USEFUL TIPS AND TRICKS FOR WRITING EXTENDIBLE AND USEFUL KERNEL OF PROGRAMS FOR PROCESSING, ANALYSIS AND VISUALIZATION OF MEDICAL IMAGES IN C++

Rafał Henryk Kartaszyński, Paweł Mikołajczak

Department of Computer Science, Maria Curie-Skłodowska University
Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Streszczenie: Niniejszy artykuł poświęcony jest dyskusji na temat różnych interesujących i przydatnych sztuczek programistycznych, mogących bardzo pomóc przy tworzeniu jądra programów graficznych, nie tylko do przetwarzania, analizy i wizualizacji danych medycznych. Omówiona została notacja węgierska, wzorzec singleton, automatyczny singleton, fabryka klas oraz implementacja listy operacji przeprowadzanych na obrazach.

Abstract: In following article we will discus several very interesting and useful programming tricks, which can be very helpful during writing kernel of programs for image processing, not only for processing, analysis and visualization of medical data. Hungarian notation, singleton pattern, automatic singleton, class factory and implementation of list of performed image transformation are elaborated.
Introduction

It is very simple to understand why C++ programming language becomes so popular among programmers. Its effectiveness and portability does not diverge a lot from C language, and in addition it offers merits of object oriented language. Inheritance is real power, it only must be correctly designed and implemented. Although principles of object programming have been invented to simplify design, portability and program extension, truth is that badly designed C++ programs can be worst from badly written C program.

We would like to discuss some very interesting programming tricks, which are using most important advantages of C++ language. Of course we will not even try to describe all issues, which must be known to programmer of image processing applications. We will only point out areas where some improvement can be made to produce more portable and extendible kernel code. In addition we would like to encourage reader to look through books provided in references. These positions are invaluable help in getting to know nuances of object programming and C++ language.

Hungarian notation

Charles Simonyi is credited with first discussing Hungarian Notation [1]. It is a variable naming convention that includes C++ information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc) [2].

Main principle of Hungarian notation is to precede variables names with identifier corresponding to type of variable For instance, integer variable named SomeVariable would be named iSomeVariable. Apart from variables types, pointer also can be indicated in the same manner. Pointer to class Foo would be: pFooObj. A prefix joining, to provide more information about variable, is also possible. For example prefix for pointer to integer number is pi and pp for pointer to pointer.

Often before prefix information about variable scope is given. Class variables should start with m_, so integer class variable would be named m_iSomeVariable. Global variables are indicated with g_, and in some variants of notation, also static variables s_ can be found. Every company and programmer seems to have their own flavour of Hungarian Notation. Following table provides some detailed information about notation:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>boolean</td>
<td>bool bStillGoing;</td>
</tr>
<tr>
<td>c</td>
<td>character</td>
<td>char cLetterGrade;</td>
</tr>
<tr>
<td>str</td>
<td>C++ String</td>
<td>string strFirstName;</td>
</tr>
<tr>
<td>si</td>
<td>short integer</td>
<td>short siChairs;</td>
</tr>
<tr>
<td>i</td>
<td>integer</td>
<td>int iCars;</td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td>li</td>
<td>long integer</td>
<td>long liStars;</td>
</tr>
<tr>
<td>f</td>
<td>floating point</td>
<td>float fPercent;</td>
</tr>
<tr>
<td>d</td>
<td>double-precision floating point</td>
<td>double dMiles;</td>
</tr>
<tr>
<td>ld</td>
<td>long double-precision floating point</td>
<td>long double ldLightYears;</td>
</tr>
<tr>
<td>sz</td>
<td>Old-Style Null Terminated String</td>
<td>char szName[NAME_LEN];</td>
</tr>
<tr>
<td>if</td>
<td>Input File Stream</td>
<td>ifstream ifNameFile;</td>
</tr>
<tr>
<td>is</td>
<td>Input Stream</td>
<td>void fct(istream &amp;risIn);</td>
</tr>
<tr>
<td>of</td>
<td>Output File Stream</td>
<td>ofstream ofNameFile;</td>
</tr>
<tr>
<td>os</td>
<td>Output Stream</td>
<td>void fct(ostream &amp;rosIn);</td>
</tr>
<tr>
<td>S</td>
<td>declaring a struct</td>
<td>struct SPoint{...}</td>
</tr>
<tr>
<td>C</td>
<td>declaring a class</td>
<td>class CPerson{...}</td>
</tr>
</tbody>
</table>

The following table contains letters that go before the above prefixes.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>unsigned</td>
<td>unsigned short usuStudents;</td>
</tr>
<tr>
<td>k</td>
<td>constant formal parameter</td>
<td>void fct(const long</td>
</tr>
</tbody>
</table>
### Singleton pattern

We can use singleton pattern [3] when single global object must be shared between many classes and / or modules. Simple not local object can be created, however this method has several drawbacks; among other things they are concerning determination of object creation time in relation to other objects creation time. Singleton pattern solves this problem by forcing access through class storing static object. Basic implementation of template can look as follows:

```cpp
class Singleton1
{
public:
    Singleton1& Instance()
    {
        static Singleton Obj;
        return Obj;
    }
private:
    Singleton1();
};
```

Above code solves swiftly described problem. If we want to create new classes on basis of above example, we need something more scalable and elegant. Changing project and requiring more clear intervention during creating and destroying object, we may extend simple singleton pattern and enable its extension.
#include <iostream.h>
#include <assert.h>

class SingletonBase
{
public:
    SingletonBase()
    {
        cout << "SingletonBase created!" << endl;
    }
    virtual ~SingletonBase()
    {
        cout << "SingletonBase destroyed!" << endl;
    }
    virtual void Access()
    {
        cout << "SingletonBase accessed!" << endl;
    }
    static SingletonBase* GetObj()
    {
        return m_pObj;
    }
    static void SetObj(SingletonBase* pObj)
    {
        m_pObj = pObj;
    }
protected:
    static SingletonBase* m_pObj;
};
SingletonBase* SingletonBase::m_pObj;

inline SingletonBase* Base()
{
    assert(SingletonBase::GetObj());
    return SingletonBase::GetObj();
}

// Create a derived singleton-type class
class SingletonDerived : public SingletonBase
{
public:
    SingletonDerived()
    {
        cout << "SingletonDerived created!" << endl;
    }
    virtual ~SingletonDerived()
    {
        cout << "SingletonDerived destroyed!" << endl;
    }
    virtual void Access()
    {
        cout << "SingletonDerived accessed!" << endl;
    }
protected:
};
inline SingletonDerived* Derived()
{
    assert(SingletonDerived::GetObj());
    return (SingletonDerived*)SingletonDerived::GetObj();
}

// Using the code...
// The complex singleton requires more work to use, but is
// more flexible. It also allows more control over object
// creation, which is sometimes desirable.
SingletonDerived::SetObj(new SingletonDerived);
// Notice that the functionality has been overridden by the
// new
class, even though accessing it through the original
method.
Base()->Access();
Derived()->Access();

// This variation on a singleton unfortunately requires
both
// explicit creation and deletion.
delete SingletonDerived::GetObj();

Modified version of singleton class is not that simple in its part with
constructors and destructors, but global access, what is main feature of singleton,
remains unchanged. Furthermore, thanks to access point elaborated in call point,
code is becoming clearer from perspective of user.
Singleton templates are often used in situations when one would use global
object or pointer, to access class clone. Managing class, when only one clone
is required, may be a good example. Classes, which are managing sound, user
interface or graphics, are good candidates to become singleton classes. Each of
these subsystems is created at program start and is being deleted when program
ends.
Some of these subsystems can be realized with global functions and static
variables. Example of such solution is system of dynamic memory allocation –
functions malloc() and free(). However, this cannot be called singleton,
because its elements haven’t been placed in class and cannot be represented with
one clone of that class. There are no reasons for placing this system in class and
using it as singleton.
Good example of singleton is file manager. We could call it DataMgr and
implement two method, for example, GetDataSet() and UseDataSet().
Its purpose would be reading files (for instance: DICOM) containing medical
data, converting it to object used by graphical system, making them available for
this system, and deleting them when they are not needed. In graphical program
only one clone of DataMgr is required, so this class should be used as singleton.
Why it is worthy to use singleton? Firstly, they ensure clarity of program
code, because their names are very important. Using appropriate nomenclature
(for example –Mgr, –Api, –Global) will determine how class should be used.
Singleton also ensures clarity of notation. Each object in C++ must belong to
something. Property template depends on the program, but mostly it corresponds
to multi-level hierarchy, in which each higher level has set of descendant objects,
which also have descendant objects. All object are making some function
available, thus access to descendants is possible. For example to gain access to
DataMgr clone, sequence of functions must be written GetApp()->
>GetServices()->GetGui()->GetDataMgr(). Each functions returns
reference to descendant object. This system is uncomfortable and non effective
due to several dereferences. Singleton soles this issue, because it is treated as
global object.

Why should we use this pattern, instead of creating global object or
pointer? There are several reasons. First of all, when creating global objects,
access to object through single function is simpler than writing extern in all
files, enabling access to global object. Furthermore moment of initialization can
be precisely controlled. Secondly, when using pointer instead of object, C++ will
control each reference to object. Thirdly, when we create singleton for
inheritance, using mentioned method, we may widen basic class with compliance
with existing basic class.

To check this scalability, let us imagine following example: library A uses
singleton class described earlier. Library B, to run, must use library A, so
depends on this class and contains its header files. Application C uses both
libraries, but due to elements connected with program, modification of library
A is required. Instead of creating new version of library (and lose all new updates
of original library made by competitive project), we should create new class (D)
inherit from library A. When we require application to be responsible for
object allocation (as a part of singleton template), we may replace class A with
derivative of class D. After opening newly named access function, returning
pointer to class D (not to class A), we will have access to all new functions of
class D. However class B still uses old access function, which returns pointer to
class A, so old functions will act as before. One should remember, that virtual
functions can be overridden, but new actions must be compliant with old, to
ensure backward compatibility.

More general solution uses template for automatic defining singleton
pointer and is occupied with setting, maintaining and deleting it. It may also
check (with assert()) if we are not trying, by mistake, to create more than
one clone. And what is the best, we may have all functions for free, inheriting
from this simple class.

template <typename T> class Singleton
{
    static T* ms_Singleton;

public:
    Singleton( void )
    {
        assert( !ms_Singleton );
        int offset = (int)(T*)1 - (int)(Singleton<T>*)1;
        ms_Singleton = (T*)((int)this + offset);
    }
    ~Singleton( void )
    {
        assert( ms_Singleton );
        ms_Singleton = 0;
    }
    static T& GetSingleton( void )
    {
        assert( ms_Singleton );
        return ( *ms_Singleton );
    }
};
To change any class into singleton, three steps must be performed:

1. Inherit publicly singleton<MyClass> in the class MyClass.
2. Remember to create the clone of the class MyClass before using it. Manner of creating is of no importance. Let the compiler worry about creating it as global or local static. One may take care of it personally by using construction new and delete in class containing it. Regardless of time and way in which clone was created, it will be tracked and can be used in rest of the system.
3. Call MyClass::GetSingleton() to use object in any place. One can be lazy and write #define g_MyClass MyClass::GetSingleton() and use g_MyClass as global object.

Following example shows usage of class

class DataMgr : public Singleton <DataMgr>
{
public:
    MEDData* GetData( const char* name );
    // ...
};

#define g_DataMgr DataMgr::GetSingleton()

void SomeFunction( void )
{
    MEDData* chestCT = DataMgr::GetSingleton().GetData( "chest" );
    MEDData* brainMRI = g_DataMgr.GetData( "brain" );
    // ...
}

Purpose of Singleton class is automatic registration and deletion of registration of each derivative class (MyClass) clone during construction and destruction. Deriving MyClass from Singleton <MyClass> we simply inherit such behaviour. It doesn’t increase class size, only adds several automatic functions calls.

How does it work? All important work is done in Singleton constructor, where relative address of derivative class is counted and written into singleton pointer (ms_Singleton). Let us point out that derivative class can be
derived from more than one Singleton class, but in this case this of MyClass may be different than this of Singleton class. Solution is to take not existing object from memory at address 0x1, cast it onto both types and check difference. Difference is in fact distance between Singleton <MyClass> and its derived type MyClass, which can be used for calculating singleton pointer.

**Factory pattern**

Factory pattern concern organization of object creation. Pattern form has been defined as method enabling abstract classes determination of time when to create exact class implementations. This method is often required by application skeleton or other class hierarchies. However graphical applications programmers are often using specific kind of factory pattern – they are using factory objects with queue of created objects, placed in main class, mainly as single method.

This means that one object is responsible for creating other objects, often tied together with common base class. This class mostly takes form of class with one method, which argument is certain identifier and returning created object. Allocating objects in one place has following merits:

- Dynamic memory allocation is time consuming, so we want to monitor it. Creation of object in one place significantly facilitates this.
- Often, for all objects in particular class hierarchy, same initialization method must be called. If we gather all object allocating code in one place, we will be able to easily carry out operations often executed on objects (for example placing them in resources manager).
- Factory pattern increases extendibility, because objects can be derivatives of existing pattern. Giving new class identifier (which can be easily comprised not in code but in data), we are enabling extendibility, during action, by new classes, without modification existing, basic code.

Last point stresses out extendibility, as a merit of using factory pattern. For this reason one should avoid using simple functions and static classes, because derivative classes cannot be created from them. Following code presents simple class factory, used for dynamic creation of graphical operations classes. Factory is organized as singleton class and therefore can be used in any part of application code.

class TGOperationClassFactory : public Singleton<TGOperationClassFactory>
{
private:
public:
As we can see there is nothing special in writing factory method, but gathering code creating objects in one place increases code organisation and extendibility.

**Graphical transformation listing**

Very important feature of application for image analysis can be listing of operations performed by user on displayed image. As we know order of transformation is very important, for example different result will be achieved by applying dilation filter after erosion, and different for reverse combination of these filters. Therefore providing user with list of transformation performed by him will largely increase usefulness of program. We would like not only to view
performed operations but also delete selected, change their order or execute only selected.

How to organize such a mechanism? Obvious answer to this question is using STL library [7][8]. STL provides programmer with several storage containers, i.e. stacks, vectors, lists, double direction queues, maps. These storages, and all operations on them (for example sorting) are implemented very effectively and optimized for best performances, what more, objects stored there can be of any type. Therefore idea of using STL for mentioned purpose is very simple. Firstly we must define virtual class after which all newly added operations will inherit. Pointers to this class will be stored in some kind of container, thus our goal will be achieved.

class TGOperation
{
public:
    virtual ~TGOperation() {};
    virtual void SetData(TDataStruct<DPoint> *ds) = 0;
        //set data which will be processed by operation
    virtual TDataStruct<DPoint>* GetData() = 0;
        //return processed image data
    virtual void ExecuteOperation() = 0;
        //name of operation
    virtual GraphicOperations OperationID() = 0;
        //operation identifier
    virtual bool IsCumulative() = 0;
        //some operations are cumulative, i.e. when their
        //are appearing
            //one after another they should be cumulated to
            //single operation
            //for example threshold on different intensity
    virtual char *Name() = 0;
};

Second step is to chose which container is most suitable for us. We have chosen vector, because it is sufficient enough for our purpose. It is providing direct access to data, unlike lists and is less complex than maps and double direction queues. Of course, changing order of elements is also quicker. Vectors are behaving almost identically as standard C arrays, with one small difference: dynamic change of size. They have been implemented as arrays, which are periodically reallocating memory and moving data to new array. For programmer it means two things. First of all, vectors can allocate more memory, than there are actually needing, because they are expecting to “grow” at any moment. Secondly, adding new element at the end of vector takes constant time, theoretically. In other words, some functions responsible for adding new element will require significant resources for allocating memory, coping existing array into new place.
and releasing not used memory, but these additional work won’t be needed always.

Following code is a skeleton for class managing operation listing (for simplicity some code has been omitted) it also shows how to use vector container:

class TDataMgr : public Singleton <TDataMgr>
{
private:
    vector<TGOperation *> OperationVect;

public:
    TGOperation *GetOperation(int i) { return OperationVect[i]; }

    void ExecuteOperations()
    {
        for(int i = 0; i < OperationVectLen; i++)
            OperationVect[i]->ExecuteOperation();
    }

    void AddOperation(TGOperation *op)
    {
        if((OperationVectLen != 0) && op->IsCumulative()
            && (OperationVect[OperationVectLen - 1]->Name() == op->Name()))
        {
            delete OperationVect[OperationVectLen - 1];
            OperationVect[OperationVectLen - 1] = op;
            return;
        }
        OperationVect.push_back(op);
        OperationVectLen ++;
    }

    void MoveOperationTo(int dest, int source)
    //moves operation from source to dest
    //– let us suppose these are correct indexes
    {
        OperationVect[dest] = OperationVect[source];
        OperationVect[source] = op;
        ListOperations();
    }

    void DeleteOperation(int which) {...};
};
Presented method can also be used to store and list edited, by user, files, read data sets etc., that is why, it is good to implement similar mechanism, adding additional functionality to our program.

Conclusions

Designing good object techniques is not enough. This conception should permeate each peace of code, what, in wider perspective, will save us a lot of troubles and shorten time of project development. Well written object code is elastic, easy to maintenance and more extendible than procedural one.

STL library is very useful tool, available for C++ programmers. Understanding its merits and limitations allows to create code, which is compromise between execution speed and flexibility.

All presented methods have been implemented in program which is currently being developed by us and did significant improvement to its kernel. Adding new graphical transformation is very simple and can be quickly implemented, also process of displaying data (two- and three- dimensional) is clear and easy to extend.

We would like once again encourage reader to look through following books provided in references (for more tips and tricks see especially [4][5][6][9]).

References
