Chapter 25
Generic Programming

General notions are generally wrong.
Letter to Mr. Wortley Montegu, 28th March 1710.

Aims

This chapter looks at some examples that implement generic programming in Fortran.

25.1 Introduction

Fortran 77 had several generic functions, e.g. the sine function could be called with arguments of type real, double precision or complex. Fortran 90 extended the idea so that a programmer could write their own generic functions or subroutines. For example we can now write a sort routine which works with arguments of a variety of types, e.g. integer, real etc.

25.2 Generic Programming and Other Languages

Generic programming has a wider meaning in computer science and effectively is a style of computer programming in which an algorithm is written once, but can be made to work with a variety of types.

This style of programming is provided in several programming languages and in a variety of ways.

Languages that support generics include

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_25

- Ada
- C#
- Eiffel
- Java
- C++

To quote the generic programming pioneer Alexander Stepanov;

> … Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.

and quoting Bjarne Stroustrup:

> … lift algorithms and data structures from concrete examples to their most general and abstract form.

We'll look at a concrete example in Fortran next.

25.3 Example 1: Sorting Reals and Integers

In Chap. 20 Example 5 had a module called `sort_data_module` that contained a `sort_data` subroutine. The `sort_data` subroutine in turn contained an internal `quicksort` subroutine that did the actual sorting.

Here is the start of the `sort_data` subroutine.

```
subroutine sort_data(raw_data, how_many)
  implicit none
  integer, intent (in) :: how_many
  real, intent (inout), dimension (:) :: raw_data
```

and we called this subroutine as shown below from the main program.

```
call sort_data(x,n)
```

The subroutine worked with an array of default real type. We will use the module `sort_data_module` and subroutine `sort_data` as the basis of a module that will work with arrays of four integer types and three real types.

The first thing we need are modules that defines kind type parameters for the three real types and four integer types.

These two modules are shown below.

```
module precision_module
  implicit none
```

```
  integer, parameter :: sp = selected_real_kind( &
    6, 37)
  integer, parameter :: dp = selected_real_kind( &
    15, 307)
  integer, parameter :: qp = selected_real_kind( &
    30, 291)
end module precision_module


module integer_kind_module
  implicit none
  integer, parameter :: i8 = selected_int_kind(2 &
    )
  integer, parameter :: i16 = selected_int_kind( &
    4)
  integer, parameter :: i32 = selected_int_kind( &
    9)
  integer, parameter :: i64 = selected_int_kind( &
    15)
end module integer_kind_module
```

We can now use these modules in the new module `sort_data_module` and main program.

We must use an interface to link the common calling name (`sort_data`) to the specific subroutines that handle each specific type.

Here is the interface block from the module `sort_data_module`.

```
  interface sort_data
    module procedure sort_real_sp
    module procedure sort_real_dp
    module procedure sort_real_qp
    module procedure sort_integer_8
    module procedure sort_integer_16
    module procedure sort_integer_32
    module procedure sort_integer_64
  end interface sort_data
```

In the original subroutine in Chap. 20 we had a call

```
call sort_date(raw_data,how_many)
```

and the subroutine `sort_data` had two arguments or parameters, a real array, and an integer for the size.

So the call is still the same, but now we can call the `sort_data` subroutine with an array of any of the four integer types or three real types.

The compiler will then look at the type, kind and ranks of the parameters in the call to the `sort_data` subroutine and call the appropriate module procedure.

Here is the new module `sort_data_module`.

```
module sort_data_module

  use precision_module
  use integer_kind_module

  interface sort_data
    module procedure sort_real_sp
    module procedure sort_real_dp
    module procedure sort_real_qp
    module procedure sort_integer_8
    module procedure sort_integer_16
    module procedure sort_integer_32
    module procedure sort_integer_64
  end interface sort_data

contains

  subroutine sort_real_sp(raw_data, how_many)
    use precision_module
    implicit none
    integer, intent (in) :: how_many
    real (sp), intent (inout), dimension (:) :: &
      raw_data

    call quicksort(1, how_many)

  contains

    recursive subroutine quicksort(l, r)
      implicit none
      integer, intent (in) :: l, r
      integer :: i, j
      real (sp) :: v, t

      include 'quicksort_include_code.f90'

    end subroutine quicksort

  end subroutine sort_real_sp
```

```
subroutine sort_real_dp(raw_data, how_many)
  use precision_module
  implicit none
  integer, intent (in) :: how_many
  real (dp), intent (inout), dimension (:) :: &
    raw_data

  call quicksort(1, how_many)

contains
  recursive subroutine quicksort(l, r)
    implicit none
    integer, intent (in) :: l, r
    integer :: i, j
    real (dp) :: v, t

    include 'quicksort_include_code.f90'

  end subroutine quicksort
end subroutine sort_real_dp

subroutine sort_real_qp(raw_data, how_many)
  use precision_module
  implicit none
  integer, intent (in) :: how_many
  real (qp), intent (inout), dimension (:) :: &
    raw_data

  call quicksort(1, how_many)




contains
  recursive subroutine quicksort(l, r)
    implicit none
    integer, intent (in) :: l, r
    integer :: i, j
    real (qp) :: v, t

    include 'quicksort_include_code.f90'

  end subroutine quicksort
end subroutine sort_real_qp
```

```
subroutine sort_integer_8(raw_data, how_many)
  use integer_kind_module
  implicit none
  integer, intent (in) :: how_many
  integer (i8), intent (inout), &
    dimension (:) :: raw_data

  call quicksort(1, how_many)

contains
  recursive subroutine quicksort(l, r)
    implicit none
    integer, intent (in) :: l, r
    integer :: i, j
    integer (i8) :: v, t

    include 'quicksort_include_code.f90'

  end subroutine quicksort
end subroutine sort_integer_8

subroutine sort_integer_16(raw_data, how_many)
  use integer_kind_module
  implicit none
  integer, intent (in) :: how_many
  integer (i16), intent (inout), &
    dimension (:) :: raw_data

  call quicksort(1, how_many)

contains
  recursive subroutine quicksort(l, r)
    implicit none
    integer, intent (in) :: l, r
    integer :: i, j
    integer (i16) :: v, t

    include 'quicksort_include_code.f90'

  end subroutine quicksort
end subroutine sort_integer_16



subroutine sort_integer_32(raw_data, how_many)
```

```
    use integer_kind_module
    implicit none
    integer, intent (in) :: how_many
    integer (i32), intent (inout), &
      dimension (:) :: raw_data

    call quicksort(1, how_many)

  contains
    recursive subroutine quicksort(l, r)
      implicit none
      integer, intent (in) :: l, r
      integer :: i, j
      integer (i32) :: v, t

      include 'quicksort_include_code.f90'

    end subroutine quicksort
  end subroutine sort_integer_32

  subroutine sort_integer_64(raw_data, how_many)
    use integer_kind_module
    implicit none
    integer, intent (in) :: how_many
    integer (i64), intent (inout), &
      dimension (:) :: raw_data

    call quicksort(1, how_many)

  contains
    recursive subroutine quicksort(l, r)
      implicit none
      integer, intent (in) :: l, r
      integer :: i, j
      integer (i64) :: v, t

      include 'quicksort_include_code.f90'

    end subroutine quicksort

  end subroutine sort_integer_64

end module sort_data_module
```

In this module we have implementations for each of the module procedures listed in the interface block.

Here is the include file,

```
i = l
j = r
v = raw_data(int((l+r)/2))
do
  do while (raw_data(i)<v)
    i = i + 1
  end do
  do while (v<raw_data(j))
    j = j - 1
  end do
  if (i<=j) then
    t = raw_data(i)
    raw_data(i) = raw_data(j)
    raw_data(j) = t
    i = i + 1
    j = j - 1
  end if
  if (i>j) exit
end do
if (l<j) then
  call quicksort(l, j)
end if
if (i<r) then
  call quicksort(i, r)
end if
```

which is used in each of the seven subroutines and is effectively a common algorithm between all seven subroutines.

Here is the main program to test the generic sort module.

```
include 'integer_kind_module.f90'
include 'precision_module.f90'
include 'sort_data_module.f90'

program ch2501

  use precision_module
  use integer_kind_module
  use sort_data_module
```

```
  implicit none
  integer, parameter :: n = 1000000
  real (sp), allocatable, dimension (:) :: x
  integer (i32), allocatable, dimension (:) :: y
  integer :: allocate_status

  allocate_status = 0

  print *, ' Program starts'
  allocate (x(1:n), stat=allocate_status)

  if (allocate_status/=0) then
    print *, ' Allocate failed.'
    print *, ' Program terminates'
    stop 10
  end if

  print *, ' Real allocate complete'
  call random_number(x)
  print *, ' Real array initialised'
  call sort_data(x, n)
  print *, ' Real sort ended'
  print *, ' First 10 reals'
  write (unit=*, fmt=100) x(1:10)
100 format (5(2x,e14.6))
  allocate (y(1:n), stat=allocate_status)
  if (allocate_status/=0) then
    print *, ' Allocate failed.'
    print *, ' Program terminates'
    stop 10
  end if
  y = int(x*1000000)
  deallocate (x)
  print *, ' Integer array initialised'
  call sort_data(y, n)
  print *, ' Sort ended'
  print *, ' First 10 integers'
  write (unit=*, fmt=110) y(1:10)
110 format (5(2x,i10))
  deallocate (y)
  print *, ' Deallocate'
  print *, ' Program terminates'

end program ch2501
```

This is obviously a very significant facility to have in a programming language.

Have a look at the following two examples which show the code for a generic quicksort in C++ and C#.

25.3.1 Generic Quicksort in C++

Here is the C++ program.

```
template <class Type>
void swap(Type array[],int i, int j)
{
  Type tmp=array[i];
  array[i]=array[j];
  array[j]=tmp;
}

template <class Type>
void quicksort( Type array[], int l, int r)
{
  int i=l;
  int j=r;
  Type v=array[int((l+r)/2)];
  for (;;)
  {
    while (array[i] < v) i=i+1;
    while (v < array[j]) j=j-1;
    if (i<=j)
      { swap(array,i,j); i=i+1 ; j=j-1; }
    if (i>j) goto ended ;
  }
  ended: ;
  if (l<j) quicksort(array,l,j);
  if (i<r) quicksort(array,i,r);
}

template <class Type>
void print(Type array[],int size)
{
  cout << " [ " ;
  for (int ix=0;ix<size; ++ix)
    cout << array[ix] << " ";
  cout << "] \n";
```

```
}

#include <iostream>
using namespace std;
int main()
{
  double da[] =
  {1.9,8.2,3.7,6.4,5.5,1.8,9.2,3.6,7.4,5.5};
  int ia[] = {1,10,2,9,3,8,4,7,6,5};
  int size=sizeof(da)/sizeof(double);
  cout << " Quicksort of double array is \n";
  quicksort(da,0,size-1);
  print(da,size);
  size=sizeof(ia)/sizeof(int);
  cout << " Quicksort of integer array is \n";
  quicksort(ia,0,size-1);
  print(ia,size);
  return(0);
}
```

25.3.2 Generic Quicksort in C#

Here is the C# version.

```
using System;
public static class generic
{

  public static void
swap< Type > (Type[] array,int i, int j)
  {
    Type tmp=array[i];
    array[i]=array[j];
    array[j]=tmp;
  }

  public static void
quicksort< Type > ( Type[] array, int l, int r)
    where Type : IComparable< Type >
  {
    int i=l;
    int j=r;
    Type v=array[(int)((l+r)/2)];
```

```
    for (;;)
    {
      while (array[i].CompareTo( v) < 0 ) i=i+1;
      while (v.CompareTo(array[j]) < 0) j=j-1;
      if (i<=j)
        { swap(array,i,j); i=i+1 ; j=j-1; }
      if (i>j) goto ended ;
    }
    ended: ;
    if (l<j) quicksort(array,l,j);
    if (i<r) quicksort(array,i,r);
  }

  public static void
print< Type > (Type[] array,int size)
  {
    int i;
    int l;
    l=array.Length;
    for (i=0;i<l;i++)
      Console.WriteLine(array[i]);
  }

  public static int Main()
  {
    double[] da =
    {1.9,8.2,3.7,6.4,5.5,1.8,9.2,3.6,7.4,5.5};
    int[]    ia = {1,10,2,9,3,8,4,7,6,5};
    int size;
    size=da.Length;
    Console.WriteLine("Original array");
    print(da,size);
    quicksort(da,0,size-1);
    Console.WriteLine("Sorted array");
    print(da,size);
    size=ia.Length;
    Console.WriteLine("Original array");
    print(ia,size);
    quicksort(ia,0,size-1);
    Console.WriteLine("Sorted array");
    print(ia,size);
    return(0);
  }

}
```

In C++ and C# we only have one version of the sort procedure and the compiler generates the code for us for each type of array we call the procedure with, which we have to actually write in Fortran.

25.4 Example 2: Generic Statistics Module

In this example we extend the statistics module from Chap. 20 (Example 4) to work with all three real kind types.

Here is the statistics module.

```
module statistics_module

  use precision_module

  interface calculate_statistics
    module procedure calculate_sp
    module procedure calculate_dp
    module procedure calculate_qp
  end interface calculate_statistics

contains


  subroutine calculate_sp(x, n, mean, std_dev, &
    median)
    implicit none
    integer, intent (in) :: n
    real (sp), intent (in), dimension (:) :: x
    real (sp), intent (out) :: mean
    real (sp), intent (out) :: std_dev
    real (sp), intent (out) :: median
    real (sp), dimension (1:n) :: y
    real (sp) :: variance
    real (sp) :: sumxi, sumxi2

    sumxi = 0.0
    sumxi2 = 0.0
    variance = 0.0
    sumxi = sum(x)
    sumxi2 = sum(x*x)
    mean = sumxi/n
    variance = (sumxi2-sumxi*sumxi/n)/(n-1)
    std_dev = sqrt(variance)
    y = x
```

```
    if (mod(n,2)==0) then
      median = (find(n/2)+find((n/2)+1))/2
    else
      median = find((n/2)+1)
    end if
  contains

    function find(k)
      implicit none
      real (sp) :: find
      integer, intent (in) :: k
      integer :: l, r, i, j
      real (sp) :: t1, t2
include 'statistics_module_include_code.f90'
    end function find
  end subroutine calculate_sp

  subroutine calculate_dp(x, n, mean, std_dev, &
    median)
    implicit none
    integer, intent (in) :: n
    real (dp), intent (in), dimension (:) :: x
    real (dp), intent (out) :: mean
    real (dp), intent (out) :: std_dev
    real (dp), intent (out) :: median
    real (dp), dimension (1:n) :: y
    real (dp) :: variance
    real (dp) :: sumxi, sumxi2

    sumxi = 0.0
    sumxi2 = 0.0
    variance = 0.0
    sumxi = sum(x)
    sumxi2 = sum(x*x)
    mean = sumxi/n
    variance = (sumxi2-sumxi*sumxi/n)/(n-1)
    std_dev = sqrt(variance)
    y = x
    if (mod(n,2)==0) then
      median = (find(n/2)+find((n/2)+1))/2
    else
      median = find((n/2)+1)
    end if
```

```
  contains
    function find(k)
      implicit none
      real (dp) :: find
      integer, intent (in) :: k
      integer :: l, r, i, j
      real (dp) :: t1, t2
include 'statistics_module_include_code.f90'
    end function find
  end subroutine calculate_dp

  subroutine calculate_qp(x, n, mean, std_dev, &
    median)
    implicit none
    integer, intent (in) :: n
    real (qp), intent (in), dimension (:) :: x
    real (qp), intent (out) :: mean
    real (qp), intent (out) :: std_dev
    real (qp), intent (out) :: median
    real (qp), dimension (1:n) :: y
    real (qp) :: variance
    real (qp) :: sumxi, sumxi2

    sumxi = 0.0
    sumxi2 = 0.0
    variance = 0.0
    sumxi = sum(x)
    sumxi2 = sum(x*x)
    mean = sumxi/n
    variance = (sumxi2-sumxi*sumxi/n)/(n-1)
    std_dev = sqrt(variance)
    y = x
    if (mod(n,2)==0) then
      median = (find(n/2)+find((n/2)+1))/2
    else
      median = find((n/2)+1)
    end if
  contains
    function find(k)
      implicit none
      real (qp) :: find
      integer, intent (in) :: k
      integer :: l, r, i, j
      real (qp) :: t1, t2
include 'statistics_module_include_code.f90'
```

```
    end function find
  end subroutine calculate_qp

end module statistics_module
```

Here is the common include file.

```
l = 1
r = n
do while (l<r)
  t1 = y(k)
  i = l
  j = r
  do
    do while (y(i)<t1)
      i = i + 1
    end do
    do while (t1<y(j))
      j = j - 1
    end do
    if (i<=j) then
      t2 = y(i)
      y(i) = y(j)
      y(j) = t2
      i = i + 1
      j = j - 1
    end if
    if (i>j) exit
  end do
  if (j<k) then
    l = i
  end if
  if (k<i) then
    r = j
  end if
end do
find = y(k)
```

Here is the main program to test the statistics module.

```
include 'precision_module.f90'
include 'statistics_module.f90'
include 'timing_module.f90'

program ch2502

  use precision_module
  use statistics_module
  use timing_module

  implicit none
  integer :: n
  real (sp), allocatable, dimension (:) :: x
  real (sp) :: x_m, x_sd, x_median
  real (dp), allocatable, dimension (:) :: y
  real (dp) :: y_m, y_sd, y_median
  real (qp), allocatable, dimension (:) :: z
  real (qp) :: z_m, z_sd, z_median
  character *20, dimension (3) :: heading = [ &
    '  Allocate    ', '  Random      ', &
    '  Statistics  ' ]

  call start_timing()
  n = 50000000
  print *, ' n = ', n

  print *, ' Single precision'

  allocate (x(1:n))
  print 100, heading(1), time_difference()
100 format (a20, 2x, f8.3)
  call random_number(x)
  print 100, heading(2), time_difference()
  call calculate_statistics(x, n, x_m, x_sd, &
    x_median)
  print 100, heading(3), time_difference()
  write (unit=*, fmt=110) x_m
110 format (' Mean               = ', f10.6)
  write (unit=*, fmt=120) x_sd
120 format (' Standard deviation = ', f10.6)
  write (unit=*, fmt=130) x_median
130 format (' Median             = ', f10.6)
```

```
    deallocate (x)

    print *, ' Double precision'

    allocate (y(1:n))
    print 100, heading(1), time_difference()
    call random_number(y)
    print 100, heading(2), time_difference()
    call calculate_statistics(y, n, y_m, y_sd, &
      y_median)
    print 100, heading(3), time_difference()
    write (unit=*, fmt=110) y_m
    write (unit=*, fmt=120) y_sd
    write (unit=*, fmt=130) y_median
    deallocate (y)

    print *, ' Quad precision'

    allocate (z(1:n))
    print 100, heading(1), time_difference()
    call random_number(z)
    print 100, heading(2), time_difference()
    call calculate_statistics(z, n, z_m, z_sd, &
      z_median)
    print 100, heading(3), time_difference()
    write (unit=*, fmt=110) z_m
    write (unit=*, fmt=120) z_sd
    write (unit=*, fmt=130) z_median
    deallocate (z)

end program ch2502
```

Here are some results for the gfortran, Intel, Nag and Oracle compilers (Table 25.1).

Table 25.1 ch2502 results

Compiler	gfortran	Intel	Nag	Oracle	
n = 50,000,000					Average time
Single precision					
Allocate	0.000	0.000	0.000	0.000	0.000
Random	0.484	0.469	0.484	1.230	0.667
Statistics	1.312	0.766	1.031	0.773	0.971
Total time	1.796	1.235	1.515	2.003	1.637
Mean	0.335544	0.335544	0.335544	0.335544	
Standard deviation	0.465684	0.442725	0.442758	0.442686	
Median	0.500006	0.499965	0.500044	0.499957	
Double precision					
Allocate	0.020	0.016	0.016	0.000	0.013
Random	1.105	0.859	0.359	1.312	0.909
Statistics	1.520	0.953	1.172	1.055	1.175
Total time	2.645	1.828	1.547	2.367	2.097
Mean	0.500017	0.499931	0.499984	0.499984	
Standard deviation	0.288686	0.288691	0.288699	0.288695	
Median	0.500011	0.499889	0.499935	0.500012	
Quad precision					
Allocate	0.027	0.031	0.031	0.004	0.023
Random	6.363	2.500	0.734	2.395	2.998
Statistics	7.766	6.453	4.109	10.840	7.292
Total time	14.156	8.984	4.874	13.239	10.313
Mean	0.500019	0.499995	0.500030	0.500084	
Standard deviation	0.288659	0.288660	0.288662	0.288688	
Median	0.500041	0.499994	0.500065	0.500125	

25.5 Problems

25.1 Write a generic swap routine, that swaps two rank 1 integer arrays and two rank 1 real arrays.

25.2 Using Example 2 from Chap. 22 as a starting point convert it to a generic variant which handles files of integer data type and real data type.

25.6 Bibliography

25.6.1 Generic Programming References

This site is a collection of Alex Stepanov's papers, class notes, and source code, covering generic programming and other topics.

```
http://www.stepanovpapers.com/
```

25.6.2 Generic Programming and C++

C++ Templates: The Complete Guide, David Vandevoorde, Nicolai M Josuttis, 2003 Addison-Wesley. ISBN 0-201-73484-2

25.6.3 Generic Programming and C#

Visit the following site

```
http://msdn.microsoft.com/en-us/
library/512aeb7t(v=vs.80).aspx
```

for a very good coverage of generics and C#.