SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya School of Engineering
(formerly K J Somaiya College of Engineering)

Structured Programming Methodology

Making Source Code Readable

By

Dr. Vaibhav P. Vasani

Assistant Professor

Department of Computer Engineering

K. J. Somaiya School of Engineering

Somaiya Vidyavihar University

vaibhav.vasani@somaiya.edu,
vaibhav.vasani@gmail.com

Somaiya TRUST

Making Source Code Readable:

- **Meaningful Naming:** Use descriptive and consistent names for variables, functions, classes, and modules that clearly indicate their purpose and content.
- **Consistent Formatting:** Adhere to a consistent style guide for indentation, spacing, and line breaks. This improves visual clarity and makes the code easier to scan and understand.
- **Clear Structure:** Organize code logically, using functions, classes, and modules to encapsulate related functionality. Minimize nesting and keep functions concise.
- **Comments (Used Wisely):** Employ comments to explain why certain decisions were made, document complex logic, or clarify non-obvious parts of the code. Avoid commenting on what is already clear from the code itself.
- **Refactoring for Clarity:** Regularly refactor code to improve its structure, eliminate redundancy, and simplify complex sections without altering its external behavior.

Documentation

Documentation complements readable code by providing broader context and detailed explanations that cannot be conveyed solely through the code itself. Essential aspects of documentation include:

- **Purpose and Overview:** Clearly explain the software's goals, target audience, and the problem it aims to solve.
- **Installation and Setup:** Provide detailed instructions for installing, configuring, and running the software, including dependencies.
- **Usage Instructions:** Explain how to use the software, including examples of inputs, outputs, and common use cases.
- **API Documentation:** Document all public functions, methods, classes, and their parameters, return values, and potential exceptions.
- **Design and Architecture:** Describe the overall design principles, architectural decisions, and key components of the system.
- **Maintenance and Contribution Guidelines:** For open-source projects, include instructions for potential contributors on how to build, test, and contribute to the codebase.
- **Readme Files:** Provide a concise overview of the project, including its purpose, installation, and basic usage, often in the project's root directory.
- **Keeping Documentation Updated:** Ensure documentation remains accurate and reflects the current state of the codebase. Outdated documentation can be more detrimental than no documentation at all.
- **Target Audience Consideration:** Tailor documentation to its intended audience (end-users, developers, maintainers), providing appropriate levels of detail and technical language.

- **External Documentation (for users/developers)**
- **README file**: Purpose of the project, installation steps, usage, examples.
- **API docs**: Explains functions, classes, and methods (can be auto-generated with tools like Doxygen, Sphinx, Javadoc, or pydoc).
- **Design Docs**: Describe architecture, flowcharts, data models, or algorithms.

- **Internal Documentation (inside the code)**
- **Comments**: Short explanations about tricky logic, assumptions, or formulas.
- **Docstrings** (in Python) or **JavaDoc-style comments** (in Java) for functions, classes, and modules.

Making Source Code Readable

- Readable code is easy to understand, maintain, and extend.
- **Key practices:**
- **Naming conventions**
  - Use descriptive variable, function, and class names (calculate_total() vs. ct()).
  - Follow language-specific style guides (e.g., PEP8 for Python, camelCase for Java/C++).
- **Consistent formatting**
  - Proper indentation.
  - Limit line length (usually 80–100 chars).
  - Group related code with blank lines.
- **Code structure**
  - Break large functions into smaller, reusable ones.
  - Keep functions doing one thing only (Single Responsibility Principle).
  - Organize files logically (modules, packages).
- **Avoid magic numbers**
  - Replace with constants (TAX_RATE = 0.18 instead of 0.18).
- **Error handling & logging**
  - Use meaningful error messages.
  - Don’t hide errors silently.
- **Comments wisely**
  - Don’t over-comment obvious things (i = i + 1 // increment i by 1 is redundant).
  - Explain complex logic, constraints, or references.

Example: Bad vs. Good Code

- Bad Code

```
#include <bits/stdc++.h>
using namespace std;
int f(int x){int y=1;for(int i=1;i<=x;i++)y*=i;return y;}
int main(){int n;cin>>n;cout<<f(n);}
```

- Good Code

```
#include <iostream>
using namespace std;

/**
 * @brief Calculate factorial of a number.
 * @param n Non-negative integer
 * @return Factorial of n
 */
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int main() {
    int number;
    cout << "Enter a number: ";
    cin >> number;

    cout << "Factorial = " << factorial(number) << endl;
    return 0;
}
```

vaibhav.vasani@gmail.com

Bad code

```
int f(int x){int y=1;for(int i=1;i<=x;i++)y*=i;return y;}
```

- Good one

```
/**
 * @brief Calculate factorial
 */
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}
```

- Readable code saves time, reduces bugs, and helps others (and *future you*) understand it quickly.

Common Mistakes in Writing C++ Code

- **Documentation & Readability**
- No comments or too many obvious comments.
- Using meaningless variable names (a, b, c, temp) instead of descriptive ones.
- Writing long, unstructured functions without breaking into smaller parts.
- Mixing logic and input/output in the same function (no separation of concerns).

- **2. Syntax & Formatting**
- Forgetting semicolons ;.
- Misplaced/missing braces { }.
- Inconsistent indentation → makes code hard to read.
- Using magic numbers instead of constants.

```
area = 3.14 * r * r;  // Bad one
area = PI * r * r;    // good one
```

- **Variables & Data Types**
- Not initializing variables (using garbage values).
- Using wrong data type (e.g., int for money when double is needed).
- Mixing signed and unsigned types incorrectly.
- Using global variables unnecessarily.

- **Loops & Conditions**
- Infinite loops due to wrong condition.

```
while (i <= 10) {  // forgot to increment i
    cout << i;
}
```

- Off-by-one errors in loops (<= vs <).
- Misuse of = instead of == in conditions.

```
if (x = 5)  // assigns 5, doesn't compare
```

- **Functions**
- No return statement in non-void functions.
- Wrong parameter types (mismatched during function calls).
- Writing one huge main() instead of modular functions.

- **Pointers & Memory**
- Forgetting to free memory allocated with new.
- Dereferencing null or uninitialized pointers.
- Memory leaks by not using delete[] for arrays.

- **Input/Output**
- Not prompting the user clearly.
- Mixing cin and getline() without handling buffer issues.
- Not validating user input.

- **Error Handling**
- Ignoring possible errors (e.g., divide by zero).
- Not using exceptions where appropriate.

Write code for humans first, then for the computer.

vaibhav.vasani@gmail.com

Important Links

- https://www.software.ac.uk/guide/writing-readable-source-code
- https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_and_Computation_Fundamentals/Programming_Fundamentals_-_A_Modular_Structured_Approach_using_C_(Busbee)/07%3A_Program_Control_Functions/7.05%3A_Documentation_and_Making_Source_Code_Readable