Common Mistakes, and Logic Guide

(updated: Saturday, August 22, 2015)

This document lists mistakes and observations derived from assignment 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.

Contents

Coding Rules

1. `<algorithms>` library not allowed unless explicitly said otherwise
2. `auto` data type not allowed unless explicitly mentioned
3. 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`.
4. Never use global variables
5. Do not repeat code: If you write the same lines of code more than once, then you must re-examine your logic. Maybe you need to use a function?
6. All literal values that could one day change must always be declared as constants *(example: prices, rates, number of items, etc)*
7. dgfdfgdfgdf

INPUT / OUTPUT

1. Never use `cin` , always use `getline()`
2. Use `stringstream` to convert to/from strings

CHARACTERS

1. Always use character functions (cctype library)

•

LOOPS

1. All loops must have 1-entry-point & 1-exit-point

•

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. Always utilize member functions
4. Use `stringstream` to convert to/from strings

•

VECTORS & Arrays

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

Important Rules:

- You must use proper operators and built-in functions. Example:
  - variable = variable + 10 → variable += 10;
  - variable * variable * variable → pow(variable,3)
- Never assume sizes or values inside a function unless if they were passed in the parameter list.
- The function is a self-contained black-box. Nothing outside the function exists except to what was passed in the parameter list
- Classes should not have public variables, unless explicitly specified in a question
-

DOs & DON'Ts

GOOD (i.e. DO)

- Use `const` variables
- Always name `const` in ALL CAPS
- Always capitalize structure datatypes.

BAD (i.e. AVOID)

- Function prototypes
- Typedef
- Static variables (esp. in Recursion)

TERRIBLE (i.e. DO NOT EVER)

- Use global variables *(aside from consts)*
- "`return`" from anywhere except the end of the function *(ok only for Recursion base case)*
- Use *`break`* in a loop.
- Use "`goto`"

Basics, I/O, Output Formatting, Strings:

Literals & Constants:

- Literals can NOT be used directly into a program. They must be declared as constants at the beginning of program.
- Literals are: anything that has a fixed value throughout the program *(except for output string literals, though, in multi-lingual programs these strings also get declared as constants)*.
- **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;
- **Rule-Of-Thumb**: any fixed number (price, tuition, PI, discount rate, etc) must be declared as a constant.

Stringstream & Data Input:

- **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.
- Generally, it is best to use the *getline* and *stringstream* method for all user data input.
- 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
```

- For a list of data that may require a loop or processing in more than one statement, a variable will be needed
- **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);
}
```

- 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
```

- REMEMBER: if inserting a new value into a previously-used stringstream variable *(i.e. reusing the stringstream variable)* , you need to clear the stream 1st using .clear() function:

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

Strings:

- Using string.**find**(): 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
}
```

- 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;
}
```

Miscellaneous

Random Number Generation

- Classic Method **<u>(DO NOT USE)</u>**:
  - 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`.
- **<u>C++11</u>**
  - <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)*

Date & Time

- Classic Method (platform dependant)
  - <u>Windows:</u>
    - #include <windows.h>   //use: Sleep (milliseconds)
  - <u>Mac OSX:</u>
    - #include <unistd.h> //use usleep(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
```

Selection & Logical/Relations:

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

Boolean Values

Boolean values (true/false) can interchangeably use 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.

Range Checking

- 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) part in red are wrong:

You won't get here unless average is less than 100, so why check again?

```
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;
}
```

Stacking & Nesting

- 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.

- 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";
}
```

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

- **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.

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

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";
}
```

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";
}
```

File I/O

- Common mistake: Opening output file for input or vice versa.
- 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
- Flags can be combined using bit-wise "OR" (like logical OR but only one character) "|"
- Typical flags are:
    - ios::in ios::out ios::app
- **Rule-Of-Thumb: once you open a file, you must immediately check if it was opened before you attempt any file operation.**
- **Rule-Of-Thumb: When you open a file SUCCESSFULLY, you must close it when done.**
- **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.**
- Best way to read from a file is to put the read operation in the loop condition itself in order to avoid the extra read inside the loop:

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

-

Loops

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

for loops:

- If counter variable is not used outside the loop, it must be initialized in the loop's init section

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

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

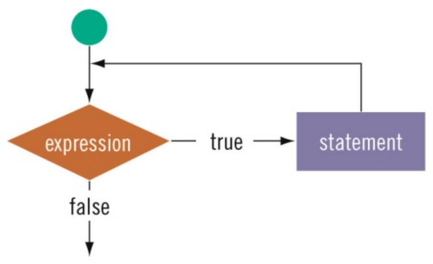
```
for (int x=0, y=100; x<1000; x+=10, y-=10)
```

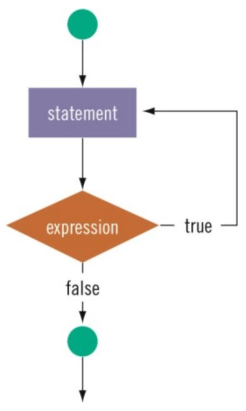
- If you are using a "for" loop that also relies on a flag, remember to check for the flag in the loop condition

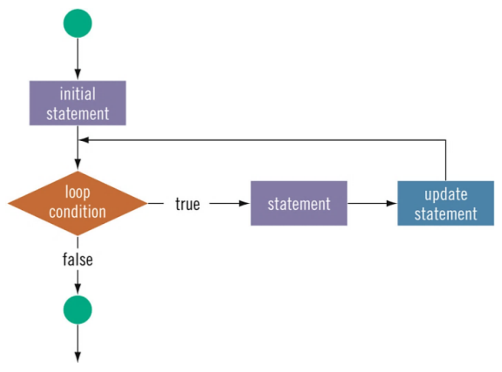
Which Loop To Choose?

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		Count-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


expression
true
statement
false


statement
expression
true
false


initial statement
loop condition
true
statement
update statement
false