Calculation of Levenshtein Distance with Dynamic Array Sizes

No explicit evaluation of this exercise (no points).

Doing this exercise is, however, needed to be able to do the next exercise.

Learning Objectives

- new and delete, new[], delete[]
- Testing (Test-First, i.e. Think, Red-Bar, Green-Bar, Refactor)
- Code and files
- Command line parameters (argv, argc)

Exercise at a glance

- Re-Implement your class `Levenshtein` now with dynamic array sizes, so that the internal matrix is not restricted any more to 1000 elements
- Split your implementation into different files.
- Use command line parameters for main

Detailed Exercise Descriptions

In this exercise we do not implement new functionality to reach the final aim of extraction of callsigns from ATC transcriptions, but just learn how to implement dynamic C-arrays and split the code into different files.

Exercise 2.1

Re-implement your code from the previous exercise, so that dynamic C-Arrays are possible inside the Levenshtein distance class. Do not use the STL-template class `vector` for the matrix. For the rest of the code you may use the vector template class.

```
class Levenshtein
{
public:
   // astr1
   Levenshtein(
      const std::vector<std::string>& astr1,  // first string compared to astr2
      const std::vector<std::string>& astr2); // i.e. LD between
                                              // astr1 and astr2 is calculated
   /* … */

private:
   // contains the matrix, used for LD calculation as a vector
   int* mpi_mat; // Use here a pointer to int and not a C-Array
   /*… */
```

```
};[1]
```

Do not forget to implement suitable tests.

Do not forget to free all Heap-memory allocated by new or new[] with delete and delete[], respectively.

Exercise 2.2

Split your code into different files, e.g.

- One main file, which executes the tests
- Header and Source-Code file for the class Levenshtein distance calculation
- Header and Source-Code files for the different tests
- …

Use include guards (#ifndef, #define or/and #pragma once).

Exercise 2.3

Split your main program into two parts. If it is called with the command line parameter `--test` all the tests are executed.

Otherwise a test file (you decide which) with some ATC utterances is read in and the contents is output to cout or a file. Later you will also output the extracted callsigns. Currently it is enough to just use the `--test` parameter in main.

Your `main` could look like this:

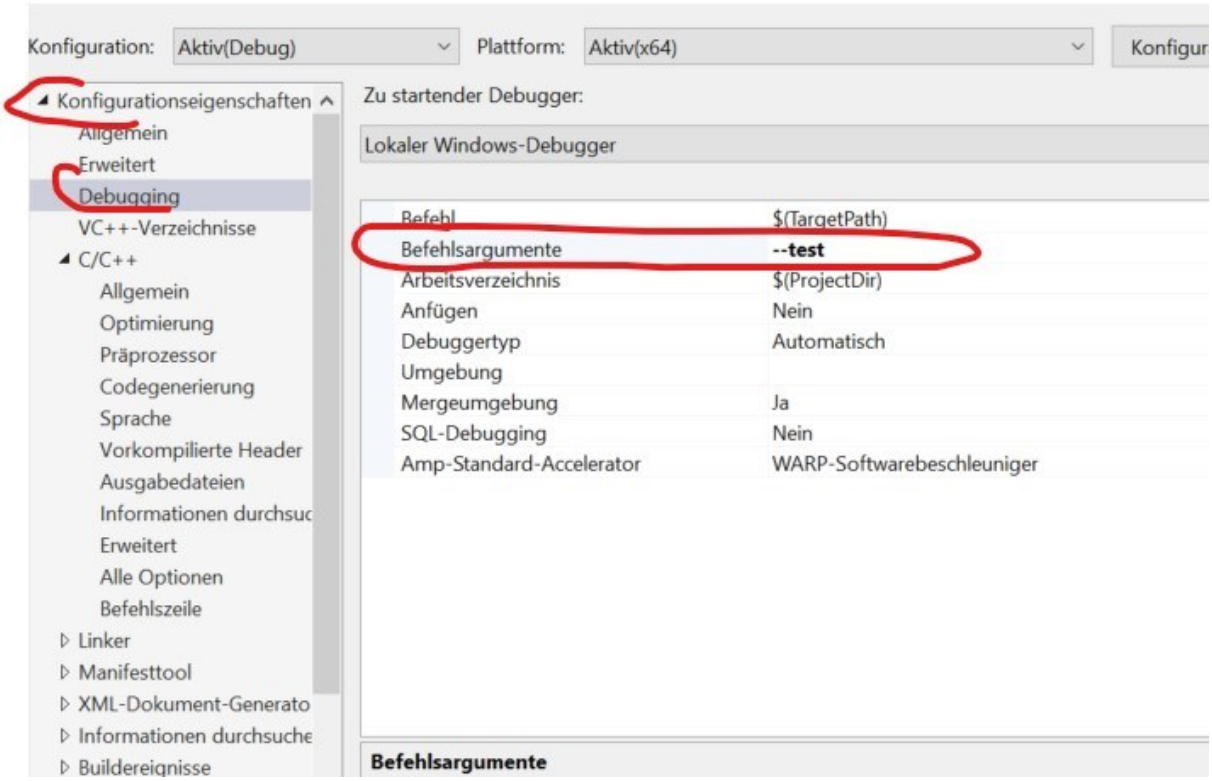
```
int main(int argc, char* argv[])
{
   if (argc > 1 && string(argv[1]) == "--test")
   {
      bool result = true;
      PERFORM_AND_OUTPUT(checkTop8InBigFile); // or other tests
      PERFORM_AND_OUTPUT(checkTop5InMediumFile); // or other tests
      if (true == result) {
         printScreenColorOnceVal(cout,
                GREEN_SCREEN_COLOR, "Tests erfolgreich\n");
         return 0;
      }
      else {
         printScreenColorOnceVal(cout, RED_SCREEN_COLOR,
```

[1] It might be easier to implement it as a vector<vector<int>>, but for learning progress and future exercise, we just use this approach.

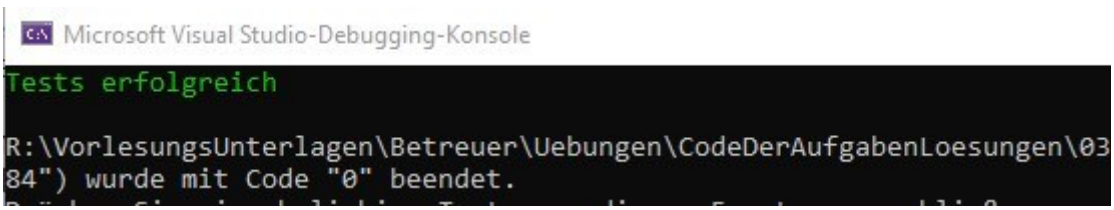
```
            " Fehler in Tests aufgetreten! ***\n");
        return -1;
      }
   }
   ReadFileAndPrintContents();
   return 0;
}
```

The above program benefits from the possibility that you can pass parameters via the command line, when you call a C++program.

You can provide parameters via the IDE of Visual Studio as shown in the following Figure



Execute your program now in test mode (parameter --test) and create a screen dump. The screen dump could look like shown in Figure \ref{exercise5-3-ProgramRunsTest}



Execute your program now NOT in test mode and create a screen dump, when you read your input file.

Tipps, Tricks and Constraints

If you are not developing under Windows (e.g. Linux) and not with Visual Studio, please make sure that all your files are in one directory (header, source, data files)[2] or provide a cmake file.[3],

Do not use path dependent file paths at different places in your program, i.e. do not write

```
   ofsteam str("C:\\VorlesungsUnterlagen\\Betreuer\\Uebungen\\"
               "CodeDerAufgabenLoesungen\\03b_ExeManyFiles\\x.txt";
```

better use a function as e.g. `AppendFilenameToPath`, shown in the following code fragments:

```
bool checkBlaBla()
{
   string filename =
     AppendFilenameToPath("data\\BigWordSeqPlusCmdsFile.txt");
/* … */
```

You could use the following implementation[4] (of course your basic path is different):

```
string GetPathPrefix()
{
#ifdef _WIN32  // on Windows system
   return "R:\\Uebungen\\CodeDerAufgabenLoesungen\\03b_ExeManyFiles";
#else
   // \toDo not implemented
   return "./";
#endif
}

string AppendFilenameToPath(std::string astr_filename)
{
#ifdef _WIN32  // on Windows system
   return GetPathPrefix() + "\\" + astr_filename;
#else
   return GetPathPrefix() + "/" + astr_filename;
#endif
}
```

[2] I will always try to run and compile on my Windows system. If your code is split in different directories, creating a Visual Studio project is too difficult for me.
[3] Do not forget to test, that your project also compiles and runs under Windows on my system. Ask a colleague with Windows Visual Studio installation.

[4] In that case I might have a chance to make your code running on my system, if you did not implement directory independent file names.