Appendix

Programming practices

Recommended programming practices

The following is a list of recommended programming practices designed to increase the reliability of scientific computing software, along with a brief description of each practice.

Use strongly-typed programming languages

Although there is some ambiguity in the definition, here we refer to a strongly-typed programming language as one which (1) requires that a variable or object maintain the same type (e.g., integer, floating point number, character) during program execution and (2) has strict rules as to which types can be used during operations (i.e., implicit type conversions are not allowed). Common examples of the latter are the use of integer division on floating point numbers and the use real functions (e.g., **cos, log**) on integers. BASIC and C are considered weakly-typed languages, while C^{++}, Java, Fortran, Pascal, and Python are considered strongly-typed. For type information on other programming languages, see en.wikipedia.org/wiki/Template:Type_system_cross_reference_list. A type-safe program can be written in a weakly-typed language by using explicit type conversions (to convert integers to real numbers, real numbers to integers, etc.). In other words, explicit type conversions should be used, even when not required by the programming language.

Use safe programming language subsets

In order to avoid error-prone coding constructs, safe programming language subsets are recommended. An example of a safe subset for the C programming language is Safer C (Hatton, 1995).

Use static analyzers

Hatton (1997) estimates that approximately 40% of all software failures are due to static faults which are readily found with the use of static analyzers (see Chapter 4).

Use long, descriptive identifiers

Most modern programming languages allow long, descriptive names to be used for variables, objects, functions, subroutines, etc. The extra time spent typing in these longer names will be more than made up by the reduction in time spent figuring out what a variable contains or what a routine does.

Write self-commenting code

The good software developer will endeavor to write code in such a way that the coding itself clearly explains its purpose (Eddins, 2006). This is certainly aided by the use of descriptive identifiers as discussed above. While the use of comments is still a subject of debate, an extreme example where 100 lines of comments are needed to explain the workings of 10 lines of executable source code suggest that the code is not well written.

Use private data

Accidental over-writing of data can be minimized through the use of private data, where data is made available only to those objects and routines that need to process it. Both C^{++} and Fortran 90/95 allow for the use of private data, the latter through the use of `Public` and `Private` attributes within modules.

Use exception handling

Exceptions can occur due to internal conditions (e.g., division by zero, overflow) or external factors (e.g., insufficient memory available, input file does not exist). Exception handling can be as simple as letting the user know the local state of the system and the location where the exception occurred, or it could transfer control to a separate exception-handling code. Some programming languages such as Java and C^{++} have built-in exception-handling constructs.

Use indentation for readability

Indent the coding blocks to denote different levels of looping structures and logical constructs to make them more readable. An example of an indented Fortran 95 code construct is given below.

```
if(mms == 0) then
  !Set standard boundary conditions
  Call Set_Boundary_Conditions
elseif(mms == 1) then
  !Calculate exact solution for temperature
```

```
  do j = 1, jmax
  do i = 1, imax
  Temperature_MMS(i,j) = MMS_Exact_Solution(x,y)
  enddo
  enddo
  !Set MMS boundary conditions
  Call Set_MMS_Boundary_Conditions
else
  !Check for invalid values of variable mms
  write(*,*) 'Error: mms must equal 0 or 1 !!!'
  Error_Flag = 1
endif
```

Use module procedures (Fortran only)

The use of module procedures (functions and subroutines) in Fortran rather than standard procedures provides an explicit interface between the procedure and its calling program. Thus interface consistency can be checked and interface errors can be found during code compilation rather than at run time.

Error-prone programming constructs

Although allowed in certain programming languages, the following programming constructs are known to be error prone and should be avoided when possible.

Implicit type definitions

Implicit variable type definitions, where new variables can be introduced in a program without a corresponding type specification, should be avoided. For the Fortran programming language, this means that ``Implicit None'' should appear at the beginning of every routine.

Mixed-mode arithmetic

With the exception of exponentiation, integer and real variables should not be used in a single expression. When they do occur together, explicit type conversions (e.g., real to integer or integer to real) should be used.

Duplicate code

When the same coding construct appears multiple times in a program, it is a good indication that the piece of coding should be replaced with a function or subroutine. Duplicate code

can make software development tedious since a modification to one instance requires the developer to also search out all other instances of the repeated code. Eddins (2006) cautions that "anything repeated in two or more places will eventually be wrong in at least one."

Equality checks for floating point numbers

Since floating point (real) numbers are subject to machine round-off errors, equality comparisons between them should be avoided. For example, instead of checking for equality between the floating point numbers A and B, one could instead check to see if the absolute value of $(A - B)$ is less than a specified tolerance.

Recursion

Recursion occurs when a component calls itself, either directly or indirectly. A recursive programming construct can be difficult to analyze, and errors in a recursive program can lead to the allocation of a system's entire available memory (Sommerville, 2004).

Pointers

A pointer is a programming construct that contains the address of a direct location in machine memory (Sommerville, 2004). The use of pointers should be avoided since pointer errors can cause unexpected program behavior (e.g., see aliasing below), can be extremely difficult to find and correct, and can be inefficient for scientific computing codes.

Aliasing

Aliasing "occurs when more than one name is used to refer to the same entity in a program" (Sommerville, 2004) and should be avoided.

Inheritance

Inheritance occurs when an object "inherits" some characteristics from another object. Objects that employ inheritance are more difficult to understand since their defining characteristics are located in multiple locations in the program.

GOTO statements

`GOTO` statements should be avoided as they make the program difficult to follow, often resulting in a complex and tangled control structure (i.e., "spaghetti" code).

Parallelism

Although the use of parallel processing in large scientific computing applications is usually unavoidable, the developer should be aware of the potential for unexpected behavior due to timing interactions between processes. Accounting for parallelism during the initial architectural design can result in more reliable software as compared to the case where parallelism is added after the serial version of the software is developed. In general, these issues cannot be detected with static analysis and may be platform dependent.

References

Eddins, S. (2006). Taking control of your code: essential software development tools for engineers, *International Conference on Image Processing*, Atlanta, GA, Oct. 9, (see blogs.mathworks.com/images/steve/92/handout_final_icip2006.pdf).

Hatton, L. (1995). *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, McGraw-Hill International Ltd., UK.

Hatton, L. (1997). Software failures: follies and fallacies, *IEEE Review*, March, 49–52.

Sommerville, I. (2004). *Software Engineering*, 7th edn., Harlow, Essex, England, Pearson Education Ltd.