Structured programming

- top-down programming
- stepwise refinement
- restricted control structures
- bottom up programming (Naur: action clusters),
  idea of a virtual machine: split the program into layers, each of which is programmed using the tools of lower levels (cf. Unix)

Top down method

1. Define exactly what data the program will get and what it has to do with them.
2. If the task is simple enough, write the program code.
3. Otherwise, split the task into smaller parts and define exactly the duty of each part and interface to the rest of the program.
4. Repeat the steps 1–4 separately for each subtask.

Properties of a good subroutine/function

Internal strength:
Executes one clearly defined task.
The task may not be simple, but can defined in a simple manner.

Splitting the code just to avoid repetition or to keep subprograms small may lead to poor internal strength.

External connections:
The more the subprogram depends on global variables and other subprograms the more difficult it will be to modify.

Ideal case: all data transmitted in parameters (cf. sin, sqrt).

Efficiency

1) Select the proper algorithm. Bad programming can ruin even a good method. Each method has its restrictions that cannot be exceeded even by efficient programming.

Select the method so that it is efficient for typical problems. In big problems an $N \log N$ method is much more efficient than an N^2 method.

2) Calculate only the necessary. E.g. sets of linear equations are often solved by computing the inverse of the coefficient matrix although it is not needed. If the matrix is big, a lot of computer time is wasted.

3) Don't reinvent the wheel. Subroutine libraries contain many good routines for most numerical probles. If you have access to them try them first.

4) Assumptions related to the problem. You may not have to solve the most general problem but a simpler special case. E.g. for solving sets of linear equations there are many different methods for different kinds of coefficient matrices.

5) Don't just try to polish bad code. Rewrite it.

6) If the execution will take a long time, store every now and then the intermediate results. If the executions is terminated for some reason (power failure etc.), the whole work is not lost, but can be continued from the last checkpoint.

Locality of memory references

In virtual memory computers the program is usually divided into pages of a fixed size and stored on a disk. Pages are loaded to the central memory as needed. When a program refers to a page that has to be read from disk, a page fault interruption takes place and the execution of the program is suspended. When the memory becomes full pages that have not been needed for a long time will be moved back to disk.

```
do i=1,100000
  x=a(i)
   ...
end do
```

In most cases the referenced element is on the same page as the previous one, and the page needed is alredy in memory with a high probability.

If a program refers to elements that are far away from each others the referenced elements are often on different pages. If the array is very big, previously referenced pages may have already been returned to disk, and in the worst case each array reference will cause a page fault interrupt.

Locality of memory references means that successive references point to memory addresses that are close to each others. Locality improves the efficiency of the virtual memory; it is also useful in optimising the usage of the cache memory.

In some number crunching machines there is no virtual memory (some models of Cray). Still the memory architecture may require certain kind of coding to get optimum performance.

Locality of memory references is always advantegous!

Problems may arise with multidimensional arrays. In Fortran arays are stored columnwise.

```
do i=1,1000
  do j=1,1000
    x=a(i,j)
    ...
  end do
end do
```

In the inner loop the distance of successive array references is 1000 words. Thus the values are on different pages.

The program is more efficient if the order of the loops is interchanged:

```
do j=1,1000
  do i=1,1000
    x=a(i,j)
    ...
  end do
end do
```

Miscellaneous ideas about programming style

A program should do one thing properly instead of doing many things somehow. Before starting to write a program, it is essential to define precisely what it has to do.

Whatever the method, the idea of structured programming is to split the task into smalle, easily manageable and relatively simple modules.

A procesure should be understandable as a unit. The more complex control structures it contains the shorter it should be.

Each procedure will accomplish one precisely defined task. The action does not have to be simple, but it has to definable in a simple way, like "sort a file", "solve a partial differential equation satisfying given boundary conditions", "prove Glodbach's conjecture". One procedure should not contain operations that do not form a logical unity.

Procedures should be as independent as possible. Global variables should be used sparingly. If a procedure has very many arguments, something might be wrong.

Properties of a good program

As in any art, no unique definition can be given.

1. Device independence
   - presentation of variables
   - machine dependent I/O
2. Arithmetic accuracy
3. Completeness
   - no missing parts
   - checking input data (form, sensibility)
   - checking of array indices
   - functionality in all cases with all inputs

4. Consistency
    - external (usage, restrictions, ...)
    - internal (variable names, types, constants, ...)
5. Efficiency
6. Accessibility
    - how to change properties of the program (input form, accuracy, constants, ...)
7. Communication
    - user interface (often neglected!)
    - form of input data
    - clarity of output
    - error messages
8. Structuring
    - modularity
    - definition of modules

9. Self-explanatory
    - e.g. descriptions of functions and interfaces of modules
    - intuitive variable names
10. Compactness
    - no unnecessary variables
    - no inaccessible code
    - no unnecessary addresses
    - often repeating operations programmed as procedures
    - constant code moved outside loops
11. Readability
    - typography
    - variable declarations
    - comments
12. Extensibility

Typography

The following Fortran 77 program is perfectly legitimate:

```
      DO100I=1,10
      IF(X(I).GT.0.0.AND.X(I).LT.1.0)THEN
      Y(I)=SQRT(X(I))+123.4*SIN(X(I))+(A-1.0)*SQRT(
     11.0-X(I))
      ELSE
      Y(I)=0.0
      ENDIF
  100 CONTINUE
```

A more readable version could be

```
      do 100 i=1,10
         if(x(i) .gt. 0.0 .and. x(i) .lt. 1.0) then
            y(i)=sqrt(x(i))+123.4*sin(x(i))+
     .           (a-1.0)*sqrt(1.0-x(i))
         else
            y(i)=0.0
         end if
  100 continue
```

A F90 solution might be

```
y(1:10)=0
do i=1,10
   if(x(i) > 0.0 .and. x(i) < 1.0) then
      y(i)=sqrt(x(i))+123.4*sin(x(i))+ &
           (a-1.0)*sqrt(1.0-x(i))
end do
```

or even

```
y=0
where (x > 0.0 .and. x < 1.0) &
  y=sqrt(x)+123.4*sin(x)+(a-1.0)*sqrt(1.0-x)
```

Even proper typography can improve the program:

- Use indentations to show control structures; thus it is easier to follow the cotnrol flow.
- In Fortran 77 spaces are ignored. However, it is better to use them as in ordinary text; to separate syntacctic items, but not inside identifiers.
- Using too much uppercase letters makes the text difficult to read.
- Split long and complicated expressions to several lines at logical breakpoints.
- In F77 the character indicating a continuation line should be chosen in such a way that it is clearly distinct from the actual program code. In this respect the style of F90 is much better.

Avoid tab characters, since they may have different effects on different devices, and the output may look confusing.

Comments

Comments are neede to help the work of the people that will have to read or modify the program. And you are one of them.

Each program file should begin with a description of the purpose of the code in that file. While developing programs it is not uncommon to create files with such descriptive names as `a`, `test`, `prog` or `xx`. After a few days even the programmer cannot remember what they are supposed to do.

If other modules must be linked to the program, the initial comments should give instructions for compiling and linking the program. Explanation of possible data files needed by the program is also useful information.

Don't explain a messy program but rewrite it.

Comments must correspond to the program code

```
if (x*y .le. 0.0) ... ! negative product
```

Such errors arise when the program code is modified but comments are left as they were.

Comments shouls give essential information:

```
i=i+1  ! add one to i
```

This does not help to understand the program any better. In another form the comment might contain useful information:

```
i=i+1  ! move to the next line of the matrix
```

Usually it is not necessary to explain each individual statement, It is better to concentrate on larger pieces of code, like procedures and control structures as a whole.

At the beginning of each procedure there should be a brief explanation: purpose of the procedure, its arguments, possible global variables, side effects, and the value returned by a function.

A good practice is to comment variable declarations (the essential ones, not all temporary auciliary variables and loop indices).

```
c-----------------------------------------------
c solve the n degree equation x**n-1=0
c the solution is returned in vector z(1), ..., z(n)
c-----------------------------------------------
subroutine solven(n,z)
  implicit none
  integer, intent(in):: n        ! degree of the equation
  complex, intent(out):: z(:)    ! solution vector
  real, parameter:: pi=3.141592654
  real phi
  integer k
phi=2*pi/n
do k=0,n-1
  z(k+1)=cmplx(cos(k*phi),sin(k*phi))
enddo
return
end
```

Variable declarations

The purpose of implicit declaerations is to reduce the work needed to write a program (and number iof punched cards). The price paid is in no proportion to the relatively small savings, when a lot of time is wasted for tracking errors for the strange behaviour of the program.

What the following program will do?

```
      NZERO=0
      DO 10 I=1.5
       IF (NUM(I).EQ.0) NZERO=NZERO+1
   10  CONTINUE
```

The program contains two errors, which are avoided if the implicit declarations are forbidden.

Most compilers accept the definition `implicit none`, but it does not belong to the F77 standard.

Variable names should be meaningful. In some environments the names of global identifiers (names of procedure and `common` areas) must be shorter than names of local variables, due to properties of the linker that must be able to link together modules written in different languages.

The `dimension` declaration is useless. Instead of

```
dimension x(10)
```

the same thing cna be expressed in a shorter and clearer manner:

```
real x(10)
```

In Fortran 90, it would be better to use consistently the form käyttää määrittelyä

```
real, dimension(10) :: x
```

Control structures

Due to indentations lines will move further right. If there are many indentations the space on the screen will become short and finding the corresponding starting and closing lines of a control structure will become harder. This is an indication that something in the program design might be wrong. Maybe the module should be split into smaller procedures or redesing the whole structure.

Do-statements are special cases of an $n + \frac{1}{2}$ loop. Very often the termination condition cannot be easily tested at ehe eginning (or end) of the loop.

```
    do i=0,n
      ..
      if (x(i) > 0) goto 100
        ...
    end do
100 continue

    do i=0,n
      ..
      if (x(i) > 0) then
        ...
      end if
    end do
```

In F90 this can be written in a more readable form

```
do i=0,n
   ..
   if (x(i) <= 0) exit
     ...
end do
```

Goto statements

In assembly languages go to statements are usually the only way to change the normal sequential control flow of statements. Also, in the very first versions of Fortran it was practically the only control structure. Thus they are still abundant in old programs.

A careful program design often leads naturally to a program where go to statements are not needed.

`goto` can transfer control to anywhere; to the reader of the program code it is not obvious where the execution will continue before the corresponding label is found.

Each `goto` requires a matching address. In fact the address is even worse than the go to statement, since it does not tell wherefrom the control is transferred.

```
      if (x .gt. 0.0) goto 200
      goto 100
200   y=sqrt(x)
100   ...
```

A simpler and more readable solution is

```
      if (x .gt. 0.0) y=sqrt(x)
```

What does the following program do?

```
        if (a .gt. b) then
        big=a
        goto 100
        end if
        big=b
100     if (big .gt. c) goto 200
        big=c
200     ...
```

The program could be written in a more readable form

```
      big=a
      if (b .gt. big) big=b
      if (c .gt. big) big=c
```

This will reveal that actually it suffices to say

```
big=max(a,b,c)
```

```
681  FORMAT( .. )
     DO 7813 LL=1,NTOT
     IF(LL.LE.N)WRITE(6,781)LL,XLOW(LL),XBE(LL),XUPP(LL)
     IF(LL.LE.N)WRITE(3,781)LL,XLOW(LL),XBE(LL),XUPP(LL)
     IF(LL.GT.N)WRITE(6,782)LL,XBE(LL)
     IF(LL.GT.N)WRITE(3,782)LL,XBE(LL)
781  FORMAT( .. )
782  FORMAT( .. )
7813 CONTINUE
     WRITE(6,7812)
7812 FORMAT( .. )
 78  CONTINUE
600  CONTINUE
 60  CONTINUE
999  KNC=0
     ...
     IF (N.NE.0) GOTO 100
100  CONTINUE
```

The previous fraction is from a real program.

The program contains many nested loops that end with `continue` statements. It is, however, difficult to find where the loops begin. The addresses are not in any order, and therefore it is hard to see which way the control is transferred.

Particularly confusing are the arithmetic `if` statements and assigned `go to` statements. In Fortran 90 standard these were marked as obsolescent features, and will not be accepted in later versions of the language.

The following example is from the book Kernighan, Plauger: *Elements of programming style*:

```
 1   SUBROUTINE MERGE(P,Q,R,S,N)
 2   DIMENSION P(N), Q(N), R(N), S(N)
 3   LP=1
 4   LQ=1
 5   LR=1
 6   LS=1
 7   CALL ORDER(P,Q,R,LP,LQ,LR,N)
 8   IF (P(LP)) 10,9,10
 9   IF (Q(LQ)) 10,13,10
10    CALL ORDER(P,Q,S,LP,LQ,LS,N)
11    IF (P(LP)) 7,12,7
12   IF (Q(LQ)) 7,13,7
13   RETURN
14   END
```

Clever solutions

A clever way:

```
real x(100,100)
do i=1,n
do j=1,n
 x(j,i)=(i/j)*(j/i)
end do
end do
```

An efficient way:

```
real x(100,100), y(10000)
equivalence (x(1,1), y(1))
do i=1,10000
 y(i)=0.0
end do
do i=1,10000,101
 y(i)=1.0
end do
```

The natural solution:

```
real x(100,100)
do i=1,100
 do j=1,100
  x(i,j)=0.0
 end do
 x(i,i)=1.0
end do
```

In Fortran 90:

```
real, dimension (100,100) :: x
integer i
x = 0.0
do i=1,100
 x(i,i)=1.0
end do
```

Side effects

Functional program modules are functions that return the value of the calculation. They have no side effects: they do not modify their argumets or gloabl variables or do any I/O operations. Using such functions is safe.

Subroutines are based solely on their side effects. Since they do not return any value, they must do something else in order to have any effect at all.

Also functions with side effects are potentially dangerous.

```
real function f(x)
x=x+1.0
f=x**2
return
end
```

The function could be called in the following way

```
x=1.0
z=f(x)+f(x)
if (f(x).gt.z .or. f(x).lt.0.0) z=x
```

What is the value of `z` now?

Does the following optimised version give the same result?

```
x=1.0
y=f(x)
z=2*y
if (y.gt.z .or. y.lt.0.0) z=x
```

And what is the output of this program?

```
call s(1)
write(*,*) 1
end
subroutine s(i)
i=i+1
return
end
```

Remember that in Fortran all parameters are refenrence parameters, i.e. the procedure is given the address of the actual parameter.

Input and output

The user must not be asked to provide information that the program can calculate.

The input format should be flexible and as simple as possible. However, fixed format can be used for big tables, which are probably produced by some other program.

In output it is usually better to use fixed format, since tables are more readable if the columns are properly aligned.

The output must be unserstandable without reading the document of the program, or (even worse) the progrm code.

Test that input values are meaningful, give messages for erroneous values, and try to recover if possible. Test unexpected end-of-file situations.

Don't scatter I/O statement everywhere in the code, but keep them in a small number of procedure.

If possible, use reasonable default values for missing data.

Avoid `format` statements. Formats used only in a few places can be written as a constant character string in the I/O statement; it is then easier to compare the format and the correspondign variable list. More often used formats can be defined as character string variables.

End the input data with a proper information indicating the end, end of file etc. Don't ask the number of input:

```
      do i=1,1000
        read (1,*,END=100) z
          x(i)=z
       end do
  100  continue
```

Miscellanea

- Put machine dependent parts in separate procedures

- Select data structures carefully

- Let the computer do the "dirty work"

```
I='41'O
I=ICHAR('A')
```

- Avoid unnecessary auxiliary variables

```
xx = x0-x1
yy = y0-x1
z = xx**2+yy**2

z = (x0-x1)**2+(y0-y1)**2
```

- Test limiting cases. Be careful with array indices and "off by one" cases of loop counters. Check that the program works correctly when the variables get special (usually limiting) values.

- Initialize all variables. (Exception: large arrays. Usually their area is allocated page by page as needed; unused area is not allocates. Zeroing the array will require allocating the whole area.)

- Be careful with rounding errors: 10.0×0.1 is hardly ever 1.0! Don't compare equality of real numbers.