Selected topics in software development

Today:

Programming style

Speaker:

Jyrki Katajainen

- code idioms (cf. design patterns)

Course home page:

`http://www.diku.dk/forskning/performance-engineering/Software-development/`

Potential topics on programming practice

- testing
- debugging
- portability
- performance
- design alternatives
- **style**

[Brian W. Kernighan & Rob Pike, The practice of programming, p. ix]

Why is good style important?

Clarity: Well-written code is easier to read and to understand.

Correctness: Sloppy code is often broken.

Simplicity: Well-written code is likely to be smaller than code that has been carelessly tossed together and never polished.

Maintainability: The best layout schemes hold up well under code modifications.

[Brian W. Kernighan & Rob Pike, The practice of programming, §1]

Write clearly — don't be too clever.

[Brian W. Kernighan & P. J. Plauger, The elements of programming style, p. 2]

Say what you mean, simply and directly.

[Brian W. Kernighan & P. J. Plauger, The elements of programming style, p. 11]

Write first in an easy-to-understand pseudo-language;
then translate into whatever language you have to use.

[Brian W. Kernighan & P. J. Plauger, The elements of programming style, p. 39]

Format a program to help the reader understand it.

[Brian W. Kernighan & P. J. Plauger, The elements of programming style, p. 123]

Recall the seven pillars of pretty code

Blend in: Code changes should blend in with the original style.

Bookish: Keep columns narrow.

Disentangle code blocks: Break code into logical blocks and disentangle their purpose so that each does a single thing.

Comment code blocks: Set off code blocks with whitespace and comments that describe each block.

Declutter: Remove anything that will distract the reader.

Make alike look alike: Two or more pieces of code that do the same or similar thing should be made to look the same.

Overcome indentation: The left side of the code defines the structure, the right side the detail. Fight indentation to safeguard this!

[Christopher Seiwald, Seven pillars of pretty code]

Formatting theorem

Theorem: Good visual layout shows the logical structure of a program.

Making the code look pretty is worth something, but it is worth less than showing the logical structure of the code. If one technique shows the structure better and another looks better, choose the one that shows the structure better. Pretty does not necessarily mean readable or maintainable!

[Steve McConnell, Code complete, 2nd edition, §31.1]

Typographic styles

Macro-typographic factors:

- program formatting
- global and intermodule commenting
- module separation conventions
- identifier-naming conventions
- conventions for special-case font styles

Micro-typographic factors:

- statement formatting
- indentation and embedded spacing
- use of blank lines
- intra-module commenting

Typographic principles

Macro typography:

1. Make the components and organization of the program obvious.
2. Identify the purpose and use of each component.
3. Make the execution control and information flow between components apparent.
4. Make the program readable and easy to browse through by providing different access paths into the code.

Micro typography:

1. Make the sections and organization of the module obvious.
2. Identify the purpose and use of each section.
3. Make the underlying control and information flow within the module obvious.
4. Make statements readable and easy to scan by providing spatial clues and white space to indicate statement grouping and separation.

Book-format paradigm

Macro-typographic factors:

- creation of a preface
- table of contents
- chapter divisions
- pagination
- indices

Micro-typographic factors:

- identification/creation of code sections
- code paragraphs
- sentence structures
- intra-module comments

Online exercise (5 min)

Comment the macro-typographic style used in the CPH STL reports 2005-1 and 2007-4.

Code idioms

Code idioms are the implementation analog of the design patterns discussed earlier. If we agreed on the code idioms used, it would be much easier for us to understand each others code.

Pick one coding standard, preferably the one agreed on the project, use it consistently, and do not waste time in arguing.

I will concentrate on the elements of C/C++ style, but much of what I say can be applied for other languages too.

Which variable-naming convention is best?

A: Variable names are all in lower case as in `variable_name`.

B: Variable names use lower-case letters for the first word, with the first letter of each following word capitalized as in `variableName`.

C: Variable names include a standardized prefix indicating its type or its intended use as in `strVariableName`.

STL conventions

- `c` is a character variable
- `i`, `j`, and `k` are integer indexes
- `n` is a number of something
- `p` and `q` are pointers
- `s` is a string
- Preprocessor macros are all in upper case as in `ALL_CAPS`
- Variable and routine names are all in lower case as in `all_lower_case`
- The underscore (_) character is used as a separator between words as above.

What would be a good single-letter variable name for an iterator?

Variable-naming idiom

Use descriptive names for global (and class) variables, and short names for local variables. To indicate the origin of a variable, a (private/protected) class variable can be qualified with `(*this)` and a global variable with `::` (meaning that the variable is found in the global namespace).

Select sensible names for your variables. E.g. `it` is a good name for an iterator, but not for a pointer. A misleading name can result in mysterious bugs.

Use active names for functions. Also, functions that return a boolean should be named so that the return value is unambiguous (e.g. `is_empty`).

Indentation idiom

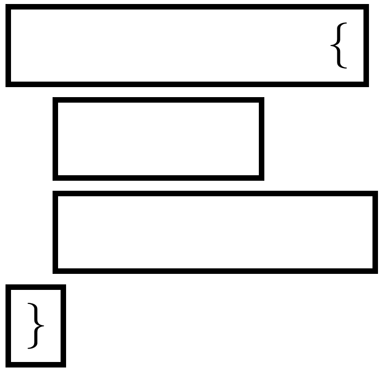
Indentation should be idiomatic, too. Use 2-4 space indentation to show the logical structure of a program. If you use tabs, make sure that your code-beautifier and other tools handle them appropriately.

```
  for (++n; n != 100; ++n) {
    field[n] = '\0';
  }

? for (
?   n++;
?   n != 100;
?   field[n++] = '\0';
? )
? {
?   ;
? }
```

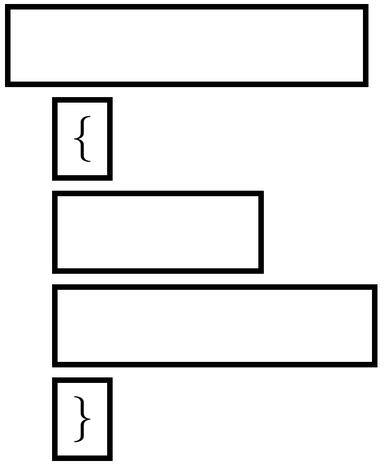
Which brace style is best?

Pure blocks:



```
while (colour == red) {
  statement₁
  statement₂
  ...
}
```

Braces as block boundaries:



```
while (colour == red)
  {
  statement₁
  statement₂
  ...
  }
```

Basic-loop idiom

Most loops, where the exit test is performed at the beginning, should have the form:

```
  for (i = 0; i != n; ++i) {
    ...
  }

  for (p = list; p != 0; p = (*p).next) {
    ...
  }

? i = 0;
? while (i <= n - 1) {
?   ...
? }

? for (i = 0; i < n; ) {
?   ...
?   i++;
? }

? for (i = n; --i >= 0; ) {
?   ...
? }
```

Infinite-loop idiom

For infinite loops use one of the following alternatives; do not use other forms.

```
  while (true) {
    ...
  }

  for (;;) {
    ...
  }

? while (1) {
?   ...
? }
```

End-exit idiom

A `do-while` loop always executes at least once; in many cases that behaviour is a bug waiting to bite, but when it is needed, write `}` in front of `while` to indicate clearly that `while` does not start a `while` loop.

```
safety_counter = 0;
do {
  node = (*node).next;
  ...
  ++safety_counter;
  if (safety_counter >= SAFETY_LIMIT) {
    std::cerr << "Internal error: Safety-counter violation.\n"
    ASSERT(false);
  }
} while ((*node).next != 0);
```

Middle-exit idiom

Few programming languages provide direct support for a loop construct whose exit point is in the middle. An idiomatic way of writing such a loop is shown in the following example:

```
  while (true) {
    c = getchar();
    if (c == EOF) {
      break;
    }
    putchar();
  }

? while ((c = getchar()) != EOF) {
?   putchar();
? }
```

Switch-break idiom

Cases should almost always end with a `break`, with the rare exceptions commented. For an unusual structure, a sequence of `else-if` statements can be even clearer.

```
  if (c == '-') {
    sign = -1;
    c = getchar();
  }
  else if (c == '+') {
    c = getchar();
  }
  else if (c != '.' && !isdigit(c)) {
    return 0;
  }

? switch (c) {
? case '-': sign = -1;
? case '+': c = getchar();
? case '.': break;
? default:  if (!isdigit(c))
?             return 0;
? }
```

```
? switch (c) {
? case '-':
?   sign = -1;
?   /* fall through */
? case '+':
?   c = getchar();
?   break;
? case '.':
?   break;
? default:
?   if (!isdigit(c)) {
?     return 0;
?   }
?   break;
? }
```

Operation-per-line idiom

Avoid using multiple operations per line; in particular, be careful with statements that have side effects.

```
  char* copy_string(char* s, char const* t) {
    char* const r = s;
    while (*t != '\0') {
      *s = *t;
      ++s;
      ++t;
    }
    *s = '\0';
    return r;
  }

? void strcpy(char *s, char *t) {
?   while (*s++ = *t++)
?     ;
? }
```

No performance penalty

```
jyrki@Per:~/Software-development/Slides/Style$ g++ string-copy.c++
jyrki@Per:~/Software-development/Slides/Style$ ./a.out
Performance testing ::strcpy ...
  Running time per character (ns): 5.9
Performance testing ::copy_string ...
  Running time per character (ns): 5.8
Performance testing std::strcpy ...
  Running time per character (ns): 5.5

jyrki@Per:~/Software-development/Slides/Style$ g++ -O3 string-copy.c++
jyrki@Per:~/Software-development/Slides/Style$ ./a.out
Performance testing ::strcpy ...
  Running time per character (ns): 1.2
Performance testing ::copy_string ...
  Running time per character (ns): 1.2
Performance testing std::strcpy ...
  Running time per character (ns): 1.1
```

Declaration-per-line idiom

Use only one data declaration per line.

```
  FILE* input_file;
  FILE* output_file;

? FILE *input_file, *output_file;
```

Separate the type names and variable names clearly in data declarations. In C and C++, data declarations are read from right to left (and from the inside out). Keep this in mind when writing data declarations; `typedefs` can be used to improve readability.

`int const* p;` Pointer to a constant integer, i.e. the integer pointed to does not change.

`int* const q;` Constant pointer to an integer, i.e. the pointer is kept constant but the underlying integer can be modified.

```
  typedef char* routine(char*, char const*);
  routine* copy = copy_string;
```

Memory-allocation idiom

In a real program, the return value of `malloc`, `realloc`, or any other allocation routine should always be checked.

```
  p = (char*) malloc(strlen(buffer) + 1);
  if (p == 0) {
    reset();
    return 0;
  }
  strcpy(p, buffer);

? p = malloc(strlen(buffer) + 1);
? strcpy(p, buffer);
```

More comments on readability

Operators: Use spaces around operators: `x = y + z;`

,: Add a single space after commas: `a = f(x, y, z);`

`!`: Negations are hard to understand and should be avoided.

`if`: Deeply nested `if` statements are difficult to follow.

`->`: Arrows clutter the code; my preference is `(*p).next`.

Instead of following the rules slavishly, it is more important to be consistent.

DiffMerge's chequered past

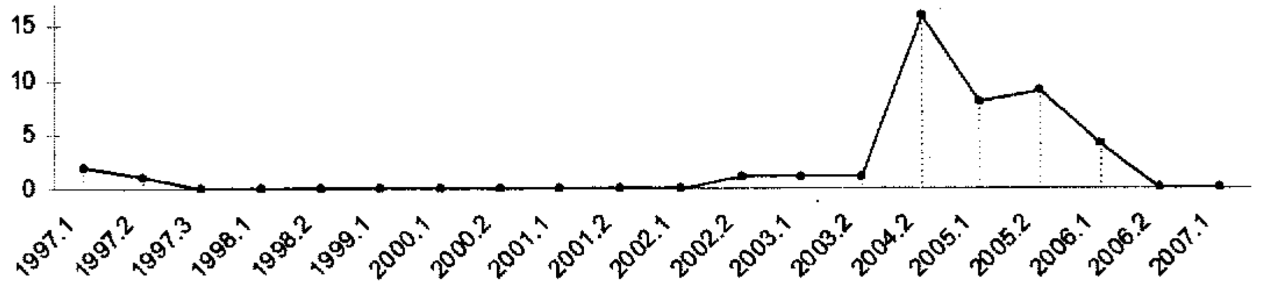


FIGURE 32-4. Number of patches applied to DiffMerge per release

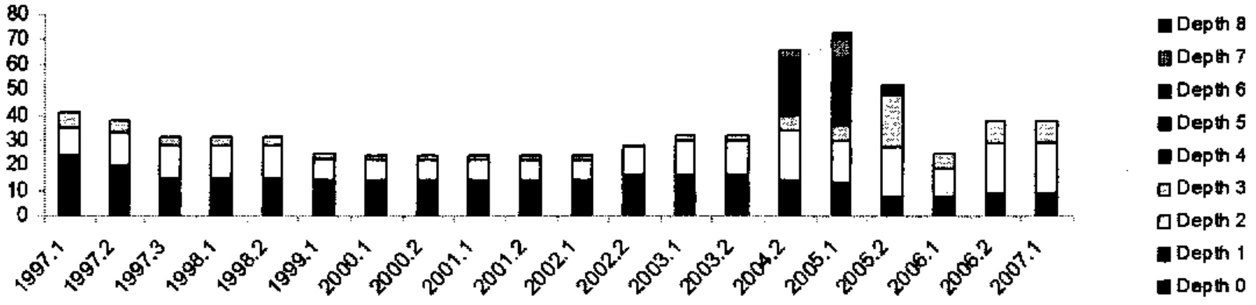


FIGURE 32-5. Number of DiffMerge's if statements at successive indentation depths per release

[Laura Wingerg & Christoper Seiwald, Beautiful code, p. 535]

Comments on maintainability

- Give names to magic numbers.
- Be prepared for changes.
- Use a commenting style that is easy to maintain.

```
#define SIZE(array) (sizeof(array) / sizeof(array[0])

char buffer[100];

for (i = 0; i != SIZE(buffer); ++i) {
  // Use braces even if the loop body only contains one statement.
}
```

Laying out class interfaces

Header comment: Describe the class and provide any notes about the overall usage of the class.

Public types:

Public routines: Describe constructors and destructors first.

Protected types:

Protected routines:

Private types, routines, and member data:

[Steve McConnell, Code complete, 2nd edition, §31.8]

Laying out class implementations

Header comment: Describe the contents of the file where the implementation is in.

Public routines: Describe constructors and destructors first.

Protected routines:

Private routines:

Separate routines in a file clearly, and group related routines together or put the routines in alphabetical order.

[Steve McConnell, Code complete, 2nd edition, §31.8]

Example problem

Task: Remove all comments from a LaTeX file; that is, remove private notes before distributing the file.

```
\begin{theorem} ...
\end{theorem}

\begin{proof} Omitted. \qed
% Our back-of-the-envelope proof must be spelled out in
% full length. In particular, the tricky special case ...
\end{proof}
```

LaTeX manual: When LaTeX encounters a % character while processing an input file, it ignores the rest of the present line, the line break, and all whitespace at the beginning of the next line. An escaped % character (\%) does **not** start a comment.

Online exercise (5 min)

Write pseudo-code for a program that removes all comments from a given LaTeX file.

Solution?

```
/* Poista kommentit tex-tiedostosta. */

#include <stdio.h>
#include <stdlib.h>

void poista_kommentit(FILE *, FILE *);
```

Main routine

```
void main(void) {
  char inname[512], outname[512];
  FILE *infile, *outfile;

  printf("infile = ");
  scanf("%s", inname);
  infile = fopen(inname, "r");
  if (infile == NULL) {
    printf("%s%s\n", "Could not open file", inname);
    exit(2);
  }
  printf("outfile = ");
  scanf("%s", outname);
  outfile = fopen(outname, "w");
  if (outfile == NULL) {
    printf("%s%s\n", "Could not open file", outname);
    exit(3);
  }
  poista_kommentit(infile, outfile);
  fclose(infile);
  fclose(outfile);
  return;
}
```

Key subroutine

```
void poista_kommentit(FILE *infile, FILE *outfile) {
  int c, e;

  e = getc(infile);
  while (e != EOF && e == '%') {
    do {
      e = getc(infile);
    } while (e != EOF && e != '\n');
    if (e != EOF) e = getc(infile);
    while (e == ' ') e = getc(infile);
  }
  if (e != EOF) {
    do {
      c = getc(infile);
      while (c != EOF && e != '\\' && c == '%') {
        do {
          c = getc(infile);
        } while (c != EOF && c != '\n');
        if (c != EOF) c = getc(infile);
        while (c == ' ') c = getc(infile);
      }
      putc(e, outfile); e = c;
    } while (e != EOF);
  }
}
```

Using `make` for testing

```
options=-ansi -pedantic -Wall -O3
program=nocomments.c
input=lecture.tex
output=temp.tex
input-base=$(basename $(input))
input-pdf=$(addsuffix .pdf, $(input-base))
output-base=$(basename $(output))
output-pdf=$(addsuffix .pdf, $(output-base))

nocomments:
        @gcc $(options) $(program)
        @echo "$(input)\n$(output)" | ./a.out
        @make -s -f../makefile copy; latex $(input) 1>/dev/null
        @make -s -f../makefile copy; latex $(output) 1>/dev/null
        @diff --brief $(input-pdf) $(output-pdf)
        @rm $(output) $(output-pdf)
        @make -s clean

shell> make nocomments input=lecture.tex output=temp.tex
nocomments.c:9: warning: return type of 'main' is not 'int'
infile = outfile =
Files lecture.pdf and temp.pdf differ
make: *** [nocomments] Error 1
```

Bug reports

I wrote the program in Mar 1997; I expected that it would be used only once, but my co-author took it in production and used it routinely.

Mar 1997: Code is difficult to understand.

Mar 1997: It does not remove comments of the form \omitted{. . . }.

Mar 1997: It may create very long lines, which is dangerous for transmission by email.

Jul 1997: It fails for a two-line (TeX) input `\bye%⤶Torben⤶`. (The output is `\byeTorben⤶`.)

Feb 2008: It removes "comments" inside a verbatim environment even if it shouldn't.

Other criticism

- Bilingualism is disturbing for non-Finnish readers.
- The key subroutine, which does the actual job, is unreadable.
- Compilation produces a warning that is not corrected.
- Unix command-line idiom is not used for inputting the file names.
- The error messages are not printed to `stderr` as they should.
- Declaration-per-line idiom is not followed.
- Operation-per-line idiom is not followed.
- Indentation is not idiomatic for all `if` and `while` statements.

Online exercise (5 min)

Comment the programming style used in the CPH STL component `cphstl::stack`, see files `stack.h++` (bridge class), `stack.c++` (implementation of the bridge class), `list_based_stack.h++` (realization class), and `list_based_stack.c++` (implementation of the realization class).

Summary

- At **macro** level, the main concerns are: obvious organization of the program, easy navigation through the code, and apparent interconnections between different components.
- At **micro** level, the main concerns are: descriptive names, clarity in expressions, straightforward control flow, readability of code and comments, the importance of consistent use of conventions and idioms.