STAT 534 Handout 3

April 8, 2002

How to write good code

©Marina Meilă

mmp@stat.washington.edu

Good code

- does what it's supposed to do
- handles exceptions (you know what it's doing)
- is efficient (time and resources)
- easy to understand, debug, modify, reuse

These goals are sometimes conflicting:

- too much error checking can make the code slow, hard to read
- making the code more efficient sacrifices readability. For example:
  - dynamic memory allocation
  - higher level languages: programming is faster, running the code is slower (in general)

However, there are some simple rules that help you write code that's easy to understand and modify AND don't conflict with the other 3 goals (possibly even help you achieve them easier too).

1 Some guidelines for writing good code

Clear program specifications. Every function should be accompanied by documentation (C comments) stating as clearly as possible:

- what the function does (optional)
- what is **required** (exception conditions that are not checked)
- what is **ensured**
  - normal case
  - exceptional cases

Mathematical notation is encouraged, since it makes for precise and short statements.

Clear program specifications lead to programs that are both faster (by eliminating redundant error checks) and are easier to understand.

Use libraries – including libraries written by yourself.

Programming style is partly a matter of taste. Strive to develop a consistent programming style that suits you but is not far from the common norm so that others can also appreciate it.

- suggestive, self-explanatory names for variables and functions
- the "Hungarian" naming convention: each variable is prefixed with a short string encoding its type. Useful especially for long programs with many variables.
- indent, make "paragraphs", use functions to encapsulate
- comment sparingly and to the point
- write short functions, few parameters

2 Errors in code

Errors can appear either

- at compilation and link time (for ex. syntax errors, forgetting to include a header file)

- at run time (for ex. segmentation fault, due to a pointer that contains garbage)
- in the program output. The program runs but the results are not correct.

Common sources of errors in C

- typos – break long expressions into shorter, more readable ones
- uninitialized variables – some compilers can check this
- memory leaks (allocating memory and forgetting to free it) – make a little cleanup section at the end of each function
- arrays out of bounds – use debugger, use `assert`
- infinite loops – use loop invariants, use `assert`
- programming errors – use `assert`