ART OF CLEAN CODE

HOW TO WRITE CODES FOR HUMAN

Roosnam Seefan

Art of Clean Code

How to Write Codes for Human

HELLO WORLD

Overview

Since the first “Hello World” program by Dennis Ritchie, the world has observed dramatic changes in the digital landscape. With the growing advancement of technology, the impact of digitization is unavoidable in all spheres of modern society. Hence efficient program is the demand of the hours which also needs to ensure security as well as scalability.

The modern era has identified software development as an ever-developing process. In other words, this also ensures the scalability of the program to some extent. Hence, software development needs to go beyond the spatial domain as well as the temporal domain. This phenomenon ignites the importance of code readability in the software industry.

Therefore, readability of the code is becoming the vital feature of the product in software development. Hence, code readability needs to be one of the very first things to learn and adapt for the young programmers. Furthermore, besides the fundamental courses, such an important topic should be cultured in the higher-level courses and industrial pieces of training. However, might be due to the negligence, lessons, and tips on code readability at the higher and professional level are missing.

On a different note, comments are considered as the explanation of a programming code that is applied by the programmers to make the code human-readable. And hence, a lot of code readability discussions ended up with a chicken and egg situation; which is the importance of comment. Till date, this is an inconclusive debate since commenting strategy includes both benefits and setbacks to the code. Well, those who are working in the software development industry, most likely have already experienced such discussions.

If the developers' community is asked about the necessity of comments in the codes, we will get diversified feedback. Some might say comments are must for the developers while the others would state it's a lost cause.

Meanwhile, if I was asked about my opinion on the necessity of comments, my stand is not purely black and white. To me, if you feel commenting adds value to your code, you shouldn't give any second thought and need to incorporate commenting effectively. Apparently, some basic and fundamental tips are very often violated by the codes which make comments as the black sheep in the code. Our discussion will mostly focus on what not to do while commenting on this book.

Moreover, in order to make a clean and readable code, we need to ensure,

DEVIL'S ADVOCATE

Introduction

Despite the mixed opinions, as far as commenting is concerned, the standpoint of this book is quite clear. Rather directly arguing on the importance and necessity of comment in a code, we will try to focus on the impacts of proper and improper comments. Therefore, the main discussion in this chapter will be on the do's and don'ts of commenting. Hence, we are anticipating efficient comment is a must component for readability. Thus, the book highly recommends to patch comments with the code for readability.

The following sections of this chapter will address the impacts of good and bad comments from both individual and organizational perspectives. Along with these discussions, the chapter will also discuss on documentation.

Impacts: Organization's Viewpoint

Before getting into the personal interest, let us first see when and why commenting becomes the angel for an organization. We will also see when and why commenting takes the devil's role for the organization. The first view is that commenting and documentation will protect the firm from

The opposing view is that there are more effective ways to mitigate the risk of bad and disappearing programmers (like mandated shared ownership of code and sufficient oversight), that comments are not necessary for clarity and can be dangerous if not kept up to date.

However, comments and documentation of the application acts as training material. These approaches can help new recruits to learn the existing's faster. Hence, the newcomers will be tacitly oriented as the documentation can be used to give lessons about the applications. Therefore, comments and documentation act as the training manual and covers the syllabus as well as provide notes to educate.

Furthermore, documentation of an application may also be introduced to the marketing and sales personnel to understand what the functional aspects and capabilities of the program. This sort of information always helps the marketing and sales

department to design their strategies while promoting and to determine what can be promised and what can be fulfilled. This will also ensure that customers will be promised with all the deliverables. This will also reduce the chances of turmoil with the customers as marketing and sales executives might not require to promote with over and/or under promises.

Impacts: Individual's Viewpoint

Though programming is considered to be pretty much materialist, however there is another analogy where computer programmers are referred to be artists. Hence, programmer's artworks are framed in the form of programming codes. Still, some may oppose the idea that art has to be something which involves emotion and passion. In such cases, a passionate programmer may advocate that there exists a different type of insanity with the coder that makes him/her work on the problem days and nights. So, codes are nothing but the personal expression and rational thoughts of a programmer.

This a common phenomenon where a programmer needs to reuse own codes. Moreover, a coder's rational thought for solving any problem is contextual. Hence, it's a must to relate previous constraints with the current requirements prior to the reuse. So, commenting in the code will always help you to recall the context or the thought process you had while developing the code. While reusing, comments in the code will also show you the gaps of the existing code to the current requirements. Meanwhile, while reusing, one may use the entire script or partial. In both cases, a well commented code tells you all the dependencies and links a code exhibits in a flash.

MORE IS NOT ALWAYS GOOD

Introduction

In the previous chapter, we have learnt why to comment in the code. In this part of the book we will discuss the common misconceptions while commenting. This is very natural that if comments are good, the first beneficiary will be the coder himself. It is also true the coder himself will be the first victim for the bad comments.

Understanding the Value of a Comment

It is observed that whenever someone gets introduced to something new, he or she is more inclined to the usage of it. However, we need to understand the value of a particular comment before applying. Consider the following example where while declaring the coder uses comments to describe the datatype of the variable.

```
int x; // x is an interger type variable
```

Here, we need to first identify the value of the comment. From any aspect such comment doesn't add any value to the code. This comment conveys the same understanding as the code part, which is nothing but repetition. Such comments discomfort the reader and hence, we need to ensure to comment only on those thoughts that the code fails to express.

Don't Overdo

While commenting in the code, we need to ensure that its not overdone. Consider the following example,

```
// get the username using get_username method where user ID is the argument
username = get_username(user_ID);

// if username matches with current user
if (username == current_user) {
    // allow to edit profile by setting edit_profile to be true
    edit_profile = 1;
}
```

In this particular case, comments before each statement make the comments to be less productive. If you are trying to explain the purpose of this particular portion, it's better to combine the comments into one and write it just before the block starts as follows,

```
// enable current user to edit profile
username = get_username(user_ID);
if (username == current_user) {
    edit_profile = 1;
}
```

INDENTATION & SPACING

Introduction

Indentation in the code is one of the most essential features for readability. Still, coders are often neglecting it while being reluctant to follow indentation systematically. Hence, it has been seen over the years that long and complex codes become abandoned just due to defective and inconsistent indentation styling.

In a programming environment, code indentation is mainly considered as the styling method in order to produce a readable and understandable program. Furthermore, adding the aesthetic values, indentation can be interpreted as the process a programmer follows to divide and organize the source code into blocks. Therefore, this approach will ensure that the reader to have a comfortable journey while sailing through the code for his or her understanding. Even, when someone revisits his/her source code, consistency in the indenting style can save lots of time of that particular programmer.

Importance of Indentation

Though being consistent with the indentation style ensures the code to be pretty looking, however, code indenting is also used to indicate a specific block associated with any comment. Yet, programmers often bristle on the importance of indentation, some programming environments such Python takes indentation quite seriously. Indentation has been given additional attention in Python such that even extra spacings before any statement may cause compilation error to the program.

Nevertheless, a good number of programmers are also practicing code indentation naturally.

While coding, a programmer needs to bear in mind that though following any particular indentation styling may require additional time, however it will save more time in the long run. Hence, if you are required to give more time while coding, this will benefit you and your readers in the long terms. However, just to meet the deadline or for any other reasons, if the indentation styles are not followed properly, rectification of the code may incur at a later stage.

We should remember that a code which follows the proper indentation style makes it:

In a nutshell, a programmer should remember that a code is coded once however this code has the chance to be read several times in the later phases.

Indentation Styles

From the previous sections we have realized that indentation style plays a vital role from the code's readability view point. On the contrary, there exists no universal indentation style which can be followed by everyone or can be considered to be the standard styling scheme. The fact of the matter is any indentation styling is a good style in programming as long as it has been followed throughout the code. Hence, indentation style enigma can be resolved by being consistent.

We may now say that in order to ensure a readable code, the best indentation style needs to be always consistent. A practical implication can be if a new member joins any ongoing development project and becomes the part of a team, as far as indentation is concerned, the new member needs to follow the exact same indentation styling that has been followed for this project.

In the following section we will see three different indentation styles for the same block of code. Again, these are few of the many different indentation styles and we are just using these three for our visual understandings. As discussed earlier, indentation mainly divides the code to different parts, which is commonly known as the blocks. Apart from this, in any indentation styling, statements of equal importance or same level will also need to indented with exact indented space.

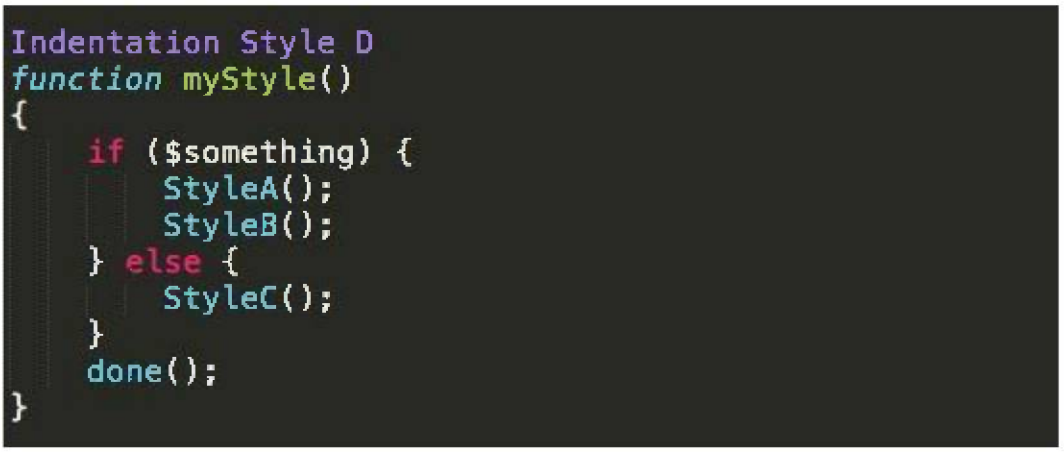
```
Indentation Style A
function myStyle() {
    if ($something) {
        StyleA();
        StyleB();
    } else {
        StyleC();
    }
    done();
}
```

```
Indentation Style B
function myStyle()
{
    if ($something)
    {
        StyleA();
        StyleB();
    }
    else
    {
        StyleC();
    }
    done();
}
```

```
Indentation Style C
function myStyle()
{   if ($something)
    {   StyleA();
        StyleB();
    }
    else
    {   StyleC();
    }
    done();
}
```

In the Indentation Style A, the opening bracket "{" goes on the same line as the function definition and control structure, where the opening bracket goes to the next line in both cases in Style B.

Unless observed closely, one may fail to identify the features of styling in the Indentation Style A, B, C since they are somewhat similar. This also tells us that there are styles which are derived from other existing styles. For instance, the following Style D is a derived one from the previous example styles. Also known as PEAR standard, in Style D the opening bracket "{" goes on the same line as the function definition and goes to the next line in control structure. Hence, we as a programmer always need to be fully clear about the indentation styling rules before applying. Moreover, for the spacing purpose, it is always advised to use tabs instead of spacebars for indentations.



Indentation Settings

The first sign that strikes the mind whenever someone refers to indentation is the spacebar. Hence, we feel on hitting the spacebar various numbers of time before different levels in the code. On the other hand, if someone is familiarized with the processing applications, he or she might be feeling to use tabs for indenting purposes.

No matter you use the spacebars or the tabs concerning indentation, you are doing good. However, the concern is raised whenever you start mixing these two keys in the code. This is because length settings are the opposite for spacebar and tabs. For spacebar, spacing length is fixed for all the systems, where tab length is a user setting dependent. Hence, one can set tabs to be 4 spaces where others may set it at 6. Whenever tabs and spaces are mixed in the code, the code becomes messy in the screen of other who has different tabs length.

The following code looks clumsy due to the improper use of spaces and tabs.

```
void addition_subtraction_multiplication(int a, int b, int opt){
 // two numbers will be added if user makes opt variable to be 1
  if (opt==1){
     int result = a+b;
      cout << "Result is " << result << endl;
 }
   // two numbers will be subtracted if user makes opt variable to be 2
   else if (opt==2){
     int result = a-b;
      cout << "Result is " << result << endl;
  }
    // for other values in opt, multiplication will be executed
    else{
     int result = a*b;
      cout << "Result is " << result << endl;
  }
}
```

Therefore, for a well-organized readable code, we should not combine spaces and tabs for indentation purpose. An implementation of this concept has been shown in the following code which is a reproduction of the previous one.

```
void addition_subtraction_multiplication(int a, int b, int opt){
    // two numbers will be added if user makes opt variable to be 1
    if (opt==1){
        int result = a+b;
        cout << "Result is " << result << endl;
    }

    // two numbers will be subtracted if user makes opt variable to be 2
    else if (opt==2){
        int result = a-b;
        cout << "Result is " << result << endl;
    }
    // for other values in opt, multiplication will be executed
    else{
        int result = a*b;
        cout << "Result is " << result << endl;
    }
}
```

CLEAN NAMES

Introduction

If a code explains its working mechanism, comments become groundless for it. However, one may argue that since I'm using comments to explain my code, why should I bother on self-describing codes. Again, we can reduce a good amount of reading time if back and forth movement is avoided to read the comments. Moreover, comments become the main mess of code if they are properly managed with the version changes. Even ensuring the up to date comments needs extra efforts and time. We can avoid such hurdles in the code if the code is self-explaining.

Word Boundaries

While naming a variable or a method, we may need to use more than a word in order to make it descriptive. Names containing more than a word makes the code more readable and understandable. However, if no technique is applied to distinguish words, the names will be meaningless and in cases misleading. In order to avoid such situations, we need to adopt one of the two popular techniques to set the boundaries. These popular approaches are called camelCase and underscoring approach. In the camelCase approach, first alphabet in each word is capitalized, other than the first word. Meanwhile, underscores are used between words to differentiate one from another in the underscoring approach.

Name to Exhibit Intention

A variable name needs to be explicit in the code in order to make readable. Hence, the name will exhibit the coder intention as well as the purpose of the variable. If the name is self-explanatory, we can easily ignore the comment requires to express its purpose.

For instance, consider the following example, where a wrong approach is used to name a variable which requires to keep track of the number of users. If you name the variable to be i, you are required to use the comment to express the functionality of this variable. Furthermore, while reading the code, whenever i appears, the reader needs to recall the meaning or the purpose over and over. This introduces extra effort for the user, as well as additional reading time. However, if we just make the variable name self-explanatory, such as UserCount, doesn't require the reader to memorize the purpose of the variable. Similarly, while dealing with locations, it's better to user destination and source as the variable names rather loc1 and loc2.

Avoid Confusing Names

While naming a variable or a method, we need to ensure that the name wouldn't confuse or mislead the reader. For instance, if you are using a variable named user to hold the name of the user, its better to name it UserName. Moreover, if required to hold the first and last names of the user, the variables should be named UserFirstName and UserLastName instead of fname and lname.

In short, a coder needs to remember that he is writing the code for the human to understand. Moreover, it's better not to glue keywords or reserved words in the name. If we name a method ObjectLocation that calculates the location of any entity, using Object will introduce confusion.

Avoid Abbreviated Names

While naming, it's better not to use abbreviated wordings, even it carries obvious meaning. Hence, we should use the variable name DateOfBirth instead of DoB in order to avoid any confusion. Similarly, partial abbreviated names are also advised to be ignored. Therefore, the variable names should be UserFirstName and UserLastName instead of UserFN and UserLN.

Furthermore, it's better to ignore the single alphabet variable names. Hence, instead of using variable names as i, x, a; it is better to use their purpose as the name of the variable.

Names for Classes and Methods

If you are coding in an object-oriented environment, you should know that we often use objects for modeling purpose. That exists in physical form is represented as an object in coding environments. Hence, it is rational to name the classes in nouns or in noun phrases.

On the other hand, we use a method or function to perform any task. Hence, it's rational to name the method using a verb. Hence, methods, named as getInput(), setMessage(), etc, are pointing.

Furthermore, there are some general activities a coder needs to perform in the code. Such activities are advised to be defined as method rather than historical variable. For instance, in order to check connection, the most intuitive method would be calling a method named isAlive() sounds more rational than keeping the status in an array.

Consistent Naming Scheme

The naming scheme for variables and methods will ensure comfort and ease in understanding code. Hence, in order to assure the code to be well readable, we need to be consistent in the naming. For instance, if we have a common method in several classes, names for this method for all the classes need to be the same.

Moreover, in case of temporary variables, the good practice is also to be consistent. For instance, in case of a loop counter, a well readable code will be using a variable name counter throughout the code.

Moreover, naming convention should maintain the credibility of the code. Hence, humorous names in variables and methods may sound funny for the first read, however, a code may lose credibility since such scheme doesn't add any value for the readers in terms of understanding the code.

Similar to indentation, again in variable and method naming, there is no prescribed "best" method. However, likewise indentation, for naming we should be consistent.

CODE GROUPING

Introduction

As mentioned earlier, the clean code needs to be well organized for the readers. Such code involves the art of grouping statements. We have methods and classes in the coding environment to group statements based on the tasks. Benefits and features of classes and functions are out of the scope of this book; however, I hope readers are already familiar with these features.

The “One Thing” Function

Function is one of most popular approaches to split a long code into blocks. Programmers usually apply functions for debugging and testing purpose. Apart for the naming convention discussed earlier, we should develop functions in the code so that they accomplish the main object. Well, in the programming arena, the main object for any function is termed to be the "one thing". However, what is this one thing and how to determine whether any function meets the “one thing” or not.

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

//functions to perform addition, subtraction and multiplication on two numbers
//user selects the operation
void addition_subtraction_multiplication(int a, int b, int opt){
    // two numbers will be added if user makes opt variable to be 1
    if (opt==1){
        int result = a+b;
        cout << "Result is " << result << endl;
    }

    // two numbers will be subtracted if user makes opt variable to be 2
    else if (opt==2){
        int result = a-b;
        cout << "Result is " << result << endl;
    }
    // for other values in opt, multiplication will be executed
    else{
        int result = a*b;
        cout << "Result is " << result << endl;
    }
}

int main()
{

    addition_subtraction_multiplication(1,2, 3);

}
```

In this code, user can perform three basic mathematical operations over two values. The code uses a variable opt which is used to select user opinion. Looking at the arguments, it's obvious that this function is performing multiple tasks within while ignoring the One Thing concept. Moreover, though it a very tiny code, yet it looks messy.

Now look at the following code, which is a refined one.

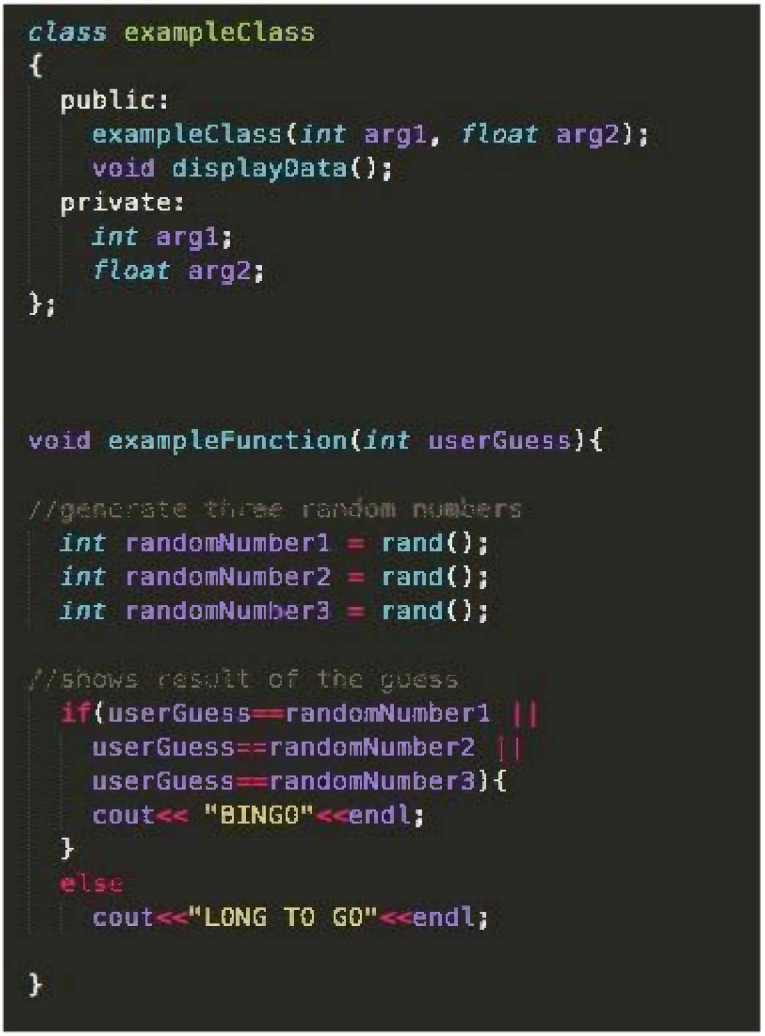
```
#include <iostream>
using namespace std;

// Function to add two values
void addition(int a, int b){
        cout << "Result is " << a+b << endl;
}

// Function to subtract one value from another
void subtraction(int a, int b){
        cout << "Result is " << a-b << endl;
}

// Function to add two values
void multiplication(int a, int b){
        cout << "Result is " << a*b << endl;
}

int main()
{
    addition(1,2);
}
```

In the revised version of the code, we have divided the messy function into 3 functions. Now, we are getting a clear idea about the code with little efforts. Meanwhile we can also straightway modify any section of the code without affecting another function.

Embrace the Brackets

If we were questioned, "what makes unobscured definition for function and class?". Brackets won't be the perfect answer, however, it's not the incorrect one as well. In fact, overlooking the odd programming environments, brackets are the proper answer. Hence, we can feel the implicit importance of bracket for grouping codes.

The following codes show function and class definitions fused with braces.



```
class exampleClass
{
  public:
    exampleClass(int arg1, float arg2);
    void displayData();
  private:
    int arg1;
    float arg2;
};



void exampleFunction(int userGuess){

//generate three random numbers
  int randomNumber1 = rand();
  int randomNumber2 = rand();
  int randomNumber3 = rand();

//shows result of the guess
  if(userGuess==randomNumber1 ||
    userGuess==randomNumber2 ||
    userGuess==randomNumber3){
    cout<< "BINGO"<<endl;
  }
  else
    cout<<"LONG TO GO"<<endl;

}
```

In case of clean coding, braces are the formatting sugar which defines the group of related statements and indicates the starting and ending points. Applying braces, we can even control the domain of a variable.

Though the introduction of braces does increase the code's readability, however, it also has some limitations. Since with brackets, we may restrict the domain of a variable and trigger vulnerabilities to the code. Even a program may crash due to the improper use of brackets.

Make Blocks

In addition to the previous discussion, grouping similar statements makes a code more reading friendly. As we may agree that a function can be divided into smaller tasks or sub-tasks. Even we may have several lines to accomplish a sub-task or we may have several statements for similar type of task in the code. In order to give the reader a quick understanding of these statements, we can group them in blocks of code. In such approach each block will have some generic features or functionalities. More often, if codes are organized in blocks, a single comment may be enough to express the functionalities of several statements. Following example states the visual presentation of several statements in a block.

```
class exampleClass
{
  public:
    exampleClass(int arg1, float arg2);
    void displayData();
  private:
    int arg1;
    float arg2;
};



void exampleFunction(int userGuess){

//generate three random numbers
  int randomNumber1 = rand();
  int randomNumber2 = rand();
  int randomNumber3 = rand();

//shows result of the guess
  if(userGuess==randomNumber1 ||
    userGuess==randomNumber2 ||
    userGuess==randomNumber3){
    cout<< "BINGO"<<endl;
  }
  else
    cout<<"LONG TO GO"<<endl;

}
```

Capitalizing Query Statements

In case of database driven applications, query statements interplay a big role in the code's readability. Hence, if we have raw SQL queries in the code, the best practice is to make them also readable and clean.

As we know that SQL statements are case insensitive for keywords. However, a general practice coder applies to capitalize on the keywords in the query statements in order to differentiate keywords from user tables and its schemas.

```
SELECT UserID, UserName
FROM UserData
WHERE UserID = 'U123';

UPDATE UserData
SET LogInTime = NOW()
WHERE UserName = 'Abc';
```

REUSABILITY & REDUNDANCY

Introduction

Removing duplicates is another smart approach to make a code efficient. Moreover, reusable code saves substantial coding time for any developers. Hence this chapter will focus how can we reuse codes in order to enhance the readability.

Templates and Includes

While developing any application, repetitive tasks are a very common phenomena. If same assignment is given to two different coders, one who has years of experience and the other one who is relatively young in coding, it's most likely the seasoned one will deliver early. If you are thinking that the experience one knows more hence solves the problem early, you are most likely making a mistake. If you ignore the experience issue, the senior coder delivers the product early just because he has larger number of resources or his archive is richer than the young coder. The moral of this is whenever we solve any problem we will most likely use a lot of codes that we had prepared previously. Hence, a better reable code makes life easier whenever we need to reuse it. And a smarter coder doesn't code the same thing over and over, rather reuse the code to get the most out of it.

One of such example could be explaining with the help of Template in C++. As you may recall, with this feature, we may use the same code functionalities for various variables. In Templates we just pass the data type as parameter in the code instead of coding the same functionalities several times for various data types.

On the other hand, if we consider web-based applications, functionalities and features of this application may be expressed via several web pages. And interesting all these will always have

few parts to be common elements. And the most obvious common parts of the web pages are the header and footer sections. In such case, the most inefficient approach would be coping the same code in all the pages considering that are these pages will be having the name header and footer section. Now, let's assume a scenario where a site contains 100 pages and you have the same header and footer code copied in 100 places. However, you have realized there is a typo in the footer. Now you just have two choices, correct the typo in 100 pages manually or leave the typo as it is. Definitely, the second option is unprofessional and the first one is cumbersome. This scenario is the best resemble for code duplicacy setbacks.

Such situations can be avoided by reusing or refering the code instead of duplicating. If we consider a web application on Laravel, the route directory contains all the route files which will be automatically loaded by the laravel framework.

LIMIT LENGTH

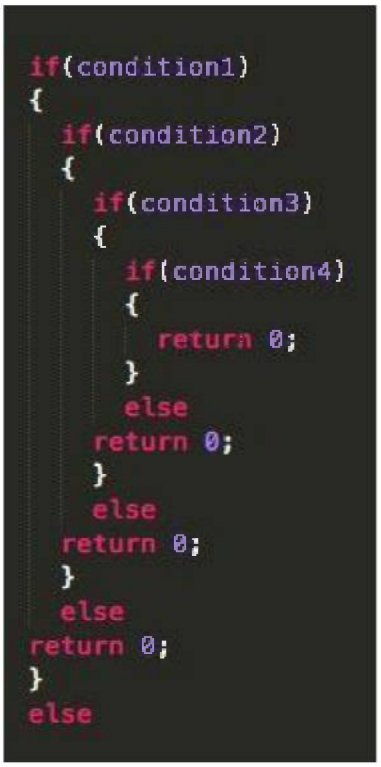
Introduction

It is often referred that readability of the source code prevails in the reader's eyes. Let's image a gigantic sized picture where the viewer needs to see the picture in parts. Later, he/she requires to imagine the whole picture fusing all the parts. Ignoring the odds, such an approach definitely discomforts the viewer. Similarly, whenever we read a code we want to see everything in front of the eye, rather sliding the screen while remembering.

Hence, this chapter discusses the line length of the code and how to reduce the length to make the code more reading friendly. Moreover, deep nesting conditions are also discussed here. We will also learn a few tips to eliminate the deep nesting loops.

Avoid Deep Nesting

Now let us consider a situation where the code contains deep nesting conditional statements. Such state in the code makes it harder to understand or follow. Concerning the state of code's readability, we need to make the adjustments and necessary modifications to decrease this nesting levels.



```
if(condition1)
{
  if(condition2)
  {
    if(condition3)
    {
      if(condition4)
      {
        return 0;
      }
      else
    return 0;
    }
    else
  return 0;
  }
  else
return 0;
}
else
```

Again, we have reached a situation or rather to call a problem state where the solution is not well bounded. As we may realize the deep nesting statements obstruct the ease of understanding, however there is no explicit definition on the non-deep nesting. Taking this opportunity, I would advocate on simpler nesting as

the acceptable one. The simpler nesting can be seen as a two leveled nesting. Even we all have or will have the experience on dealing with 3D array, hence even Three Level deep nesting may also be considered to be readable.

However, it the level grows to four, things will get tough to be read or to trace. Hence, again we need to devise a method that limits the growth of the deep nestings. I personally follow two tricks whenever I have to stop the growth, where complexity of the deepening loops may trigger which approach to be followed. The first one is relatively easier than the second one, since it is quite straightforward. The tip here is to relocate the inner loops into a separate function. The limitation of this approach is such method may introduce additional functions which might require extra efforts to be maintained as well. Furthermore, in the second approach, we need merge or combine two or even three nested conditions into one statement and reproduce all the possible states and their actions.

Trim the Long Lines

While coding for human, I would like to first highlight on the eyes and its natural turns. In medical science, this is an established concept that wider columns strain the eye and which will eventually make the reading difficult.

Moreover, narrow sized columns are easier to be parsed as compared to the wider ones. This is the reason why we have newspaper articles in narrow columns. Likewise, while coding, long-lined statements may hamper code readability. Bearing this in mind, while coding we should avoid the long-lined statements. Even if we have some long lines in our code, we can always slash them and present in multiple lines.

END-WORDS

Clean coding happens to be one of major concerns in today's open-source era where readability of the code is becoming the vital feature in software development. Besides, code readability needs to be one of the very first things to learn and adapt for the young programmers. Hence this book introduces clean code along with few tips and tricks. If adequate steps on readability are followed while developing a program, the impact of any code would be much bigger than otherwise. Moreover, bug fixing will also be much easier for such codes.

Here, in the book, I have presented my perceptions that I have obtained for the last 12 years both in industry and academia.