Assignments P2.7 - Part 1

General remarks

All assignments are to be programmed in *portable* Fortran 2003/2008 except where explicitly noted. If an assignment requires a modification of a main program from a previous section, a copy under a new name should be made. Prefix main program sources by the section number (e.g. *01_readint.f90*, *02_readfloat.f90*). Create a new branch for each part of the assignments (*part1*, *part2*, *part3*, and *part4*) where each new branch starts with the code required to be completed from the previous part, since later assignments will depend on code from previous ones, but may also require incompatible changes to code written for previous parts.

00 CMake Build system

Study the talk slides with the CMake introduction and compare with the files in the “01-cmake-demo” folder. More introductory information about writing “modern” CMake build systems can be found at https://cliutils.gitlab.io/modern-cmake/ and at https://hsf-training.github.io/hsf-training-cmake-webpage/01-intro/index.html and – of course – there also is https://cmake.org/documentation

Use the folder “programming” in the top level of the repository for writing the Fortran codes for this class. This will be an incrementally growing project and the folder already includes a dummy Fortran source, a *CmakeLists.txt* file example for building Fortran programs and running a test with I/O redirection and checking of the output of the program. Expand this build system as needed. Remember that for Fortran modules the order of compilation matters because of using modules. Supporting modules are best compiled into a library.

01 Read in integer arrays

Write a program that can read in a list of integer numbers into an array from a provided file, reading from standard input (channel 5) via i/o redirection. The files have filenames of the pattern “d1_#.dat” and the following format: the first line has a single integer number signaling the length of the array; then the array elements spread over multiple lines, and finally a single number on the final line containing the sum of all elements, which should be used to check whether the reading was done correctly. The program should read the first line, then allocate sufficient storage and read in the array data and finally read and store the “checksum”. It shall output the length of the array and whether the checksum matches or not. If the checksum does not match, both, the expected and the computed values should be printed as well.

02 Read in real arrays

Write a program, similar to the one from section 01, which handles real data (i.e. 32-bit floating point numbers) instead of integer. The corresponding files have names following the pattern “d2_#.dat”. Real numbers may be provided in fixed or exponential format. Please note that floating point numbers may be truncated or rounded on output, so do not test for identify when comparing with the checksum. Rather use a relative accuracy of 10^{-5}

03 Check data for being sorted

Write a program, similar to the one from section 01, where you add a function “is_sorted()” in the same source file to test if the array elements are sorted in ascending order. This function shall take the integer array as argument and return a logical as result. After reading the data, call the function and output a line of text indicating whether the array is sorted or not.

04 Check real data using a module

Now write a program similar to the one from section 03, which handles real numbers instead of integers and also the “is_sorted()” function is provided in a separate file as part of a “list_tools” module.

05 Optional function arguments

Expand the “list_tools” module so that it contains two constants (parameters), “ascending” and “descending”, both are of type logical and have the value .true. and .false., respectively. Now add to the “is_sorted()” function an optional argument so that the direction of the order can be indicated. Do this in such a fashion that the program from 04 remains functional without modification, yet uses the same module. The main program shall differ from the version in section 04 only by explicitly requesting the sort order to check for (ascending).

06 Function overloading

Now rename the “is_sorted()” function in “list_module” to “is_sorted_real()” and implement a version for integers called “is_sorted_int()” and define an interface for “is_sorted()” so that either of the two functions is being called, depending on the datatype of the arguments. The programs from sections 04 and 05 have to remain functional without change. In addition write a variant of the main program from section 03 that now imports “is_sorted()” from the “list_tools” module and checks integer arrays for being sorted.

07 Derived types

Write a program similar to the programs of sections 01 and 02 that can read in pairs of numbers as key-value pairs, where the key is integer and the value a real number. The corresponding data files have names “d3_#.dat” and the format is similar to previous cases: one line with the number of items (i.e. pairs), then the integer-real pairs, and then a single line with the checksum, that is is the sum of all real values. To implement this create a derived type “pair” with two entries: an integer “key” and a real “val” and store the data from the files in an array of this derived type.

08 Multiple modules

Now put the definition of the derived type “pair” info a new module “list_types”. Import this module into “list_tools” and write a variant of “is_sorted()” for the pair type. This function shall have a second optional argument, a logical indicating whether the check should be performed on the key or the value of the pair. For that two new constants, “bykey” and “byvalue” shall be added to the module as well. Finally write a main program, similar to that from section 06 that will read the pair data files and perform the sorted check. Test data files if they are ordered by ascending value and descending keys.