Generic Programming in FORTRAN Language: Realization of High-Performance Generic Container by Using Preprocessor

Ruihua Zhu[a], Lina Ning

Department of Physics, Mudanjiang Normal University, Mudanjiang 157012, China

[a]ruihuazhu@126.com

Keywords: Generic programming, generic container, FORTRAN, preprocessor

Abstract: The Performance of several containers, including the generic type containers simulated by object-oriented programming in FORTRAN language and the ordinary container also written in FORTRAN language, are investigated. The container of highest performance and the lowest performance is the ordinary container and the generic container by object-oriented programming, respectively. In order to achieve a high-performance generic container, a preprocessor is written used to translate the generic programming codes to the ordinary codes. With preprocessing, the abstract programming and high-performance codes are implemented at the same time.

1. Introduction

Generic programming is dedicated to writing highly reusable code, which greatly reduces the amount of program code and improves code maintainability [1-5]. Generic containers, as achievements of generic programming, have been implemented in a variety of computer languages and are widely used in the development of various software. A famous generic container is due to the Standard Template Library (STL) as a part of the c plus plus language [6].

However generic programming is not directly supported by the FORTRAN language, according to the FORTRAN 2008 standard. But the generic containers in FORTRAN language are still intriguing topics, for there are a lot of libraries to simulate generic containers [7-10]. The difficulty to archive a type safe generic container is memory operations such as the location of memory and data copy. These matters are usually treated by three ways, such as using mixing programming, using the object-oriented types and virtual copy functions. Both the memory and data operations govern the performance of generic containers. Thus, the performance of the generic containers is quite different, which will be discussed in the after section.

2. Performance of the different kind of containers

Vectors are sequence containers representing arrays that can change in size, which are well realized by different libraries and extensively used. Four kinds of vector are selected for comparison. The First container is from the open source project qcontainer in GitHub and the main contributor to the project is darmar-lt [7]. This project is written by mixing programming with c and FORTRAN. The second container is written by using “type (*)” and “select type” which are object-oriented grammar added by FORTRAN 2003 standard. The third container is just a common container written in FORTRAN language. At last, the c plus plus STL vector container are added to this comparison.

In order to show the time performance of the four containers, a comparison is performed based on the “push_back () function. This function adds a new element at the end of the vector, after its current last element. If and only if the new size surpasses the current vector capacity, the capacity will be increased by one. Programs are written to push serial integers to each container, and finally show the time during the push-operation in running the program. The first container library is compiled using its “makefile”. The test programs of the first, second and third programs are compiled by gfortran compiler with “-O3” optimization indication. The test program using c plus

plus vector container are compiled by g++ compiler with "-Ofast" optimization indication. These programs are running in a computer with I7 4790 CPU under Ubuntu operating system.

The run time in seconds of each containers are shown in table 1. It's clear that the time used of these operations are increased almost linearly with the increasing of cycle times, since the time used per cycles are nearly constant. It's clear that ordinary containers used least time and object-oriented container used most time. The program with qcontainer use more time than the program with c plus plus STD. The "select type" sentence used in the object-oriented containers which is necessary for the data copy operations, cost additional time. It's the reason for the weakness performance of the object-oriented containers. The qcontainer library is worked by passing data through an interface to c library, this data passing progress will also cost some time. Although the time performance is based on the "push_back" function, we believe similar results will be presented on the test of other functions such as "insert" or "remove" functions.

It seems that abstract and performance are trade-off relations in code programming and it's difficult to have both abstract and high-performance codes. In some time, computer runs heavy load calculations, the weak performance of the object-oriented container will be impressive. An object-oriented container is used depends on whether we can tolerate its poor time performance.

However, the poor time performance is not due original to the generic programming but due to the simulations of generic programming. The generic container also can be established as its original process of generic programming. The progress is down as follow. First, it's necessary to write a template with generic types. Second, a code generator as a preprocessor are used which automatically translate generic codes to common codes, according to the template. The programs compiled after the translation will performs as common codes, without restriction of the trade-off relations.

After sections of this paper are mainly about how to write the template and the preprocessor.

Table. 1 The time performance of the different containers

loop times	qcontainer	object-oriented container	ordinary container	c plus plus std
1000000	0.0149	0.0159	0.0045	0.0077
10000000	0.1269	0.1715	0.0523	0.0854
100000000	1.1849	1.5093	0.4393	0.6952

3. The container templates in Fortran language

A vector template is written in FORTRAN language, which are composed of two parts. The one is the declaration of the container type and the other is realization of member functions. The shape of declaration of the container type are shown in Figure 1, and one detailed member functions "push_back_fun" are shown in figure 2.

```
type, public:: <<container_name>>
    private
    integer :: capacity = 0
    integer :: used     = 0
    …
    <<type_name>>, dimension(:), pointer :: ptr
    contains
    procedure, public:: push_back => <<container_name>>_push_back_sub
    …
end type
```

Figure. 1 the declaration of the type declarations

```
subroutine <<container_name>>_push_back_sub(this, obj)
    implicit none
    class(<<container_name>>), intent(inout) :: this
    <<type_name>>,             intent(in)    :: obj
    <<type_name>>, dimension(:), pointer     :: midptr
    if( this%capacity < this%used+1 ) then
        if(this%capacity.eq.0) then
            allocate(this%ptr(1))
            this%capacity = 1
            this%ptr(1)   = obj
        else
            allocate(midptr(this%capacity*2))
            midptr(1:this%capacity) = this%ptr
            this%ptr(this%used+1)   =  obj
            deallocate(this%ptr)
            this%ptr => midptr
           this%capacity = this%capacity * 2
        end if
    end if
    this%used = this%used+1
end subroutine
```

Figure. 2 the realization of the member function

The notation of “<<container_name>>” and “<<type_name>>” marked by “<<” and “>>”

Appears in the template both the declaration partition and the realization partition. They are generic names which should be replaced by detailed container name and detailed type name respectively, while the template is used. As normal type declarations, the container name can’t be repeated in one function, in one module or in the main program. The container name precedes the name of the target member function in order to prevent duplication with the name of the other container's member function. This method is general which can be applied to containers of other type, such as list, map and so on. The member variable of “capacity” and the member variable of “used” is for saving the used capacity and the used capacity of the vector, respectively.

The subroutine “<<container_name>>_push_back_sub” is worked for save new data at the end of the vector. If the vector is full, the subroutine begins with the memory operations. In this condition,

The “capacity” is equivalent to “used”, and vector is of no capacity for the new income data. It’s necessary to allocate new memories for both the new data and previous data. If the “capacity” of the vector is zero, it is necessary to locate “ptr” with size equivalent to one. If the capacity is greater than zero, the new memory by “midptr” will be located whose size is one times larger than the previous. After that, copy the “ptr” to “midptr” release “ptr” and set “ptr” to the pointer “midptr”. After the location and the copy operations if necessary, then appending the new data to the end of the used data.

There are some notations about the previous location function. Since the “allocate” function has the “source” parameter. An idea is by combine the location operation and the copy operation into the allocate operations like in figure 3. However, it’s unsure that we can allocate memory larger than memory of source. The location functions in figure 3 will pass the compiling progress, but the running result is different due to different compilers. A wrong result will be presented, during running the program compiled by the Intel FORTRAN 2019 compiler and a correct result will be shown during running the program compiled by GCC compiler. However, combining the location and copy operations show nearly no better time performance than doing the location operation and the copy operation independently as shown in Figure 4.

It’s also need to pay attention to the copy of data. If the type is an intrinsic type of FORTRAN, such as integer, double precision, and so on, no extra work is required. However, if it’s a user-defined type, it’s necessary to overload the “=” operator which ensure a deep copy.

```
integer, dimension(:), allocatable :: a
integer, dimension(20)             :: b
allocate(a(30), source = b)
```

Figure. 3 combining the location and the copy operatons.

```
integer, dimension(:), allocatable :: a
integer, dimension(20)             :: b
allocate(a(30))
a(1:20) = b
```

Figure. 4 doing location and copy separately

4. Preproccessor with the template

The preprocessor is code generators which translate the generic codes to common codes and place it to the proper position of the input documents. An example of code generator for FORTRAN language is given by [12].

The input to the preprocessor is a document file and the output of the preprocessor is a FORTRAN source code file. The preprocessor works begin with scanning the input files for the derive sentence like this "! PCF vector<integer> vint". The "! PCF" is a declaration which is short for "Preprocess containers in Fortran". The "vector" is declared the container types. The "integer" between the "<" and ">" are the type name and "vint" is the container name. These derive sentences works as declaring a new type whose name is "vint", in the after section, the "vint" type variable can be declared by the sentence "type (vint): v1".

It's easy for preprocessor to locate the derive sentence. However, there are something should be treated carefully in the type declaration. If the type name is an intrinsic type of FORTRAN, this type variable is usually declared by the sentence "integer: a". However, if a type is a derived type, this type variable must be declared like this sentence "type (some_type): a". The "type ( )" must be presented. Hence, a direct method is by determining the type between the "<" and ">" whether it's an intrinsic type or a derived type. However, the sentence "type (integer): a" is also accepted by most compilers. All the types can be treated as derived types if you like.

The generated code of the container type declaration, should directly replace the derive sentence and placed in the output document, but codes of member functions should be placed in the "contains" partition of program subroutines, functions or modules. Sometimes, functions and functions are nested together, the member functions can't be placed directly. An integer variable "layer" is used. The layer is initialized by zero, and vary during scanning the input document. If scanner reads "start flags" such as "subroutine", "function" or "module" that "module" can't be "module, procedure", layer will be increased by one. If the scanner reads end flags such as "end subroutine", "end function" or "end module", layer will be decreased by one. The codes of member functions are generated with the declaration of the container types, but it is pushed into a stack together with the layer number. While scanner read end flags, it should compare the layer in the top of the stack. If the layer is equivalent to the layer in the top of stack then insert the codes before the end flag and pop the stack. The previous operation will be done until the layer is different with the layer in the top of the stack or the stack is empty.

According to previous method a code generator as a preprocessor are written by c plus plus language and it's compiled by g++ compiler. The test codes are written with the derive sentence according to the preprocessor. The target codes generated by the preprocessor are well compiled by the gfortran compiler, which has the same time performance with the hand-writing containers.

5. Conclusion

A generic container in FORTRAN language can be achieved by using preprocessor which is worked as a code generator. The quality of the codes generated by the preprocessor is the same as hand-writing codes for the same performance of programs compiled from the two kinds of codes. The method to writing the preprocessor is discussed in detail. These methods are generic, not only for writing vector type containers, but also for other type containers, such as map and list as well as generic functions.

Acknowledgments

This work was financially supported by the recording project of educational department of Heilongjiang province though grant number 1352MSYYB011 and it's also supported by youth project of by Mudanjiang Normal University with grant number YB2017004.

References

[1] D.R. Musser, A.A. Stepanov, Generic programming, Lecture notes in computer science 358 (1989) 13-25.

[2] R. Backhouse, P. Jansson, J. Jeuring, L. Meertens, Generic programming - An introduction, Lecture notes in computer science 1608 (1999) 28-115.

[3] J.C. Dehnert, A. Stepanov, Fundamentals of generic programming, Lecture notes in computer science 1766 (2000) 1-11.

[4] J.G. Siek, A. Lumsdaine, A language for generic programming in the large, Science of computer programming 76 (2011) 423–465.

[5] D. Gregor, J. Jarvi, M. Kulkarni, A. Lumsdaine, D. Musser, S. Schupp, Generic programming and high-performance libraries, International journal of parallel programming 33 (2005) 145-164.

[6] C. Grelck, H. Wiesinger, Persistent asynchronous adaptive apecialization for generic array programming, International journal of parallel programming 47 (2019) 164.

[7] A. Davidson, Generic containers in c++, Dr Dobbs Journal, 16 (1991) 50.

[8] Qcontainer on https://github.com/darmar-lt/qcontainers

[9] FIAT on https://github.com/Fortran-FOSS-Programmers/FIAT

[10] Fortran-container on https://github.com/dongli/fortran-container

[11] Fortran-collections on https://github.com/bast/fortran-collections

[12] M.V. Waveren C. Addison, P. Harrison, D. Orange, N. Brown, H. Iwashita, Code generator for the HPF library and Fortran 95 transformational functions, Concurrency Computat.: Pract. Exper. 14 (2002) 589–602.