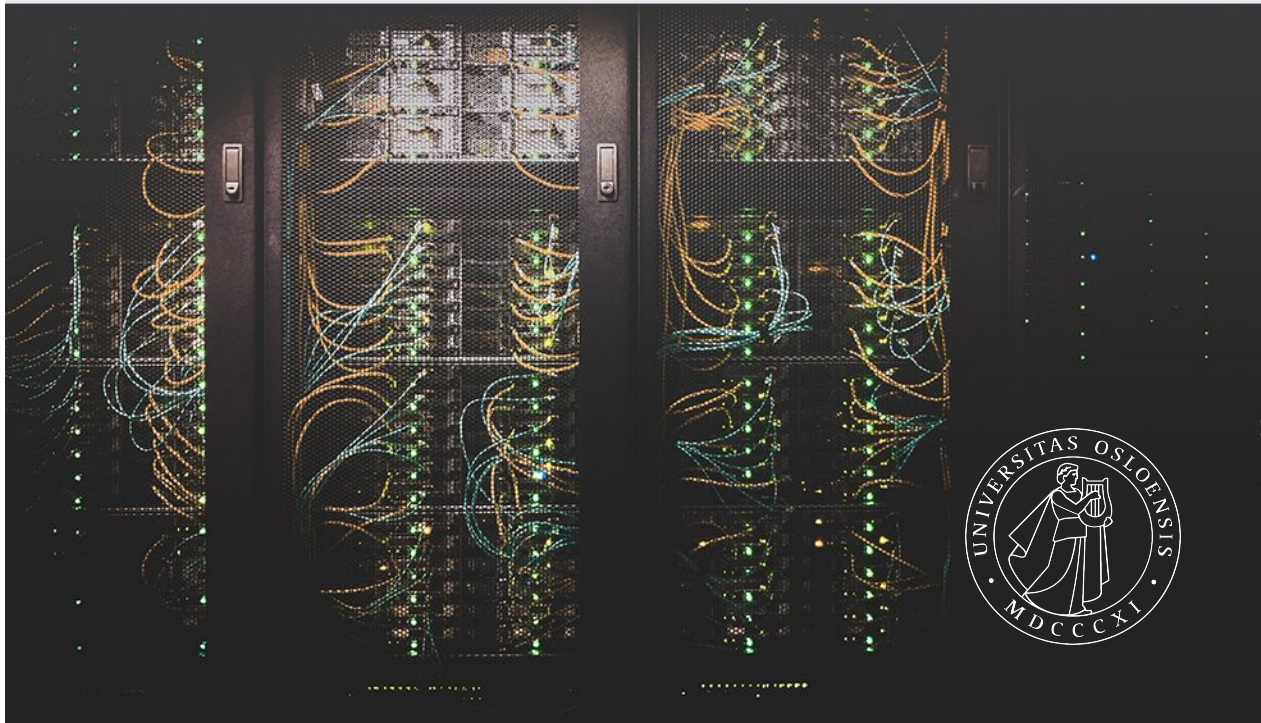
UNIVERSITETET I OSLO

IT-avdelingen



Ole W. Saastad, Dr. Scient (Kjemi)
IT-Avdelingen, Beregningstjenester

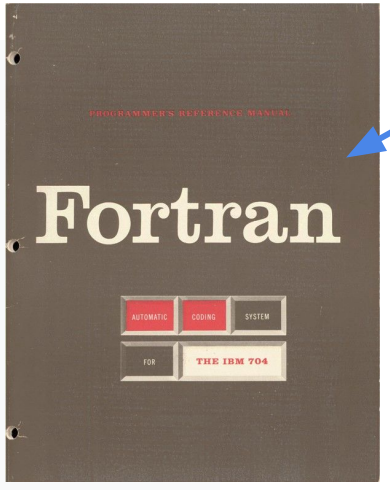
From your Grandmother’s fortran to the future.

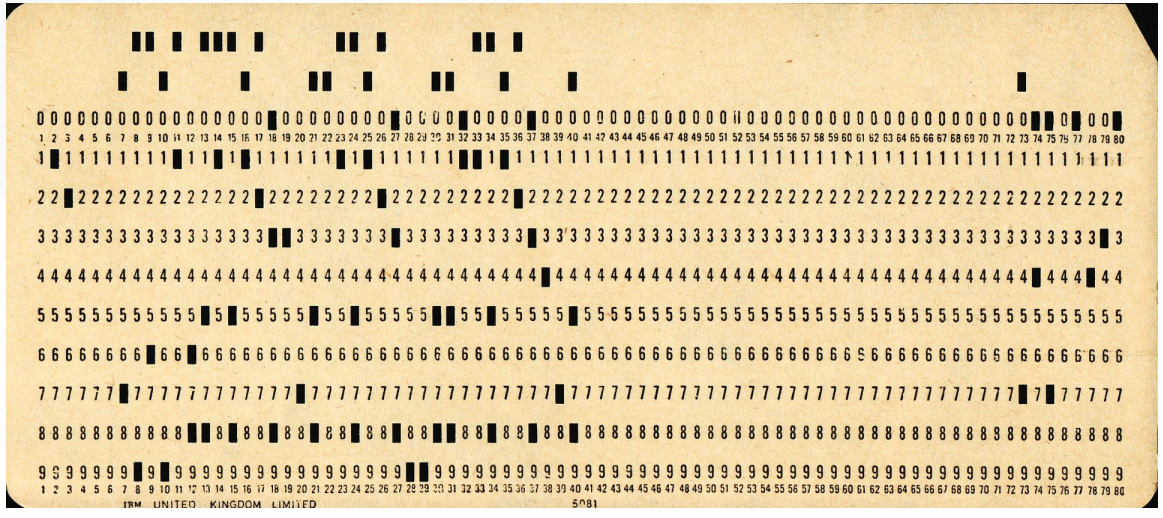
History, FORmula TRANslation, syntax development, data types, why still in use?, scientific language, compilers, efficiency, dusty decks, compatibility

UNIVERSITETET
I OSLO

In the beginning, there was FORTRAN – FORmula TRANslator for the IBM 704 (1954).

```
C AREA OF A TRIANGLE - HERON'S FORMULA
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT -
C INTEGER VARIABLES START WITH I,J,K,L,M OR N
      READ(5,501) IA,IB,IC
  501 FORMAT(3I5)
      IF (IA) 701, 777, 701
  701 IF (IB) 702, 777, 702
  702 IF (IC) 703, 777, 703
  777 STOP 1
  703 S = (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - IA) * (S - IB) * (S - IC) )
      WRITE(6,801) IA,IB,IC,AREA
  801 FORMAT(4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,
     $13H SQUARE UNITS)
      STOP
      END
```





Introduction to modern fortran

This training is an introduction to FORTRAN from old legacy code to more modern f90.

The rationale was that many students and scientists are meeting old legacy FORTRAN codes to maintain and develop.
It’s expected programming experience and knowledge of numerical programming.
As the F66 standard was introduced generations ago, knowledge of the totally renewed language is a good idea. Since early 90s and forward many new concepts are introduced.

This not a parallel programming course, hence co-arrays are not covered, but «do concurrent» will be. In addition offloading to accelerators, while still using ISO fortran is covered.
Modern fortran is object oriented like all modern programming languages.
With the new 2023 standard just voted in we’re good at least for the next few decades.

Why Fortran in 2023 ? - easy : **PERFORMANCE !**

Matrix matrix multiplication as an example, the major operation since Konrad Zuse (1941) up to today's machine learning.

Two major languages Python and Fortran (F77 code from 1989)

N=2000 , two 2000 x 2000 matrices multiplied together.

Python : `22067.96 secs` *(10000x10000 : estimated 51 days)*

Fortran : `0.6897 secs` *(10000x10000 : 139.895 secs)*

Fortran+lib : `0.3707 secs` *(10000x10000 : 43.9182 secs)*

Python code :

```
for j in range(n):
    for l in range(n):
        for i in range(n):
            z[i,j] = z[i,j] + x[i,l]*y[l,j]
```

Fortran code :

```
z = matmul(x,y)
```

Why Fortran in 2023 ? -
Simple **vector syntax** - less loops

```
a = sqrt(b)  ;                  call random(b)
z = matmul(x,y) ;               c = matmul(b(1:3,1:3), b(1:3,1:3))
y = sum(x(1:2,1:6:2)) ;         y = product(x(1,:), 1, x(1,:)<15 )

where (x>=0)
     x=sqrt(x)                  pack(v1, v1(1:) <= v1(1) )
elsewhere
     x=sqrt(-x)                 qsort_reals(pack(data(2:), data(2:) > data(1)) )
end where
```

In the beginning, there was FORTRAN – **FOR**mula **TRAN**slator for the IBM 704 (1954).

Versions

Version	Note	Release
FORTRAN 66	First standardization by ASA (now ANSI)	1966-03-07
FORTRAN 77	Fixed Form, Historic	1978-04-15
Fortran 90	Free Form, ISO Standard, Array operations	1991-06-15
Fortran 95	Pure and Elemental Procedures	1997-06-15
Fortran 2003	Object Oriented Programming	2004-04-04
Fortran 2008	Co-Arrays	2010-09-10

Syntax, fixed format, free format came in f90

Fixed format:

Column 1 through 5 are for labels
A ‘C’ in the 1st column signify a comment
Any character in column 6 signify a continuation of the previous line
Columns 7-72 are for statements.

Columns from 73 to 80 are not significant.

Comments after column 72, often used to number cards

Blanks are insignificant hence ‘DO 15 I’ is a variable.

IBM

FORTRAN Coding Form

GX28-7327-6 U/M 050** Printed in U.S.A.

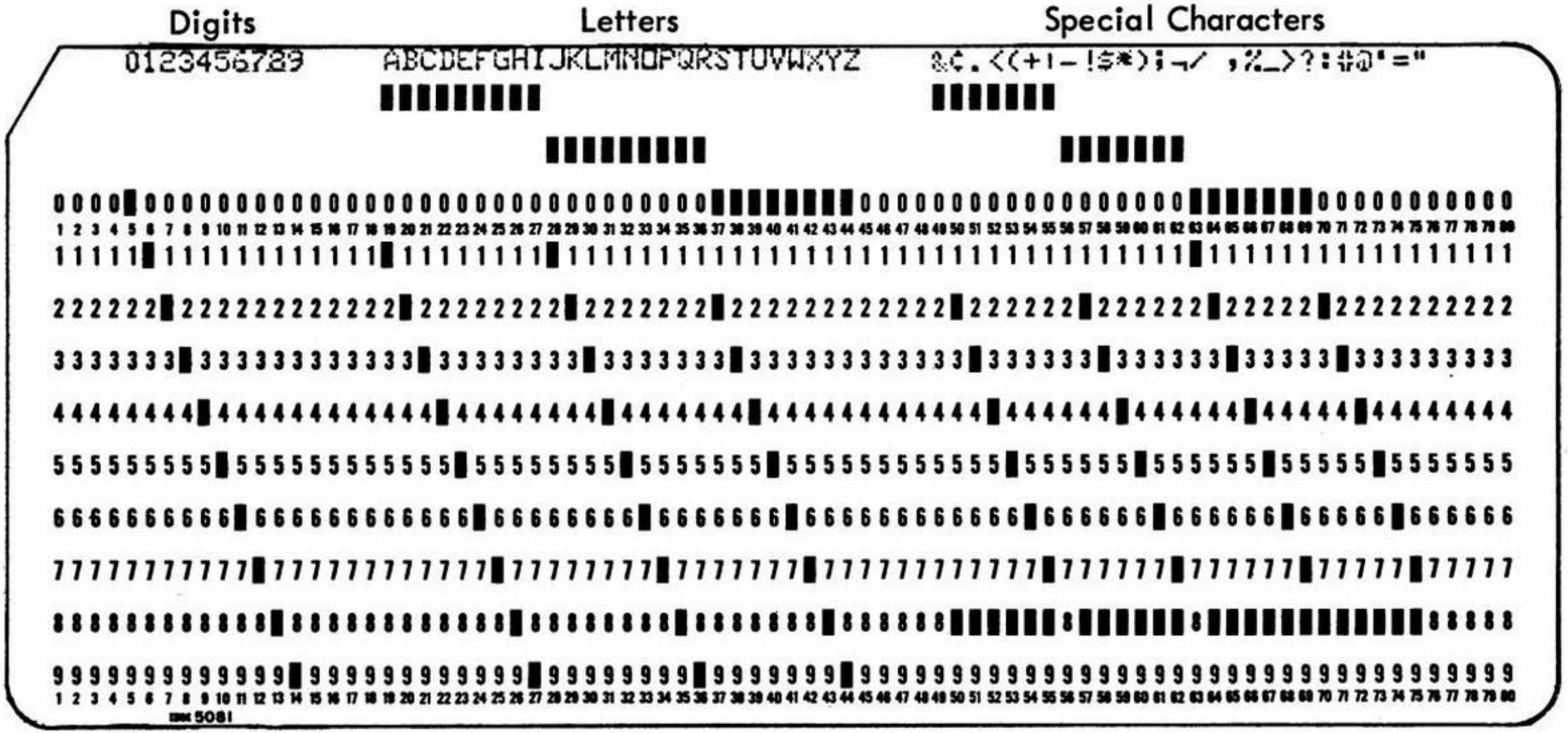
PROGRAM PHONBIL | PUNCHING INSTRUCTIONS | GRAPHIC | PAGE 3 OF 4

PROGRAMMER MPG | DATE 3/12/77 | PUNCH | CARD ELECTRO NUMBER*

COMMENT	STATEMENT NUMBER	CONT	FORTRAN STATEMENT	IDENTIFICATION SEQUENCE
			ccol(77),0"99V99") skip;	490
				500
			get file(Master) Edit(Master_card)(col(10),A(12),col(51),A(10))	510
		c	skip;	520
				530
			if (Telco_card.number_called<Telco_number_temp) then goto ERROR;	540
			if (Master_card.number_called<Master_number_temp) then goto ERROR;	550
			Telco_number_temp=Telco_card.number_called;	560
			Master_number_temp=Master_card.number;	570
			do while("1"b);	580
			do while (Telco_card.number_called=Master_card.number);	590
			put file(sysprint) Edit("****unsigned:",charge*(01.00+Tax),	606
		c	substr(number_called,1,3)!!"-"!!substr(number_called,4,3)!!	610
		c	"-"!!substr(number_called,7),substr(date,1,2)!!"/"!!substr	620
		c	(date,3),place_called)(A,F(4,2),X(4),A,X(2),A,X(2),A) skip;	630
			put file(unsign) Edit(charge*(01.00+Tax),substr(number_calle	640
		c	d,1,3)!!"-"!!substr(number_called,4,3)!!"-"!!substr(number_	650
		c	called,7),substr(date,1,2)!!"/"!!substr(date,3),place_called	660
		c	)(col(35),F(4,2),col(51),A,x(1),A,x(1),A) skip;	670
			get file(Telco) Edit(Telco_card)(A(9),col(20),A(10),col(60),	680
		c	A(10),col(77),0"99V99") skip;	690
			if (Telco_card.number_called<Telco_number_temp) then goto	700
		c	ERROR;	710
			Telco_number_temp=Telco_card.number_called;	720

*A standard card form, IBM electro 888157, is available for punching statements from this form.

**Number of forms per pad may vary slightly


Digits
Letters
Special Characters
0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
&¢.<(+!−!$*);¬/ ,%_>?:#@'="
IBM 5081

Hello world in fortran 90

This hello-world is the first program in the new language.

part1/hello.f90

Demo - try to compile and run it. Fortran syntax is easy to learn, it’s verbose, in some way self explanatory. We’ll go through the compile and link process using the command line.

Syntax, fixed format, free format came in f90

```
123456789012345670123456789012345678901234567890123456789012345678901234567890
0
      PROGRAM SYNTAX                                                  CARD 1
      DO 10 I=1, 10, 2                                                CARD 2
      PRINT *,I                                                       CARD 3
10    CONTINUE                                                        CARD 4
      PRINT *,'DONE'                                                  CARD 5
      END                                                             CARD 6
```

comments after column 72, often used to number cards

Blanks are insignificant hence DO15I is a variable.

The newer standards have superior flow control.

Variables - naming could be important

Names starting with I,J,K,L,M,N declared as integers, all other real.
Implicit real a-z was commonly used
Implicit none - introduced in fortran90, before one could use implicit logical a-z

implicit none - is ubiquitous today, gfortran support : `-fimplicit-none`

5 types of standard variables integer, real, complex, character and logical

Plus any number of derived types.

Control flow - labels - old style and bad programming

Labels have been with us since the beginning of Fortran in 1954.
Labels are numericals located in column 1 through 5.
They are fixed points at which execution can be directed to.

Some standards existed like two digits for loops, 3 digits for formats.
This all up to the programmer.

DO NOT USE LABELS IN 2023 (or in any other year after 1990  **)**

Control flow - labels - 1970s do loop style

```
12345678901234567012345678901234567890123456789012345678901234567890123456
90
      PROGRAM SYNTAX                                                  CARD 1
      DO 10 I=1, 10, 2                                                CARD 2
      PRINT *,I                                                       CARD 3
10    CONTINUE                                                        CARD 4
      PRINT *,'DONE'                                                  CARD 5
      END                                                             CARD 6
```

Control flow - jumps using GOTO and labels

Jumps to a certain label to alter control flow was common before.
Assembly (machine code instructions have only jumps).
It turned out to be a very bad practice with unreadable code.

GOTO STATEMENTS SHOULD BE PROHIBITED ! - no joke.

Older code uses these statements.

Control flow - machine code has only jumps - Fortran is high level !

```
MOV  AX, 00    ; Initializing AX to 0
MOV  BX, 00    ; Initializing BX to 0
MOV  CX, 01    ; Initializing CX to 1
L20:
ADD  AX, 01    ; Increment AX
ADD  BX, AX    ; Add AX to BX
SHL  CX, 1     ; Shift left CX, this in turn doubles the CX value
JMP  L20       ; Repeats the statements
```

Modern x86-64 assembly/machine code

hello in machine code, close to how it was back in 1954.

part1/hello.asm (or .nasm)
Demo - it should compile on most machines using x86-64 processors.

Maybe the only time in your life you'll see an assembly program :)

Control flow - GOTO and labels - arithmetic GOTO

```
12345678901234567012345678901234567890123456789012345678901234567890
      PROGRAM BADSYNTAX                                              CARD 1
      IF (I) 60,70,80                                                CARD 2
 60   PRINT *,'NEGATIVE',I                                           CARD 3
      GOTO 91                                                        CARD 4
 70   PRINT *,'NULL',I                                               CARD 5
      GOTO 91                                                        CARD 6
 80   PRINT *,'POSITIVE',I                                           CARD 7
 91   END                                                            CARD 8
```

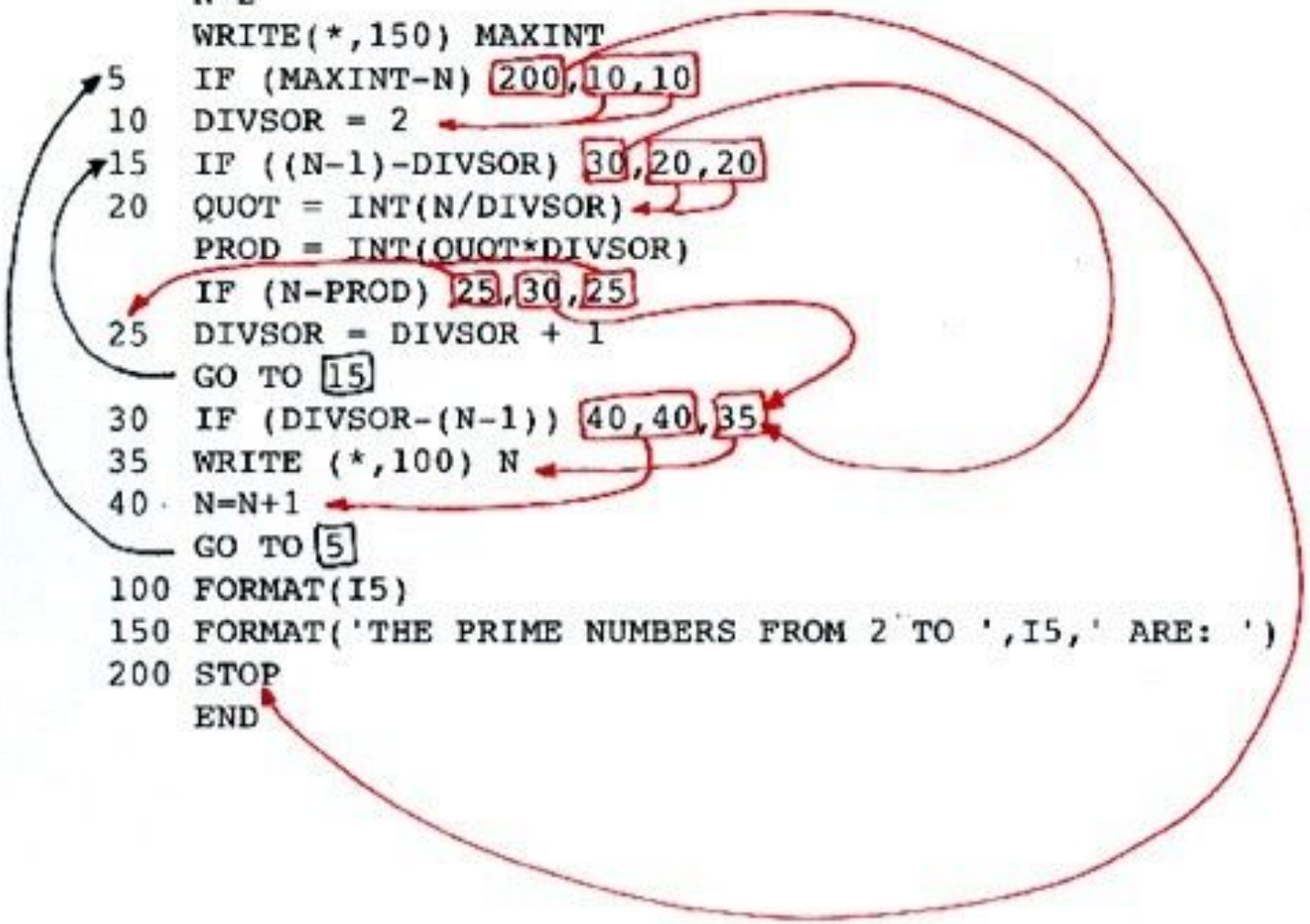
Control flow - GOTO and labels - computed GOTO

```
12345678901234567012345678901234567890123456789012345678901234567890
      PROGRAM BADSYNTAX                                                 CARD 1
100   FORMAT(/'GIVE AN INTEGER'/)                                       CARD 2
      WRITE(*,100)                                                      CARD 3
      READ(*,*) I                                                       CARD 4
      GOTO (30,40,50) I                                                 CARD 5
      PRINT *,'ONLY 1, 2 OR 3'                                          CARD 6
      GOTO 90                                                           CARD 7
 30   PRINT *,'ONE'                                                     CARD 8
      GOTO 90                                                           CARD 9
 40   PRINT *,'TWO'                                                     CARD 10
      GOTO 90                                                           CARD 11
 50   PRINT *,'THREE'                                                   CARD 12
 90   END
```

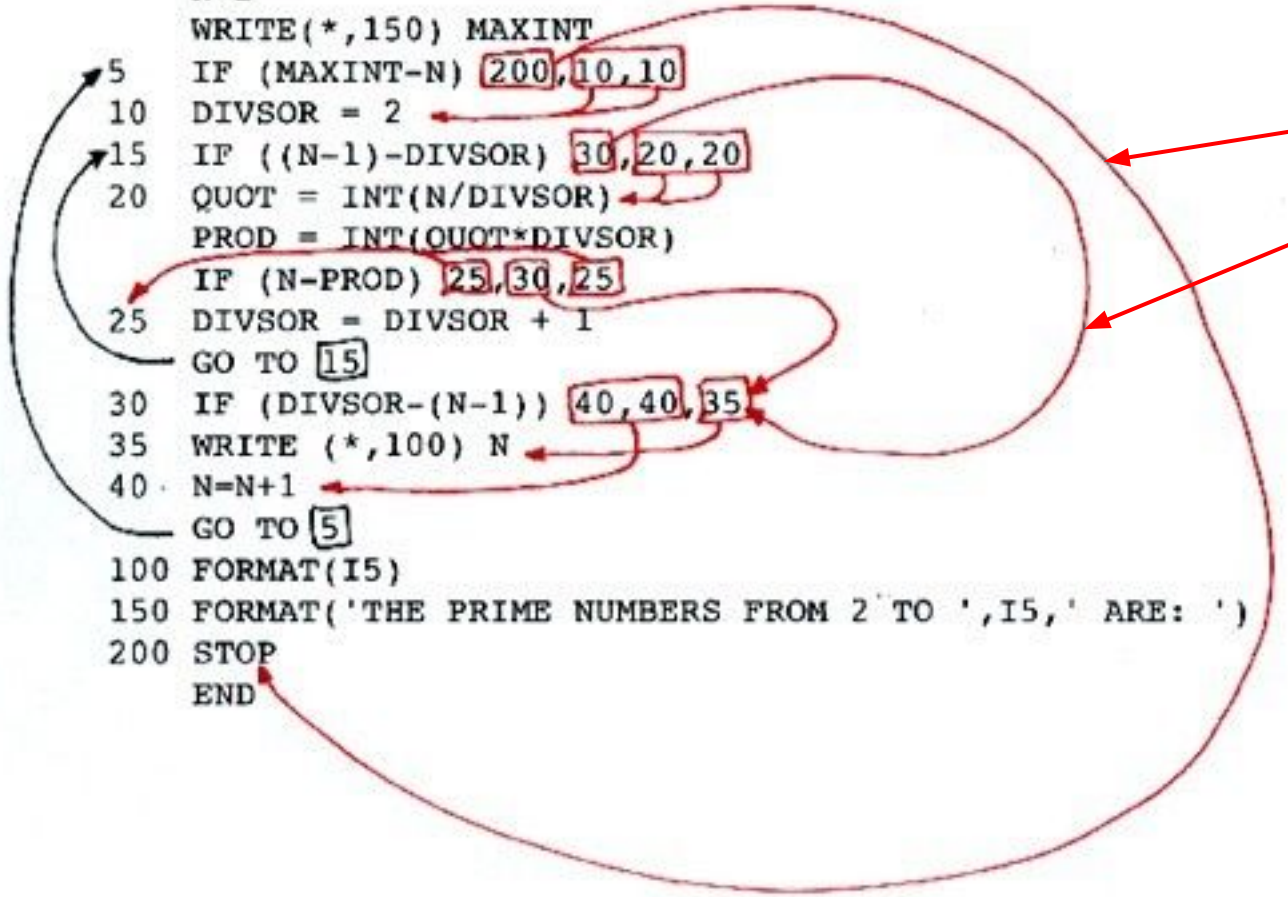
One of the classics of computer science is Edsger Dijkstra's "Go To Statement Considered Harmful", written in 1968. This missive argued that the GOTO statement (present in several languages at the time, including Fortran) was **too primitive for high-level programming languages**, and should be avoided.

https://degenerateconic.com/goto-still-considered-harmful.html
https://craftofcoding.wordpress.com/2020/02/07/is-goto-evil/

```
C       PROGRAM TO COMPUTE PRIME NUMBERS
C
        PROGRAM PRIME
        INTEGER MAXINT, N, DIVSOR
        INTEGER QUOT, PROD
        READ(*,100) MAXINT
        N=2
        WRITE(*,150) MAXINT
5       IF (MAXINT-N) 200,10,10
10      DIVSOR = 2
15      IF ((N-1)-DIVSOR) 30,20,20
20      QUOT = INT(N/DIVSOR)
        PROD = INT(QUOT*DIVSOR)
        IF (N-PROD) 25,30,25
25      DIVSOR = DIVSOR + 1
        GO TO 15
30      IF (DIVSOR-(N-1)) 40,40,35
35      WRITE (*,100) N
40      N=N+1
        GO TO 5
100     FORMAT(I5)
150     FORMAT('THE PRIME NUMBERS FROM 2 TO ',I5,' ARE: ')
200     STOP
        END
```



```
C       PROGRAM TO COMPUTE PRIME NUMBERS
C
        PROGRAM PRIME
        INTEGER MAXINT, N, DIVSOR
        INTEGER QUOT, PROD
        READ(*,100) MAXINT
        N=2
        WRITE(*,150) MAXINT
5       IF (MAXINT-N) 200,10,10
10      DIVSOR = 2
15      IF ((N-1)-DIVSOR) 30,20,20
20      QUOT = INT(N/DIVSOR)
        PROD = INT(QUOT*DIVSOR)
        IF (N-PROD) 25,30,25
25      DIVSOR = DIVSOR + 1
        GO TO 15
30      IF (DIVSOR-(N-1)) 40,40,35
35      WRITE (*,100) N
40      N=N+1
        GO TO 5
100     FORMAT(I5)
150     FORMAT('THE PRIME NUMBERS FROM 2 TO ',I5,' ARE: ')
200     STOP
        END
```



Such code is often called Spaghetti code

Would you like to review and maintain a million lines of such code ?

Spaghetti code - how bad can it be ?

Bad code should in most cases not be shown when teaching.

part1/gotos.f

Demo - would you like to work with such code ?

Variables - not so simple…

integer, real, complex, character and **logical**

Fortran standard makes no assumptions of hardware logic nor sizes, it care about science not computers.

Variables - not so simple…

integer, real, complex, character and **logical**

Fortran standard makes no assumptions of hardware logic nor sizes, it care about science not computers.

You, the scientist know what you need - review your task before starting to program !
Fortran is and have always been a science language.

Need to specify explicit (default are available)
(human or machine approach)
Note that all integers are signed, this is science not computer architecture/science.

Variables - Human or machine approach ?

Human:
Integer range (lowest to highest, number of digits)
Real/Complex (lowest to highest exponent and decimal places)

Machine:
Integer 8, 16, 32, 64 or 128 bit size (whatever range that is?)
Real/Complex 32 or 64 bits components (complex 64 and 128) what range (exponent) and number of decimal places is this ?

Both approaches are available and widely used.

Variables - Human or machine approach ?

Humans need some help:

print *, "maxint i4 ",huge(ii4)

print *, "Precision and range r4 ",precision(rr4), range(rr4)

(i4 is a 32 bit integer and r4 is a 32 bit/float/single precision variable)

Variables - human approach

What do you need ? (the human approach, range, precision/decimals and exponent range) :

Review your code and data needs, what variables do I need to hold my data ?

What is the outcome of any calculation ? What's kind of variable is needed ?

While Fortran only uses signed variables you still should review if negative answers from symmetrical solutions are valid or not. Same for complex solutions, are they valid for your calculation ? Make a plan for data management before beginning to write code.

Variables - human approach - paper & pencil exercise

What do you need ? (the human approach, range, precision/decimals and exponent range) :

integer, parameter :: i4 = selected_int_kind(r=9) - at least 9 digits
integer, parameter :: i8 = selected_int_kind(17) - at least 17 digits

integer, parameter :: r4 = selected_real_kind(p=6,r=20) - at least 6 decimal places, ± 20 exp
integer, parameter :: r8 = selected_real_kind(p=15,r=307) - at least 15 dec. places, ± 307 exp

The value of i4, i8, r4 and r8 are undefined by the standard, it's compiler specific !

Variables - human approach

```
integer(i8)     :: long_int
real(r8)        :: long_real
complex(r8)     :: long_complex
logical(i1)     :: short_logical
```

actually integer(kind=i8), we drop 'kind='

No need to use i8, longint is also used,
int is a function so int cannot be used.

constants : 2.0_r8 , 1.0_r4 , 1_i2 , 2_i4 , 1_i8

Variables - hardware approach

Use standard environment ,
use iso_fortran_env

integer(int64) :: long_int
real(real64)   :: long_real
complex(real64) :: long_complex
logical(int8) :: short_logical

How many digits do an integer16 have ?
How many decimals do a real32 have ?
What is the range of the exponent for a real64?

The hardware approach expects the user to have these numbers at hand for each individual system then use.

For most machines these numbers are standardised.

Variables - not as simple as before -hardware approach

The machine approach, size in bits, who knows what precision and exponent range are?
(VAX 11 had two 64 bit formats D_float: 16 decimals ± 38 exp, while G_float: 15 decimals ±308 exp).
Standards: IEEE 754-2019 and ISO/IEC 60559:2020 define it today

integer(kind=int64) :: long_int

2.0_real64 or 2_int64

integer*4 is portable in the sense it always means 4 bytes.
But on the other hand it is not portable because it has never been part of any standard. Programs using this notation are not valid Fortran 77, 90 or any other Fortran. Avoid any usage of <datatype>*N !! - rewrite !!

Declaration of variables

This file contain an example of declaring variables and shows sizes, ranges etc.

part1/variables_iso.f08
fortran standard 2008 code.

Demo

Derived types - declaration of variables user specified type

With the fortran 90 standard came user defined derived types.
Simple example:

```
type :: t_location
    real :: lat, lon
 end type t_location
```

The new type is t_location, just like integer, real, logical etc.

Derived types - demo

With the fortran 90 standard came user defined derived types.

part1/location.f90

The new type is t_location, just like integer, real, logical etc.

«Poor programming kills people»

stated at a Supercomputing tutorial on better programming

https://arstechnica.com/civis/threads/mariner-1-fortran-bug.862715/

```
program do10i
do 10 i = 1.10
print *,do 10 i
end
```

Mariner fault 1962 - simple mistake

Spot the error

part1/mariner.f

Demo - try to run it, it’s all about the details.

The importance of keeping control over data types and their ranges cannot be stressed more than pointing to two disasters :

Ariane-5 failure : Ariane flight V88 - Wikipedia

An even worse example, killing people, the Therac-25 incident Therac-25 - Wikipedia

The examples are just variable overflow examples…..

Overflow - what is 127 + 1

This exercise is showing examples of overflow

part1/overflow.f90

Demo - do you have control over the range and precision of your variables ?
Are you sure that you don’t make an out of range mistake ?

Formatting output

The most common format code letters are:

A - text string
D - double precision numbers, exponent notation
E - real numbers, exponent notation
F - real numbers, fixed point format
I - integer
X - horizontal skip (space)
/ - vertical skip (newline)

The format code F (and similarly D, E) has the general form F*w.d* where *w* is an integer constant denoting the field width and *d* is an integer constant denoting the number of decimal digits.

Formatting output (more on this later)

The most common format code letters are:

(A) - A is for text

(I3) - I is an integer, here I3 with up to 3 digits.

(F10.3) - F is for reals, in this case width of total 10 characters and 3 decimal places.

(E12.8) - Scientific notation decimals and exponent, single precision (32 bit), width 12 with 8 decimals

(D12.4) - Scientific notation decimals and exponent, double precision (64 bit), width 12 with 4 decimals

(3I4) - Three integers, each with a width of up to four digits.

(5(A,F3.0)) - 5 tuples of text and float/real with 3 places, and zero decimals, 2 digits and decimal.

Decision - test - flow control

if then else construction

```
if (test) statement

if (test) then
    statement
else
    statement
end if
```

Decision - test

```
logical :: p=.true.

if (p) then print *,’true’
if (a==b) write(*,'(a)') 'They are equal'


if (a>0) then
     x=1/a
else
     a=0
end if
```

Decisions - if-then-else

This exercise is showing how to make decisions.

part1/if.f90

Demo

Loops - do loop - the backbone of fortran

Syntax of do loop:

do variable = start, end, step ; where all must be integers.

Simple do loop :

```
do i = j, n, k
     print *,i
end do
```

Loops - do loop - cycle and exit

Syntax of do loop:

```
do
    print *,i
    i=i+1
    if (i>3) exit
end do
```

The statement exit finish execution of the loop.

Loops - do loop - cycle and exit

Syntax of do loop:

```
do
    i=i+1
    if (i<5) cycle
    if (i>9) exit
    print *, i
end do
```

Print only the last elements using cycle and exit.

Loops - do while loop

Syntax of do while loop:

```
do while (test)
        print *, ‘here’
        test = .false.
end do
```

Test can be a test like (a>0) or a logical. Cycle and exit as before.

Loops , fixed loops if decision loops

This exercise is showing how loops works in Fortran

part1/loops.f90 , part1/loop2.f90 , part1/control-flow.f90

Demo - loops and different control flows.

Decision - case

Syntax of the case construct.

```
do i=1, 10
  select case (j)                ! j must be integer, character or logical
     case(1)                     ! Case must be a constant, or a parameter variable.
           print *, "One"
         case(2)
        print *, "Two"
     case default
           if (j > 4) exit ! Exit the do loop, not the case.
           print *, "Many"
   end select
end do
```

Decision - case

Syntax of the case construct.

```
character :: c

select case(c)
     case ('0':'9')              ! A range is allowed, integers and characters.
       print *, 'Numbers !'
end select
```

Case - select from multiple of cases

This exercise is showing how to select from multiple options.

part1/case.f90

Demo - select from a range of options with different control flows.

Input / Output - format

```
print ‘(a)’, ‘hello world’

unit=6 for terminal
format = ‘(a)’

write(unit,fmt=format) ‘Hello world‘
```

Input / Output - terminal

```
print *, ‘hello world’
print ‘(a)’, ‘hello world’

write(*,’(A)’) ‘Hello world‘

form=’(i3,x,a,x,f5.2)’
write(*,form) 127,’is int’,3.14
```

Variables supplied MUST match the format,
no compile error, run time error.

Formatting output

The most common format code letters are:

(A) - A is for text, character string

(I3) - I is for integer, here I3 with up to 3 digits, i6.4 yield leading zeros.

(F10.3) - F is for reals, in this case width of total 10 characters and 3 decimal places.

(E12.8) - Scientific notation decimals and exponent, width 12 with 8 decimals.

(en10.2) - Engineering notation decimals and exponent, width 10 with 2 decimal places,.

(es8.1) - Scientific notation, decimals and exponent, width 8 with 2 decimal places.

(3I4) - Three integers, each with a width of up to four digits.

(5(a,f3.0)) - 5 tuples of text and float/real with 3 places, and zero decimals, 2 digits and decimal.

(G12) - Scientific format decimal number of exponent format depending on value, humal format.

(L) - Logical

Formatting output

The most common special codes are:

(x) - Emit a space

(/) - Emit a linefeed

(sp) - Numeric, print ‘+’ for positive numbers

(dc) - Decimal symbol, can be comma ‘,’ or period ‘.’

Format string

Format statements can be kept in a string.

```
form='(a)'                  ! One character or a string
form='(a,2x,i3,i3,f4.2)' ! One string, two spaces, one integer, one integer, one real.
form='(a, "," i3 "," i3 "," f4.2 "," e8.2 "," g10.3)'! string, comma, one integer, comma,
      one integer, comma, one real, comma, one real, comma, one real
form='(10(ai3))' ! Ten tupples of string + integer
```

print format, <variables>
write(*,format) <variables>

Format string - internal file

Format statements can be kept in a string, this is called internal file.

A string can be written to :
write(string,’(a)’) ‘Hello world’

string will now contain the text, formats can also be written to format strings.

Demos on formatting

This demo explore formatting.

part2/format.f90
part2/format2.f90

Introduction to implied do loop, (i, i=1,10) a powerful tool.

implied-do-loop.f90

Working with stored files - Input / Output

Important concepts :
Unit number, related to 5 and 6 for keyboard and terminal.

Files can be assigned to a unit, keyword *newunit* assign an unsed number.

```
open(newunit=myunit,file='myfile.txt')
read(myunit,*) a
```

Working with stored files - Input / Output

Binary output, data structures can be written.

```
real(real64), dimension(100,100) :: x
open(myunit,file='myfile.txt', form='unformatted')  ! Binary 'raw', written as stored.
write(myunit) x
```

Demos on input/output - I/O

This demo explore file I/O files in part2.

formatted, ascii text format:
fileio_w.f90 and fileio_r.f90

Binary file format:
fileio2_w.f90 and fileio2_r.f90

Vector syntax - declaration of variables

With the fortran 90 standard came vector syntax and more strict declarations of indexed variables of often referred to as vectors (of multidimensions).

The keyword is dimension

dimension(n1,n2,n3,...,n15) up to 15 dimensions (2008 std).

Vector syntax - declaration of variables

Declare vectors of different dimensions :

```
integer(int8), dimension(100) :: k
logical(int8), dimension(5) :: ok
real(real64), dimension(10,10) :: x
real(real32), dimension(10,10,10) :: y
complex(real64), dimension(10,10,10) : z
complex(real32), dimension(5,5,5,5) :: w = (1.4_real32, 1.5_real32)
```

Vector syntax - no do loops

We can operate on vectors, matrices and higher dimensional (up to 15) arrays as if they were scalars. This is element by element by element operations.

```
A = 0          ! all elements in A is set to zero
A = B + C      ! All elements in A are assigned to the sum of B + C
B = B*10       ! Elements of B are multiplied by ten
C = C**2       ! Elements of C are squared
```

Vector syntax -no do loops

This is element by element by element operations. Important - remember.

Assume A,B & C are arrays - then C = A+B is equivalent to

```
do j=1,n
  C(j)=A(j)+B(j)
```

As you need one do loop per dimension, operations on 3d or higher dimensions become easier and easier.

While C=A+B is vector syntax.
A*B is different from matrix-matrix multiplication, element by element.

Vector syntax - no do loops anymore (almost)

Vectors of any dimension

assigning	A=1.0
extracting	A(1:2)
slicing	A(1:2:2)
shaping	A(1:4)
reshaping	A(1:4) => A(2,2)
masking	L = A<0

Demos on vector syntax

This demo explore vectors, files in part2:

vectors1.f90 , vectors2.f90, vectors3.f90, vectors4.f90

Advanced usage, functions this is homework and exercise
slicing.f90 , pack.f90
A challenge is the qsort.F90 code, a simple quicksort using vector syntax.

vector / array operations

We can operate on vectors, matrices and higher dimensional structures.

packing
unpacking
masking
shifting
rotating
locate (locate a value in a vector)

modern fortran:

```
mask =  v1 /= 0
```

F77 style (n must be known):

```
do j=1,n
if (v1(j) .ne. 0) then
  mask(j)=.true.
else
  mask(j)=.false.
end
```

Vector / array operations

We can operate on vectors, matrices and higher dimensional structures.

shape
reshape
pack

Change shape of vectors of any dimension.

Change a N*N*N long 1-d vector to a 3-d N x N x N cube,
or to a 2-d N*N x N matrix.

or pack, mask and reshape together.

Demos on vector syntax

This demo explore vectors, files in part2

Advanced usage, functions this is homework and exercise
slicing.f90 , pack.f90
A challenge is the qsort.F90 code, a simple quicksort using vector syntax.

Functions and routines

Functions and subroutines.

Functions are well known, they return a value.

Subroutines are called, exchange data via parameters

In fortran everything is by reference. Any variable is an address (e.g. like a pointer to a chunk of memory).

Functions

Functions return something,

a = func(x) where a gets assigned to the result of the function

Like s = sqrt(2.0) , t = exp(1.0) etc.

The input parameter should not change, not be updated, or any other values should be changed. These are called side effects and can cause issues and errors.

Intrinsic routines

Built in routines, functions and subroutines:

https://gcc.gnu.org/onlinedocs/gfortran/Intrinsic-Procedures.html

A large selection of useful routines.

Be careful with the GNU extensions.

Pure functions

Pure functions have no side effects !

Any assignments of parameters, allocation or I/O are not allowed.

allocation could cause the system to run out of memory

I/O could in principle fill up the storage.

Or other effects the compiler will flag as impure.

Subroutines

Subroutines are called, no return value.

Subroutines exchange data via parameters

In fortran everything is by reference. Any variable is an address (e.g. like a pointer to a chunk of memory).

Subroutines

call name(a,b,c)

Calling subroutine with 3 parameters, which can be:

a) in
b) out
c) inout

It also possible to have optional parameters.

Pure subroutines

Pure subroutines have no side effects !

Any assignments of input parameters, allocation or I/O are not allowed.

allocation could cause the system to run out of memory

I/O could in principle fill up the storage

Or other effects the compiler will flag as impure.

Interface to functions and subroutines

The compiler need to know how to call a routine.

Without the interface there is no way the compiler can make sure the calling of a routine is correct. You might send a long integer (64 bit) to an unsuspecting receiver (routine) who expect a long real (64 bit).

Or an index to an array which might be far off and cause a segment violation and abort.

If possible use an interface, it’s there to guide and provide safer programs.

Interface to functions and subroutines

The compiler need to know how to call a routine.
Number, size and type of parameters. A template of just the calling interface:

```
interface
  pure subroutine multwo(a,b,c)
     use iso_fortran_env
     implicit none
     integer(int8), intent(in) :: a,b
     integer(int8), intent(out) :: c
  end subroutine multwo
 end interface
```

Demos functions and subroutines

This demo show how fortran call by reference, and how pure works, and introduce the concept of interface. Files found in part4:

functions1.f90
through functions4.f9 , subroutines1.f90 and subroutines2.f90

Contiguous layout : subroutines3.f90

Functions calling themselves

Functions can call itself, this is called recursive functions.

In Fortran the default is not to allocate new memory when a function is called the second time. Old values will keep their value.

The keyword recursive must be used.

Typical example is the Fibonacci number generator :

fib(n) = fib(n-1) + fib(n-2)

Demos functions and subroutines

This demo show recursive functions.

part4/functions5.f90

Simple Fibonacci number sequence generator.

Elemental Functions

Elemental functions operate on element by element.

It can take input of various length and ranks.

Operation is done element by element on the arrays given.

Example, a sum : s=a(1,1)+b(1,1) + a(2,1)+b(2,1) etc etc.

arguments can have any higher ranks.

like example addint([1,2,3], [4,5,6]) would yield 1+4, 2+5, 3+6

Take any ranks and dimension!

Demos functions and subroutines

This demo show elemental functions.

part4/functions6.f90

Dynamic memory allocation

Dynamic memory allocation allows programs to be more flexible and adaptable. It allows the allocation and deallocation of memory as needed during runtime, enabling the program to handle varying amounts of data.

Efficient memory management: Dynamic memory allocation provides more efficient memory management compared to static memory allocation. With dynamic allocation, memory can be allocated and deallocated on-demand, reducing wasted memory space.

Dynamic memory allocation

Performance is comparable when using static or dynamic allocation.

Performance is not a big issue when deciding on static or dynamic allocation.

Dynamic memory allocation

However,

- Allocation takes time, same for static or dynamic, Linux is lazy so static memory is allocated by the OS as it’s written to (init or written).

- if you allocate a large chunk, then deallocate and allocate again it will impact performance.

- If your structure is static during a run you might keep the variables allocated for the duration of the run (static or dynamic)

Dynamic memory allocation

Static memory allocation :

Stores static variables

Static Memory Allocation is done before program execution.

In Static Memory Allocation, there is no memory re-usability

In this allocated memory remains from start to end of the program.

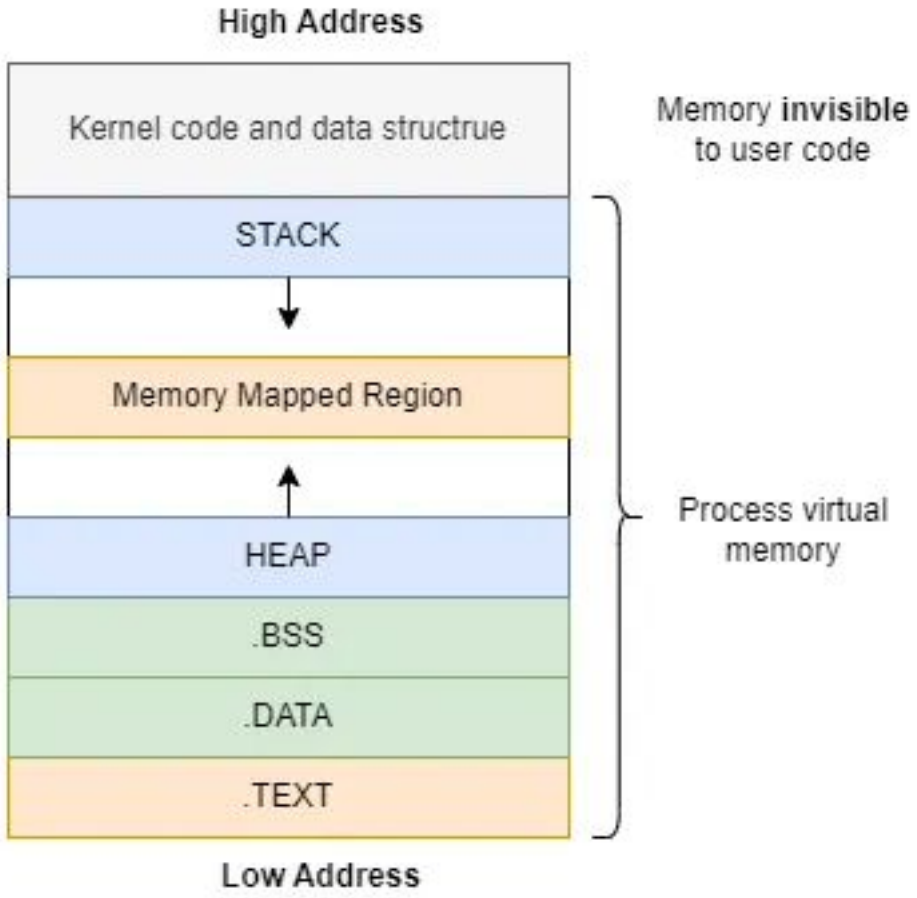
Dynamic Memory allocation :

Stores dynamic variables

Dynamic Memory Allocation is done during program execution.

In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required

In this allocated memory can be released at any time during the program.



Your memory space start at 0 and extend linearly to N.

Behind the scenes the OS and memory management system fragment and place it at various places (even write it out to storage).

https://www.linux.com/training-tutorials/stack-vs-heap-whats-difference-and-why-should-i-care/

Dynamic memory allocation

Practical issues, too small stack size, run time error abort program

solution (some times) : ulimit -s unlimited

Check if already set : ulimit -a

Dynamic memory allocation

Important statements, dimension is unknown, rank must be known :

```
real(real64), allocatable, dimension(:) :: x

allocate(x(size))
```

Only difference is that x is allocated, use as before, no change.

Dynamic memory allocation

Important statements, dimension is unknown, rank must be known :

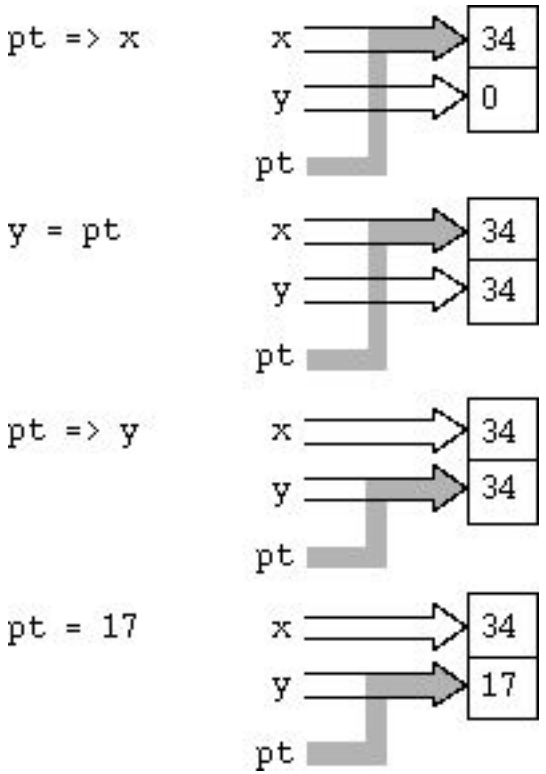
```
real(real64), allocatable,  target, dimension (:) :: x

real(real64), pointer, dimension(:) :: pt

allocate(x(size)) or allocate(pt(size))
```

The pointer can now point to vector x. Only difference with x is that's a possible target for a pointer.

Pointers



pointers are strongly typed

- Data type
- Rank of array

real(real64), pointer, dimension(:) :: pt

Demos on dynamic memory allocation

This demo show dynamic allocation and pointer.

part3/alloc2.f90
part3/alloc3.f90

Dynamic memory allocation

Important statements, dimension is unknown, rank must be known :

allocate(z, source=x)

allocate(z, mold=pt)

allocate(z(vectorsize), source=3.14_real64)

Source make a new vector identical to source with content of source.

Mold make a new vector identical to source without any copy of data.

Demos on dynamic memory allocation

This demo show dynamic allocation mold and source, using a template.

part4/alloc4.f90

Accelerated code - GPU accelerators

Use only ISO standard Fortran code

```
do concurrent(j = 1:n)
      do l = 1,n
      do i = 1,n
            z(i,j) = z(i,j) + x(i,l)*y(l,j)
      enddo
      enddo
  end do
```

The do concurrent construct is ISO Fortran standard for parallel execution.

Demos with acceleration

This demo show accelerated matrix matrix multiplication, files in part5.

module load NVHPC/23.1-CUDA-12.0.0

Need NVIDIA Fortran compiler and : -gpu=cc60 -stdpar=gpu

This demo need a node with GPU.

draft of the fortran 2018 standard are available at :

std-2018-007r1.pdf

Useful links include :

Fortran Wiki

Fortran-lang.org

Fortran Tutorial

The GNU Fortran Compiler

https://wg5-fortran.org/N2201-N2250/N2212.pdf (whats'n new in 2023 standard)