Exercise in C++ Programming for CE
ExC++PCE
Winter Term 2014/2015

Assignment sheet B

Assignments that are marked with *StudOn submission* are **mandatory** and must be submitted via StudOn in time – please see there for deadlines.

Redirection revisited

As this assignment is mainly about transforming I/O, mastering file redirection provides great advantages. It does not only save you from mindlessly typing test input, but also allows to automatically check the output for the respective test case by writing it to a file and comparing that to a reference solution.

```
# ./program < input.txt > output.txt
# diff -qs output.txt reference.txt
Files output.txt and reference.txt are identical
```

Basically all OSes have low-level tools for comparing files pre-installed, on Unix-like systems the command `diff` (please see `man diff` for details) and its Windows counterpart `fc`. There are, however, many tools that can also graphically visualize differences.

Error conditions during stream I/O

In the lecture, you already learned basic I/O operations using streams, like `std::cin` and `std::cout`. For some of the exercises, you will also be able to to check the success of I/O operations. How can one, for instance, detect that one has read up to the end of a redirected file? Especially novice C++ programmers want to find out in advance *if there will be more data to read.* However, this is *not* the way it works.

Instead, C++ streams rely on trial-and-error: One attempts an I/O operations and checks *afterwards* if it was successful. The result of operations is stored in a flag field associated with each stream that also allows to differentiate the type of failure. Common reasons are reaching the end of an input file or the impossibility to convert the input, for instance when reading “Didgeridoo” and trying to store it in a numeric value. Consequently, there is no (simple) way to detect if the last operation read the last character of file! Only the succeeding operation will fail and allow to detect this.

Feel free to already read up on the details of error handling for streams, but being able to check success or failure of an I/O operation should be sufficient for this assignment sheet. This is done through the `good()` member function of a stream.

```
float f;
std::cin >> f;
if ( ! std::cin.good() ) {
    std::cerr << "Something went wrong, "
                 "perhaps I read after end-of-file "
                 "or I encountered a non-numeric value!"
              << std::endl;
    exit( EXIT_FAILURE ); // bail out - requires #include <cstdlib>
}
```

Through mechanism that cannot be explained for now, streams, references to streams, and expressions that result in such a reference can be used directly in conditionals, allowing to write more curtly:

```
std::string str;
double d;

if ( ! ( std::cin >> d ) ) {
    // something went wrong
}

while ( std::getline( std::cin , str ) ) {
    // successfully retrieved another line
}
```

Even typing manually into a terminal window, you can instruct the command shell to stop forwarding input to a controlled program in way it will encounter an end-of-line. On Unix-like systems, you will have to type *control+D* (probably with subsequent return key, because shells usually buffer input line-wise), on Windows *control+Z*. Try not to confuse them, using control-Z on Unix-like systems will suspend the current program – you can continue using the shell command `fg`.

1 Scalar product (`B1_scalp.cpp`)

Write a program that computes the $\mathcal{L}_2$-norm a vector. It shall first prompt for the vector size (you may want to use `unsigned long` or `std::vector< double >::size_type` for this) and subsequently read as many values to populate a `std::vector< double >`. Afterwards it shall compute and write the vector norm to `std::cout`. The required square root is defined in the `cmath` library header.

2 Prefix sum (`B2_scan.cpp`)

The scan for a series $x_1, \ldots, x_n$ results in the series $x'_1, \ldots, x'_n$ with $x'_k := \sum_{i=1}^{k} x_i$, which can be computed more efficiently by trivially deducing $x'_1 = x_1$ and $x'_k = x'_{k-1} + x_k$ for $k > 1$.

Write a program that reads from `std::cin` as long as possible and collects these values in a `std::vector< double >`. Now use iterators or a range-based for-loop to compute the prefix sum in-place and eventually write it to `std::cout` with one value per line.

3 Matrix-matrix product (`B3_mmp.cpp`)

The purpose of the program is to read two matrices from `std::cin` and write their product to `std::cout`. As storage for the three matrices, `std::vector< std::vector< double > >` are to be used.

The program shall first read three integral numbers, in the following denoted s_1, s_2, and s_3. They specify the dimensions of the matrices, $m_1 \in \mathcal{R}^{s_1 \times s_2}, m_2 \in \mathcal{R}^{s_2 \times s_3}, m_3 \in \mathcal{R}^{s_1 \times s_3} = m_1 m_2$.

It then reads $s_1 \times s_2$ numbers that are used to populate m_1 row by row, and then $s_2 \times s_3$ numbers that are used to populate m_2 analogously.

It now computes m_3 before it is eventually written to `std::cout` with line breaks after each row and white space as separators. Tests that are provided for your convenience use a tab as separator after *every* coefficient, including the last element in a row.

4 Matrix-matrix product with linearized multi-dimensional arrays (`B3_mmp_lin.cpp`)

Copy and modify your `B2_mmp.cpp` so that is uses only a single, larger `std::vector< double >` for each matrix and computes the effective index for the multi-dimensional access.

5 Pointers, references, and const-ness

StudOn submission

Which of the following initializations are legal and which are not? If previous variables are referred, only consider the declared type of that variable and ignore if the respective definition was legal.

(a) `int i = -1;`	☐ correct	☐ invalid
(b) `int const ci = i;`	☐ correct	☐ invalid
(c) `int i2 = ci;`	☐ correct	☐ invalid
(d) `int & ri = ci;`	☐ correct	☐ invalid
(e) `int const & rci = &ci;`	☐ correct	☐ invalid
(f) `int const * pci = &i;`	☐ correct	☐ invalid
(g) `int * const cpi = &ic;`	☐ correct	☐ invalid
(h) `int const * const & rcpci= pci;`	☐ correct	☐ invalid
(i) `int const * const cpci = &ci;`	☐ correct	☐ invalid
(j) `int const & rci2 = *cpci;`	☐ correct	☐ invalid
(k) `int const & icr3 = *cpi;`	☐ correct	☐ invalid
(l) `int i3 = rci2;`	☐ correct	☐ invalid

6 String duel (`B6_stringduel.cpp`)

Write a program that reads two words from standard input and stores them in `std::string`s. Use iterators or a range-based for-loop to convert both strings to lower case – please see documentation of the `cctype` library header for this functionality. Now compare the two words as follows: If either is longer than the other, it "wins". If they are equally long, the one that comes later in lexicographic comparison[1] wins. If both strings are identical, it's a tie.

Optimally, your code should also justify its rating.

7 Punctuation (`B7_punctuation.cpp`)

The program shall read a line from standard input and store it in a `std::string`. Use *iterators* to seek the string and copy every character that is no punctuation (see library header `cctype` for this check) to another string, before printing this string. This should be repeated until reading another line fails.

The resulting program should be usable as a filter like this:

```
# ./B3_punctuation < with_punct.txt > no_punct.txt
```

8 Lowercase reversion (`B8_lcrev.cpp`)

The program shall fill a `std::vector< std::string >` by reading from `std::cin` as long as possible. After no more lines can be read, the program uses iterators to print the whole input reverse (i. e. starting with the last character of the last line), converting all characters to lower case. Once again, the library header `cctype` will become useful.

[1] this functionality is already provided by the standard library, there is no need for extended coding

9 Iterators

StudOn submission

The following variable definitions are given:

```
std::vector< double > dvec = { 0 , 1 , 2 };
std::vector< double > const cdvec = { 0 , 1 , 2 };
```

Now indicate for the following code fragments if they are correct or, if not, which error they contain. Please note that due to the initialization used in this example the code must be compiled either with the *-std=c++0x* or the *-std=c++11* switch, depending on the version of you compiler.

(a)

```
for ( auto dvit = dvec.begin() ; dvit!=dvec.end() ; ++dvit ) {
    ++(*dvit);
}
```

(b)

```
double s( 0 );
auto cdvit = cdvec.cbegin();
while ( ( *cdvit < 42 ) && ( cdvit!=cdvec.cend() ) ) {
    s += *cdvit;
}
```

(c)

```
for ( auto dvit = dvec.rbegin() ; dvit!=dvec.rend() ; ++dvit ) {
    *dvit = *dvit / 2;
}
```

(d)

```
for ( auto dvit = dvec.end() ; dvit!=dvec.begin() ; --dvit ) {
    *dvit = *dvit * *dvit;
}
```

(e)

```
double prod( 0 );
for (std::vector< double >::iterator const cdvit = cdvec.begin() ;
     cdvit != cdvec.end() ;
     ++cdvit
) {
    prod *= *cdvit;
}
```
