* Optimization Tips

(for beginners)

Optimization Tips (for beginners)

- **Rule # 1:**
  - **FIRST**… Write **CORRECT** code that produces the **CORRECT** result.
  - **THEN**… **REVISE** it to make it more **efficient**.

- ✓ Avoid calling a function multiple times if the return value is always the same.

```
// Vector v contains several elements.

for (unsigned int i = 0; i < v.size(); ++i)
    cout << v[i] << "";

// Call to the function size will execute at each iteration and it will
// always give the same result. Use a variable instead.
```

A better way to do this…

```
int size = static_cast<int>( v.size()); // Function size does NOT return
                                        // an int => need to cast it.
for (int i = 0; i < size; ++i)
    cout << v[i] << "";
```

- ✓ Do not perform unnecessary operations multiple times when you can reduce it to one statement.

```
while (looping n times)
{
        // some code
        // keeping some count
        ++count;
}
// Call to the function size will execute at each iteration and it will
// always give the same result. Use a variable instead.
```

A better way to do this…

```
while (looping n times)
{
        // some code
}
// if possible, set the count outside the loop
count = // some value that is already known
```

- ✓ Do not check the same value twice.

```
// Finding the minimum value.
int min = a[0]; ... // Setting the minimum value to the first element in the array.

for (int i = 0; i < numOfElements; ++i)
    if (a[i] < min) ... // the first element is already the minimum value.
```

A better way to do this…

```
// Start the loop from the second element.
int min = a[0];
for (int i = 1; i < numOfElements; ++i)
    if (a[i] < min) ... // the first element is already the minimum value.
```

✓ Declare your variables only when you are ready to use them.

```
int n1 = 0,
    n2 = 100,
    n3 = 200;
cout << n1;
// Several lines of code and only n1 is being used.
// A better implementation would declare each variable right before using it.
```

A better way to do this…

```
int n1 = 0;
cout << n1;
// other code
int n2 = 100;
cin >> n2;
// more code
int n3 = 200; ...// n3 is not needed until this point
if (n3 < n2)…
```

- A nested if...else statement performs much faster that a series of single-selection if statements.

```
if (grade == 'A') ...
if (grade == 'B') ...
if (grade == 'C') ...
if (grade == 'D') ...

// By writing a nested if...else statement, there is a possibility of early
// exit after one of the conditions is satisfied.
// Tip: When more than one if, test the conditions that are more likely to
// be true at the beginning.
```

A better way to do this…

```
if (grade == 'A') ... // If this is true, none of the others will be considered.
else if (grade == 'B') ...
else if (grade == 'C') ...
else if (grade == 'D') ...
```

- ✓ **Passing an object by reference is good for performance reasons, because it can eliminate the pass-by-value overhead (copying the data).**

```
void func(string str, vector v);
// The function will make a copy of the string and the vector.
```

A better way to do this…

```
void func(string& str, vector& v);
// The function will receive only the respective addresses.


// When passing by reference, you need to be careful. If the original
// string and vector do not need to be modified, then you should add
// a const modifier.
void func(const string& str, const vector& v);
```

- ✓ Whenever possible, initialize member object explicitly, instead of using the default constructor and then re-setting the values.

```
MyClass obj;     // Will call default constructor and initialize member variables
                 // to default values.

obj.setMemberVar1(someValue1);
obj.setMemberVar2(someValue2);
obj.setMemberVar3(someValue3);
```

A better way to do this…

```
// If available, use the overloaded constructor instead; it is faster and more readable.
MyClass obj(someValue1, someValue2, someValue3);
```

✓ When to use cout and when to use cerr.

```
cout
// console output
// To display standard output stream.
cerr
// console error
// To display standard output stream for errors.
// Can be use to collect errors and write them to a file.
```

A simple example

```
// Assume the user is searching for a specific book in the database.
if ( the database is empty )
        cerr << "There are no books in the database.");
else if ( this specific book is not in the database )
        cout << "The book is not available at this time.");
else...
```

- ✓ Avoid calling an accessor function when you have direct access to the private member variables of a class.

```
void MyClass::func(const MyClass& otherObject) const
{
     cout << getMemberVar() << otherObject.getMemberVar();
}

// Why call the accessor function when you can access the value of the
// member variable that belongs to the same class of the object?
```

A better way to do this…

```
void MyClass::func(const MyClass& otherObject) const
{
     cout << memberVar << otherObject.memberVar;
}
```

✓ Do NOT make a FOR loop look like a WHILE loop.

```
int i = 0;
for ( ; i < limit;  )
{
     // some code
     ++i;
}

// The loop above looks like a WHILE loop, doesn't it?
// Make it a WHILE loop to improve readability.

int i = 0;
while( i < limit )
{
     // some code
     ++i;
}
```

Avoid!

- ✓ **Never** use an EQUAL (==) or NOT (!=) in a FOR loop.
  Never use a LOGICAL operator in a FOR loop.
  Use a WHILE loop instead.

// This can produce unexpected results.

```
for ( int i = 0; i != someValue; ++i)
{
    // do something…
}
```

Avoid!

// Here there are too many conditions. Using a WHILE loop is much more readable.

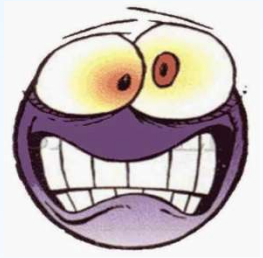
```
for ( int i = 0; i < someValue || j < someOtherValue; ++i)
{
    // do something…
}
```

Avoid!

*Note: A **FOR** loop is meant to have 3 conditions only. If we change the format, the statement becomes less readable.*

✓ Avoid redundancy.

```
bool func(int num)
{
     if (num < 3)
          return true;
     else
          return false;
}
```



// (num > 3) is a Boolean expression and will be either true or false.

A better way to do this…

```
bool func(int num)
{
     return (num < 3);
}
```

And more redundancy!

```
bool done = false;
...
if (done == false)
{
     ...
}
```

// done is a Boolean expression and will be either true or false.

A better way to do this…

```
if (!done)
{
     ...
}
```

And one more thing…

- Make your code **READABLE** for other programmers AND yourself
  - Choose descriptive identifiers.
  - Do not abbreviate (prefer `numOfElem` to `noe`).
  - Separate your implementation in meaningful blocks of code.
  - Leave a space around operators (prefer `a = 2 + 3` to `a=2+3`).
  - Keep your indentation style consistent (either open a bracket at the end of a statement or on a new line).
  - Keep statements short to avoid horizontal scrolling (no more than 70 characters).
  - Keep naming scheme consistent (whether you decide to use camelCase or under_scores, use it throughout the whole program, BUT for this class, you are **required** to use **camelCase**.)

* Optimization Tips (end)