## **Good Programming Practice**

Patrick Guio

Patrick Guio Good Programming Practice

ヘロト 人間 とくほとくほとう





Programming Practice



4 Debugging & Optimising



#### (Readability and Maintainability)

(Performance Enhancements)

## Code Formatting: Be Consistent!

Rule of thumb:

Consistency more important than specific formatting style

- No matter how many SPACE's you use for an indent, use it consistently throughout the source code. SPACE's and TAB's do not mix well!
- Indent code to better convey the logical structure of your code. Without indenting, code becomes difficult to follow.

if the if the		
else		
end if		
else		
end if		

Patrick Guio Good Programming Practice

# Code Formatting (Contd)

- Establish a maximum line length for comments and code to avoid having to scroll the window of the text editor (and allow clean hard-copy even though printing is not recommended!).
- Use SPACE after each "comma" in lists, such as array values and arguments, also before and after the "equal" of an assignment

Energy =  $0.5 * k_b * \text{Temp}(i, j, k)$ 

- Use empty lines to provide organisational clues to source code, blocks ("paragraphs"-like structure) help the reader in comprehending the logical segmenting.
- When a line is broken across several lines, make it obvious that the line is incomplete using indentation.

Code Formatting (Contd)

 Avoid placing more that one statement per line, an exception is loop in C and C++

for (i = 0; i < 100; i++)

 In FORTRAN, avoid to define format statements "far away" from the READ/WRITE statement itself

```
write(fileld, 99062)iter
...
99062 format ('Number of iterations = ',i7)
```

Even better, avoid label at all and include the format within the statement



## Code Formatting (Contd)

- Enable syntax highlighting in your text editor.
- Use freely available program that help to indent, format, and beautify your source code automatically and consistently.
  - indent for C, astyle for C, C++, C# and Java.
  - floppy for FORTRAN 77, tidy for FORTRAN 77/90.
- Break large, complex sections of code into smaller, comprehensible modules (subroutine/functions/methods). A good rule is that modules do not exceed the size of the text editor window.
- Arrange and separate your source code logically between files.

ヘロト 人間 ト 人 ヨ ト 人 ヨ ト

## Naming convention: Be Consistent!

Rule of thumb:

Consistency more important than specific naming convention

- Choose and stick to a style for naming various elements of the code, this is one of the most influential aids to understand the logical flow.
- Difficulty to find a proper name for a routine/variable may indicate a need to further analysis to define its purpose... "Ce que l'on conçoit bien s'énonce clairement", from "l'art poétique", Nicolas Boileau, 1674.
- A name should tell what rather than how, avoid names that expose underlying implementation.
- Ideally you would like to be able to read the code as prose.

・ロト ・四ト ・ヨト・

Naming convention (Contd)

Avoid elusive names, open to subjective interpretation like

Analyse(...) // subroutine or function or method nnsmcomp1 // variable

It brings ambiguity more than abstraction...

• Use a verb-noun method to name routines that perform some operation-on-a-given-object. Most names are constructed by concatenating several words, use mixed-case formatting or underscore to ease reading.

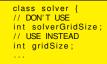
calculateKineticEnergy(...) calculate\_kinetic\_energy(...)

or any other derivatives.

ヘロト ヘアト ヘヨト ヘ

Naming convention (Contd)

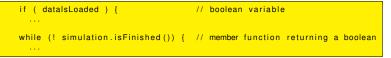
• In Object-Oriented languages, it is redundant to include the class name in the name of a member field or function, like



- In languages with overloading capability (C++, Matlab,...), overloaded functions should perform a similar task.
- Append/Prepend computation qualifiers like Av, Sum, Min, Max and Index to the end of a variable when appropriate.
- Use customary opposite pairs for names such as min/max, begin/end, start/stop, open/close.

## Naming convention (Contd)

 Boolean type variable names (and functions returning boolean) should contain is/Is to "imply" a True/False value



 Avoid using terms such as "Flag" for status variable different from boolean type

integrationFlag // expect a boolean type integrationMethodType // expect a status value

• Even for short-lived variable, use a meaningful name. Use single-letter variable (i, j) for short-loop indexes only.

## Naming convention (Contd)

- If using Charles Simonyi's Hungarian Naming Convention, or some derivative, develop a list of standard set of prefixes for the project (for instance arrldf, arrldi, arr2df, etc...).
- For variable names, it can be useful to include notation that indicate the scope of the variable, such as the prefix g\_ for global or 1\_ for local.
- "Constants" (literals) should be all uppercase with underscores between. For instance, the C header file "math.h" define  $\pi$  and  $\sqrt{2}$  as the literal

```
# define M_PI 3.14159265358979323846 /* pi */
# define M_SQRT2 1.41421356237309504880 /* sqrt(2) */
```

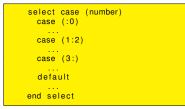
(日)

- Coding techniques Programming Practice Compiling code Debugging & Optimising Maintaining code
- Keep lifetime of variables as short as possible when the variable represents a finite resource such as a file descriptor.
- Keep scope of variables as small as possible to avoid confusion and ensure maintainability.
- Use variables and routines for one purpose only. Avoid multipurpose routines that perform a variety of unrelated tasks...
- Keep in mind what control flow constructs do, for instance

```
if ( isReady ) then
...
else if ( .NOT.isReady ) then
...
end if
```

contains an unnecessary construction. Which one?

 Take advantage of the control flow construct capabilities of a language, for instance in FORTRAN 90 use



#### instead of

```
if (number .lt. 0) then
...
else if (number .eq. 1 .or. number .eq. 2) then
...
else if (number .ge. 3) then
...
else
...
end if
```

Patrick Guio Good Programming Practice

◆□ ▶ ◆□ ▶ ◆ □ ▶ ◆ □ ▶ ● □ ● ● ● ●

- Coding techniques Programming Practice Compiling code Debugging & Optimising Maintaining code
- Some common sense and critical analysis can help to avoid such "flaw". nn is never modified in the subroutine, called once with value zero.... What is wrong?

```
subroutine update_fields(..., nn)
...
do j = 1, maxDim2
do i = 1, maxDim1
do k = 1, maxDim3
...
if (nn .eq. 0) then
...
else
...
end if
...
end do
end do
end do
```

 Don't forget how array are stored internally in C and FORTRAN, "optimal" encapsulated loops over several dimensions is different for C and FORTRAN.

- Use literals. A good rule is to gather related literals in a single "header" file to include wherever needed.
- Don't assume output formats. Functions should return values in original type, the caller should decide what to do, reformat, sent to standard output, etc...
- Wrap built-in functions and third-party library functions with your own wrapper functions can be beneficial. This is a good practice to serve several purposes:
  - If third-party libraries interface change, only the the wrapper functions need change, not the main application.
  - Easier to add code or breakpoints at one place in the wrapper when debugging for instance.

- Test returned status for error conditions. When you try to open a file, write to, read from, or close it, doesn't mean you will succeed. When calling a function that can "throw" an error, you should add code to deal with that potential error.
- Recover or fail "gracefully". Robust programs should report an error message (and optimally attempt to continue).
- Provide useful error messages. Expanding on the previous point, you should provide a user-friendly error message while simultaneously logging a programmer-friendly message with enough information that they can investigate the cause of the error.

ヘロト ヘアト ヘビト ヘビト

1

### Understanding "Makefile"

• "Makefile" contain statements like

target: dependencies command\_to\_update\_target # optional

When the "command\_to\_update\_target" is provided, the statement is called an explicit rule.

• "Makefile" can define variables

```
CXX=g++
CXXFLAGS-O
hello: hello.cpp
$(CXX) $(CXXFLAGS) -o hello hello.cpp
```

Variables are useful as they can be redefined on the command line, allowing to change compiler, compiler options without editing the file. For instance

% make hello CXX=icpc CXXFLAGS=-g\_

### Understanding Makefile (contd)

• "Makefile" variables can be substituted

OBJECTS=\$(SOURCES:.cpp=.o)

• "Makefile" can define inference rules, such as

```
.SUFFIXES: .cpp
.cpp.o:
$(CXX) $(CXXFLAGS) -o $@ -c $<
# or equivalently
%.o: %.cpp
$(CXX) $(CXXFLAGS) -o $@ -c $<
```

Use  ${\tt make}\ -{\tt p}$  to get the database of variables and inference rules.

- \$@ contains the target name
- \$^ contains the list of dependencies
- \$< contains the first element of the list of dependencies

#### Writing a "Makefile"

 There is no unique way to write a "Makefile", here is one way to compile a FORTRAN 90 code

```
FC=ifort

FFLAGS=-O

SOURCES=main.f90 init.f90 integrate.f90

OBJECTS=$(SOURCES:.f90=.0)

EXECUTABLE=myProg

default: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)

$(FC) $(FFLAGS) -0 $@ $^

%.0: %.f90

$(FC) $(FFLAGS) -0 $@ $<
```

You can force another compiler and other options on the command line with

% make FC=myCompiler FFLAGS=myOptions

ヘロン 人間 とくほ とくほ とう

3

# Debugging

- All compilers have the option -g which can be used to generate extra information to be used together with a symbolic debugger (commands of gdb such as run, break, start/step, list, print and of course help can do a lot to debug!)
- Intel FORTRAN compiler ifort have options to check run time condition like -check uninit for uninitialised variables or -check bounds for access outside allocated array.
- Reading the documentation of your compiler/debugger is always a good start (man ifort/icpc/idb/g++/gcc/gfortran/gdb/...).

イロト 不得 とくほ とくほとう

# Optimising

- All compilers have the option -0 which provides default and "safe" optimisation.
- Note that some compiler optimise without any options, while others don't. Be careful!
- Intel FORTRAN compiler ifort (as well as the C++ compiler icpc) have options for more aggressive optimisation such as -fast, and hardware-dependent such as the -xSSE family.
- Information about your CPU capability on a Linux system can be found in the file /proc/cpuinfo.

# Setting up a Version Control System (cvs)

- You need first to set up your cvs repository. Choose a directory with disk space for several times the size of your actual source package, then set the CVSROOT environment variable to this path. For instance, for the Berkeley/C shell family
  - % setenv CVSROOT

:ext:yourId@yourServer:pathToYourRepo then run the command to initialise the repository

% cvs init

This command is run only once and creates the repository and the special module containing the configuration files for this repository.

くロト (過) (目) (日)

# Using cvs

• You can import your code into the sub directory myProject of the repository by running the "import" command from the root directory of your project

% cvs import myProject mySoft START
mySoft is a so-called vendor tag, START is the initial
release tag.

- Then you can get a working copy of your project with the command
  - % cvs checkout myProject

It will create the sub directory myProject and put all the files you have imported into the repository to allow further development.

ヘロト ヘアト ヘビト ヘビト

# Using cvs (contd)

 If you make any changes (and you are happy with them), you can commit your changes into the repository with the command

% cvs commit [filename(s)]
without forgetting to log your changes.

 If you have "checked out" your project on another machine, you can synchronise these with the following command % cvs update

## Conclusion

- Programming is not an exact science, but the more you practice, the more you develop skills...
- Using such "cooking recipes" and a bit of common sense should hopefully help you to develop your awareness for good practice.
- Due to the limited time, this course introduced a limited amount of aspects to good programming practice, feel free to drop by my office to discuss any programming problems, I will try to help you.