Some Do’s & Don’ts in C++

Unnecessary temporaries

- Problem:
  - If you pass in an object by value, you are creating a copy of it. This is a temporary variable created, and this may be expensive.

- Solution:
  - Pass it in by const reference.

Inlining

- You can inline single statement methods.
  - Example:
    - Get and Set methods like
    - int GetValue() const { return m_Value; }
    - void SetValue(int value) { m_Value = value ; }

- Be careful inlining functions with more than a couple of lines of code.

Use initialization lists

- This is to prevent an extra initialization (technically initialization is inside the initialization list, but outside, i.e., in the constructor body, it is assignment).

- Example:
- Do
  - MyClass (const string & name)
    : m_Name (name)
    {}

  Instead of :

  - MyClass (const string & name)
    {
        m_Name = name;
    }

Initialization lists

- Do not depend on the order of initialization in initialization lists.
- Example;
  - MyClass :: MyClass (int a, int b)
    : m_ValueA (a),
      m_ValueB (b),
      m_Sum (a + b)
    {}

  - Potentially incorrect way of doing above would be :

  - MyClass :: MyClass (int a, int b)
    : m_ValueA (a),
      m_ValueB (b),
      m_Sum (m_ValueA + m_ValueB) ← ***m_ValueA*** *or* ***m_ValueB***
      {} *may still be uninitialized*

NOTE: Order of initialization is the order in which data members are **declared** in the class.

Pre increment iterators

- If you don't really need to post increment iterators, you should prefer pre increment because those are usually cheaper than post incrementing iterators.

- Example
  - vector <MyClass> myClassVector;
    vector< MyClass>::iterator iter = myClassVector.begin();
    vector< MyClass>::iterator endIter = myClassVector.end();

    for (; iter != endIter; ++ iter) ← Don't say iter ++ in this case
      …

Complex initialization of global/static

- You should avoid complex initialization of global variables or static data members because if the code involved in such initialization throws an exception, you cannot catch it.

Order of static initialization

- You should not depend on the order of initialization of static data members.
- What this means is that if you use a static data member of a class A for the initialization of static data member of another class B, there is no guarantee that A's static data members have been initialized before B's static members.
- Example:
  - int B::m_S_staticDataMember = A::m_StaticDataMember;
  - Here, A::m_StaticDataMember may not be initialized.

Writing to static data members

- Be careful when updating static data members in a multi threaded application. More than one thread can potentially modify the data, causing incorrect results.
- Use critical sections / semaphores / whatever method is available on your platform to guard writes to the static data member.
- NOTE: If you are using a read-only static data member, this is not an issue.

Const correctness

- Make your methods and parameters const correct, i.e., if a method is supposed to be a const, declare it as a const.
- Example:
  - Get methods are const methods.

- Otherwise you can get into ripple effects, like if you are writing a method that should be const, and you make a call to, say, GetValue() type of a method which is not declared as a const, then the method you are writing also cannot be declared as a const.

- So, not making GetValue() a const made it so that this new method also cannot be made constant.

Virtual destructors

- If you expect a class to be used as a base class, and
  - Either the derived class(es) are going to do dome memory/resource allocation in the constructor
  - OR
  - The derived class(es) consist of objects (composition) that do some memory/resource allocation in their constructors.

  - THEN

  - You should make the destructor of the base class virtual.

Copy constructor, =, ~

- If you have reason to define a copy constructor, then you have the same reason to define an assignment operator and a destructor.
- NOTE: You don't need to define the destructor if it does not have to do anything.
- By the same token, if you don't want an object to be copied, you should make the copy constructor and the assignment operator private. Otherwise, the compiler will generate a default which will get used.

Public, protected, then private

- Order of declaration inside the class should be :
- Public first
- Then protected
- Then private
- This is because the protected and private sections have implementation details, and clients need to know the interface of the class, so they should not have to look through the protected / private declarations to get to the public interface.

Pointers are passed by value

- Example:

```
- void FreeMemory( char * ptr )
  {
    delete ptr;
    ptr = NULL;          ← This will not be reflected at the caller level
  }
```

Fix is to do either of the following:

```
void FreeMemory( char * & ptr )
{
  delete ptr;
  ptr = NULL;
}
```

```
void FreeMemory( char * * ptr )
{
  delete * ptr;
  *ptr = NULL;
}
```

Reference data member

- If a class has reference data members, these should not be initialized with an argument that is passed in by value.

- Example:
  - Let MyClassA have m_B as a reference data member, so it is declared inside the MyClassA class declaration as :
    - MyClassB & m_B;

  - MyClassA :: MyClassA (int value, MyClassB b) ← *Incorrect*
    : m_B (b),
    m_Value (value)
    {}

  - MyClassA :: MyClassA (int value, MyClassB **&** b) ← *Correct*
    : m_B (b),
    m_Value (value)
    {}

Using namespace

- Never put a `using namespace` directive in a header file as it might be included in a different context and can cause clashes.
- This is because the using directive causes all classes/functions/objects inside the name space to become global.

- Its ok to put this inside the .cpp file.
- Its definitely ok to put it inside a function.

```
int main()
{
   using namespace MySpace;
   MyString s1;
}

void foo ()
{
   MyString s2;        ← Will be a compile error now, will have to say
                              MySpace::MyString  s2;
}
```