Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Part 11 – Testat Review

IFS INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad / Thomas Corbat

Rapperswil, 24.11.2016

HS2016

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL

FHO Fachhochschule Ostschweiz

Word Class (word.h) – Header File

- **Include Guard**
  - Usually done correctly
  - Generated by IDE
- **Namespace**
  - Usually missing
  - Recommended for ADL

```
#ifndef WORD_H_
#define WORD_H_

<Includes>

namespace word {

<Declarations>

}

#endif /* WORD_H_ */
```

Word Class (word.h) – Class Declaration

- **Word Class Declaration**
- **Member for value**
- **Default Constructor**
  - Defaulted default constructor
  - Word(std::string)
    - `explicit` avoids implicit conversion
    - `const &` avoids unnecessary copy (More on that topic and proper alternatives in C++ Advanced)
- **Member Functions**
  - Constness

```
class Word {
  std::string value;
  static bool isValidWord(std::string const & value);

public:
  Word() = default;
  explicit Word(std::string const & value);

  void print(std::ostream & os) const;
  void read(std::istream & is);

  bool operator <(Word const & rhs) const;
};
```

Word Class (word.h) – Constructor/Function Parameters

	value	reference
non-const	`Word(std::string value)` ■ Argument gets copied ■ Modification of value in Constructor does not affect the call-site ■ Used for primitive and small types	`Word(std::string & value)` ■ Argument is used as is in memory ■ Modification of value in Constructor affects the call-site ■ Used when side-effect is desired
const	`Word(std::string const value)` ■ Argument gets copied ■ value cannot be modified in the Constructor ■ Used for primitive and small types	`Word(std::string const & value)` ■ Argument is used as is in memory ■ value cannot be modified in the Constructor ■ Used to pass (potentially) large objects

- **Non-Member Operators**
  - Outside the Word class
  - inline when implemented in header
- **When to implement a member in a header?**
  - Short (obvious) implementation
  - No additional (dependencies) Includes required

```
namespace word {
<Word Class>

inline std::ostream & operator <<(std::ostream & os, Word const & w) {
  w.print(os);
  return os;
}

inline std::istream & operator >>(std::istream & is, Word & w) {
  w.read(is);
  return is;
}
}
```

- **Relational Operators**
  - As members or free functions
  - Implementation can be in header
    - Short implementation
    - No additional dependencies
- **Similar to the Date class**
  - Use other (existing) operators
  - Beware of dependency loops!
    - Less (<) calls Greater (>) and vice versa
- **Implicitly inline if implemented in class**

```
namespace word {
<Word Class>
...
inline bool operator !=(Word const & lhs, Word const & rhs) {
  return !(lhs == rhs);
}

inline bool operator >(Word const & lhs, Word const & rhs) {
  return rhs < lhs;
}
}
```

Word Class (word.h) - Friends

- **Friend declaration**
  - `friend` keyword
  - Classes and (free) functions can be friends
- **Friends can access private/protected members from "outside"**
  - High coupling!
- **Almost always a design error!**
  - General rule: Avoid at any cost!
- **Implementing the input (>>) and output (<<) operators as friend arguably makes sense**
  - No read/print functions required
  - Input/output operators cannot be members
- **Don't implement the relational operators as friends**
  - They can be members

```
class Word {
  std::string value;
  ...
  friend bool operator <(Word const & lhs, Word const & rhs);
  friend std::ostream & operator <<(std::ostream & os, Word const & w);
};
```

Word Class (word.h) – Required Includes

- **<string>**
  - std::string
    - Instance created
- **<iosfwd>**
  - std::ostream (&)
    - Only passed as reference
    - No memory allocated or member used
  - std::istream (&)
    - Only passed as reference
    - No memory allocated or member used

```
class Word {
  std::string value;
  static bool isValidWord(std::string const & value);

public:
  Word() = default;
  explicit Word(std::string const & value);

  void print(std::ostream & os) const;
  void read(std::istream & is);

  bool operator <(Word const & rhs) const;
};
inline std::ostream & operator <<(
    std::ostream & os, Word const & w) {
  w.print(os);
  return os;
}
inline std::istream & operator >>(
    std::istream & is, Word & w) {
  w.read(is);
  return is;
}
```

- **Default Constructor**
  - Either in header with = default
    `Word() = default`
  - Or explicitly in source file when setting a default value

```
Word::Word()
    : value{"default"} {
}
```

- **Type conversion constructor**
  - Set the value in initializer list
  - Throw an exception if not value to prevent creation of an instance that violates the invariant

```
Word::Word(std::string const & value)
    : value { value } {
  if (!isValidWord(value)) {
    throw std::invalid_argument{"msg"};
  }
}
```

Word Class (word.cpp) – Validity Check

- **Provide a function to test validity of a word value**
  - Check for emptiness
  - Check all characters are isalpha[1]
    - std::isalpha is ambiguous as it is a template and overloaded
      - Error: <unresolved overloaded function type>
      - Cast explicitly to required function type: `static_cast<int(*)(int)>(std::isalpha)`
    - Or use the C isalpha implementation not in namespace `std: ::isalpha`

```
bool Word::isValidWord(std::string const & value) {
  return !value.empty() &&
    std::all_of(
      std::begin(value),
      std::end(value),
      static_cast<int(*)(int)>(std::isalpha));
}
```

[1] http://en.cppreference.com/w/cpp/string/byte/isalpha

Word Class (word.cpp) – Optimization for Equality

- **Implementation of Less (<) Operator**
  - Same as the comparator in the *wlist* exercise
  - std::lexicographical_compare
  - One pass over both string values
- **Implementation of Equal (==) Operator**
  - Either use less (<) operator
    `return !(lhs < rhs) && !(rhs < lhs)`
- **Slightly more efficient with std::equal**
  - Especially for equal words (2x)

```
bool Word::operator <(Word const & rhs) const {
  return std::lexicographical_compare(
    std::begin(value), std::end(value),
    std::begin(rhs.value), std::end(rhs.value),
    [](char l, char r) {
      return std::tolower(l) < std::tolower(r);
    });
}
```

```
bool Word::operator ==(Word const & rhs) const {
  return std::equal(
    std::begin(value), std::end(value),
    std::begin(rhs.value), std::end(rhs.value),
    [](char l, char r) {
      return std::tolower(l) == std::tolower(r);
    });
}
```

Word Class (word.cpp) – Printing Words

- **Printing function to implement non-friend Output (<<) Operator**
  - const since it does not modify the `this` object
  - Just output `value`

```
void Word::print(std::ostream & os) const {
  os << value;
}
```

Word Class (word.cpp) – Reading Words (Stream States)

- **Set `failbit` when Word could not be read successfully: Since Input (>>) Operator from Word skips all leading non-alpha characters this only happens at EOF**
  - A stream can recover from the fail state with `clear()`
- **The eofbit is set when trying to read beyond the last character in the input**
- **A stream may end up in bad state (badbit set) when reading in non-"good" state**

eofbit	failbit	badbit	good()	fail()	bad()	eof()	operator bool	operator!
false	false	false	true	false	false	false	true	false
false	false	true	false	true	true	false	false	true
false	true	false	false	true	false	false	false	true
false	true	true	false	true	true	false	false	true
true	false	false	false	false	false	true	true	false
true	false	true	false	true	true	true	false	true
true	true	false	false	true	false	true	false	true
true	true	true	false	true	true	true	false	true

Table from: http://en.cppreference.com/w/cpp/io/basic_ios/good

- **Skipping non-alpha characters first**

```
while (is.good() && !std::isalpha(is.peek())) {
  is.ignore();
}
```

  - Use is.good() in the condition as this will prevent to read from the stream in EOF state with peek()
    - Better than `is && !std::isalpha(is.peek())`
  - Use `is.ignore()` over `is.get()` since it communicates the intention better

- **Read alpha characters into a separate buffer**

```
std::string buffer{};
while (is.good() && std::isalpha(is.peek())) {
  buffer += is.get();
}
```

Word Class (word.cpp) – Checking Input

- **Use same function to check input as in the constructor**
  - `buffer.empty()` might be sufficient here and now, but that might change later
- **If no valid word could be read from the input...**
  - ... don't overwrite value, in order to retain the class invariant
  - ... set the failbit to signalize the problem
  - ... if your implementation might have set the failbit when reading a word successfully, clear the state!

```
if (isValidWord(buffer)) {
  value = buffer;
} else {
  is.setstate(std::ios_base::failbit);
}
```

You may read the element multiple times (*it), BUT...
... after increment (++) all previous copies of the iterator become obsolete!

Input Iterator

```
  struct input_iterator_tag { };
T const operator *()
operator++()
operator++(int)
operator==(myiter) // and !=
```

- Supports reading the "current" element, ~~but only once (*it)~~
  - ~~must increment the iterator afterwards with ++~~
- Allows for one-pass input algorithms (++)
- Models the std::istream_iterator and std::istreambuf_iterator
- can compare iterators == !=
- Most other iterators are also input iterators

Beware! According to this you must NOT use an input iterator as follows: *it++
However, the C++ Standard explicitly allows it...

Word Class (word.cpp) – Reading with Iterators

- **Possibility to avoid loops using iterators**

```
void Word::read(std::istream & is) {
  std::istreambuf_iterator<char> input{is}, eof{};
  auto alpha_start = std::find_if(input, eof, ::isalpha);

  std::string buffer{};
  std::find_if(alpha_start, eof, [&buffer](char c) {
    bool end_of_word {!std::isalpha(c)};
    if (!end_of_word) {
      buffer += c;
    }
    return end_of_word;
  });

  if (isValidWord(buffer)) {
    value = buffer;
  } else {
    is.setstate(std::ios_base::failbit);
  }
}
```

Word Class (word.cpp) – Required Includes

- **"word.h"**
  - Own header
- **<algorithm>**
  - std::all_of
  - std::lexicographical_compare
  - std::find_if
- **<iterator>**
  - std::begin
  - std::end
  - std::istreambuf_iterator
- **<cctype> (maybe <locale>)**
  - std::isalpha
- **<stdexcept>**
  - std::invalid_argument
- **<istream>**
  - Use of istream members (ignore, peek, get)
- **<ostream> / <iosfwd>**
  - Use of ostream
- **<string>**
  - std::string
  - operator << (std::ostream&, std::string)

- **One ASSERT per test**
  - General rule
  - Provides better overview when tests fail
- **Relational Operators**
  - Never ASSERT_EQUAL(true, XXX)
  - Prefer the corresponding ASSERT macros
    - E.g. ASSER_LESS for Less (<) Operator
  - Don't invert actual and expected

```
ASSERT_EQUAL(true, expected == actual);
ASSERT(expected == actual);
ASSERT_EQUAL(actual, expected);
```

- **If you don't understand the meaning of given tests, ask...**

```
void test_exercise_example() {
  std::istringstream input{
    "compl33tely ~ weird !!??!! 4matted in_put "};
  Word w{};
  input >> w;
  ASSERT_EQUAL(Word{"compl"}, w);
  input >> w;
  ASSERT_EQUAL(Word{"tely"}, w);
  input >> w;
  ASSERT_EQUAL(Word{"weird"}, w);
  input >> w;
  ASSERT_EQUAL(Word{"matted"}, w);
  input >> w;
  ASSERT_EQUAL(Word{"in"}, w);
  input >> w;
  ASSERT_EQUAL(Word{"put"}, w);
  input >> w;
  ASSERT_EQUAL(Word{"put"}, w);
}
```

```
std::istringstream input{"compl33tely ~ weird !!??!! 4matted in_put "};
```

```
..\src\Test.cpp:170:50: warning: trigraph ??! converted to | [-Wtrigraphs]
  std::istringstream input{"compl33tely ~ weird !!??!! 4matted in_put "};
```

Primary	Digraph	Trigraph(until C++17)
{	<%	??<
}	%>	??>
[	<:	??(
]	:>	??)
#	%:	??=
\		??/
^		??'
|		??!
~		??-

- **Alternative operator representations**
  - Required when corresponding characters are not present in the used character set
    - E.g. ISO 646:1983
- **Will be removed in C++17**
- **Can have surprising effects**

```
//Print nothing ??/
std::cout << "nothing\n";
```

Table from: http://en.cppreference.com/w/cpp/language/operator_alternative

Kwic Function (kwic.h)

- **Include Guard**
- **Namespace**
  - Maybe "word" too
  - Not as important as for word because we don't have operators (externally)
- **Include of iosfwd**
  - std::istream &
  - std::ostream &

```
#ifndef KWIC_H_
#define KWIC_H_

#include <iosfwd>

namespace kwic {

void kwic(std::istream & in, std::ostream & out);

}

#endif /* KWIC_H_ */
```

Kwic Function (kwic.cpp) – kwic() Function

- **Kwic basically has to do three things**
  - Read lines from the input
  - Rotate all lines
  - Print all rotations
- **So we need more or less three statements performing the corresponding action**

```
void kwic::kwic(std::istream & input, std::ostream & output) {
  sorted_lines input_lines = readLines(input);
  sorted_lines rotated_lines = rotate_lines(input_lines);
  std::copy(
     std::begin(rotated_lines),
     std::end(rotated_lines),
     std::ostream_iterator<line>{output, "\n"});
}
```

```
sorted_lines input_lines = readLines(input);
```

- **What is `sorted_lines`?**
  - We don't need to create a type explicitly if another type already satisfies the functionality
  - Due to template arguments and name qualifiers, names can become long
  - So let's make the code a bit shorter with the `using` keyword

```
using word::Word;
using line = std::vector<Word>;
using sorted_lines = std::set<line>;
```

- **`using word::Word` allows us to write Word, instead of word::Word**
- **`using line = std::vector<Word>` gives us a nice name for a type representing a line**
- **`using sorted_lines = std::set<line>` holds multiple lines in lexicographical order**

Kwic Function (kwic.cpp) – std::set vs. std::multiset

- **We used std::set in the previous slide**
  - The exercise did not specify whether we allow duplicates
  - In the original purpose (of kwic) it was unlikely to have the same title twice
- **If you want to allow duplicate lines use std::multiset instead**

```
using sorted_lines = std::multiset<line>;
```

- **You could argue that after reading the input the lines need not be sorted**

```
using unsorted_lines = std::vector<line>;
```

Kwic Function (kwic.cpp) – Reading Lines

- **Read input lines in while-Loop and std::getline**
  - Could be solved using a (custom) line input iterator – we haven't covered this topic though
- **Convert each line (`std::string`) into a stream**
- **Fill a `line` (alias for `std::vector<Word>`) using `std::istream_iterator<Word>`**
- **Add the `line` to the unsorted input lines**

```
unsorted_lines readLines(std::istream & in) {
  unsorted_lines all_input_lines{};
  std::string input_line{};
  while(std::getline(in, input_line)) {
    std::istringstream line_stream{input_line};
    std::istream_iterator<Word> word_it{line_stream}, eof{};
    all_input_lines.push_back(line{word_it, eof});
  }
  return all_input_lines;
}
```

Kwic Function (kwic.cpp) – Rotating Lines

- **Take unsorted_lines by const &**
- **For each line add all rotations into a `sorted_lines` (`std::set<line>`) container**
- **For the rotation use either `std::rotate_copy` or `std::rotate`**
  - std::rotate has a side-effect on the input

```
sorted_lines rotate_lines(unsorted_lines const & input_lines) {
  sorted_lines rotated_lines{};
  for_each(begin(input_lines), end(input_lines), [&](line line_to_rotate) {
    for (auto it = begin(line_to_rotate); it != end(line_to_rotate); it++) {
      line rotated_line{line_to_rotate.size()};
      std::rotate_copy(
        std::begin(line_to_rotate), it, std::end(line_to_rotate),
        std::begin(rotated_line));
      rotated_lines.insert(rotated_line);
    }
  });
  return rotated_lines;
}
```

Kwic Function (kwic.cpp) – Printing Sorted Lines

- **What is required for this to work?**

```
std::copy(
      std::begin(rotated_lines),
      std::end(rotated_lines),
      std::ostream_iterator<line>{output, "\n"});
```

- **Output (<<) Operator for line aka std::vector<Word>**
- **Where do you have to define this operator? (std, word, global?)**
  - Argument Dependent Lookup
  - Namespace `word` is the best option

```
namespace word {
  std::ostream & operator<<(std::ostream & os, line const & l) {
    std::copy(
      std::begin(l), std::end(l),
      std::ostream_iterator<Word>{os, " "});
    return os;
  }
}
```

Kwic Function (kwic.cpp) – Required Includes

- **"kwic.h"**
  - Own header
- **"word.h"**
  - Word class
  - Input (>>) Operator for Word
  - Output (<<) Operator for Word
- **<vector>**
  - std::vector
- **<set>**
  - std::set or std::multiset
- **<string>**
  - std::string
- **<sstream>**
  - std::istringstream
- **<iterator>**
  - std::begin
  - std::end
  - std::istream_iterator
  - std::ostream_iterator
- **<algorithm>**
  - std::for_each
  - std::rotate or std::rotate_copy
  - std::copy

Main Function (main.cpp)

- **Nothing but a call of kwic()**
  - Arguments `std::cin` and `std::cout`
- **No logic unless for user interaction**
  - Not testable
- **Requires Includes**
  - `kwic.h` for `kwic::kwic`
  - `iostream` for `std::cin` and `std::cout`

```
#include "kwic.h"
#include <iostream>

int main() {
  kwic::kwic(std::cin, std::cout);
}
```

Questions?

No additional exercises this week!