation is implemented in `vector`.

Erasing elements in place

```
32  vector<Student_info> extract_fails_2(vector<Student_info>& students) {
33      vector<Student_info> fails;
34      vector<Student_info>::size_type i = 0;
35
36      // Invariant: elements `[0,i)` of `students` are passing grades
37      while (i != students.size())
38          if (fgrade(students[i])) {
39              fails.push_back(students[i]);
40              students.erase(students.begin() + i);
41          } else
42              ++i;
43
44      return fails;
45  }
```

No version of `erase` operates on indices: specify element through `students.begin()` and offset.

Remember that `erase` changes the `vector`'s size.

Iterators

```
1  for (vector<Student_info>::size_type i = 0;
2          i != students.size(); ++i)
3      cout << students[i].name << endl;
```

Another way to do the same thing:

```
1  for (vector<Student_info>::const_iterator iter = students.begin();
2          iter != students.end(); ++iter)
3      cout << (*iter).name << endl;
```

Iterator is a value that:

- identifies elements in a container;
- let us examine value of that element;
- has operation for moving between elements;
- only support efficient operations on container.

container-type`::const_iterator` gives read-only access.
container-type`::iterator` gives full read-write-erase access.

More on iterators

- `begin()` function returns an `iterator` to the first element of the collection.
- `end()` function returns an `iterator` to the first element **past the end** of the collection.
- Dereferencing: `*iter` provides access to element referred to by `iter`.
- `iter->name` is the same as `(*iter).name`.
- `students.begin() + i` is an `iterator` to the *i*th element in `students`.
- Note how we used `iter != students.end()` and *not* `iter < students.end()`. Operator `<` is not defined for all iterators.

Using iterators instead of indices

```
47  vector<Student_info>
48  extract_fails_3(vector<Student_info>& students) {
49      vector<Student_info> fails;
50      vector<Student_info>::iterator iter = students.begin();
51
52      while (iter != students.end())
53          if (fgrade(*iter)) {
54              fails.push_back(*iter);
55              iter = students.erase(iter);
56          } else
57              ++iter;
58
59      return fails;
60  }
```

Need `iter = students.erase(iter);` because `erase` **invalidates** iterators for all elements from the one erased.

A note on vectors

- `vector` is a great container for adding "at the end" and for random access;
- but not that good when erasing in the middle, because of required shifting of elements.

$\Rightarrow$ Our implementation may get very slow with large number of students.

$\Rightarrow$ We need a better container for erasing in the middle.

A faster version, using the `list` type

```
62  list<Student_info>
63  extract_fails_4(list<Student_info>& students) {
64      list<Student_info> fails;
65      list<Student_info>::iterator iter = students.begin();
66
67      while (iter != students.end())
68          if (fgrade(*iter)) {
69              fails.push_back(*iter);
70              iter = students.erase(iter);
71          } else
72              ++iter;
73
74      return fails;
75  }
```

Shorter iterator declarations using `auto`

Iterator syntax can be quite heavy:

```
1  for (std::vector<double>::const_iterator it = xs.begin();
2            it != xs.end(); ++it)
3      // Do something with `it`
```

`auto` can help:

```
1  for (auto it = xs.begin(); it != xs.end(); ++it)
2      // Do something with `it`
```

. . . but beware!

Shorter iterator declarations using `auto`

Iterator syntax can be quite heavy:

```
1  for (std::vector<double>::const_iterator it = xs.begin();
2            it != xs.end(); ++it)
3      // Do something with `it`
```

`auto` can help:

```
1  for (auto it = xs.begin(); it != xs.end(); ++it)
2      // Do something with `it`
```

. . . but beware!

`begin()` can return either an `iterator`, or a `const_iterator`.
`auto it = xs.begin()` defines a read-write-erase `iterator`.

`cbegin()`/`cend()` always return a `const_iterator`.

C++11 for-each loops

An even shorter and clearer syntax is provided by **ranged-based for loops**. Once again, beware of access types!

```
1  for (auto x : xs) {
2    // x iterates over xs by COPYing values
3    ++x; // Only modifies local variable x, NOT xs!
4  }
5
6  for (auto& x : xs) {
7    // x iterates over xs by reference, no copy
8    ++x; // Modifies xs
9  }
10
11 for (const auto& x : xs) {
12   // x iterates over xs by reference, no copy
13   ++x; // COMPILE ERROR, cannot modify a const ref
14 }
```

More on `strings`

`string` is a special kind of container, that:

- contains only characters;
- supports some container operations:
  - indexing;
  - iterators.

Splitting a string

```
16  vector<string> split(const string& s) {
17      vector<string> ret;
18      string::size_type i = 0;
19
20      // Invariant: we have processed characters `[0,i)`
21      while (i != s.size()) {
22          // Find word first character
23          while (i != s.size() && isspace(s[i]))
24              ++i;
25          // Find end of next word
26          string::size_type j = i;
27          while (j != s.size() && !isspace(s[j]))
28              ++j;
29          // If we found some non-whitespace characters
30          if (i != j) {
31              // Copy word to vector
32              ret.push_back(s.substr(i, j - i));
33              i = j;
34          }
35      }
36      return ret;
37  }
```

Splitting a `string` (2)

`isspace` requires `<cctype>`

`substr`:

- member function of `string`;
- creates a new `string`;
- first parameter: start index of new string;
- second parameter: length of new string.

Framing string "boxes"

```
39  string::size_type width(const vector<string>& v) {
40      string::size_type maxlen = 0;
41      for (auto& s : v) // No need for const here
42          maxlen = max(maxlen, s.size());
43      return maxlen;
44  }
45
46  vector<string> frame(const vector<string>& v) {
47      vector<string> ret;
48      string::size_type maxlen = width(v);
49      string border(maxlen + 4, '*');
50
51      // Write the top border
52      ret.push_back(border);
53      // Write each interior row, bordered by an asterisk and a space
54      for (auto& s : v)
55          ret.push_back(
56              "* " + s + string(maxlen - s.size(), ' ') + " *");
57      // Write the bottom border
58      ret.push_back(border);
59
60      return ret;
61  }
```

Vertical concatenation of `string` "boxes"

No facility to concatenate `vectors`: do it yourself.

```
63  vector<string> vcat(const vector<string>& top,
64                      const vector<string>& bottom) {
65      // Copy top picture
66      vector<string> ret = top;
67      // Copy bottom picture
68      for (auto& s : bottom)
69          ret.push_back(s);
70
71      return ret;
```

Code in lines 68 – 69 could be replaced by:

```
68  ret.insert(ret.end(), bottom.begin(), bottom.end());
```

Horizontal concatenation of `string` "boxes"

```
74  vector<string> hcat(const vector<string>& left,
75                      const vector<string>& right) {
76      vector<string> ret;
77      // Add 1 to leave a space between pictures
78      string::size_type width1 = width(left) + 1;
79      // Indices to look at elements from `left` and `right` respectively
80      vector<string>::size_type i = 0, j = 0;
81      // Continue until we've seen all rows from both pictures
82      while (i != left.size() || j != right.size()) {
83          // Construct new string to hold characters from both pictures
84          string s;
85          // Copy a row from the left-hand side, if there is one
86          if (i != left.size())
87              s = left[i++];
88          // Pad to full width
89          s += string(width1 - s.size(), ' ');
90          // Copy a row from the right-hand side, if there is one
91          if (j != right.size())
92              s += right[j++];
93          // Add `s` to the picture we're creating
94          ret.push_back(s);
95      }
96      return ret;
97  }
```

Local variable defined in loop

The `hcat` function defines a local variable (`s`) *inside* a loop.

This variable is:

- created;
- initialised (if appropriate);
- destroyed;

at *each* loop iteration.