# Common Mistakes, and Logic Guide

(updated: Saturday, September 21, 2019)

This document lists mistakes and observations derived from assignment & lab comments. It is created in order to share this knowledge with all students such that they all know of such mistakes and how to avoid them.

**This document only lists what to do, not why**. The "why" was taught to you in class and lecture. If you do not remember, ask.

# Contents

# 1) Basic Coding Rules

1. <algorithms> library and `auto` data type are not allowed unless explicitly
2. Variables and function names must be meaningful. Short variable names like `int x` is allowed only for loop counters and temp variables.
3. You are NOT an English language translator. So, don't translate English sentences into the same in computer language. Think of the logic to do it best and generally avoid needless "`if`" statements
4. All code must have 1-entry-point & 1-exit-point
    a. That includes: loops, functions, conditional statements, etc.
    b. In other words: you are not allowed to use keywords like: `break, exit, continue`.
    c. This also means that a function can ONLY have 1 return statement at the end of function.
    d. There are no exceptions to the above rules. *(aside from the use of `break` within switch statement)*
5. Users are dumb *(golden rule of programming)*
    a. never assume you will get the proper capitalization, always use `toupper`() or `tolower`() if checking for a character based on user input. Never compare a full user-inputted string assuming capitalization was done correctly.
    b. Example: user may input month name as: "January" or "January" or "JANUARY" or "jANuary",..etc
6. Do not repeat code:
    a. If you write the same lines of code more than once, then you must re-examine your logic. Re-organizing your code or re-examining your logic should be the answer. If both failed, then you may need to use a function?
7. Comments: do not write unnecessary comments. Generally simple rule for comments is:
    a. You need to explain something not clear from code *(maybe you did something clever?)*
    b. When it would otherwise require the reader to go lookup something somewhere else in order to understand/verify what you did.
8. All literal values that could one day change must always be declared as constants
    a. *(example: prices, rates, number of items, etc)*
9. **`If`** statements are considered to be relatively heavy. Don't be too liberal in using them,
    a. See also coding rule 5-b
    b. think of your logic 1st and remember you are not translating English.
10. You must use proper operators and built-in functions. Example:
    a. variable = variable + 10 → variable += 10;
    b. variable * variable * variable → pow(variable,3)
11. Never assume sizes or values within inside a function unless if they were passed in the parameter list.
12. The function is a self-contained black-box. Nothing outside the function exists except to what was passed in the parameter list
13. Classes should not have public variables, unless explicitly specified in a question
14. Each line in your source code file can NOT exceed 70 characters in length.
15.

# 2) DOs & DON'Ts

## a) DO (i.e. good)

1. **DO** Use `const` variables
2. **DO** Always name `const` in ALL CAPS
3. **DO** Always capitalize structure datatypes.
4. **DO** Define all local variables within a scope at beginning of function/scope *(not in the middle of code)* unless you have a good reason.
5.

## b) DO NOT EVER use

1. **Do NOT use**: Function prototypes
2. **Do NOT use**: Typedef
3. **Do NOT use**: Static variables unless absolutely necessary and there is no other way (esp. in Recursion)
4. **Do NOT use**: global variables
5. **Do NOT use:** online compilers
6. **Do NOT use**: "`return`" from anywhere except the end of the function *(ok only for Recursion base case)*
7. **Do NOT use**: *`break`* in a loop.
8. **Do NOT use**: `goto`
9.

## c) DO NOT (i.e. DO NOT EVER) do the following:

### i) Data Types

1. Do not use `float`
2. Do not use other obsolete ways for typecasting. Always use `static_cast<>` *(see book 3.5)*
3. Do not create uninitialized variables.
   - **ALWAYS** initialize variables of simple data types (int, double, ..etc) .
   - **DO NOT** initialize class/object variables (like string, vector) unless you need to start with a non-empty value
4. Beware of the size of used data types. You should not copy value from a larger data type to a smaller data type even if the value you are currently using will not be affected.
   - Example (NO): long → int , double → float …etc.
   - NOTE: using some rationalization is required. There are cases when this rule does not apply. for example: `toupper`() returns an int but it is ok to assign the result to a char because the value returned from `toupper`() will always represent a char even though it is returned in a larger "container"
5. It is OK however to typecast or directly convert between integer-type values and floating-point types
   - Example (OK) : double → int , int → double

### ii) INPUT / OUTPUT

1. *(TRACK-A)* Never use `cin` , always use `getline()` for input and `stringstream` to convert to/from strings to other data types. See rule 3-c
2. **DO** use `endl` instead of `\n` in `cout` statements. Example:
   - **OK :** cout<<"hello"<<endl<<"how are you?"<<endl;
   - **NO:** cout<<"hello\nhow are you?\n";

### iii) CHARACTERS

1. Do not do character conversion manually, always use character functions (cctype library)
2. Do not use ASCII values for chars *(use the char itself for example 'b')*
3.

### iv) LOOPS

1. All loops must have 1-entry-point & 1-exit-point
2. A loop must terminate immediately after the result is known. See Rule 6
3. Do not use `break` or `continue` or `return` in a loop. Always use a flag to invalidate the loop condition if you desire early termination. See rule 6

### v) STRINGS

1. NEVER use C-style strings or string::C_str() function. Always use C++ string class.
2. Use `.at()` member function to access individual elements, never use `[..]`
3. ONLY use string class member functions *(like find, find_firsT_of, insert,…etc).* NEVER use external string manipulation functions like: *reverese*
4. *(TRACK-A)* ALWAYS Use `stringstream` to convert to/from strings

### vi) VECTORS (*optional for online classes)*

1. *(TRACK-A)* Never use built-in arrays, use vectors instead
2. Always Use `.at()` member function to access individual elements, never use `[..]`
3. Always utilize member functions

### vii) Conditions & Boolean

1. Never write: `if (x==true)` …. Always write `if (x).`
   - Generally: writing `==true` or `==false` or `!=true` or `!=false` is normally unnecessary and meaningless
2. Never write lateral 0/1 to mean true/false for Boolean variables. Always use true/false
3. Never write a condition with empty body.

# 3) Basics, I/O, Output Formatting, Strings:

## a) Literals & Constants:

1. Literals can NOT be used directly into a program. They must be declared as constants at the beginning of program.

2. Literals are: anything that has a fixed value that can/should not change throughout the program *(except for output string literals, though, in multi-lingual programs these strings also get declared as constants)*.

## b) Constants must be named per style guide: in ALL CAPS

- Example: item prices, tax rates, etc. they should be declared as: const PRICE=15.5;

3. **Rule-Of-Thumb**: any fixed number *(price, tuition, PI, discount rate, etc)* must be declared as a constant.

## c) *(TRACK-A)* Stringstream & Data Input:

1) **Rule-Of-Thumb**: If user input includes any string input at any point, then use *getline* for all data input in the entire program. Then use *stringstream*() to convert values.

2) Generally, it is best to use the *getline* and *stringstream* method for all user data input.
3) For fixed set of input variables, stringstream will split the string based on its ability to separate the data types:

```
int a, c;
char b;
string sline;
getline(cin,sline)                    //user inputs  7:5
stringstream (sline)>> a>> b>>c;      //a→7    c→5
```

4) For a list of data that may require a loop or processing in more than one statement, a variable will be needed

5) **REMEMBER: stringstream returns NULL when empty.**

```
const int SIZE=20;
int ar[SIZE];
getline(cin,sline)
stringstream ss(sline);
for (int i=0; (i<SIZE) && ss; i++){ //either end of array or
                                    // end of input items
  ss >> ar[i];
}
```

```
vector <int> v;
string instr;
int tempvar;
getline (cin, instr);
stringstream ss(instr);
while (ss>>tempvar){      //there are items to extract
   v1.push_back(tempvar);
}
```

6) In order to insert an integer TO a string we can write:

```
string mystr;
int myint = 1234;
stringstream mystream;
mystream << myint;       //insert integer into the stream
mystream >> mystr;       // extract it into a string
```

7) REMEMBER: if inserting a new value into a previously-used stringstream variable *(i.e. reusing the stringstream variable)*:
   - you need to clear the stream flags 1st using .clear() function
   - you may need to reset the stream using .str(""); if you previously read past stream's end.

```
string mystr1,mystr2;
int myint1=100,myint2=200;
stringstream mystream;
mystream << myint1;       //insert integer into the stream
mystream >> mystr1;       //mystr1→"100"
mystream.clear();
mystream.str("");
mystream << myint2;       //insert integer into the stream
mystream >> mystr2;       //mystr2→"200"
```

## d) Strings:

8) Using string.**find**(): always check result using **string::npos** constant, it means no position was found

```
size_t found = s.find("sqrt");
if (found  !=  string::npos){
        //i.e. "sqrt" was found
}
```

9) Using string.**insert():** you need position to insert. The following example converts from decimal to binary by inserting digits into string from the left side:

```
while (num > 0){
     s.insert(s.begin(),((num%2)+'0'));  //insert from the left
     num/=2;
}
```

# 4) Miscellaneous

## a) Random Number Generation

- Classic Method **(DO NOT USE)**:
  - When using the random number generator seed function srand, call it ONCE ONLY in your program, and "sacrifice" the first value, like this: `srand(time(0)); rand();` at the top of main. Be sure to #include both `ctime` and `cstdlib`.
- **C++11**
  - <random> header introduces random number generation facilities.
  - This library allows to produce random numbers using combinations of generators and distributions:
    - Generators: Objects that generate uniformly distributed numbers.
    - Distributions: Objects that transform sequences of numbers generated by a generator into sequences of numbers that follow a specific random variable distribution, such as uniform, Normal or Binomial.
  - Distribution objects generate random numbers by means of their *operator()* member, which takes a generator object as argument:

```
#include <random>
……
random_device rd;    //ensures new set of numbers every time
default_random_engine generator (rd());
uniform_int_distribution<int> distribution(1,6);
// generates number in the range 1..6
int dice_roll = distribution(generator);
```

Codeblocks workaround:
- add #include <ctime>
…
- replace rd() with time(0)
- call dice_roll twice or more *(1st call will always return the same number)*

## b) Pause for a certain time:

- Classic Method (platform dependent) **DO NOT USE**
  - Windows:
    - #include <windows.h>  //use: Sleep (milliseconds)
- **C++11**

```
#include <thread>       // std::this_thread::sleep_for
#include <chrono>       // std::chrono::seconds
this_thread::sleep_for (chrono::seconds(1));   //chrono::minutes , chrono::hours
```

## c) Date & Time

```
#include <ctime>
#include <chrono>
system_clock::time_point today = system_clock::now();
std::time_t tt = system_clock::to_time_t(today);
string s = ctime(&tt); // see ctime  reference. The string includes a '\n'
```

# 5) Selection & Logical/Relations:

1. From this point on, you are graded on the efficiency of your code
2. "if" statements are relatively heavy operations. Hence, their use should be optimized

## a) Boolean Values

Boolean values (true/false) can be interchangeably used with integer numbers. Boolean values (just like char type) are actually integers. Computer sees false as 0, and true 1 or as anything not 0.

Always remember that false → 0 and true → 1 *(do NOT write 0 or 1 in your code)* which means you can use logical expressions in mathematical calculations. *See more details in section b below*

## b) *(TRACK-A)* AVOID IF statements when possible

"If" statements are generally inefficient. However, there is always a certain level of compromise between computational efficiency and readability. Most of "if" statements misuse is due to people who think they are English-to-C++ translators. This is not programming.

Always think of the logic and find out it can be best implemented. In many cases "if" statements can be avoided by using a simple calculation instead or in other cases by using the ternary operator.

**Rule of thumb: If you found yourself writing "true" "false" within an expression, it can usually be improved**

- Example1: given bool acON , turn ac on when temp is above 80f or otherwise turn it off.

BAD

```
If (temp > 80){
      acON = true;
} else{
      acON = false;
}
```

OK

```
acON = (temp > 80)? true : false;
```

G O O D

```
acON = (temp > 80);
```

- Example2: person's tax rate is:
  - 10% if income is less than $10,000
  - 15% if income is more than 10,000 and less than 25,000
  - 25% if income is greater than 25,000

S M A R T

```
taxRate = (income* 0.10) + ((income>10000)*income* 0.05) +((income>25000)*income* 0.10)
```

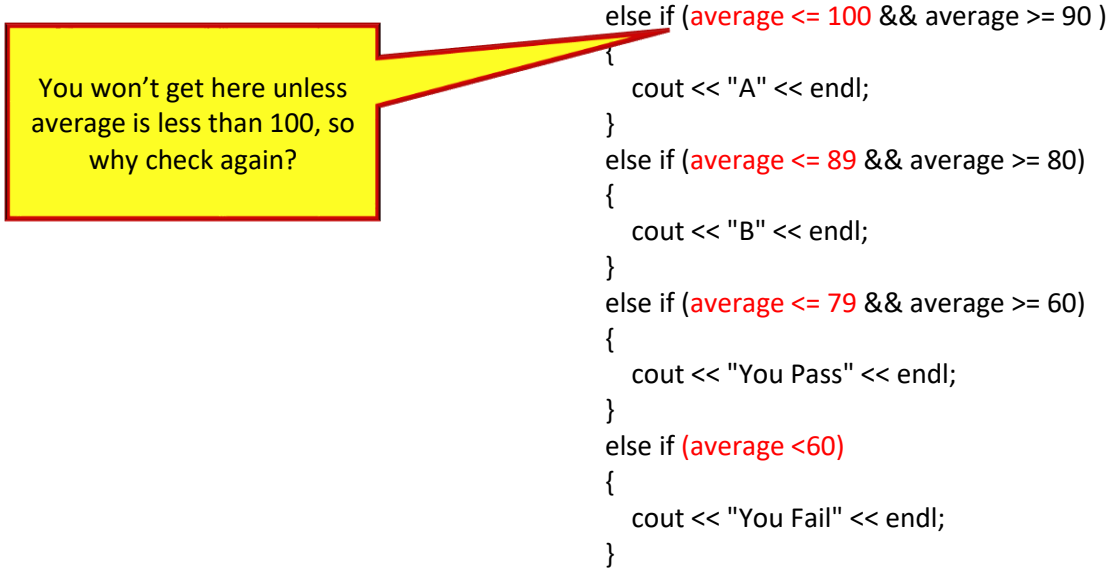
How? Look at the rates: everyone will get to pay 10% , people with income above 10k pay extra 5%, people with income above 25k pay extra 5% … this is how a programmer should look at the above requirement *(vs English translator)*

And since (anything * 0 = 0) and (anything * 1 = 1) you can use the Boolean expression as part of the claculation

## c) Range Checking

3. You should never create an "if" statement for every possible value. If example: if you are doing something based on an answer that is either yes or no: you only check for "yes" and don't go afterwards and check for "no" because by if the answer if not "yes" it means it is "no". The same applies for multiple values or data ranges: if you check the range in sequence then you only need to check for one end of the range and no need to check for the last value.

Example: checking for letter grades against grade value ranges (lab problem) **parts in red are wrong**:




```
if (average > 100)
{
   cout << "Invalid data" << endl;
}
else if (average <= 100 && average >= 90 )
{
   cout << "A" << endl;
}
else if (average <= 89 && average >= 80)
{
   cout << "B" << endl;
}
else if (average <= 79 && average >= 60)
{
   cout << "You Pass" << endl;
}
else if (average <60)
{
   cout << "You Fail" << endl;
}
```

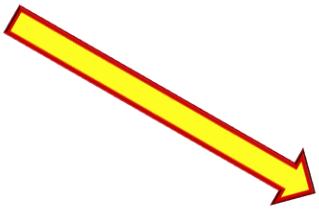
## d) Stacking & Nesting

1. Aside from the two types of "if" statement (one-way selection "if" , and two-way selection "if..else") there are two ways of concatenating/nesting multiple if statements together.
   - **Concatenating** is mainly stitching few "if" statements together like example above.
   - **Nesting** is when having a totally new independent "if" statement inside the braces of a different "if" statement. If new nested "if" will get executed as part of the code that gets executed when the evaluation of outer "if" expression causes this block of code to be executed.

2. You should not use multiple "if" statements if one expression can give the same result.
   - Example:

```
if (grade=='A')
        cout <<"You Passed";
else if (grade=='B')
        cout <<"You Passed";
else if (grade=='C')
        cout <<"You Passed";
else
        cout <<"You Failed";
```

```
if (grade=='A' || grade=='B' || grade=='C')
{
        cout <<"You Passed";
}
else
        cout <<"You Failed";
```



```
switch (grade){
case 'A':
case 'B':
case 'C': cout << "You Passed";
        break;

default: cout<< "You Failed";
}
```

3. **Tip:** Avoid repeating the exact same code.

4. **Rule-Of-Thumb:** if you wrote the same block of code more than once as a result of two or more different "*if*" statements, then your logic is likely flawed.

5. **Rule-Of-Thumb:** Always use braces in conditional statements.

## e) Using switch statements:

Switch statements are used to check for a finite **set of values that are either integers or characters**.

```
char grade;
…….
switch (grade){
case 'A':
case 'B':
case 'C': cout << "You Passed";
        break;
default: cout<< "You Failed";
}
```

```
int  grade;
…….
switch (grade){
case 100:
case 70:
case 60: cout << "You Passed";
        break;
default: cout<< "You Failed";
}
```

## f) Input Validation (menus):

There are various methods for checking user input. It also depends if the operation require only one selection or more.

A) Typical Solution:

Typically a switch statement is the most efficient; in that case, the "default" statement will catch the invalid input

B) Too many, or Two or more selections validated together:

For example: selecting residency status (I or O) for school tuition and selecting if you want boarding included (Y / N).

B.1) for such a simple example, an *"if"* statement should be enough:

```
if ((state == 'I' || state == 'O') && (room == 'Y' || room == 'N'))
```

B.2) for larger number of options, say state options are (i, n, o p) and room options are (y,n,m,x,w)

There are many ways including loops etc. One clever way is using string's find function:

```
string s="inopymxw";
if (s.find(a) != string::npos;){
     cout<<"a valid option";
}
```

# 6) Loops

- **Never, EVER, use the following in a loop: return, break, goto.**

## a) Loop termination:

1) A loop MUST terminate once the result is known. It cannot be allowed to continue needlessly
2) A loop can only terminate though its condition. In other words: the loop condition must contain all needed logic to terminate the loop whenever needed (including early terminating).

Example: **You can NOT** terminate counter-based "for" loop by changing the counter value inside the loop. The Boolean expression of the loop condition must have additional condition to accommodate loop termination instead

## b) `for` loops:

***1)*** For loop is used when you have control over when the loop will end based on something you update every iteration ***(does not have to be a counter)***
2) If a variable will not be used outside the loop, it must be initialized within the loop

```
for (int x=0; x<1000; x+=10)
```

3) Counter can increment by any value with the for loop
4) for loop can have more than one counter simultaneously

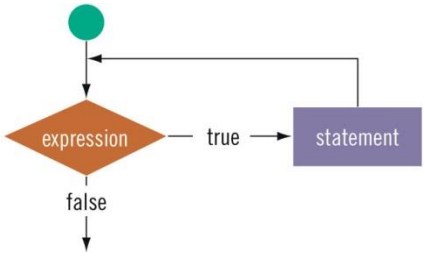
```
for (int x=0, y=100; x<1000; x+=10, y-=10)
```

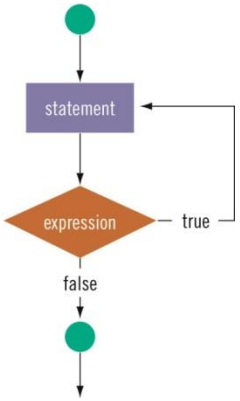
5) If you are using a "for" loop that also relies on a flag, remember to check for the flag in the loop condition

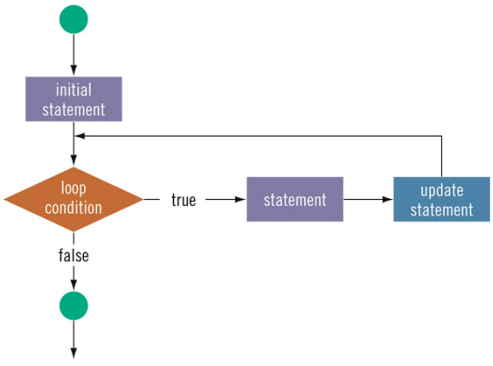
## c) Which Loop to Choose?

There are 3 types of loops. Each serves a specific need. It is true that you can programmatically get any type to do the job of the other types, but it does not mean this is the proper way to use it.

| Condition-Controlled | | UPDATE-Controlled |
|---|---|---|
| **`while`** | **`do-while`** | **`for`** |
| Pre-test | Post-test | Pre-test |
| Executes 0 or more times | Executes 1 or more times | Executes 0 or more times |



# 7) File I/O

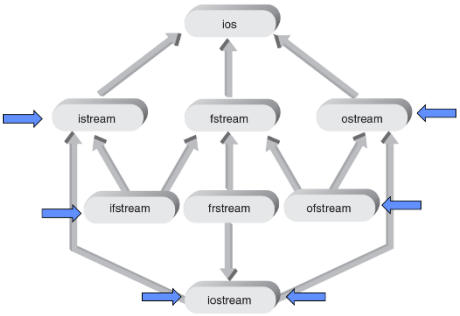
1) Common mistake: Opening output file for input or vice versa.
2) There are 5 steps to using a file, must be followed no exception:

| | | |
|---|---|---|
| 2.1 | Name the file | `string filename ("myfileName.txt");` |
| 2.2 | Open the file | Either:<br>`ifstream myInputfile (filename);`<br>or<br>`ifstream myInputfile;`<br>`…..`<br>`myInputfile.open(filename);` |
| 2.3 | Check if it is open<br>*(only check using the object itself, DO NOT use functions like: isgood, etc)* | `If (myInputfile){`<br>//the file is open, write code here<br>} |
| 2.4 | Read using the getline statement in the loop condition<br>*(do NOT use eof() etc only getline..) .*<br>**Note**: exception to the rule if you are reading one-character-at-a-time using get(). | `While (getline(myInputfile, myline)){`<br>*//process the line. The loop will terminate at end of file.*<br>} |
| 2.5 | When you will no longer need to use the file, **close it** | `myInputfile.close();` |

3) Remember:
   - **o**fstream: Output file stream, do not need added flags
   - **i**fstream: Input file stream, does not need added flags
   - **f**tream: General-purpose file stream. Must have added flags
4) Flags can be combined using bit-wise "OR" (like logical OR but only one character) "|"
5) Typical flags are:
   - ios::in ios::out ios::app
6) **Rule-Of-Thumb: once you open a file, you must immediately check if it was opened before you attempt any file operation.**
7) **Rule-Of-Thumb: When you open a file SUCCESSFULLY, you must close it when done.**
8) **REMEMBER: End-Of-File (EOF) is a character by itself. This means it requires 1-extra read operation before EOF flag is triggered. In other words: for EOF flag to be triggered, you must read the EOF character 1st.**
9) *(TRACK-A)* To read from a file you must put the read operation as part of the loop condition itself in order to avoid the extra read inside the loop:

```
while (getline(myFile, str)){
     cout<<str<<endl;
}
```

10) *(TRACK-A)* Relation between file stream and console stream:




- Because they inherit from the same base, they can be used as one datatype in function parameters.
- Example:

```
void printline (ostream & st){
   st<<"Hello there, how are
you"<<endl;
}
int main(){
      ofstream of("outfile.txt");
      printline (of); //will write to the file
      printline (cout);//will write to the screen
      return 0;
}
```