— November 15, 2024 —
Introduction to Fortran Templates
Binary Search Example

Magne Haveraaen[1], Thomas Clune[2], Brad Richardson[3], Aury Shafran[4], Ondřej Čertík[5], and Damian Rouson[6]

[1] Bergen Language Design Laboratory
Department of Computer Science, University of Bergen, Norway
`https://bldl.ii.uib.no/`
[2] NASA GSFC
`thomas.l.clune@nasa.gov`
[3] NERSC
`brad.richardson@lbl.gov`
[4] Intel
`aury.shafran@intel.com`
[5] LFortran
`ondrej@certik.us`
[6] Lawrence Berkeley National Laboratories, University of California
`rouson@lbl.gov`

Abstract. This a document in a series of pamphlets introducing Fortran 2028 templates. Each pamphlet is centered around an example, showing how to templetise it. The pamphlets also discusses how to harden such code and prepare it for inclusion in a public template library.
This pamphlet uses *binary search* as its motivating example.

1 Introduction

Binary search for an element in an ordered array is among every software developer's reportoire. It is a reusable algorithm, and without templates it needs to be hand specialised for every use case. With templates we can make one standalone implementation of binary search, and reuse that in all settings.

However, there are some technical pitfalls we should be aware of when implementing binary search for others to use. We pinpoint these, ending up with a solid and reusable implementation.

1.1 Background: Fortran Templates

For decades the ISO and INCITS Fortran committees, informally known as WG5 and J3, respectively, have looked at proposals for templates or generics.[7] The work towards a generics mechanism got traction at the Fortran meeting July 2019 in Tokyo with the presentation of the paper [2].

Dan Nagle (1953-2023) from NCAR, head of J3 at the time, initiated a "generics" subgroup with Tom Clune as its lead. This work has lead to a *type-safe template* proposal.[8]

The LFortran project provides a prototype compiler for the Fortran template mechanism.[9] LFortran's template syntax deviates from the official INCITS Fortran proposed notation. The examples in this paper uses the LFortran notation, but some of the code is too advanced for the LFortran compiler to handle.

[7] Peruse the archive of the INCITS Fortran committee at `https://www.j3-fortran.org` for these.
[8] See the relevant papers on `https://j3-fortran.org/doc/year/24`
[9] See `https://lfortran.org` for the LFortran project. On `https://dev.lfortran.org` an in-browser version of the compiler and runtime system is available. The in-browser compiler also contains a selection of template Fortran example codes.

2 Binary Search

Binary search is a very efficient algorithm for finding elements stored in a sorted array. It works by repeatedly halving the segment of the array where the sought after value may be stored, until it finds the value or the array segment is empty.

2.1 A plain implementation

Writing such an algorithm is a programming text book exercise. Below we give a version satisfying the following specification.

- The algorithm assumes the first `n` elements of the array `arr` are sorted in ascending order.
- It looks for the element value `target` among the first `n` elements of `arr`.
- It returns an integer which is to be interpreted as an index into `arr`:
  - A positive tnteger if the target is found at that index.
  - A negative integer of the next larger element if the target is not found.
  - `-(n+1)` if the target is larger than all elements in the array.

In this way the algorithm is useful also for non-discrete data, e.g., real numbers and colour spaces where an exact match could be rare.

```
simple integer function binary_search(arr, n, target)
  implicit none
  ! The elements are of integer type, arr(1:n) must be sorted
  integer, intent(in) :: arr(:)
  ! The target element to be searched for in the array
  integer, intent(in) :: target
  ! Index into the array: upper limit of array segment to be searched
  integer, intent(in) :: n

  ! Indices into the array (local variables)
  integer :: low, high, mid

  low = 1
  high = n

  do while (low <= high)
      mid = (low + high) / 2
      if (arr(mid) < target) then
          ! Target in upper half of array segment
          low = mid + 1
      else if (target < arr(mid)) then
          ! Target in lower half of array segment
          high = mid - 1
      else ! arr(mid) == target
          binary_search = mid
          return
      end if
  end do

  ! If not found, return the negative index for the next larger element
  binary_search = -low
end function binary_search
```

In order to use an implementation across different types of data we will templatise the code. If we are thinking like a C++ template developer, we introduce a type argument, a *deferred* type, `T` for the elements.

```fortran
  simple integer function binary_search{T}(arr, n, target)
    ! Declaring template argument type T
    type, deferred :: T
    ! The elements are of type T, arr(1:n) must be sorted
    type(T), intent(in) :: arr(:)
    ! The target element of type T to be searched for in the array
    type(T), intent(in) :: target
    ! Index into the array: upper limit of array segment to be searched
    integer, intent(in) :: n

    ! Indices into the array (local variables)
    integer :: low, high, mid

    low = 1
    high = n

    do while (low <= high)
        mid = (low + high) / 2
        if (arr(mid) < target) then
            ! Target in upper half of array segment
            low = mid + 1
        else if (target < arr(mid)) then
            ! Target in lower half of array segment
            high = mid - 1
        else ! arr(mid) == target
            binary_search = mid
            return
        end if
    end do

    ! If not found, return the negative index for the next larger element
    binary_search = -low
  end function binary_search
```

This is invalid for Fortran templates, since there is no way the compiler can know if a type `T` will have an `operator(<)`. The only types in Fortran that intrinsically admits `operator(<)` are `INTEGER` and `REAL`. The LFortran compiler gives the following error message.

```
semantic error: Compare: only Integer or Real can be on the LHS and RHS.
  If operator is .eq. or .neq. then Complex type is also acceptable
  --> BinarySearch.f90:28:13
   |
28 |        if (arr(mid) < target) then
   |            ^^^^^^^^^^^^^^^^^
```

Ensuring type-safety forces us to explicitly declare the comparison operator as a argument. This will also open up for greater flexibility, e.g., using `operator(>)` for binary search in an array sorted in descending order.

2.2 A naïve standalone template procedure

In the previous section we provided code for searching for integers in an integer array. Obviously the same algorithm would work for any data. So we would like to introduce a template type argument `T` for the target and elements of the array. As noted above, we also need to introduce the comparison operation as a template argument.

For Fortran templates we can define a *requirement* with the declaration of the comparison predicate. It uses Fortran's existing *interface* construct, a somewhat longwinded notation.

```
requirement binary_predicate_r( T, bin )
    type, deferred :: T
    interface
      simple logical function bin(x, y)
        type(T), intent(in) :: x, y
      end function
    end interface
end requirement
```

This requirement defines a binary predicate, a logical function with two arguments. We also state that the type `T` is *deferred*, i.e., it is a type argument to the requirement.[10] Such requirements will be natural to include in a standard library for template Fortran.

Now the naïve templated version of binary search can look like this.

```
24  simple integer function binary_search {T,lt} ( arr, n, target )
25    ! Declaring template arguments type T and strict comparison relation lt
26    require :: binary_predicate_r(T,lt)
27    ! The elements are of type T, arr(1:n) must be sorted according to lt
28    type(T), intent(in) :: arr(:)
29    ! The target element of type T to be searched for in the array
30    type(T), intent(in) :: target
31    ! Index into the array: upper limit of array segment to be searched
32    integer, intent(in) :: n
33
34    ! Indices into the array (local variables)
35    integer :: low, high, mid
36
37    low = 1
38    high = n
39
40    do while (low <= high)
41        mid = (low + high) / 2
42        if ( lt( arr(mid), target ) ) then
43            ! Target in upper half of array segment
44            low = mid + 1
45        else if ( lt( target, arr(mid) ) ) then
46            ! Target in lower half of array segment
47            high = mid - 1
48        else ! arr(mid) == target
49            binary_search = mid
50            return
51        end if
52    end do
53
54    ! If not found, return the negative index for the next larger element
55    binary_search = -low
56  end function binary_search
```

Note the *require* statement on line 26. It uses the requirement `binary_predicate_r` to both declare `T` as a deferred type and `lt` as a binary predicate. This is a very compact way to get both the declarations in place by using the requirement.

[10] It is not necessary to introduce a requirement, as the body of the requirement can be placed directly in the body of the standalone procedure instead of using a require statement.

We consider this implementation naïve since it does not cater for Fortran having multiple integer types. Such types are differentiated by the *kind* attribute, varying in what ranges they cover, e.g., 32bit integers (-2 147 483 648 through 2 147 483 647) or 64 bit integers (-9 223 372 036 854 775 808 through 9 223 372 036 854 775 807).

The other problem with this algorithm is that it does not handle searching very large arrays well. Consider an array with 4GB of BFloat16 data (a 16bit floating point commonly used in machine learning applications). This is just about the range that can be handled using signed 32bit integers, a typical Fortran integer. When searching in the upper end of such an array, the sum `low + high`, see line 38, will exceed the 32bit range, causing an overflow in the computation, leaving `mid` with an irrelevant value. Hopefully the situation is detected and not silently ignored. This problem was discovered back in 2006 when standard computer memory went beyond 2GB [1].

2.3 A hardened template procedure

Here we harden the templated binary search algorithm to ensure that no user of it will trip up.

We let the user choose which kind of integers should be used for indexing, the template argument `k`. Thus the user may use a large range integer, such as 64bit, if the arrays are extremely large, e.g., more than 16GB of double precision floating point data.

We reformulate the computation of the value `mid` such that no internal overflow will occur, however big the data set becomes.

As above, we exploit the flexibility of using templates in letting the user choose what type of data to be searched, the template type argument `T`, and which ordering relation to be used, the `lt` argument.

```
simple integer(kind=k) function binary_search {T,lt,k} ( arr, n, target )
  ! Declaring template arguments type T and strict comparison relation lt
  require :: binary_predicate_r(T,lt)
  ! Declaring constant k for use as kind argument for the indices
  integer, deferred, argument :: k
  ! The elements are of type T, arr(1:n) must be sorted according to lt
  type(T), intent(in) :: arr(:)
  ! The target element of type T to be searched for in the array
  type(T), intent(in) :: target
  ! Index into the array: upper limit of array segment to be searched
  integer(kind=k), intent(in) :: n

  ! Indices into the array (local variables)
  integer(kind=k) :: low, high, mid

  low = 1
  high = n

  do while (low <= high)
      mid = low + (high-low) / 2
      if ( lt( arr(mid), target ) ) then
          ! Target in upper half of array segment
          low = mid + 1
      else if ( lt( target, arr(mid) ) ) then
          ! Target in lower half of array segment
          high = mid - 1
      else ! arr(mid) == target
          binary_search = mid
          return
```

```
      end if
    end do

    ! If not found, return the negative index for the next larger element
    binary_search = -low
  end function binary_search
```

In order to use the templated code, we need to instantiate it with the relevant arguments.

3 Instantiating Binary Search

To use the templated binary search algorithm we need to instantiate it for the circumstance where we want to use it. If we want exactly the same behaviour as the very first binary search implementation, we need to select the default integer type as our element type, use `operator(<)` as our comparison operation, and provide the argument `k` with `kind(0)` which gives the kind value corresponding to the default integer. Thus we achieve this goal by using the template arguments `{integer,operator(<),4}` to the function call. In the call we let `arr` be an integer array with 10 elements and 7 be the target, both with the default integer kind.

```
  position = binary_search {integer,operator(<),kind(0)} ( arr, 10, 7 )
```

This replicates the call from Section 2.1. The second template arguments provides the ordering `operator(<)`, since it is impossible for the compiler to decide wihich ordering operation is intended, e.g., `operator(<)` versus `operator(>)`.

It might feel unnecessary to list `integer` in the template argument list, since this could be derived from the type of the arguments `arr` and 7. But the `operator(<)` is *overloaded* (a *generic name* in Fortran terminology) for all integer and real kinds in Fortran, and may also be provided by user defined derived types. However, since the exact declaration of this operator is defined by the requirement `binary_predicate_r`, the compiler is able to determine precisely which of these alternatives will be used. This is different from arguments to procedures, which must have a unique name (a *specific name* in Fortran terminology) in order to type-check the procedure call. When needing a specific name for the comparison operator or other generic operators or procedures, we will have to write wrapper functions, a significant inconvenience.

The tradeoff to declare the type argument explicitly allows the deduction of which operation to use from overload sets–an important ergonomic for using templates.

We can also instantiate binary search for complex numbers, but then we need to define an ordering relation to use for the search. The following logical function orders complex numbers first on their real component, then by their imaginary component.

```
   pure logical function complex_less(zx, zy) result(less)
      complex, intent(in) :: zx, zy
      less = (zx%re < zy%re)
      if (zx%re /= zy%re) return
      ! real parts equal - imaginary part is tie breaker
      less = (zx%im < zy%im)
   end function complex_less
```

We can then instantiate binary search for complex numbers using `complex_less`.

```
  position = binary_search {complex,complex_less,kind(0)} ( carr, 10, (7.,.4) )
```

This will search for the complex number `(7.,.4)` in an array `carr` of 10 complex numbers.

4 Summary

When preparing library code, it is important to consider corner cases that may arise for users of the library. In the above sequence of binary search algorithms, we started out with a version that would return useful indices for data that might not be discrete like the integers. Then we templatised this procedure, ensuring that the code could be used for any type `T` and any strict comparison operation for that type. We then hardened the code, allowing a user of the template to choose which of Fortran's integer kinds to use for indexing, and carefully computing internal data in such a way not to cause overflow issues.

Fortran's template mechanism is type-safe.

The body of a standalone procedure is type-checked by the compiler and possible problems are flagged immediately. This is the same as one would expect non-template code to be type-checked.

An instantiation is type-checked based on the declarations of the template arguments, ensuring that instantiation errors are flagged as such. There will be no error message relating to template internals at this stage, since the compiler will not need to look inside the template in order to discover inconsistencies.

We also discussed the trade-off of the user having to provide explicitly type arguments to a template instantiation, with the benefit of allowing overloaded operators and procedures as arguments. The compiler then uses the explicit type arguments to deduce which specific procedure from an overload set is intended.

The appendices contain the standard Fortran, non-hardened version of the plain code written specifically for integers, and the hardened template Fortran version which is fully reusable.

Both these versions come with a simple test program that builds confidence in the code.

We hope this excursion into template Fortran and binary search has been a useful venture for the reader.

References

1. Bloch, J.: Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken, `https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`, accessed 2020-07-23
2. Haveraaen, M., Järvi, J., Rouson, D.: Reflecting on generics for Fortran (2019), `{https://j3-fortran.org/doc/year/19/19-188.pdf}`, presented at the Fortran Standardisation Meeting, ANSI PL22.3 (J3) & ISO/IEC JTC 1/SC 22/WG 5 in Tokyo (Japan), 2019-08-05 through 2019-08-09.

A Integer Binary Search with test program

```
module binary_search_m
  implicit none
contains

  !! The binary search function searches through the first n elements of arr looking for target.
  !! It returns:
  !! - A positive index in arr to the target if found.
  !! - A negative index of the next larger element in arr if the target is not found.
  !! - -(n+1) if the target is larger than all elements in the array.
  simple integer function binary_search(arr, n, target)
    implicit none
    ! The elements are of integer type, arr(1:n) must be sorted
    integer, intent(in) :: arr(:)
    ! The target element to be searched for in the array
    integer, intent(in) :: target
    ! Index into the array: upper limit of array segment to be searched
    integer, intent(in) :: n

    ! Indices into the array (local variables)
    integer :: low, high, mid
```

```fortran
    low = 1
    high = n

    do while (low <= high)
       mid = (low + high) / 2
       if (arr(mid) < target) then
          ! Target in upper half of array segment
          low = mid + 1
       else if (target < arr(mid)) then
          ! Target in lower half of array segment
          high = mid - 1
       else ! arr(mid) == target
          binary_search = mid
          return
       end if
    end do

    ! If not found, return the negative index for the next larger element
    binary_search = -low
  end function binary_search

end module binary_search_m


program binary_search_program
    use binary_search_m
    implicit none

    !! Test data
    logical :: testerr
    integer :: arr(10) = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
    integer :: testdata(7) = [12,22,13,3,5,1,0]
    integer :: testsizes(5) = [10,8,2,1,0]
    integer :: answers(5,7)
    integer :: i, j, position
      answers(1,:) = [-7,-11, 7, 2, 3, 1,-1]
      answers(2,:) = [-7, -9, 7, 2, 3, 1,-1]
      answers(3,:) = [-3, -3,-3, 2,-3, 1,-1]
      answers(4,:) = [-2, -2,-2,-2,-2, 1,-1]
      answers(5,:) = [-1, -1,-1,-1,-1,-1,-1]

    print *, "Binary search test program"
    print *, "Unit test data"
    print *, "testsizes=", testsizes
    print *, "answers="
    print *, answers
    print *, "Unit tests"
    testerr = .false.
    print *, "arr=", arr
    print *, "testdata=", testdata
    do j = 1,size(testsizes)
      do i = 1, size(testdata)
        position = binary_search(arr, testsizes(j), testdata(i))
        if (.not. position == answers(j,i)) then
```

```
        print *, "error: j=", j, " testsizes(j)=", testsizes(j), " i=", i, &
          " testdata(i)=", testdata(i), " position=", position, " answers(j,i)=", answers(j,i)
        testerr = .true.
      end if
    end do
  end do
  if (testerr) then
    print *, "Test errors, see diagnostics above"
  else
    print *, "SUCCESS!"
  end if

end program binary_search_program
```

B Hardened Template Binary Search with test program

```
module binary_search_m
  implicit none

    !! A binary predicate is a logical function which takes two inputs of type T.
    requirement binary_predicate_r( T, bin )
        type, deferred :: T
        interface
          simple logical function bin(x, y)
            type(T), intent(in) :: x, y
          end function
        end interface
    end requirement


contains

  !! This is a standalone template procedure for binary search.
  !! It works for any element type T with a strict comparison predicate lt.
  !! The range used for integer indices is given by kind k.
  !! The binary search function searches through the first n elements of arr looking for target.
  !! It returns:
  !! - A positive index in arr to the target if found.
  !! - A negative index of the next larger element in arr if the target is not found.
  !! - -(n+1) if the target is larger than all elements in the array.
  simple integer(kind=k) function binary_search {T,lt,k} ( arr, n, target )
    ! Declaring template arguments type T and strict comparison relation lt
    require :: binary_predicate_r(T,lt)
    ! Declaring constant k for use as kind argument for the indices
    integer, deferred, parameter :: k
    ! The elements are of type T, arr(1:n) must be sorted according to lt
    type(T), intent(in) :: arr(:)
    ! The target element of type T to be searched for in the array
    type(T), intent(in) :: target
    ! Index into the array: upper limit of array segment to be searched
    integer(kind=k), intent(in) :: n

    ! Indices into the array (local variables)
    integer(kind=k) :: low, high, mid
```

```fortran
    low = 1
    high = n

    do while (low <= high)
        mid = low + (high-low) / 2
        if ( lt( arr(mid), target ) ) then
            ! Target in upper half of array segment
            low = mid + 1
        else if ( lt( target, arr(mid) ) ) then
            ! Target in lower half of array segment
            high = mid - 1
        else ! arr(mid) == target
            binary_search = mid
            return
        end if
    end do

    ! If not found, return the negative index for the next larger element
    binary_search = -low
  end function binary_search

end module binary_search_m


program binary_search_program
    ! Import bnary search
    use binary_search_m
    ! Import integer kinds for 32bit and 64bit signed integers.
    use iso_fortran_env, only : INT32, INT64
    implicit none

    ! Defining kind for 32bit integers (index type)
    integer, parameter :: k = INT32
    ! Defining kind for 64bit integers (element type)
    integer, parameter :: d = INT64

    !! Test data
    logical :: testerr
    integer(kind=d) :: arr(10) = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
    complex :: carr(10) = [(1.,0.), (1.,.3), (5.,0.), (5.,7.), (5.,9.9), (11.,1.), (13.2.), (15.,0.)
    integer(kind=d) :: testdata(7) = [12,22,13,3,5,1,0]
    complex :: carr(10) = [(1.,0.), (1.,.3), (5.,0.), (5.,7.), (5.,9.9), (11.,1.), (13.,2.), (15.,0.
    complex :: ctestdata(7) = [(12.,-2.5), (22.,2.),(13.,2.),(1.,.3),(5.,0.),(1.0,0.),(1.,-0.4)]
    integer(kind=d) :: testsizes(5) = [10,8,2,1,0]
    integer(kind=d) :: answers(5,7)
    integer(kind=k) :: i, j, position
      answers(1,:) = [-7,-11, 7, 2, 3, 1,-1]
      answers(2,:) = [-7, -9, 7, 2, 3, 1,-1]
      answers(3,:) = [-3, -3,-3, 2,-3, 1,-1]
      answers(4,:) = [-2, -2,-2,-2,-2, 1,-1]
      answers(5,:) = [-1, -1,-1,-1,-1,-1,-1]

    print *, "Binary search test program"
```

```
    print *, "Unit test data"
    print *, "testsizes=", testsizes
    print *, "answers="
    print *, answers
    print *, "Unit tests"
    testerr = .false.
    print *, "arr=", arr
    print *, "testdata=", testdata
    do j = 1,size(testsizes)
      do i = 1, size(testdata)
        position = binary_search {integer(kind=d),operator(<),k} ( arr, testsizes(j), testdata(i) )
        if (.not. position == answers(j,i)) then
          print *, "error: j=", j, " testsizes(j)=", testsizes(j), " i=", i, &
            " testdata(i)=", testdata(i), " position=", position, " answers(j,i)=", answers(j,i)
          testerr = .true.
        end if
      end do
    end do
    print *, "carr=", carr
    print *, "ctestdata=", ctestdata
    do j = 1,size(testsizes)
      do i = 1, size(ctestdata)
        position = binary_search {complex,complex_less,k} ( carr, testsizes(j), ctestdata(i) )
        if (.not. position == answers(j,i)) then
          print *, "error: j=", j, " testsizes(j)=", testsizes(j), " i=", i, &
            " ctestdata(i)=", ctestdata(i), " position=", position, " answers(j,i)=", answers(j,i)
          testerr = .true.
        end if
      end do
    end do
    if (testerr) then
      print *, "Test errors, see diagnostics above"
    else
      print *, "SUCCESS!"
    end if

    contains

   ! Provide a lexical ordering on complex numbers: first real part, then imaginary part.
   pure logical function complex_less(zx, zy) result(less)
      complex, intent(in) :: zx, zy
      less = (zx%re < zy%re)
      if (zx%re /= zy%re) return
      ! real parts equal - imaginary part is tie breaker
      less = (zx%im < zy%im)
   end function complex_less

end program binary_search_program
```