Script for the vhb lecture

Programming in C++
Part 2

Prof. Dr. Herbert Fischer
Deggendorf Institute of Technology

Table of Contents

Primary literature:

Breymann Ulrich
Derr C++ Programmierer, Hanser, 2. Auflage, 2011
ISBN 3-4464-2691-7

Kirch-Prinz Ulla, Kirch Peter
C++ Lernen und professionell anwenden, mitp, 2.Auflage, 2002
ISBN 3—89842-171-6

May Dietrich
Grundkurs Softwareentwicklung mit C++, vieweg, 2.Auflage, 2006
ISBN 3-8348-0125-9

Einsenecker Ulrich
C++: Der Einstieg in die Programmierung, W3L GmbH, 1.Auflage, 2008
ISBN 3-9371-3712-4

Secondary literature:

André Willms
C-Programmierung lernen
ISBN: 978-3827314055, Addison-Wesley Verlag, 6. Auflage, 1998

André Willms
C++-Programmierung lernen
ISBN: 978-3827326744, Addison-Wesley Verlag, 1. Auflage, 2008

André Willms
C/C++-Workshop
ISBN: 978-3827316622, Addison-Wesley Verlag, 1. Auflage, 2000

Ute Claussen
Objektorientiertes Programmieren
ISBN: 978-3540579373, Springer-Verlag, 2. Auflage, 2013

Oliver Böhm
C++ echt einfach
ISBN: 978-3772374104, Franzis Verlag, 3. Auflage, 2006

Thomas Strasser
C++ Programmieren mit Stil
ISBN: 978-3898642217, dpunkt.Verlag, 2003

John R. Hubbard
C++ Programmierung
ISBN: 978-3826609107, mitp-Verlag, 2003

Arnold Willemer
Einstieg in C++
ISBN 978-3836213851, Rheinwerk-Verlag, 4.Auflage, 2009

Tools:

Code::Blocks for Windows, Linux, Mac OS (free Software): **http://codeblocks.org/downloads/26**

Alternatives:
CodeLite for Windows, Linux, Mac OS: **http://downloads.codelite.org**
KDevelop for Windows, Linux: **https://www.kdevelop.org/download**
Dev-C++ for Windows: **http://www.bloodshed.net/dev/**
XCode for Mac OS: **https://itunes.apple.com/de/app/xcode/id497799835**

Recommended websites:
http://www.c-plusplus.de/cms
http://community.borland.com/cpp/0,1419,2,00.html
http://www.willemer.de/informatik/cpp/index.htm

1 File processing & error handling

As we all know, the data stored in the main memory of a program is lost when a program is closed. What can we do about that? The answer is very simple. To permanently save data, it must be stored in a separate file.

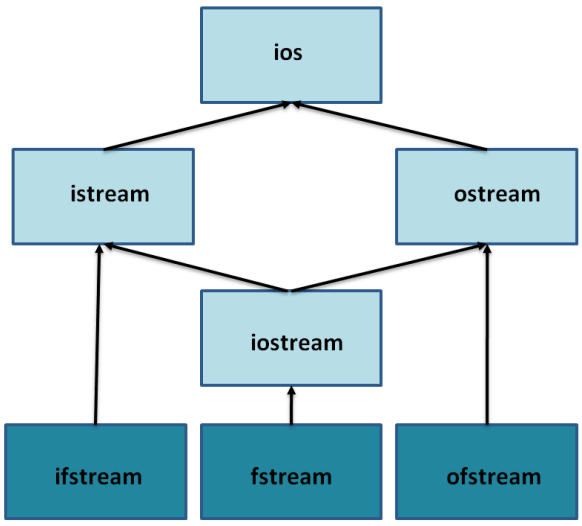
1.1 File operations

Strings or single characters can be written into a text file just like they can be printed on the screen. However, data sets are often stored in a file. But what is a data set? A data set contains data that logically belongs together, such as personal data or bank account details. During the writing operation, a data set is transferred to the file, which means that the data set in the file is updated or added. During the reading operation, however, this data set is copied from the file into a data structure of the program. In this way, objects are permanently stored.

From the point of view of a C++ program, a file is a large "byte-vector". Structuring the data sets is one of the programmer's tasks. This gives you the advantage of maximum flexibility.

Each character in a file has its own byte position. The first character has position 0, the second 1, etc. The current position in the file is the position of the byte that is read or written next. When the file is *accessed sequentially*, the data is always read or written consecutively. The first reading operation always starts at the very beginning of the file. If a certain piece of information is to be retrieved from the file, the file content must be searched consecutively from the very beginning of the file. When writing into the file, new or existing information can be overwritten. New data can also be added at the end of an existing file.
In contrast to sequential access, there is the option of *accessing files randomly*. This allows setting the current file position as you like.

C++ provides various default classes for file processing. The easy handling of files is made possible by so-called *file stream classes*. Above all, they take care of the management of stream buffers and system-specific details.



The above class hierarchy shows that the file stream classes have the stream classes we already know as base classes:

- the ifstream class is derived from the istream class and makes the reading of files possible
- the ofstream class is derived from the ostream class and makes writing to files possible
- the fstream class is derived from the iostream class and makes both reading of and writing to files possible.

1.1.1 Opening, writing to and closing a file

To open a file, an object of the fstream class is required. Afterwards the file is opened or newly created by calling the open()function. By calling the close() function the file is closed again.

To edit a file, it must be opened first. This involves
- specifying a file name, which can contain a path, and
- defining a so-called file opening mode.

If there is no path specification, the file has to be located in the same directory. The file opening mode determines in particular whether the file is opened for reading or writing, or both.

```
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
   fstream f;
   f.open("test.txt", ios::out); // Opening the file for writing
   f << "This text is written into the file. << endl; // Write string into the file
   f.close(); // Closing the file

   system("pause");
   return 0;
}
```

File opening modes:

Constant	Explanation
ios::in	Open an exisiting file for reading
ios::out	For writing
ios::trunc	File content is deleted when opened
ios::app	Written data is added at the end of the file. Before each writing operation, the position is set to the end of the file.
ios::ate	Set position pointer to the end of the file
ios::binary	Perform writing/reading operations in binary mode

Various opening modes can be combined using the bitwise operator |.

Note: The constructor and the open()function of the ifstream and ofstream file stream classes use the following default values:

```
ifstream    ios::in
ofstream    ios::out | ios::trunc
```

1.1.2 Reading from a file

You can simply use cin and cout to manage the stream objects for files.

To read from files, the stream extraction operator >> is used, which works with fstream objects in the same way as with cin. The file is read as if you were typing the content on the keyboard.
To be able to read spaces from the files, you must use the getline()member function. A pointer to char is passed as the first parameter, and the maximum number of characters that fits into the buffer is passed as the second parameter.

Example: Read from the test.txt file

```
#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
   fstream f;
   char cstring[256];
   f.open("test.txt", ios::in);                  // Opening the file
   while(!f.eof()) {                             // As long as the end of the file has not been reached
         f.getline(cstring, sizeof(cstring));    // Read a single line and save it in cstring.
         cout << cstring << endl;
   }

   f.close();

   system("pause");
   return 0;
}
```

There is, however, also a global function called getline(), which does not have fstream as its first parameter but an object of the ifstream type. This function also works with the string standard class and also accepts objects of the string type as second parameter. The example from above would look like this:

```
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main()
{
   ifstream f;
   string s;
   f.open("test.txt", ios::in);          // Opening the file
   while(!f.eof()) {                     // As long as some data is still available
         getline(f,s);                   // Reading a line
         cout << s << endl;              // Print it on the screen
   }
   f.close();                            // Closing the file

   system("pause");
   return 0;
}
```

1.1.3 User-defined file

The next example is to show how to create user-defined files with a structured setup and how to read out the stored values.

Example:

```
#include <iostream>
#include <cstdlib>
#include <fstream>

using namespace std;

int main()
{
   ofstream writingFile("employee.txt");
   ifstream readingFile("employee.txt");

   cout << "Employee no, name, salary" << endl;

   int no;
   string name;
   double salary;
   bool next;

   do
   {
      cin >> no >> name >> salary;  //Input No, Name, Salary, separated by spaces

      writingFile << no << ' ' << name << ' ' << salary << endl;  // User-defined output with the values
                                                                  // Save in employee.txt file
      cout << "NEXT? [1] YES [0] NO: ";
      cin >> next;
   }while(next == 1);                                             // Enter data as long as next equals 1

   writingFile.close();                                           // Closing file

   cout << endl << "--- READING ---" << endl;

   while(readingFile >> no >> name >> salary)          // Reading from employee.txt file
   {                                                   // Assign each column value of a line to the variable.
      cout << no << ", " << name << ", " << salary << endl;       // Output of the individual lines with the values
   }

   system("pause");
   return 0;
}
```

Explanation:
We use an *ofstream* to create a file and to store (write) the data entered to the file and an *ifstream* to read the data from the file. The user is asked to enter a number, a name and a salary. This is repeated until the user enters the value 0 when asked “Next?”. Since there are three variables at *cin*, you can enter them directly, in one go, separated by a space each. Example: 1 Max 1999.90
Each line read is then pushed into the ofstream object and the lines are stored in the employee.txt file. When you terminate the loop, the file is closed.
A loop is also created to output the data. The ifstream object is used to output the data. Since we know that our file is structured in three columns, we can tell the ifstream object to store the values read in the *no., name and salary* variables. The data is output within the loop. This continues until we reach the end of the file. To close the file, the *close()* function is not needed, since the ifstream object knows that it has reached the end and therefore closes the file automatically.

1.2 Error handling

1.2.1 Try, catch, throw

C++ provides a mechanism that makes it possible to secure a block of statements against crashes. All exception errors occurring in the statement block are forwarded to a treatment block. The secured statement block is initiated by the keyword *try*. The statement block for handling errors starts with the keyword *catch*. The statement block therefore attempts to execute the program without errors. The statement block for handling errors catches the crashes like a net.

Example: Try, catch

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
   Try                                  // Instruction where an error could occur
   {
      int divisor = 0;
      int dividend = 1;
      int quotient = dividend/divisor; // An error occurs, the program would crash!
   }
   catch (...)                          // This generally intercepts the error and throws an error message.
   {
      cerr <<  "Problem detected…"<< endl; // Output of the error using cerr
   }

   system("pause");
   return 0;
}
```

Explanation:
The division by 0 in the example would normally cause a program crash. Such an error is called an exception. If such an exception occurs within a try-block, the processing is continued in the catch-block that handles the exception.

Of course, this problem can be avoided by checking the divisor before each division. However, the try statement deliberately removes error handling from the standard program flow. This separation of code and error handling results in programs that are easier to read and to maintain. Another advantage is that not each sub-step has to be checked individually for any conceivable error situation (see examples in Section 1.2.2).

Within the brackets of the catch statement, the type of exception to be caught by this block is specified. It is possible to define several catch blocks for different types in a row. The three dots following the catch statement are a placeholder for all types and denote the general exception handling case so that all exceptions that have not yet been handled get caught at this point. This general catch must always be the very last error handling block.

Not all errors always lead to exceptional situations, but only exceptions can be caught by catch. It is therefore particularly interesting to be able to define your own exceptional situations. The *throw* command triggers such an exception. It terminates the processing immediately and directly jumps to the appropriate exception handling.

Example: Generating your own exception using *throw*

```
#include <cstdlib>
#include <iostream>

using namespace std;

void DoSomething(int Problem)
{
   if(Problem>0)
   {
      throw 0;                    // Pass error to the catch block with "error code" 0
   }
}

int main()
{
   try
   {
      DoSomething(1);
   }
   catch (int a)
   {
      cerr <<  "Exception:" << a << endl; // Output: Exception: 0
   }

   system("pause");
   return 0;
}
```

If the catch block has a variable of the *int* type as parameter, it only handles exceptions thrown by a throw-command, with a number as argument. To edit other parameters, another catch block with a different parameter type is simply appended. To handle all the other exceptions, the general catch statement with the three dots as parameters can be added at the very end. The matching catch blocks are selected according to their parameters. However, a catch always has only one parameter. An automatic type conversion as with functions is not implemented here. For example, if you use the throw command with argument 1.2, the matching catch must have *double* as parameter and not *float*, because floating point constants are treated by the compiler as double by default. The following example shows several catch blocks for different data types.

Example: Various catch blocks

```
#include <iostream>
#include <cstdlib>

using namespace std;

// DoSomething throws different types, depending on the
// value of the Problem parameter.
void DoSomething(int Problem)
{
   switch (Problem)
   {
   case 0: throw 5; break;                    // throws int
   case 1: throw (string)"test.dat"; break;   // throws string
   case 2: throw 2.1; break;                  // throws double
   case 3: throw 'c'; break;                  // throws char
   }
}

// Test program
int main()
{
   // insert a number for the Problem variable
   int Selection;
   cout << "Insert a number between 0 and 3:" << endl;
   cin >> Selection;
   // The try block catches the exception in DoSomething
   try
   {
      DoSomething (Selection);
   }
   catch(int i) // Handler for int
   {
      cerr << "Integer " << i << endl;
   }
   catch(string s) // Handler for string
   {
      cerr << "String " << s << endl;
   }
   catch(double f) // Handler for double
   {
      cerr << "Float " << f << endl;
   }
   catch(...) // Handles all the other types of exceptions
   {
      cerr << "General Error" << endl;
   }

   system("pause");
   return 0;
}
```

When you start the program, you can use the numbers 0 to 3 to select which exception is triggered. For number 1, a character string is thrown which can then be further processed in the catch block as parameter s. To do this, the character string must be converted (cast). For number 3 a char is thrown for which no catch block is provided. So the general catch block catches this exception.

1.2.2 Error handling for file access

When editing a file, various errors can occur for which the program should be prepared. For example, you might want to access a file that does not exist, or you do not have write permission, or your hard disk is already full. For this reason, the *ifstream* class is prepared for such exception handling.

With the good() member function you can at any time query the stream object whether it detected errors during the last execution. The member function has no transfer parameters. However, it returns a value of the Boolean type.

Example:

```
#include <cstdlib>
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
   ifstream output_data;
   ofstream copy_data;
   output_data.open("source.txt", ios::in);
   copy_data.open("backupcopy.txt");

   if (output_data.good()){
      if (copy_data.good()) {
      char ch;
      while(output_data.get(ch)){                    // Read character by character from the file
         copy_data.put(ch);                          // Copy character by character into the new file
      }
      } else {
         cerr << "source.txt cannot be opened!" << endl;        // File does not exist
      }
   } else {
      cerr << "backupcopy.txt cannot be opened!" << endl;      // File is write-protected
   }

   output_data.close();
   copy_data.close();

   system("pause");
   return 0;
}
```

This kind of error handling is quite simple and elegant, because we can access the stream object directly and query its state. However, a large number of stream objects can quickly become confusing because of the excessive load of error handling.

An easier way to work around this problem would be to create a single try-catch block and to throw the error message with the file name by means of a throw statement. The file name is then passed as a parameter.

Example:

```
#include <cstdlib>
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
   ifstream output_data;
   ofstream copy_data;

   try
   {
         output_data.open("source.txt",ios::in);
         copy_data.open("backupcopy.txt");
         if(!output_data.good())
            throw (string)"source.txt";                    // If the file does not exist
         else if(!copy_data.good())
            throw (string)"backupcopy.txt";                // If the file is write-protected

         char ch;
         while(output_data.get(ch)){                       // Read character by character from the file
            copy_data.put(ch);                             // Copy character by character into the file
         }
   }
   catch(string filename)
   {
      cerr << "Error: " << filename << " cannot be opened." << endl;      // cerr is used like cout
   }                                                                      // Output as an error

   output_data.close();
   copy_data.close();

   system("pause");
   return 0;
}
```