Computer programming

"Knowledge is of no value unless you put it into practice."
— Anton Pavlovich Chekhov

Best practices

Best Practices for Scientific Computing

by **Greg Wilson et al**.

- http://arxiv.org/pdf/1210.0530v3.pdf

summarized by

- Jonathan Callahan, in:

Best Best Practices Ever

- http://server.dzone.com/articles/best-best-practices-ever

Best practices. C

- style:
  - If there are several levels of *indentation*, each level should be indented the *same* additional amount of *space*.
  - Too many levels of nesting can make a program difficult to understand. As a rule, try to *avoid using more than three levels of nesting*.
  - The combination of vertical spacing before and after control statements and indentation of the bodies of control statements within the control-statement headers gives programs a two-dimensional appearance that greatly improves program *readability*.
  - *Unary operators* should be placed *directly next to their operands with no intervening spaces*.

Best practices. C

- control:
  - When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as printing an error message) rather than allowing the fatal error to occur.
  - In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.
- for:
  - Although statements preceding a `for` and statements in the body of a for can often be merged into the for header, avoid doing so, because it makes the program more difficult to read.
  - Limit the size of control-statement headers to a single line if possible.

Best practices. C

- switch:
  - Although the `case` clauses and the `default` case clause in a switch statement can occur in any order, it's common to place the `default` *clause last*.
  - In a `switch` statement when the default clause is last, the `break` statement isn't required. You may prefer to include this `break` for *clarity and symmetry* with other `case`s.
- do-while:
  - To eliminate the potential for ambiguity, you may want to include *braces* in `do … while` statements, even if they're not necessary.

Best practices. C

- re-use:
  - Familiarize yourself with the rich collection of functions in the C standard library.
  - Avoid reinventing the wheel. When possible, use C standard library functions instead of writing new functions. This can reduce program development time
- functions:
  - Although it's not incorrect to do so, do not use the same names for a function's arguments and the corresponding parameters in the function definition. This helps avoid ambiguity.
  - Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.

Best practices. C

- functions:
  - Small functions promote software reusability.
  - Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain and modify.
  - Include function prototypes for all functions to take advantage of C's type-checking capabilities. Use #include preprocessor directives to obtain function prototypes for the stan- dard library functions from the headers for the appropriate libraries, or to obtain headers containing function prototypes for functions developed by you and/or your group members.
  - Parameter names are sometimes included in function prototypes (our preference) for documentation purposes. The compiler ignores these names.

Best practices. C

- enums:
  - Use only uppercase letters in the names of enumeration constants to make these constants stand out in a program and to indicate that enumeration constants are not variables.
  - Use only uppercase letters in enumeration constant names. This makes these constants stand out in a program and reminds you that enumeration constants are not variables.
- *Strive for program clarity. Sometimes it may be worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs*

Best practices. C

- constants:
  - Use *only uppercase letters for symbolic constant names*. This makes these constants stand out in a program and reminds you that symbolic constants are not variables.
  - In *multiword* symbolic constant names, separate the words with *underscores* for readability.
  - Using *meaningful names* for symbolic constants helps make programs self-documenting.
  - By convention, symbolic constants are defined using only uppercase letters and underscores.

Best practices. C

- pointers:
  - We prefer to include the letters Ptr in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately
- input:
  - When inputting data, prompt the user for one data item or a few data items at a time. *Avoid asking the user to enter many data* items in response to a single prompt.
  - Always *consider* what the user and your program will do when (not if) *incorrect data* is entered—for example, a value for an integer that's nonsensical in a program's context, or a string with missing punctuation or spaces.

Best practices. C

- structs:
  - *Always provide a structure tag name* when creating a structure type. The structure tag name is convenient for declaring new variables of the structure type later in the program.
  - *Do not put spaces around* the `->` and `.` operators. Omitting spaces helps emphasize that the expressions the operators are contained in are essentially single variable names.
- typedef:
  - Capitalize the first letter of `typedef` names to emphasize that they're synonyms for other type names.
  - Using `typedef` s can help make a program be more readable and maintainable.

Best practices

- Write programs for people, not computers.
  - a *program* should not require its readers to hold more than a *handful of facts in memory* at once
  - *names* should be *consistent, distinctive* and *meaningful*
  - *code style and formatting* should be *consistent*
  - all *aspects of software development* should be broken down into *tasks roughly an hour long*

Best practices

- Automate repetitive tasks.
  - rely on the computer to repeat tasks
  - save recent commands in a file for re-use
  - use a build tool to automate scientific workflows
- Use the computer to record history.
  - software tools should be used to track computational work automatically (SVN, Git…)
    - Tortoise SVN

Best practices

- Make incremental changes.
  - *work in small steps with frequent feedback and course correction*
- Don't repeat yourself (or others).
  - *every piece of data must have a single authoritative representation in the system*
  - *code should be modularized rather than copied and pasted*
  - *re-use code instead of rewriting it*

Best ractices

- Plan for mistakes.
  - *add assertions to programs to check their operation*
  - *use an off-the-shelf unit testing library*
  - *use all available oracles when testing programs*
  - *turn bugs into test cases*
  - *use a symbolic debugger*

Best practices

- Optimize software only after it works correctly.
  - *use a profiler to identify bottlenecks*
  - *write code in the highest-level language possible*
- Document design and purpose, not mechanics.
  - *document interfaces and reasons, not implementations*
  - *refactor code instead of explaining how it works*
  - *embed the documentation for a piece of software in that software*

Best Practices

- Collaborate.
  - *use pre-merge code reviews*
  - *use pair programming when bringing someone new up to speed and when tackling particularly tricky problems*
- Extra (J. Callahan)
  - Maintain and update older code.

- Now, some practice...
  - See files on course pages for this lecture