qContainers Fortran library: a tutorial

This is an introduction tutorial for qContainers Fortran library. It will show you how to start using qContainers on Windows using Gfortran compiler. However, the users of other OSes and/or compilers can find the tutorial useful too.

Compilation of the library

1) Install MinGW with C and Fortran compilers. You can download it, for example, from “http://tdm-gcc.tdragon.net” or “www.equation.com”.

2) Install CMake from “cmake.org”.

3) Download or clone qContainers from “github.com/darmar_lt/qcontainers” (Fig. 1). Extract downloaded zip file to the folder of your chose. I will call a folder <qContainers_folder> below which you find other folders like “containers”, “internal”, etc. and files “CMakeLists.txt”, “Install.md”, etc.

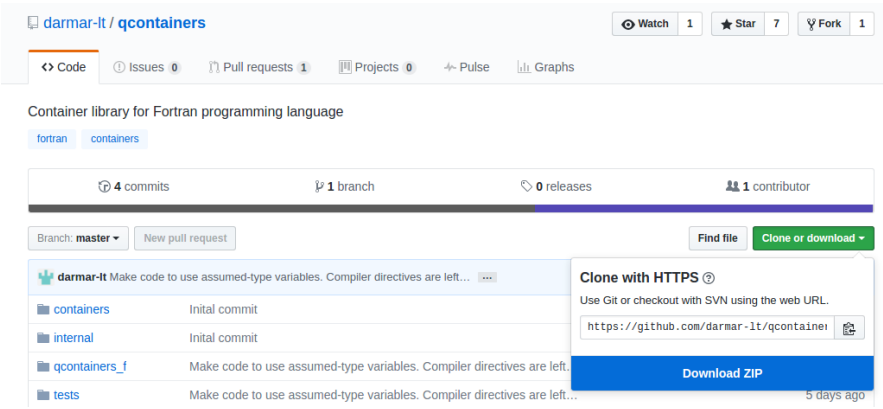

Fig. 1. qContainers on GitHub

4) Now, C files of the library should be compiled. Open CMake GUI, locate <qContainers_folder> in the first text field and some new folder in the second text field where the binaries will be compiled (see Fig. 2). Press “Configure”. In the opened dialog you are asked to specify the generator for this project. I used “MinGW Makefiles” in this tutorial. If you want to use qContainers with Intel Fortran, you should specify “Visual Studio XXX”. Select CMAKE_BUILD_TYPE (usually, “Release”) and once more press “Configure” (Fig. 3). Press “Generate”. Required files are created in your “build” folder. Close CMake window.

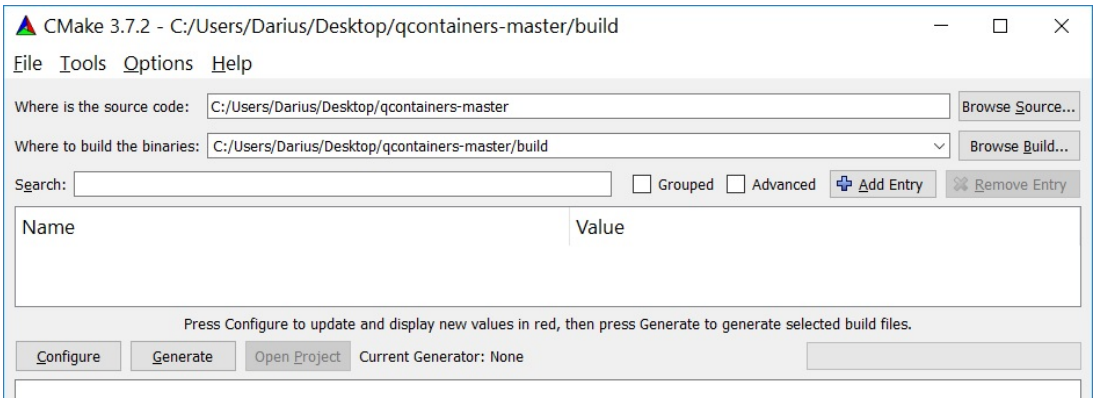

Fig. 2. CMake window

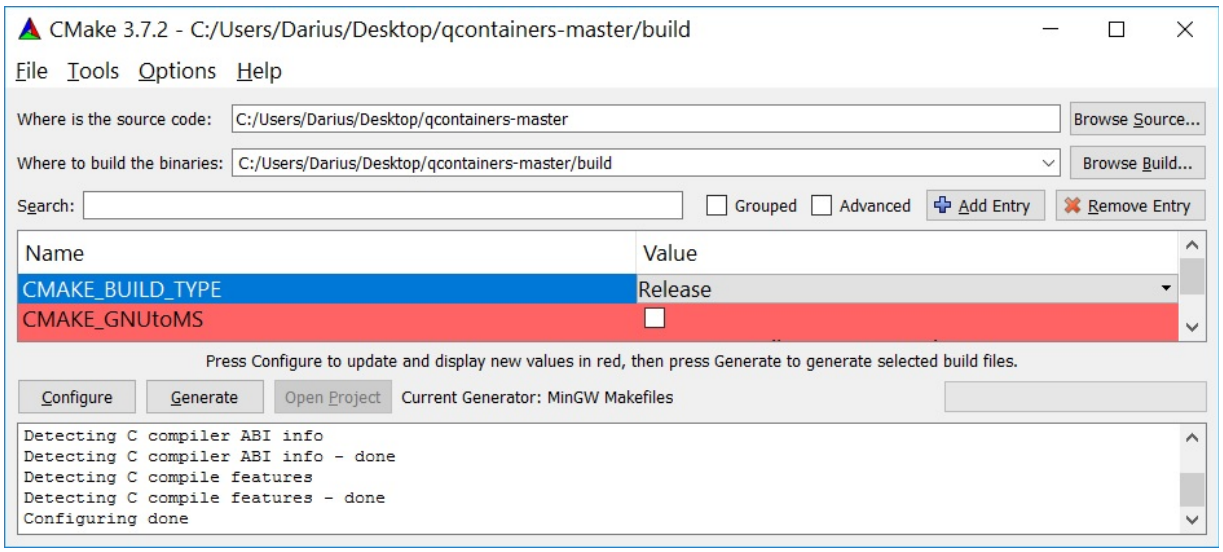

Fig. 3. CMake window after “Configure”

Go to the folder with the generated Makefile. Open “cmd”. Execute “make.exe” or, as it is in my case, “mingw32-make.exe” (it is the program installed with MinGW). In case of success, the library file “libqcontainers.a” is created in the folder “<qContainers_folder>\build\lib”.

Testing of the library

qContainers library comes with a test program. Beside the testing of the library, the test program can be used for learning how to use the library. The Code::Blocks IDE can be used for the compilation of the tests. The test files together with a Code::Blocks project file “qcontainers.cbp” are in “<qContainers_folder>\tests” folder.

Open Code::Blocks (if you don’t have it yet, download it from “cbfortran.sourceforge.net”). Open the “qcontainers.cbp” project file. The project file contains three targets: “Tests” is for the compilation with Gfortran (our case), “Tests_PGI” is for the compilation with PGI Fortran on Linux, “Tests_Oracle” is for the compilation with Oracle Fortran compiler on Linux. If not already selected, select “Tests” target and compile the project. You will get a lot of warnings complaining about Final procedures, however, it seems, it is a bug in the current Gfortran (please write me, if I am wrong). Run the compiled project.

A new project with qtreetbl container

“qtreetbl” container can be use to store unique key-value pairs. Usually, character strings are used for keys. However, “qtreetbl” allows to use other types for keys too. Another container for storing key-value pairs is “qhashtbl”. However, it allows to use only character string for keys.

I am using Code::Blocks IDE for this tutorial. However, you are free to use other tool if you prefer.

Create a new project. Select Gfortran as a compiler. Copy the “<qContainers_folder>\qcontainers_f” folder to the folder with the created project file. Add files from “qcontainers_f” to your project. Note, the files *.fi are include files and they should NOT be compiled. Usually, the file names of such files in C::B are grayed out (see Fig. 4). Actually, you can remove *.fi files from the C::B project. The compiler will find these files during the compilation anyway.

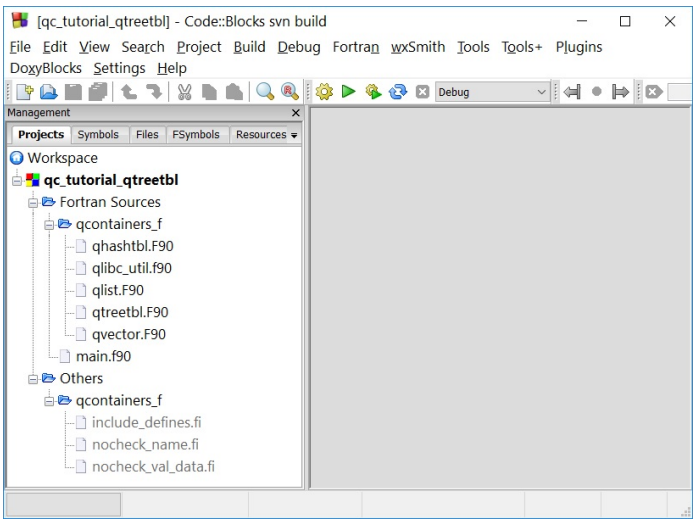

Fig. 4. Project files for use of qContainers

During the link step, Gfortran should find the compiled “libqcontainers.a” library. Therefore, point to this library at “Project build options” dialog (“Project→Build options”) (see Fig. 5).

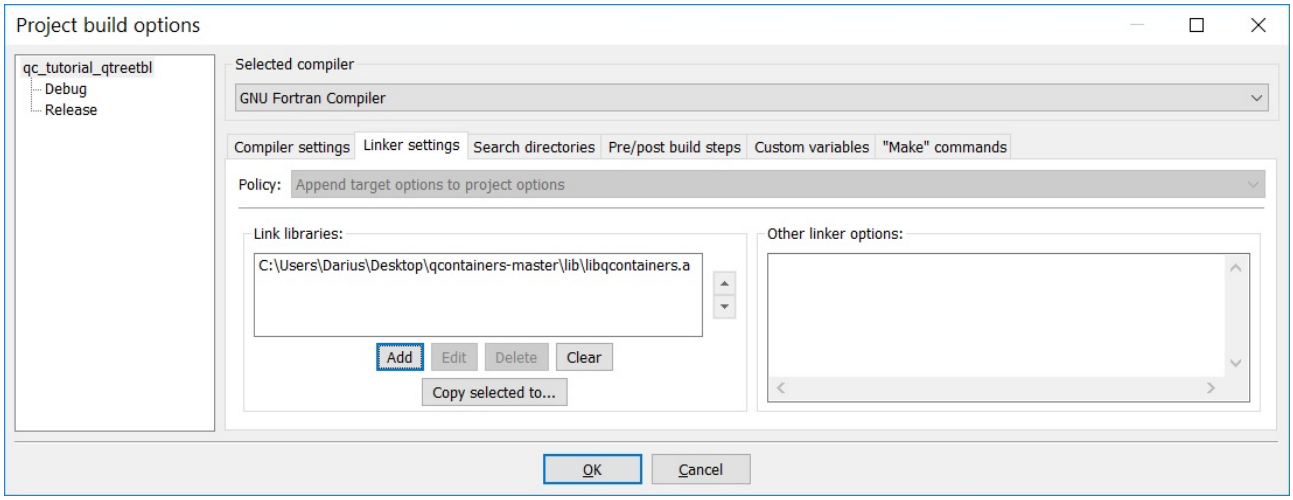

Fig. 5. Project build options dialog: Linker settings

The files with extensions *.F90 should be preprocessed before a real compilation take place. Gfortran should preprocess *.F90 files by default. However, if you are not sure, you can add the “-cpp”, compiler option (Fig. 6).

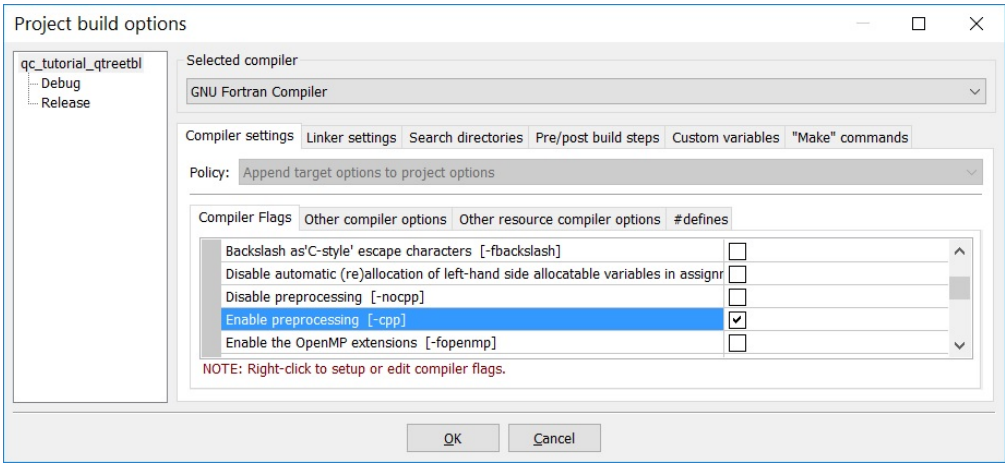

Fig. 6. Project build options dialog: Enabling preprocessing

The code:

```
program test_qtreetbl
    use qtreetbl_m
    implicit none

    type price_t
        integer :: eur
        integer :: cnt
    end type

    type(qtreetbl_t)     :: qt
    type(price_t)        :: pval, pback
    type(qtreetbl_obj_t) :: tobj
    integer :: siz
    logical :: found
    character(len=:), allocatable :: key

    ! Storage size in bytes
    siz = storage_size(pval) / 8

    ! Initialize container
    call qt%new(siz)

    ! Put some values
    pval%eur = 1
    pval%cnt = 25
    call qt%put("Apples", pval)

    pval%eur = 0
    pval%cnt = 85
    call qt%put("Milk", pval)

    pval%eur = 1
    pval%cnt = 19
    call qt%put("Bananas", pval)

    !-----------------
    ! Get values back

    !Iterate over all keys
    call tobj%init() ! should be initialized before the use
    do while(qt%getnext(tobj))
        call tobj%getname(key)
        call tobj%getdata(pback)

        print *, key, ": ", pback%eur, " Eur ", pback%cnt, " cent"
    end do

    call qt%get("Oranges", pback, found)
    if (found) then
        print *, "Oranges: ", pback%eur, " Eur ", pback%cnt, " cent"
    else
        print *, "No oranges today :("
    end if

    !------------
    ! Update value
    pval%eur = 1
    pval%cnt = 9
    call qt%put("Milk", pval)

    ! Get back
    call qt%get("Milk", pback, found)
    print *, "New price for milk: ", pback%eur, " Eur ", pback%cnt, " cent"

end program
```

Copy the text into the main.f90 file. Compile the project. Execute. Output should be like in the Fig. 7.

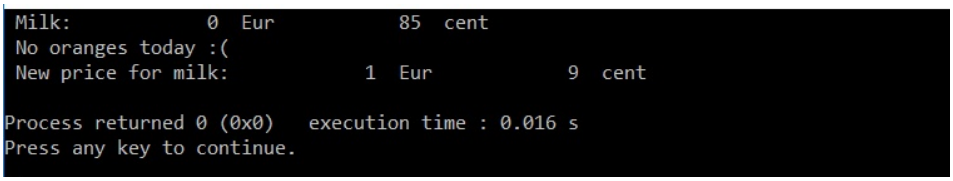

Fig. 7. Output of the program

Some more comments about the code.

Include “use qtreetbl_m” statement if you want to use qtreetbl container. Similarly, include “use qlist_m”, “use qhashtbl_m” or “use qvector_m” if you are going to use another container. In the current code, variables of the derived type “price_t” are stored in the dictionary (key-value pairs). The container should be initialized by call to the subroutine ‘new(size_data)’ before the first use. The argument ‘size_data’ is optional. It tells for the container, how much of the memory in bytes occupy each value. Alternatively, you can set the size for each value you put.

Storing derived type variables with allocatable or pointer components

In the previous example, the derived type variables were stored into the container. However, you should be careful storing derived type variables in the qContainers. If the derived type contains allocatable or pointer components, the pointer to the allocated memory will be stored only. Allocated memory will not be copied to the container. Therefore, the pointers of the derived type and type(c_ptr) variables should be used instead. Below is an example of code where qhashtbl container is used.

```
program test_qhashtbl
    use qhashtbl_m
    use iso_c_binding, only: c_ptr, c_loc, c_f_pointer
    implicit none

    type value_t
        integer :: nv
        integer, allocatable :: val(:)
    end type

    type(qhashtbl_t)     :: qh
    type(qhashtbl_obj_t) :: hobj
    type(value_t), pointer :: pval, pback
    integer :: siz, i, j
    logical :: found
    character(len=6) :: names = "abcdef"
    type(c_ptr) :: cp

    ! Storage size in bytes
    siz = storage_size(cp) / 8

    ! Initialize the container
    ! 'range' value defines the size of table is used internally.
  ! It is recommended to use a value between (total_number_of_keys / 3) ~ (total_number_of_keys * 2)
    call qh%new(range=10, size_data=siz)

    ! Put some values
    do i = 1, 6
        allocate(pval)
        allocate(pval%val(i))
        pval%nv = i
        pval%val(:) = [(j,j=1,i)]
        cp = c_loc(pval)

        call qh%put(names(i:i), cp)
    end do

    ! Take some values back
    call qh%get(names(2:2), cp, found)
    if (found) then
        call c_f_pointer(cp, pback)
```

```
        print *, "'", names(2:2), "' has values nv=", pback%nv, " val=", pback%val
    end if

    call qh%get(names(5:5), cp, found)
    if (found) then
        call c_f_pointer(cp, pback)
        print *, "'", names(5:5), "' has values nv=", pback%nv, " val=", pback%val
    end if

    ! We allocated memory for pointers.
    ! Now we have to free this memory.
    call hobj%init()
    do while(qh%getnext(hobj))
        call hobj%getdata(cp)
        call c_f_pointer(cp, pback)
        print *, "Value will be deallocated: nv=", pback%nv, " val=", pback%val
        deallocate(pback)
    end do

end program
```

Perhaps, with the code I told everything I wanted to tell.

Storing arrays

The qContainers use assumed-type dummy arguments to allow store variables of any type. This approach has a limitation, that it does not allow to use an array as an actual argument to the procedures. However, it is possible to “send” first element of a range we want to store and use size of that range. An example below should explain it more clearly.

```
program test_qlist
    use qlist_m
    implicit none

    type(qlist_t)     :: ql
    integer :: arr(2,10)
    integer :: bvec(2)
    integer :: siz, i

    ! Fill values into array
    arr = reshape([(i,i=1,size(arr))], shape(arr))

    ! Storage size for TWO array elements in bytes
    siz = storage_size(arr) / 8 * 2

    ! Initialize the list
    call ql%new(siz)

    ! Add every second column of arr into the list
    do i = 1, 10, 2
        call ql%addlast(arr(1,i))
    end do

    ! Look into the list, if we have there, what we are expecting.
    do i = 1, ql%size()
        call ql%getat(i, bvec(1))
        print *, "In i=", i, " element list has ", bvec
    end do

end program
```

I hope, this tutorial was helpful for you.