# Software Engineering - I

An Introduction to Software Construction Techniques for Industrial Strength Software

Chapter 10 – Good Programming Practices and Guidelines

### **Maintainable Code**

As we have discussed earlier, in most cases, maintainability is the most desirable quality of a software artifact. Code is no exception. Good software ought to have code that is easy to maintain. Fowler says, "Any fool can write code that computers can understand, good programmers write code that humans can understand." That is, it is not important to write code that works, it is important to write code that works and is easy to understand so that it can be maintained. The three basic principles that guide maintainability are: simplicity, clarity, and generality or flexibility. The software will be easy to maintain if it is easy to understand and easy to enhance. Simplicity and clarity help in making the code easier to understand while flexibility facilitates easy enhancement to the software.

## **Self Documenting Code**

From a maintenance perspective, what we need is what is called *self documenting code*. A *self documenting code* is a code that explains itself without the need of comments and extraneous documentation, like flowcharts, UML diagrams, process-flow state diagrams, etc. That is, the meaning of the code should be evident just by reading the code without having to refer to information present outside this code.

The question is: how can we write code that is self-documenting?

There are a number of attributes that contributes towards making the program self documented. These include, the size of each function, choice of variable and other identifier names, style of writing expressions, structure of programming statements, comments, modularity, and issues relating to performance and portability.

The following discussion tries to elaborate on these points.

## **Function Size**

The size of individual functions plays a significant role in making the program easy or difficult to understand. In general, as the function becomes longer in size, it becomes more difficult to understand. Ideally speaking, a function should not be larger than 20 lines of code and in any case should not exceed one page in length. From where did I get this number 20 and one page? The number 20 is approximately the lines of code that fit on a computer screen and one page of course refers to one printed page. The idea behind these heuristics is that when one is reading a function, one should not need to go back and forth from one screen to the other or from one page to the other and the entire context should be present on one page or on one screen.

## **Identifier Names**

Identifier names also play a significant role in enhancing the readability of a program. The names should be chosen in order to make them meaningful to the reader. In order to understand the concept, let us look at the following statement.

if (x==0) // this is the case when we are allocating a new number

In this particular case, the meanings of the condition in the if-statement are not clear and we had to write a comment to explain it. This can be improved if, instead of using x, we use a more meaningful name. Our new code becomes:

if 
$$(AllocFlag == 0)$$

The situation has improved a little bit but the semantics of the condition are still not very clear as the meaning of 0 is not very clear. Now consider the following statement:

We have improved the quality of the code by replacing the number 0 with a named constant NEW\_NUMBER. Now, the semantics are clear and do not need any extra comments, hence this piece of code is self-documenting.

## **Coding Style Guide**

Consistency plays a very important role in making it self-documenting. A consistently written code is easier to understand and follow. A coding style guide is aimed at improving the coding process and to implement the concept of standardized and relatively uniform code throughout the application or project. As a number of programmers participate in developing a large piece of code, it is important that a consistent style is adopted and used by all. Therefore, each organization should develop a style guide to be adopted by its entire team.

This coding style guide emphasizes on C++ and Java but the concepts are applicable to other languages as well.

### Naming Conventions

Hungarian Notation was first discussed by Charles Simonyi of Microsoft. It is a variable naming convention that includes information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc). Every company and programmer seems to have their own flavor of Hungarian Notation. The advantage of Hungarian notation is that by just looking at the variable name, one gets all the information needed about that variable.

Bicapitalization or camel case (frequently written CamelCase) is the practice of writing compound words or phrases where the terms are joined without spaces, and every term is capitalized. The name comes from a supposed resemblance between the bumpy outline of the compound word and the humps of a camel. CamelCase is now the official convention for file names and identifiers in the Java Programming Language.

In our style guide, we will be using a naming convention where Hungarian Notation is mixed with CamelCase.

## **General Naming Conventions**

### General naming conventions for Java and C++

1. Names representing types must be nouns and written in mixed case starting with upper case.

```
Line, FilePrefix
```

2. Variable names must be in mixed case starting with lower case.

```
line, filePrefix
```

This makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line;

3. Names representing constants must be all uppercase using underscore to separate words.

```
MAX ITERATIONS, COLOR RED
```

In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice. This form is both easier to read, and it ensures a uniform interface towards class values.

```
int getMaxIterations()// NOT: MAX_ITERATIONS = 25
{
   return 25;
}
```

4. Names representing methods and functions should be verbs and written in mixed case starting with lower case.

```
getName(), computeTotalWidth()
```

5. Names representing template types in C++ should be a single uppercase letter.

```
template<class T> ...
template<class C, class D> ...
```

- 6. Global variables in C++ should always be referred to by using the :: operator. ::mainWindow.open() , ::applicationContext.getName()
- 7. Private class variables should have \_ suffix.

```
class SomeClass
{
  private int length_;
  ...
}
```

Apart from its name and its type, the *scope* of a variable is its most important feature. Indicating class scope by using \_ makes it easy to distinguish class variables from local scratch variables.

8. Abbreviations and acronyms should not be uppercase when used as name.

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type would have to be named dVD, hTML etc. which obviously is not very readable.

9. Generic variables should have the same name as their type.

Non-generic variables have a *role*. These variables can often be named by combining role and type:

```
Point startingPoint, centerPoint;
Name loginName;
```

10. All names should be written in English.

```
fileName; // NOT: filNavn
```

- 11. Variables with a large scope should have long names, variables with a small scope can have short names. Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables for integers are *i*, *j*, *k*, *m*, *n* and for characters *c* and *d*.
- 12. The name of the object is implicit, and should be avoided in a method name.

```
line.getLength(); // NOT: line.getLineLength();
```

The latter seems natural in the class declaration, but proves superfluous in use.

### Specific Naming Conventions for Java and C++

1. The terms *get/set* must be used where an attribute is accessed directly.

```
employee.getName();
matrix.getElement (2, 4);
employee.setName (name);
matrix.setElement (2, 4, value);
```

2. is prefix should be used for boolean variables and methods.

```
isSet, isVisible, isFinished, isFound, isOpen
```

Using the is prefix solves a common problem of choosing bad boolean names like status or *flag. isStatus* or *isFlag* simply doesn't fit, and the programmer is forced to chose more meaningful names.

There are a few alternatives to the *is* prefix that fits better in some situations. These are has, can and should prefixes:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

3. The term *compute* can be used in methods where something is computed.

```
valueSet.computeAverage(); matrix.computeInverse()
```

Using this term will give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result.

4. The term *find* can be used in methods where something is looked up.

```
vertex.findNearestVertex(); matrix.findMinElement();
```

This gives the reader the immediate clue that this is a simple look up method with a minimum of computations involved.

5. The term *initialize* can be used where an object or a concept is established.

```
printer.initializeFontSet();
```

6. List suffix can be used on names representing a list of objects.

```
vertex (one vertex), vertexList (a list of vertices)
```

Simply using the plural form of the base class name for a list (matrixElement (one matrix element), matrixElements (list of matrix elements)) should be avoided since the two only differ in a single character and are thereby difficult to distinguish. A *list* in this context is the compound data type that can be traversed backwards, forwards, etc. (typically a Vector). A plain array is simpler. The suffix Array can be used

to denote an array of objects.

7. *n* prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```

The notation is taken from mathematics where it is an established convention for indicating a number of objects.

8. No suffix should be used for variables representing an entity number. tableNo, employeeNo

The notation is taken from mathematics where it is an established convention for indicating an entity number. An elegant alternative is to prefix such variables with an *i*: iTable, iEmployee. This effectively makes them *named* iterators.

9. Iterator variables should be called *i*, *j*, *k* etc.

```
while (Iterator i = pointList.iterator(); i.hasNext(); ) {
   :
}

for (int i = 0; i < nTables; i++) {
   :
}</pre>
```

The notation is taken from mathematics where it is an established convention for indicating iterators. Variables named j, k etc. should be used for nested loops only.

10. Complement names must be used for complement entities.

```
get/set, add/remove, create/destroy, start/stop, insert/delete, increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide

Reduce complexity by symmetry.
```

11. Abbreviations in names should be avoided.

There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated. Never write:

```
cmd instead of command
cp instead of copy
pt instead of point
comp instead of compute
init instead of initialize
etc.
```

Then there are domain specific phrases that are more naturally known through their acronym or abbreviations. These phrases should be kept abbreviated. Never write:

```
HypertextMarkupLanguage instead of html
CentralProcessingUnit instead of cpu
PriceEarningRatio instead of pe
etc.
```

- 12. Negated boolean variable names must be avoided.
- © Copy Rights Virtual University of Pakistan

### Software Engineering-I (CS504)

The problem arise when the logical **NOT** operator is used and double negative arises. It is not immediately apparent what !isNotError means.

13. Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do. This increases readability. Makes it clear what the unit should do and especially all the things it is *not* supposed to do. This again makes it easier to keep the code clean of side effects. Naming pointers in C++ specifically should be clear and should represent the pointer type distinctly.

```
Line *line //NOT Line *pLine; or Line *lineptr; etc
```

## File handling tips for Java and C++

1. C++ header files should have the extension .h. Source files can have the extension .c++ (recommended), .C, .cc or .cpp.

```
MyClass.c++, MyClass.h
```

These are all accepted C++ standards for file extension.

2. Classes should be declared in individual header files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to. All definitions should reside in source files.

```
class MyClass
{
  public:
    int getValue () {return value_;} // NO!
    ...
  private:
    int value_;
}
```

The header files should declare an interface, the source file should implement it. When looking for an implementation, the programmer should always know that it is found in the source file. The obvious exception to this rule is of course inline functions that must be defined in the header file.

- 3. Special characters like TAB and page break must be avoided. These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.
- 5. The incompleteness of split lines must be made obvious.

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give rigid rules for how lines should be split, but the examples above should give a general hint. In general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

### Include Files and Include Statements for Java and C++

1. Header files must include a construction that prevents multiple inclusion. The convention is an all uppercase construction of the module name, the file name and the h suffix.

```
#ifndef MOD_FILENAME_H
#define MOD_FILENAME_H
:
#endif
```

The construction is to avoid compilation errors. The construction should appear in the top of the file (before the file header) so file parsing is aborted immediately and compilation time is reduced.

#### **Classes and Interfaces**

Class and Interface declarations should be organized in the following manner:

- 1. Class/Interface documentation.
- 2. class or interface statement.
- 3. Class (static) variables in the order public, protected, package (no access modifier), private.
- 4. Instance variables in the order **public**, **protected**, package (no access modifier), private.
- 5. Constructors.
- 6. Methods (no specific order).

### Statements in Java and C++

### **Types**

1. Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
floatValue = (float) intValue;  // NOT: floatValue = intValue;
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

- 2. Types that are local to one file only can be declared inside that file.
- 3. The parts of a class must be sorted *public*, *protected* and *private*. All sections must be identified explicitly. Not applicable sections should be left out. The ordering is "most public first" so people who only wish to use the class can stop reading when they reach the protected/private sections.

#### **Variables**

- 1. Variables should be initialized where they are declared and they should be declared in the smallest scope possible.
- 2. Variables must never have dual meaning. This enhances readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.
- 3. Class variables should never be declared public. The concept of information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ struct). In this case it is appropriate to make the class' instance variables public.
- 4. Related variables of the same type can be declared in a common statement. Unrelated variables should not be declared in the same statement.

```
float x, y, z;
float revenueJanuary, revenueFebrury, revenueMarch;
```

The common requirement of having declarations on separate lines is not useful in the situations like the ones above. It enhances readability to group variables.

5. Variables should be kept alive for as short a time as possible. Keeping the operations on a variable within a small scope, it is easier to control the effects and side effects of the variable.

- 6. Global variables should not be used. Variables should be declared only within scope of their use. Same is recommended for global functions or file scope variables. It is easier to control the effects and side effects of the variables if used in limited scope.
- 7. Implicit test for  $\theta$  should not be used other than for boolean variables and pointers.

It is not necessarily defined by the compiler that ints and floats 0 are implemented as binary 0. Also, by using explicit test the statement give immediate clue of the type being tested. It is common also to suggest that pointers shouldn't test implicit for 0 either, i.e. if (line == 0) instead of if (line). The latter is regarded as such a common practice in C/C++ however that it can be used.

### **Loop structures**

1. Only loop control statements must be included in the *for()* construction.

2. Loop variables should be initialized immediately before the loop.

- 3. The use of *do* .... *while* loops should be avoided. There are two reasons for this. First is that the construct is superflous; Any statement that can be written as a *do* .... *while* loop can equally well be written as a *while* lopp or a *for* loop. Complexity is reduced by minimizing the number of constructs being used. The other reason is of readability. A loop with the conditional part at the end is more difficult to read than one with the conditional at the top.
- 4. The use of *break* and *continue* in loops should be avoided. These statements should only be used if they prove to give higher readability than their structured counterparts. In general *break* should only be used in *case* statements and *continue* should be avoided alltogether.
- 5. The form *for* (;;) should be used for empty loops.

This form is better than the functionally equivalent *while (true)* since this implies a test against *true*, which is neither necessary nor meaningful. The form while(true) should be used for infinite loops.

### **Conditionals**

1. Complex conditional expressions must be avoided. Introduce temporary boolean variables instead.

```
if ((elementNo < 0) || (elementNo > maxElement)||
    elementNo == lastElement) {
    :
}
```

should be replaced by:

```
boolean isFinished = (elementNo < 0) || (elementNo > maxElement);
boolean isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
   :
}
```

2. The nominal case should be put in the *if*-part and the exception in the *else*-part of an if statement.

```
boolean isError = readFile (fileName);
if (!isError) {
  :
} else {
  :
}
```

3. The conditional should be put on a separate line.

4. Executable statements in conditionals must be avoided.

#### **Miscellaneous**

- 1. The use of magic numbers in the code should be avoided. Numbers other than 0 and 1 should be considered declared as named constants instead.
- 2. Floating point constants should always be written with decimal point and at least one decimal.

```
double total = 0.0;  // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

This emphasizes the different nature of integer and floating point numbers even if their values might happen to be the same in a specific case. Also, as in the last example above, it emphasize the type of the assigned variable (sum) at a point in the code where this might not be evident.

3. Floating point constants should always be written with a digit before the decimal point.

```
double total = 0.5; // NOT: double total = .5;
```

The number and expression system in Java is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .5; There is no way it can be mixed with the integer 5.

4. Functions in C++ must always have the return value explicitly listed.

```
int getValue() // NOT: getValue()
{
  :
}
```

If not explicitly listed, C++ implies int return value for functions.

5. goto in C++ should not be used. Goto statements violates the idea of structured code. Only in some very few cases (for instance breaking out of deeply nested structures) should goto be considered, and only if the alternative structured counterpart is proven to be less readable.

## Layout and Comments in Java and C++

#### **Comments**

The problem with comments is that they lie. Comments are not syntax checked, there is nothing forcing them to be accurate. And so, as the code undergoes change during schedule crunches, the comments become less and less accurate.

As Fowler puts it, comments should not be used as deodorants. Tricky code should not be commented but rewritten. In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

If, however, there is a need to write comments for whatever reason, the following guidelines should be observed.

- 1. All comments should be written in English. In an international environment English is the preferred language.
- 2. Use // for all comments, including multi-line comments.

```
// Comment spanning
// more than one line
```

Since multilevel commenting is not supported in C++ and Java, using // comments ensure that it is always possible to comment out entire sections of a file using /\* \*/ for debugging purposes etc.

3. Comments should be indented relative to their position in the code.

```
while (true) {
   // NOT: while (true) {
   // Do something
   something();
   // something();
}
```

## **Expressions and Statements**

### Layout

1. Basic indentation should be 2.

```
for (i = 0; i < nElements; i++)
  a[i] = 0;</pre>
```

Indentation of 1 is to small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increase the chance that the lines must be split. Choosing between indentation of 2, 3 and 4, 2 and 4 are the more common, and 2 chosen to reduce the chance of splitting code lines.

## **Natural form for expression**

Expression should be written as if they are written as comments or spoken out aloud. Conditional expression with negation are always difficult to understand. As an example consider the following code:

```
if (! (block < activeBlock) || !(blockId >= unblocks))
```

The logic becomes much easier to follow if the code is written in the natural form as shown below:

```
if ((block >= activeBlock) || (blockId < unblocks))
```

## Parenthesize to remove ambiguity

Parentheses should always be used as they reduce complexity and clarify things by specifying grouping. It is especially important to use parentheses when different unrelated operators are used in the same expression as the precedence rules are often assumed by the programmers, resulting in logical errors that are very difficult to spot. As an example consider the following statement:

```
if (x \& MASK == BITS)
```

This causes problems because == operator has higher precedence than & operator. Hence, MASK and BITS are first compared for equality and then the result, which is 0 or 1, is andded with x. This kind of error will be extremely hard to catch. If, however, parentheses are used, there will be no ambiguity as shown below.

```
if ((x \& MASK) == BITS)
```

Following is another example of the use of parentheses which makes the code easier to understand and hence easier to maintain.

```
leapYear = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
```

In this case parentheses have not been used and therefore the definition of a leap year is not very clear for the code. The code becomes self explanatory with the help of proper use of parentheses as shown below:

## **Breakup complex expressions**

Complex expressions should be broken down into multiple statements. An expression is considered to be complex if it uses many operators in a single statement. As an example consider the following statement:

```
*x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

This statement liberally uses a number of operators and hence is very difficult to follow and understand. If it is broken down into simple set of statements, the logic becomes easier to follow as shown below:

```
if (2*k < n-m)
          *xp = c[k+1];
else
          *xp = d[k--];
*x = *x + *xp;</pre>
```

## **Shortcuts and cryptic code**

Sometimes the programmers, in their creative excitement, try to write very concise code by using shortcuts and playing certain kinds of tricks. This results in a code which is cryptic in nature and hence is difficult to follow. Maintenance of such code therefore becomes a nightmare. Following are some examples of such code.

1. Let us start with a very simple shortcut, often used by programmers. Assume that we have the following statement.

```
x *= a;
```

For some reason the code was later modified to:

```
x *= a + b;
```

This seemingly harmless change is actually a little cryptic and causes confusion. The problem lies with the semantics of this statement. Does it mean x = x\*a+b or x = x\*(a+b)? The second one is the right answer but is not obvious from the syntax and hence causes problems.

2. Let us now look at a more complex example. What is the following code doing?

```
subkey = subkey >> (bitoff - (bitoff >> 3) << 3));</pre>
```

As can be seen, it is pretty hard to understand and therefore difficult to debug in case there is any problem. What this code is actually doing is masking bitoff with octal 7 and then use the result to shift subkey those many time. This can be written as follows:

```
subkey = subkey >> (bitoff & 0x7);
```

It is quite evident that the second piece of code is much follow to read than the first one.

3. The following piece of code is taken from a commercial software:

```
a = a >> 2;
```

It is easy to see that a is shifted right two times. However, the real semantics of this code are hidden – the real intent here is to divide a by 4. No doubt that the code above achieves this objective but it is hard for the reader to understand the intent as to why a is being shifted right twice. It would have been much better had the code been written as follows:

```
a = a/4;
```

4. A piece of code similar to the following can be found in many data structures books and is part of circular implementation of queues using arrays.

```
bool Queue::add(int n)
{
    int k = (rear+1) % MAX_SIZE;
    if (front == k)
        return false;
    else {
        rear = k;
        queue[rear] = n;
        return true;
    }
}
```

```
bool Queue::isFull()
{
    if (front == (rear+1 % MAX_SIZE))
        return true;
    else
        return false;
}
```

This code uses the % operator to set the rear pointer to 0 once it has reached MAX\_SIZE. This is not obvious immediately. Similarly, the check for queue full is also not trivial.

In an experiment, the students were asked to implement double-ended queue in which pointer could move in both directions. Almost everyone made the mistake of writing something like rear = (rear-1) % MAX\_SIZE. This is because the semantics of % operation are not obvious.

It is always much better to state the logic explicitly. Also, counting is also much easier to understand and code as compared to some tricky comparisons (e.g. check for isFull in this case). Application of both these principles resulted in the following code. It is easy to see that this code is easier to understand. It is interesting to note that when another group of students were asked to do implement double-ended queue after showing them this code, almost everyone did it without any problems.

```
bool Queue::add()
{
      if (! isFull() ) {
            rear++;
            if (rear == MAX SIZE) rear = 0;
            QueueArray[rear] = n;
            size++;
            return true;
      else return false;
}
bool Queue::isFull(int n)
      if (size == MAX SIZE)
           return true;
      else
          return false;
}
```

### **Switch Statement**

In the switch statement, cases should always end with a break. A tricky sequence of fall-through code like the one below causes more trouble than being helpful.

This code is cryptic and difficult to read. It is much better to explicitly write what is happening, even at the cost of duplication.

It would even be better if such code is written using the if statement as shown below.

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

## **Magic Numbers**

Consider the following code segment:

Can you tell by reading the code what is meant by the numbers 20, 27, 3, 21, 22, and 23. These are constant that mean something but they do not give any indication of their importance or derivation, making the program hard to understand and modify. To a reader they work like magic and hence are called magic numbers. Any number (even 0 or 1) used in the code is a magic number. It should rather have a name of its own that can be used in the program instead of the number.

The difference would be evident if we look at the code segment below that achieves the same purpose as the code above.

```
enum {
     MINROW = 1,
     MINCOL = 1,
      MAXROW = 24,
      MAXCOL = 80,
      LABELROW = 1.
      NLET = 26,
      HEIGHT = MAXROW -4,
      WIDTH = (MAXCOL-1) / NLET
};
fac = (lim+HEIGHT-1) /HEIGHT;
if (fac < 1)
     fac = 1;
for (i =0; i < NLET; i++) {
      if (let[i] == 0)
            continue;
      for (j = HEIGHT - let[i] / fac; j < HEIGHT; j++)</pre>
            draw(j-1 + LABELROW,
                      (i+1) *WIDTH, '*');
}
draw(MAXROW-1, MINCOL+1, '');
for (i='A'; i <= 'Z'; i++)
      cout << i;
```

## Use (or abuse) of Zero

The number 0 is the most abused symbol in programs written in C or C++. One can easily find code segment that 0 in a fashion similar to the examples below in almost every C/C++ program.

```
flag = 0;  // flag is boolean
str = 0;  // str is string
name[i] = 0;  // name is char array
x = 0;  // x is floating pt
i = 0;  // i is integer
```

This is a legacy of old style C programming. It is much better to use symbols to explicitly indicate the intent of the statement. It is easy to see that the following code is more in line with the self-documentation philosophy than the code above.

```
flag = false;
str = NULL;
name[i] = '\0';
x = 0.0;
i = 0;
```

## **Clarity through modularity**

As mentioned earlier, abstraction and encapsulation are two important tools that can help in managing and mastering the complexity of a program. We also discussed that the size of individual functions plays a significant role in making the program easy or difficult to understand. In general, as the function becomes longer in size, it becomes more difficult to understand. Modularity is a tool that can help us in reducing the size of individual functions, making them more readable. As an example, consider the following selection sort function.

Although it is not very long but we can still improve its readability by breaking it into small functions to perform the logical steps. The modified code is written below:

### Software Engineering-I (CS504)

```
void swap(int &x, int &y)
{
      int temp;
      temp = x;
      x = y;
      y = temp;
}
int minimum(int a[], int from, int to)
      int i;
      int min;
      min = a[from];
      for (i = from; i <= to; i++) {
           if (a[i] < a[min])
                  min = i;
      }
      return min;
}
void selectionSort(int a[], int size)
      int min;
      int i;
      for (i = 0; i < size; i++){}
            min = minimum(a, i, size -1);
            swap(a[i], a[min]);
      }
}
```

It is easy to see that the new selectionSort function is much more readable. The logical steps have been abstracted out into the two functions namely, minimum and swap. This code is not only shorter but also as a by product we now have two functions (minimum and swap) that can be reused.

Reusability is one of the prime reasons to make functions but is not the only reason. Modularity is of equal concern (if not more) and a function should be broken into smaller pieces, even if those pieces are not reused. As an example, let us consider the quickSort algorithm below.

```
void quickSort(int a[], int left, int right)
      int i, j;
      int pivot;
      int temp;
      int mid = (left + right)/2;
      if (left < right) {</pre>
            i = left - 1;
             j = right + 1;
            pivot = a[mid];
            do {
                   do i++; while (a[i] < pivot);</pre>
                   do j--; while (a[i] < pivot);</pre>
                   if (i<j) {
                          temp = a[i];
                          a[i] = a[j];
                          a[j] = temp;
                   }
             \} while (i < j);
             temp = a[left];
             a[left] = a[j];
             a[j] = temp;
            quickSort(a, left, j);
             quickSort(a, j+1, right);
}
```

This is actually a very simple algorithm but students find it very difficult to remember. If is broken in logical steps as shown below, it becomes trivial.

```
void quickSort(int a[], int left, int right)
{
      int p;
      if (left < right) {</pre>
            p = partition(a, left, right);
            quickSort(a, left, p-1);
            quickSort(a, p+1, right);
      }
}
int partition(int a[], int left, int right)
      int i; j;
      int pivot;
      i = left + 1;
      j = right;
      pivot = a[left];
      while(i < right && a[i] < pivot) i++;</pre>
      while(j > left && a[j] >= pivot) j++;
      if(i < j)
            swap(a[i], a[j]);
      swap(a[left], a[j]);
      return j;
}
```

## Short circuiting | | and &&

The logical and operator, &&, and logical or operators,  $\parallel$ , are special due to the C/C++ short circuiting rule, i.e. a  $\parallel$  b and a && b are short circuit evaluated. That is, logical expressions are evaluated left to right and evaluation stops as soon as the final truth value can be determined.

Short-circuiting is a very useful tool. It can be used where one boolean expression can be placed first to "guard" a potentially unsafe operation in a second boolean expression. Also, time is saved in evaluation of complex expressions using operators || and &&. However, a number of issues arise if proper attention is not paid.

Let us look at the following code segment taken from a commercially developed software for a large international bank:

```
struct Node {
    int data;
    Node * next;
};

Node *ptr;
...
while (ptr->data < myData && ptr != NULL) {
    // do something here
}</pre>
```

What's wrong with this code?

The second part of condition, ptr != NULL, is supposed to be the guard. That is, if the value of the pointer is NULL, then the control should not enter the body of the while loop otherwise, it should check whether ptr->data < myData or not and then proceed accordingly. When the guard is misplaced, if the pointer is NULL then the program will crash because it is illegal to access a component of a non-existent object. This code is rewritten as follows. This time the short-circuiting helps in achieving the desired objective which would have been a little difficult to code without such help.

## **Operand Evaluation Order and Side Effects**

A side effect of a function occurs when the function, besides returning a value, changes either one of its parameters or a variable declared outside the function but is accessible to it. That is, a side effect is caused by an operation that may return an explicit result but it may also modify the values stored in other data objects. Side effects are a major source of programming errors and they make things difficult during maintenance or debugging activities. Many languages do not specify the function evaluation order in a single statement. This combined with side effects causes major problems. As an example, consider the following statement:

```
c = f1(a) + f2(b);
```

The question is, which function (f1 or f2) will be evaluated first as the C/C++ language does not specify the evaluation order and the implementer (compiler writer) is free to choose one order or the other. The question is: does it matter?

To understand this, let's look at the definition of f1 and f2.

In this case both f1 and f2 have side effects as they both are doing two things - changing the value of the parameter and changing the value at the caller side. Now if we have the following code segment,

```
a = 3;

b = 4;

c = f1(a) + f2(b);
```

then the value of a, b, and c would be as follows:

```
a = 6

b = 2

c = 8
```

Software Engineering-I (CS504)

So far there seem to be any problem. But let us now consider the following statement:

$$c = f1(a) + f2(a);$$

What will be the value of a and c after this statement?

If f1 is evaluated before f2 then we have the following values:

$$a = 3$$
  
 $b = 9 // 7 + 2$ 

On the other hand, if f2 is evaluated before f2 then, we get totally different results.

$$a = 2$$
  
 $b = 3 // 3 + 0$ 

#### **Common mistakes**

Following is short list of common mistakes made due to side-effects.

array[i++] = i;

If i is initially 3, array[3] might be set to 3 or 4.

2.  $\operatorname{array}[i++] = \operatorname{array}[i++] = x;$ 

Due to side effects, multiple assignments become very dangerous. In this example, a whole depends upon when i is incremented.

3.
", " is very dangerous as it causes side effects. Let's look at the following statement:

```
int i, j = 0;
```

Because of the syntax, many people would assume that i is also being initialized to 0, while it is not. Combination of , and = -- is fatal. Look at the following statement:

```
a = b, c = 0;
```

A majority of the programmers would assume that all a, b, and c are being initialized to 0 while only c is initialized and a and b have garbage values in them. This kind of overlook causes major programming errors which are not caught easily and are caused only because there are side effects.

#### Guidelines

If the following guidelines are observed, one can avoid hazards caused by side effects.

- 1. never use "," except for declaration
- 2. if you are initializing a variable at the time of declaration, do not declare another variable in the same statement
- 3. never use multiple assignments in the same statement
- 4. Be very careful when you use functions with side effects functions that change the values of the parameters.
- 5. Try to avoid functions that change the value of some parameters and return some value at the same time.

### **Performance**

In many cases, performance and maintainability are at odds with one another. When planning for performance, one should always remember the 80/20 rule - you spend 80 percent of your time in 20 percent of the code. That is, we should not try to optimize everything. The proper approach is to profile the program and then identify bottlenecks to be optimized. This is similar to what we do in databases – we usually normalize the database to remove redundancies but then partially de-normalize if there are performance issues.

As an example, consider the following. In this example a function isspam is profiled by calling 10000 times. The results are shown in the following table:

sec	%	Cum%	cycles	instructions	calls	function				
45.26	81.0	81.0	11314990000	9440110000	4835000	strchr				
6.08	10.9	91.9	1520280000	156646000	4618000	strnemp				
2.59	4.6	96.6	648080000	854500000	2170000	strstr				
1.83	3.3	99.8	456225559	344882213	2170435	strlen				
0.09	0.2	100.0	21950000	28510000	10000	isspam				
11 othe	11 other functions with insignificant performance overhead									

The profiling revealed that most of time was spent in strchr and strncmp and both of these were called from strstr

When a small set (a couple of functions) of functions which use each other is so overwhelmingly the bottleneck, there are two alternatives:

- 1. use a better algorithm
- 2. rewrite the whole set

In this particular case strstr was rewritten and profiled again. It was and found out that although it was much faster but now 99.8% of the time was spent in strstr.

The algorithm was rewritten and restructured again by eliminating strstr, strchr, and strncmp and used memcmp. Now memcmp was much more complex than strstr but it gained efficiency by eliminating a number of loops and the new results are as shown:

sec	%	Cum%	cycles	instructions	calls	function
3.524	56.9	56.9	880890000	1027590000	46180000	memcmp
2.662	43.0	100.0	665550000	902920000	10000	isspam
0.001	0.0	100.0	140304	106043	652	strlen

strlen now went from over two million calls to 652.

### Software Engineering-I (CS504)

Many details of the execution can be discovered by examining the numbers. The trick is to concentrate on hot spots by first identifying them and then cooling them. As mentioned earlier, most of the time is spent in loops. Therefore we need to concentrate on loops.

As an example, consider the following:

```
for (j = i; j < MAX_FIELD; j++)
  clear(j);</pre>
```

This loop clears field before each new input is read. It was observed that it was taking almost 50% of the total time. On further investigation it was found out that MAX\_FIELD was 200 but the actual fields that needed to be cleared were 2 or 3 in most cases. The code was subsequently modified as shown below:

```
for (j = i; j < maxField; j++)
  clear(j);</pre>
```

This reduced the overall execution time by half.

## **Portability**

Many applications need to be ported on to many different platforms. As we have seen, it is pretty hard to write error free, efficient, and maintainable software. So, if a major rework is required to port a program written for one environment to another, it will be probably not come at a low cost. So, we ought to find ways and means by which we can port applications to other platforms with minimum effort. The key to this lies in how we write our program. If we are careful during writing code, we can make it portable. On the other hand if we write code without portability in mind, we may end-up with a code that is extremely hard to port to other environment. Following is brief guideline that can help you in writing portable code.

#### Stick to the standard

- 1. Use ANSI/ISO standard C++
- 2. Instead of using vendor specific language extensions, use STL as much as possible

### **Program in the mainstream**

Although C++ standard does not require function prototypes, one should always write them.

## Size of data types

Sizes of data types cause major portability issues as they vary from one machine to the other so one should be careful with them.

#### **Order of Evaluation**

As mentioned earlier during the discussion of side effects, order of evaluation varies from one implementation to other. This therefore also causes portability issues. We should therefore follow guidelines mentioned in the side effect discussion.

## Signedness of char

The language does not specify whether char is signed or unsigned.

```
char c;
    // between 0 and 255 if unsigned
    // -128 to 127 if signed

c = getchar();
if (c == EOF) ??
    // will fail if it is unsigned
```

It should therefore be written as follows:

```
int c;
c = getchar();
if (c == EOF)
```

### **Arithmetic or Logical Shift**

The C/C++ language has not specified whether right shift >> is arithmetic or logical. In the arithmetic shift sign bit is copied while the logical shift fills the vacated bits with 0. This obviously reduces portability.

Interestingly, Java has introduced a new operator to handle this issue. >> is used for for arithmetic shift and >>> for logical shift.

## **Byte Order and Data Exchange**

The order in which bytes of one word are stored is hardware dependent. For example in Intel architecture the lowest byte is the most significant byte while in Motorola architecture the highest byte of a word is the most significant one. This causes problem when dealing with binary data and we need to be careful while exchanging data between to heterogeneous machines. One should therefore only use text for data exchange. One should also be aware of the internationalization issues and hence should not assume ASCII as well as English.

## **Alignment**

The C/C++ language does not define the alignment of items within structures, classes, an unions. data may be aligned on word or byte boundaries. For example:

```
struct X {
     char c;
     int i;
};
```

address of i could be 2, 4, or 8 from the beginning of the structure. Therefore, using pointers and then typecasting them to access individual components will cause all sorts of problems.

#### **Bit Fields**

Bit fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples:

- Packing several objects into a machine word. *e.g.* 1 bit flags can be compacted -- Symbol tables in compilers.
- Reading external file formats -- non-standard file formats could be read in. *E.g.* 9 bit integers.

C lets us do this in a structure definition by putting :bit length after the variable. i.e.

```
struct packed_struct {
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int f4:1;
    unsigned int type:4;
    unsigned int funny_int:9;
} pack;
```

Here the packed\_struct contains 6 members: Four 1 bit **flags** fl..f3, a 4 bit type and a 9 bit funny int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word.

Bit fields are a convenient way to express many difficult operations. However, bit fields do suffer from a lack of portability between platforms:

- integers may be signed or unsigned
- Many compilers limit the maximum number of bits in the bit field to the size of an integer which may be either 16-bit or 32-bit varieties.
- Some bit field members are stored left to right others are stored right to left in memory.
- If bit fields too large, next bit field may be stored consecutively in memory (overlapping the boundary between memory locations) or in the next word of memory.

Bit fields therefore should not be used.

## **Exception handling**

Exception handling is a powerful technique that separates error-handling code from normal code. It also provides a consistent error handling mechanism. The greatest advantage of exception handling is its ability to handle asynchronous errors.

The idea is to raise some error flag every time something goes wrong. There is a system that is always on the lookout for this error flag. Third, the previous system calls the error handling code if the error flag has been spotted. The raising of the imaginary error flag is simply called *raising* or *throwing* an error. When an error is thrown the overall system responds by *catching* the error. Surrounding a block of error-sensitive code with exception handling is called *trying* to execute a block. The following code segment illustrates the general exception handling mechanism.

```
try {
    ...
    throw Exception()
    ...
} catch( Exception e )
{
    ...
}
```

One of the most powerful features of exception handling is that an error can be thrown over function boundaries. This allows programmers to put the error handling code in one place, such as the *main*-function of your program.

## **Exceptions and code complexity**

A number of invisible execution paths can exist in simple code in a language that allows exceptions. The complexity of a program may increase significantly if there are exceptional paths in it. Consider the following code:

```
String EvaluateSalaryAnadReturnName( Employee e)
{
   if (e.Title() == "CEO" || e.Salary() > 10000)
   {
      cout << e.First() << " " << e.Last() << " is overpaid" << endl;
   }
   return e.First() + " " + e.Last();
}</pre>
```

Before moving any further, let's take the following assumptions:

- 1. Different order of evaluating function parameters are ignored.
- 2. Failed destructors are ignored
- 3. Called functions are considered atomic
  - a. for example: e.Title() could throw for several reasons but all that matters for this function is whether e.Title() results in an exception or not.
- 4. To count as different execution paths, an execution path must be made-up of a unique sequence of function calls performed and exited in the same way.

**Question**: How many more execution paths are there?

**Ans**: 23. There are 3 non-exceptional paths and 20 exceptional paths.

## The non-exceptional paths:

The three non-exceptional paths are enumerated as below:

- 1. if (e.Title() == "CEO" || e.Salary() > 10000)
  - if e.Title() == "CEO" is true then the second part is not evaluated and e.Salary() will not be called.
  - cout will be performed
- 2. if e.Title() != "CEO" and e.Salary() > 10000
  - both parts of the condition will be evaluated
  - cout will be performed.
- 3. if e.Title() != "CEO" and e.Salary() <= 10000
  - both parts of the condition will be evaluated
  - cout will not be performed.

### **Exceptional Code Paths**

The 20 exceptional code path are listed below.

1. String EvaluateSalaryAnadReturnName( Employee e)

The argument is passed by value, which invokes the copy constructor. This copy operation might throw an exception.

e.Title() might itself throw, or it might return an object of class type by value, and that copy operation might throw.

Same as above.

To match a valid == () operator, the string literal may need to be converted to a temporary object of class type and that construction of the temporary might throw.

Same as above.

operator == () might throw.

Same as above.

Same as above.

As per C++ standard, any of the five calls to << operator might throw.

```
14-15 cout << e.First() << " " << e.Last() << " is overpaid" << endl
    similar to 2 and 3.

16-17 return e.First() + " " + e.Last();
    similar to 14-15.

18-19 return e.First() + " " + e.Last();
    similar to 6,7, and 8.

20. return e.First() + " " + e.Last();
    similar to 4.</pre>
```

### **Summary:**

- A number of invisible execution paths can exist in simple code in a language that allows exceptions.
- Always be exception-aware. Know what code might emit exceptions.

#### The Challenge:

Can we make this code exception safe and exception neutral? That is, rewrite it (if needed) so that it works properly in the presence of an exception and propagates all exceptions to the caller?

### **Exception-Safety:**

A function is exception safe if it might throw but do not have any side effects if it does throw and any objects being used, including temporaries, are exception safe and clean-up there resources when destroyed.

#### **Exception Neutral:**

A function is said to be exception neutral if it propagates all exceptions to the caller.

### **Levels of Exception Safety**

- Basic Guarantee: Ensures that temporaries are destroyed properly and there are no memory leaks.
- Strong Guarantee: Ensures basic guarantee as well as there is full-commit or roll-back
- No-throw Guarantee: Ensure that a function will not throw.

Exception-safety requires either no-throw guarantee or basic and strong guarantee.

### Does the function satisfy basic guarantee?

Yes. Since the function does not create any objects, in the presence of an exception, it does not leak any resources.

#### Does the function satisfy strong guarantee?

No. The strong guarantee says that if the function fails because of an exception, the program state must not be change.

This function has two distinct side-effects:

- an overpaid message is emitted to cout.
- A name strings is returned.

As far as the second side-effect is concerned, the function meets the strong guarantee because if an exception occurs the value will never be returned.

As far as the first side-effect is concerned, the function is not exception safe for two reasons:

- if exception is thrown after the first part of the message has been emitted to cout but before the message has been completed (for example if the fourth << operator throws), then a partial message was emitted to cout.
- If the message emitted successfully but an exception occurs in later in the function (for example during the assembly of the return value), then a message was emitted to cout even though the function failed because of an exception. It should be complete commit or complete roll-back.

### Does the function satisfy no-throw guarantee?

No. This is clearly not true as lots of operations in the function might throw.

### **Strong Guarantee**

To meet strong guarantee, either both side-effects are completed or an exception is thrown and neither effect is performed.

```
// First attempt:
String EvaluateSalaryAnadReturnName( Employee e)
{
    String result = e.First() + " " + e.Last();
    if (e.Title() == "CEO" || e.Salary() > 10000)
    {
        String message = result + " is overpaid\n";
        cout << message;
    }
    return result;
}
What happens if the function is called as follows:
String theName;</pre>
```

1. string copy constructor is invoked because the result is returned by value.

theName = evaluateSalarayAndReturnName(someEmplyee);

- 2. The copy assignment operator is invoked to copy the result into the Name
- 3. If either copy fails then the function has completed its side-effects (since the message was completely emitted and the return value was completely constructed) but the result has been irretrievable lost.

Can we do better and perhaps avoid the problem by avoiding the copy?

```
// Second attempt:
String EvaluateSalaryAnadReturnName( Employee e, String &r)
{
    String result = e.First() + " " + e.Last();
    if (e.Title() == "CEO" || e.Salary() > 10000)
    {
        String message = result + " is overpaid\n";
        cout << message;
    }
    r = result;
}</pre>
```

Looks better but assignment to r might still fail which leaves us with one side-effect completed and other incomplete.

```
// Third attempt:
auto_ptr<String> EvaluateSalaryAnadReturnName( Employee e)
{
    auto_ptr<String> result = new String(e.First() + " " + e.Last());
    if (e.Title() == "CEO" || e.Salary() > 10000)
    {
        String message = (*result) + " is overpaid\n";
        cout << message;
    }
    return result;();  // rely on transfer of ownership
    // this can't throw
}</pre>
```

We have effectively hidden all the work to construct the second side-effect (the return value), while we ensured that it can be safely returned to the caller using only non-throwing operation after the first side-effect has completed the printing of the message. In this case we know that once the function is complete, the return value will make successfully into the hands of the caller and be correctly cleaned-up in all cases. This is because the aut\_ptr semantics guarantee that If the caller accepts the returned value, the act of accepting a copy of the auto\_ptr causes the caller to take the ownership and if the caller does not accept the returned value, say by ignoring the

return value, the allocated string will automatically be destroyed with proper cleanup.

### **Exception Safety and Multiple Side-effects**

It is difficult and some-times impossible to provide strong exception safety when there are two or more side-effects in one function and these side-effects are not related with each other. For example we would not have been able to provide exception safety if there are two output messages, one to cout and the other one to cerr. This is because the two cannot be combined.

When such a situation comes with two or more unrelated side-effects which cannot be combined then the best way to handle such a situation is break it into two separate functions. That way, at least, the caller would know that these are two separate atomic steps.

#### Summary

- 1. Providing the strong exception-safety guarantee often requires you to trade-off performance.
- 2. If a function has multiple un-related side-effects, it cannot always be made strongly exception safe. If not, it can be done only by splitting the function into several functions, each of whose side-effects can be performed atomically.
- 3. Not all functions need to be strongly exception-safe. Both the original code and attempt#1 satisfy the basic guarantee. For many clients, attempt # 1 is sufficient and minimizes the opportunity for side-effects to occur in the exceptional situation, without requiring the performance trade-off of attempt #3.