Good FORTRAN Programs

Nick West

What is a Good FORTRAN Program?

- It Works
  - May be ~ impossible to prove e.g. Operating system.
- Robust
  - Can handle bad data e.g. I/O error or unfriendly user!
- User Friendly
  - Not user hostile!
  - Questions - clear, offer help, diagnose bad replies.
  - Multi-path? Allow backoff.
  - No unnerving silence!
  - Restartable.
- Maintainable
  - Environment dynamic!
  - Well structured - minor changes - easy to find & fix and are local.

What is a GOOD FORTRAN Program? (cont.)

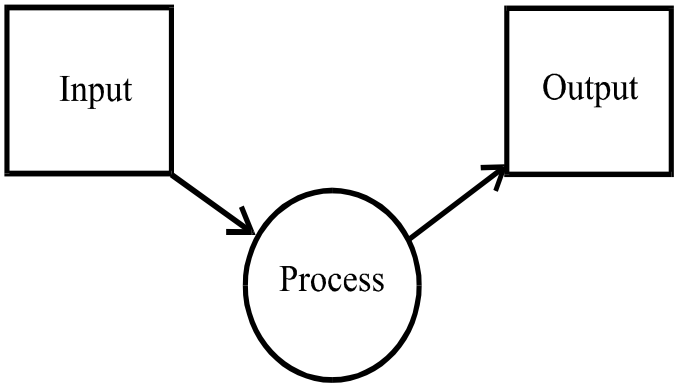
- Extendable
  - To similar tasks.
- Portable
  - Use “standard” compilers and libraries.

Stages of Program Development

- Design
  - Analysis of problem.
  - Outline design in English and Maths.
  - Detailed specification.
- Coding
  - Pseudo coding.
  - Language specific translation.
- Testing
  - Components in test-bed (bottom-up).
  - Skeleton (top-down).
- Common Fault:-
  - Not enough time on design and testing
  - Coding should be ~ mechanical - if not design incomplete.

Design: General

- Clear Statements in English and Maths
  - Write them down!
- Specify I/O & Actions



- Consider Possible Strategies
  - What are the intermediate data structures?

General Design (cont.)

- Limit Special Requirements
  - CPU intensive? Divide or make restartable.
  - Isolate specific device code (graphics packages do this for displays).
- Task Generalisation
- Plan Testing
  - Write down verification procedures.
  - Plan diagnostic printout of internal data structures and program flow.
- Take Great Care at this Stage!
  - Mistakes may be very expensive!
- In HEP have used SA/SD
  - Structured Analysis.
  - Structured Design.

Design: Detailed

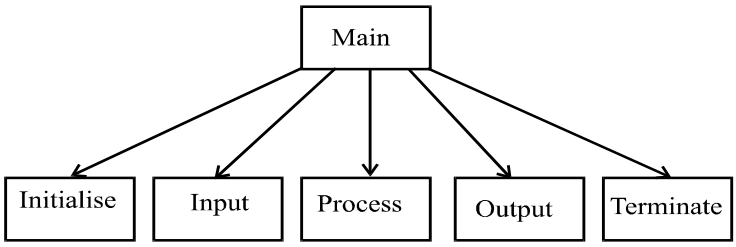
- Now use:-
  - Routine Names
    - Calling sequences & “Black Box” spec.
  - Data Structures
    - Precise descriptions of each data item for:-
      - COMMON blocks.
      - I/O records.
- Two Common Techniques:-
  - Bottom-Up.
  - Top-Down.

Bottom-Up Design

- Example, a Graphics Package

- Layer 0: Pen Up/Down. Move in X/Y
- Layer 1: Move (Pen Up, Move).
  Draw (Pen Down, Move)
- Layer 2: Draw Line (Move, Join)
- Layer 3: Draw Char (Move, Draw Line)
- ...
- ….
- Layer n: Draw Histogram,
  Plot Function.
- Principles:-
  - Each layer uses services from layers below.
  - Each layer provides complete and consistent "model" to layer above.
  - Only layer 1 talks to hardware.

Top-Down Design



- Divide and Conquer
  - Separate program into few logically independent sections.
  - Repeat with each section as necessary.
  - Implement each section at each level as a subroutine.
- Has Enormous Advantages
  - Simplifies problem.
  - Testing: Can test sections separately.
  - Maintenance: easy to find and fix.
  - Extendable: Plug in new section e.g. new processor.

Top-Down Design (cont.)

- Structuring
  - Separate and simplify.
  - Minimise Interconnections.
- Separate Out:-
  - Machine Dependencies:-
    - Generalise to machine independent concept.
    - Use bottom-up to implement.
  - I/O
  - Control & Processing "Boss & Slave" Model:-
    - BOSS - decides what to do.
    - SLAVES - do the work.
- COMMONS
  - Keep separate data in separate commons.

Top-Down Design (cont.)

- SUBROUTINES
  - Either - general function (test: could it go in library?)
  - Or - Called by one routine (its Boss).
- Interfaces
  - In FORTRAN either COMMON or Arg list.
  - Long Range i.e. upper level, use arg list (easy to replace).
  - Restrict COMMONs to lower level, tightly coupled and localise use (esp. modification).
  - E.g. Graphics Package
    - Pen X,Y in COMMON.
    - Use directly within package.
    - Provide subroutine for caller.

Design: Error Paths

- Error Paths May dominate program design (esp. interactive ones) because:-
  - If an error can occur it will (and if it can't it probably will).
  - To deal with an error must detect and analyse.
  - Standard data may have few process paths, but errors are deviations, they may have many.
- Example
  - 1000 data items, 10 bad
    => 1% error rate
  - 1 good data path, 10 error path
    => 90% error handling

Coding

- Star System:
  - * worth thinking about.
  - ** only break for good reason.
  - *** never break.
- COMMON Blocks
  - *** Meaningful names (e.g. I - useless). Although 6 letters names is FORTRAN77 standard, most FORTRAN compilers permit more.
  - ** Avoid simple names e.g. SUM (used locally).
  - *** Identical definition in all routines.
  - * Use a unique prefix on all variables..
  - * Use INCLUDE (not official FORTRAN77 - would be *** if it was).

Coding (cont.)

- SUBROUTINES
  - *** Continue top-down - divide code into blocks.
  - *** Minimise/eliminate GOTOs. If possible only jump down to next block. Sacrifice a little efficiency for fewer flow paths.
  - ** Only jump up as an iteration.
  - ** Use labels that reflect block structure e.g 100-199 for first block.
  - *** Keep labels in ascending order.
  - *** Give variables meaningful names.
  - ** Avoid long routines - 200 lines is getting too long.
  - * Collect formats at the end - helps if used multiple times and when adding more.
  - ** Use IMPLICIT NONE if possible (not FORTRAN77).

Coding (cont.)

- SUBROUTINES (cont.)
  - *** Always comment routine. A good scheme:-
    - 1 or 2 line summary.
    - Author names and a revision history of changes.
    - Description of interface:-
      - Arg. list
      - COMMONs

      } Specify what are input and output
    - Details of implementation.
    - Separate out and comment each block.
  - *** Take time to layout code professionally - indenting code and aligning material makes it more readable (and can highlight mistakes!)

Program Modification

Either Function or Performance

- Function
  - Golden Rule: How would I designed it if I were starting now?
  - Not: What is the absolute minimum change. It may get job done faster but shortens program lifetime!
- Performance
  - Optimise compilation.
  - Use Performance Analysers to find hot spots and then recode.
  - If all else fails, take a fresh look at intermediate data.

Testing

May be easy to ~ impossible. With complex programs it is very hard to prove free of bugs (except by redefining them as "features").

- Minimise Bugs
  - by Well Structured with Fewest Possible Flow Paths.
- Testing Methods:-
  - Desk Check - take time to read slowly checking:-
    - Flow paths.
    - Calling sequences.
    - Formulae.
  - Bottom-Up testing -
    - Test lowest level modules using a Test Bed.
    - Test levels above.

Testing (cont.)

- Testing Methods (cont.)
  - Top-Down Testing -
    - Start with top levels with almost dummy modules.
    - Add more function step by step.
    - Good for complex program development.
  - Minimum Condition:-
    - Test every line of code.
    - Remember to retest after fixing bugs.
- Leave in Switchable Debug Printout
  - Useful despite powerful debuggers.