Chapter 2 - Assignment #1
Bootstrap Verify
GPU Programming

February 21, 2018

1 Due

Tuesday, February 27th in the evening.

2 Overview

This is an assignment that helps you to make sure that the submission zip file contains all the files and formatting that are required. This assignment can be used to verify itself before submission, as well as all future assignments.

The intended use of the program looks like this:

$c:/>Verify.exe\ [submission-zipfile]\ [tokenfile]\ [namefile]$

To run the program on the command line, supply the command line arguments of the application with the path to the submission zip file, the path to a special check file (see below) and the path to a file containing your last name. The check file contains the specification of what should be in the zip file. An example of how to use the problem may look like:

e.g: Verify.exe C2_A1_BlackAdder.zip C2_A1_Check.txt Name.txt

So indeed, you may assume that all files are residing in the same folder as the executable. System-path handling is left out of this assignment. To build this program, we break it down in to manageable parts.

2.1 Check submission filename

First, the name of the zip file must be correct and observe the following rules:

1. The filename begins with the correct chapter numbering, indicated with a capital 'C'. The number is valid.
2. The filename continues with one underscore
3. The filename continues with the correct assignment / excercise numbering, indicated with a capital 'A' or 'E' respectively. The number is valid.
4. The filename continues with one underscore
5. The filename continues with your last name, without (white)space characters, with capitalization of each part of your last name, and without any accents or special tokens.
6. The filename ends with '.zip'.

The last name that is used in the name of the submission file zip archive must be identical to the one found in the Name.txt (which should hold your last name!), and must have no special characters in it.

To accomplish this task, we will write a number of string functions so that we can verify if the filename is correct.

2.2 Check submission content

Of course the contents of the file should also be correct. Because we can only expect some parts of the content, we check the files in the archive by extracting them and then matching each file against a file specification target s. Each target may match a file exactly n number of times. Both s and n will be supplied in a check file that describes the submission rules for the assignment. A target must be matched **atleast** n number of times.

To accomplish this task, we will use an existing piece of code that extracts the files from the zip archive first. Next we will use the string methods to try and match each of the filenames with any of the specification targets. For simplicity, we assume that specification targets do not overlap, such that each file can only match exactly one specification target.

2.3 Expected output

The program tells us if the zip file is valid for submission.

If there is a program, the program should indicate this, and report atleast the following problems:

1. The command line arguments are invalid.
2. The assignment zip filename is not correct.

3. The assignment zip filename is missing files.

4. The assignment zip contains empty files.

3 Instructions

Make sure your name is in comments at the top of each source file!

1. Write a function $StartsWith(in, start)$ in Verify.cpp that compares the start of a string with another string.

   (a) 2 arguments, both are null terminated strings, i.e. arguments are of type char in[256]

   (b) Returns true if it is found, false otherwise.

2. Write a function $EndsWith(in, end)$ in Verify.cpp that compares the end of a string with another string and returns true if it is found at the end.

   (a) 2 arguments, both are null terminated strings.

   (b) Returns true if it is found, false otherwise.

3. Write a function $Split(in, delim, index)$ in Verify.cpp that splits a string into multiple strings, based on a delimiter string. So for example, the string "Have a nice day." would be split into the string-list {"Have", "a", "nice", "day."} if the delimiter is a single space character (the delimiter could also be a multi-character string). The index argument selects the string that must be returned, so in this case Split("Have a nice day.', " ",2); would return the null-terminated string "nice".

   (a) 3 arguments: the null terminated input string, the delimeter string, and the index detailing which substring should be returned.

   (b) Returns (one of) the substring(s).

4. Write a function $Compare()$ in Verify.cpp that compares 2 strings and returns true if there is a strict match. So for example, the strings "nice" and "nIcE" would not be considered equal.

5. Write a function $CompareNoCase()$ in Verify.cpp that compares 2 strings and returns true if there is a match with uppercase/lowercase differences not being considered, so in the previous example, the strings would be considered equal.

6. Write a function $ReadName()$ in Name.cpp that accepts the file name of the "Name.txt" file (so that it could also have another file name). The function opens the file, reads the name, closes the file and returns the name as a null-terminated string. If no name is found, the function returns an empty string.

7. Write a function $ReadNumberOfTarget()$ in Check.cpp that accepts the file name of the "Check.txt" file and that returns the number of target specifications to check on. The function opens the file, reads the information (see next section), and closes the file. The number is returned as an integer. If the file does not exist, the returned number is 0.

8. Write a function $ReadTarget()$ in Check.cpp that accepts the file name of the "Check_C2_A1.txt" file and an index of the target that must be returned. The function opens the file, reads the information (see next section) until it finds the index'th target specification. It closes the file and returns the target specification as a null-terminated string. If the file does not exist or the index is not within the bounds of the number of targets in the file, the returned string is empty.

9. Write a function $MatchFileToTarget()$ in Check.cpp that accepts a filename and a target specification string. If the filename matches the target specification it returns true. More on target specifications follows below.

10. Write a function $ExtractZip()$ that takes the zip file as argument and ' 'extracts" all the file information in the zip file. In fact, extraction itself is not necessary, only information about the files in the archive (full path and file size) are necessary. Take a look at the ZipInfo.cpp file. At the end of the file you will find 3 c-style functions. Write a ZipInfo.h so that you can use these functions to obtain the number of files in the zip, the name of each file, and each file's size one by one, as demonstrated in the commented section of the ZipInfo.cpp file.

11. Write a function $VerifySubmission()$ in BootstrapVerify.cpp next to the main function. The VerifySubmission() function combines the functions to validate the name of the zip file and each of the files that are included.

Note 1: For those using char arrays for strings, you can assume that the maximum number of characters in a string never exceeds 256 characters. You can also use std::string instead.

Note 2: For those using char arrays to store filenames, you can assume that the maximum number of files in an archive should never exceed 256 number of files.

4 Target Specifications

The file is structured as follows:

1. A header section, that checks if the submission filename is correctly formatted.
   (a) "HEADER", a marker string followed by end-of-line character.
   (b) x = Integer number, followed by end-of-line character =¿ total number of filename targetfile specifications in this file, including the zip file target specification.
   (c) one "Submission zip" target specification string
2. A mandatory-files section, that can check if all necessary files are included
   (a) "REQUIRED", a marker string followed by end-of-line character.
   (b) y= Integer number, followed by end-of-line character =¿ number of filename targetfile specifications that are required.
   (c) y target specification strings
   (d) ...
3. A forbidden-files section, that can filter out file names that should not be uploaded
   (a) "FORBIDDEN", a marker string followed by end-of-line character.
   (b) z = Integer number, followed by end-of-line character =¿ number of filename targetfile specifications that are forbidden.
   (c) z target specification strings

Note1 : x = 1 + y + z!

Note2 : the file must contain 6 + x number of lines in total.

Each target specification string is formatted as follows:

Format: %[xyz]%[ext]%[delim]%[pattern1]%[pattern2]%[pattern3]%...%[EOL]

xyz - 3 digits that tell us how many entries may match the pattern at most (this field is always '001' for submission zip file, ignored for forbidden patterns)

ext - an extension string that ends before the next % character.

delim - a delimiter string that ends before the next % character.

pattern1 - After splitting, the first substring of the filename should match the pattern string. The pattern string ends before the next % character.

pattern2 - After splitting, the second substring of the filename should match the pattern string. The pattern string ends before the next % character.

pattern3 - After splitting, the third substring of the filename should match the pattern string. The pattern string ends before the next % character. See below for more on pattern strings.

etc..

EOL - the format specification can be read from the file as one string and ends with an eol (end-of-line) symbol. This symbol is \r\n (carriage-return + linefeed) on DOS/Windows and \n (linefeed) on unix/linux. (Note: In the example below, the line endings are not printed. The supplied file with this exercise has DOS-style line endings)

Pattern strings are alphanumeric character strings that may contain wildcards, such as ? and *. The ? symbol indicates a single character placeholder, while '*' implies multiple characters. Both wildcards may occur in the same pattern. The only characters that the matched substrings can never contain are the wildcard characters themselves, the '%' character (used to indicate the different fields in the pattern) and the '|' character (to indicate valid alternatives) .

Example 1 : we expect 1 zipfile filename to match correctly with the assignment numbering and a wildcard string (= your last name):

001%.zip%_%C2%A1%*

Example 2 : we expect 5 cpp files that strictly match with the following names :

005%cpp%%BootstrapVerify|Verify|Name|Check|ZipInfo

To handle wildcards in your pattern matching logic you can use the following approach: To handle wildcards of '*' type, split the pattern into sub-patterns based on the wildcard symbol. Check if each sub-pattern occurs atleast once, in the same order and without overlapping in the filename string. To handle wildcards of '?' type, first detect and locate all occurrences of '?' in the (sub-)pattern to match. In the string that is matched, replace characters at the same locations (if possible) so that the (sub-)strings follows the same wildcard (sub-)pattern. Now you can verify if the rest of the (sub-)pattern matches the other parts of the (sub-)string using a regular string compare.

Note: you can assume that extensions do not contain wildcards or alternative marker characters.

A full example C2_A1_Check.txt file is given below.

5 Submission Guidelines

HEADER

22

001%.zip%_%C2%A1%*

REQUIRED

5

001%.txt%%CMakeLists

001%.sln%%BootstrapVerify

001%.vcxproj%%BootstrapVerify

005%.cpp%%BootstrapVerify|Verify|Name|Check|ZipInfo

004%.h%%Verify|Name|Check|ZipInfo

FORBIDDEN

16

000%.obj%%*

000%.pdb%%*

000%.ilk%%*

000%.exp%%*

000%.pch%%*

000%.o%%*

000%.so%%*

000%.lib%%*

000%.dll%%*

000%.sdf%%*

000%.suo%%*

000%.log%%*

000%.tlog%%*

000%.lastbuildstate%%*

000%.stamp%%*

000%.depend%%*

Submit your ZIP file according to the **Submission Guidelines on Minerva Announcements (Aankondigingen)** as many times as you want to Minerva (Dropbox) - we will only consider the last one submitted.

Remember: the supplied file with this exercise has DOS-style line endings symbols to terminate each 'line' (i.e. \r\n)