Session of low-level optimization of memory usage in the C++ programs with the total exposure

Author:
Victor Milokum,
Team Leader of Apriorit Inc.

Table of Content

“In computing, **optimization** is the process of modifying a system to make some aspect of it work more efficiently or use fewer resources. The system may be a single computer program, a collection of computers or even an entire network such as the Internet.

Although the word "optimization" shares the same root as "optimal," it is rare for the process of optimization to produce a truly optimal system. Often there is no “one size fits all” design which works well in all cases, so engineers make trade-offs to optimize the attributes of greatest interest.

Donald Knuth made the following statement on optimization: “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” An alternative approach is to design first, code from the design and then profile/benchmark the resulting code to see which parts should be optimized."

(c) wiki

In this article we will try to make our algorithms working faster using the methods of low-level optimization of memory allocation in C++. It should be clear that all methods described in this article should be used very carefully and just in the exceptional cases: usually we have to pay for all low-level optimization elements that we used by flexibility, portability, clearness or scalability of the resulted application.

But if you have exactly that specific case and have no way back – than you’re welcome.

1. **The optimization of the “Concrete Factories”**

I think that a lot of you repeatedly met the classic ”Factory” pattern – or “Concrete Factory” in GoF. terminology:

```
struct IObject
{
     virtual ~IObject(){}
     virtual void DoIt()=0;
};
class CFactory
{
public:
  void CreateSmth(int objType, std::auto_ptr<IObject> * pResult)
  {
         if (iValue<0)
         {
             pResult->reset(new CObject1);
         }
         else
         {
             pResult->reset(new CObject2);
         }
  }
};
```

This pattern is great for avoiding endless if’s and switch’es through over the code, but it has one unpleasant disadvantage. It is concerned with excessive usage of dynamic memory that sometimes affects badly on the C++ program performance. We will try to cure this patter of it – with certain reservations of course.

Let’s follow the factory usage process again:

Action	**Sense**
`std::auto_ptr<IObject> product;`	We define some container for the production, the place where created object will be stored.
`MySuperFactory.CreateSmth(1, &product);`	We create an object in this container by means of the factory.
`product->DoIt();`	We use the object via the defined interface
`mySuperContainer.Add( product );`	or pass the container ownership to somebody else.

To optimize the described life cycle of the production we can use the following statements:

1) The special form of the **new** operator – **placement new** – enables to create objects in the custom “row” buffer. For example:
```
        char buffer[1024];
        new(buffer)CObject(a,b,c);
```
It is very useful taking into account the fact that we can allocate the buffer more effectively than the standard **new** implementation does. But actually the using of **placement new** also adds some difficulties to the developer’s life:

   a) Row buffer should be aligned by the platform-dependant range;
   b) The destructor of the created object should be implemented manually.
2) Stack is the great alternative for the heap. It would be tempting to use buffer on the stack as the container, i.e. to use

   MyWrapperAroundLocalBuffer<ObjectSize, `IObject`> product;

   instead of the original

```
        std::auto_ptr<IObject> product;
```

   The main problem of the container on the stack creation is that it's impossible to allocate an object of the custom (unknown while compiling) size in the standard C++.
3) But as soon as our factory is the concrete one (and not an abstract) and we know about all of its production types, we certainly will be able to know the maximal size of the object-production on the compilation stage. For example we can use the type lists:

```
//--------------------------------
// typelist basics
//--------------------------------
template<class Head, class Tail>
struct Node
{
};
struct NullNode
{
};
```

   We can develop the recursive compile-time function to calculate the maximal size of the object from the types in the list:

```
template <class List>
struct GetMaxSize
{
};
template <class Head, class Tail>
struct GetMaxSize<Node<Head, Tail> >
{
    static const size_t TailSize = GetMaxSize<Tail>::Result;
    static const size_t Result = (TailSize > sizeof(Head) ) ? TailSize : sizeof(Head);
};
template <>
struct GetMaxSize<NullNode>
{
    static const size_t Result = 0;
};
```

   Then we can create the list of all possible production types for the factory CFactory:

```
   typedef utils::Node<COobject1>,
                utils::Node<CObject2>,
                     utils::NullNode
```

```
                        >
                   > ObjectList;
    static const size_t MaxObjectSize = utils::GetMaxSize<ObjectList>::Result;
```

As the result now we can be 100% sure that each produced object will be placed in the buffer of MaxObjectSize (if we've developed the type list correctly, of course):

```
    char buffer[ MaxObjectSize ];
```

It can be easily allocated in the stack.

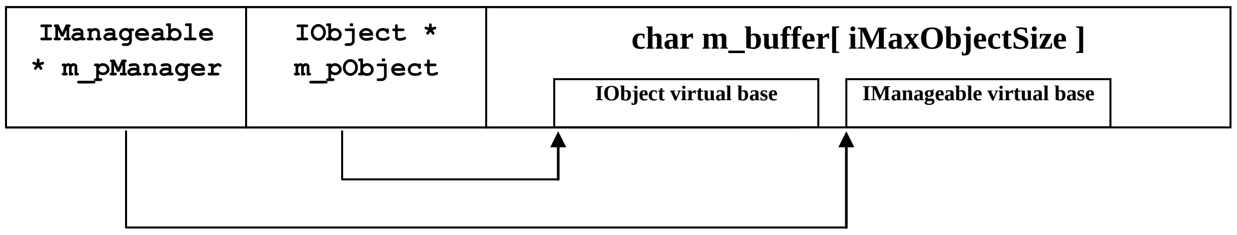
4) As far as our container should be able to store the objects of different types we have a right to expect some help from them in the form of the corresponding interface support:

```
   struct IManageable
   {
       virtual ~IManageable(){}
       virtual void DestroyObject(void * pObject)=0;
       virtual void CreateAndSwap(void * pObject, int iMaxSize)=0;
       virtual void CreateAndCopy(void * pObject, int iMaxSize)=0;
   };
```

I.e. an object that wants to live in our container should be able to:

a) Destroy the objects of its type by the certain address;
b) Use the Create And Swap technology for passing object ownership (optional);
c) Use the Create And Copy technology for object copy creation (optional).

The structure of the container can be represented in the following scheme:



I.e. our container includes row buffer and two pointers to the different virtual bases of the object placed in the row buffer:

a) Pointer to IManageable for the management of the object life cycle;
b) Pointer to the user interface IObject which methods the factory user, in fact, wants to call.

As far as we don't want to spend efforts on adding the support of the **IManageable** interface to the each production class it makes sense to develop the pattern **manageable** that will do it automatically:

```
  // manageable control flags (iFlags field)
  const int allow_std_swap = 1;
  const int allow_copy     = 2;
  const int allow_all      = 3;
```

```
 // class manageable: wrapper, provides binary interface to manage object's life
template<class ImplType, int iFlags>
class manageable:public IManageable, public ImplType
{
    typedef manageable<ImplType, iFlags> ThisType;

    virtual void DestroyObject(void * pObject)
    {
        ((ThisType*)pObject)->~ThisType();
    }
    // CreateAndSwap
    template<int iFlags>
    void CreateAndSwapImpl(void * /*pObject*/, int /*iMaxSize*/)
    {
        throw std::runtime_error("Swap method is not supported");
    }

    template<>
    void CreateAndSwapImpl<allow_std_swap>(void * pObject, int /*iMaxSize*/)
    {
        ThisType * pNewObject = new(pObject)ThisType();
        pNewObject->swap(*this);
    }

    virtual void CreateAndSwap(void * pObject, int iMaxSize)
    {
        if (sizeof(ThisType)>iMaxSize)
            throw std::runtime_error("Object too large: swap method failed");
        CreateAndSwapImpl<iFlags & allow_std_swap>(pObject, iMaxSize);
    }

    // CreateAndCopy
    template<int iFlags>
    void CreateAndCopyImpl(void * /*pObject*/, int /*iMaxSize*/)
    {
        throw std::runtime_error("Copy method is not supported");
    }
    template<>
    void CreateAndCopyImpl<allow_copy>(void * pObject, int /*iMaxSize*/)
    {
        new(pObject)ThisType(*this);
    }

    virtual void CreateAndCopy(void * pObject, int iMaxSize)
    {
        if (sizeof(ThisType)>iMaxSize)
            throw std::runtime_error("Object too large: copy method failed");
        CreateAndCopyImpl<iFlags & allow_copy>(pObject, iMaxSize);
    }

public:
    manageable()
    {
    }
    template<class Type0>
    manageable(Type0 param0)
        : ImplType(param0)
    {
    }
};
```

The pattern is parameterized by object type and flags that define what methods should be supported. For example, if we specify the **allow_copy** flag then the compiler will require the constructor of coping from the object for the **`CreateAndCopy`** method implementation; similarly, if we specify the **allow_swap** flag then the **`CreateAndSwap`** function will be generated – it will be based on the method of the **swap** object, that we should develop ourselves.

So our optimized factory now looks as following:

```
class CFastConcreteFactory
{
public:
    typedef utils::Node<utils::manageable<CObject1, utils::allow_copy>,
                 utils::Node<utils::manageable<CObject2, utils::allow_copy>,
                      utils::NullNode
                     >
                > ObjectList;

    static const size_t MaxObjectSize = utils::GetMaxSize<ObjectList>::Result;
    typedef utils::CFastObject<MaxObjectSize, IObject> AnyObject;

    void CreateSmth(int iValue, AnyObject * pResult)
    {
        if (iValue<0)
        {
            pResult->CreateByCopy(utils::manageable<CObject1, utils::allow_copy>());
        }
        else
        {
            pResult->CreateByCopy(utils::manageable<CObject2, utils::allow_copy>());
        }
    }
};
```

And it is as easy to use as the original one:

```
// usage sample
CFastConcreteFactory factory;
CFastConcreteFactory::AnyObject product;
factory.CreateSmth(-1, &product);
product->DoIt();
// copy sample
CFastConcreteFactory::AnyObject another_product;
product.Copy( &another_product );
```

But the functioning of our creation will be much faster (see fast_object_sample in the attachments).

All source, examples, performance and unit tests can be found in the lib **MakeItFaster** in the attachments. There are also projects for VC++ 7.1 and VC++ 8.0.

See: cmnFastObjects.h, fast_object_sample

2. The optimization by «Arena»

The second optimization method proposed is much simpler but the scope of its application is a little bit smaller.

Let's imagine that we have some iterative algorithm that repeats some action N times:

```
int mega_algorithm(int N)
{
   int Count =0;
   for(int i =0 ; i<N; ++i)
   {
      Count += DoSmth1( );
   }
   return Count;
}
```

Let's suppose that two conditions are met:

1) Algorithm doesn't have any side effects;
2) We can predict the maximal size of the dynamic memory allocated for an iteration (let's name it MaxHeapUsage).

In this case we can use the "Arena" pattern for its optimization. The essence of the pattern is rather simple. For its implementation we:

a) replace the standard **new/delete** with our own ones;
b) register some buffer-arena pBuffer with MaxHeapUsage size before the algorithm starts and set the index pointing on the start of free space FreeIndex to 0;
c) in the **new** handlers we allocate memory directly in the buffer by moving FreeIndex on the allocation value. Naturally, we return ((char *) pBuffer + oldFreeIndex);
d) set FreeIndex to 0 after each iteration and so dispose all memory that iteration has allocated for its needs;
e) unregister our buffer-arena after algorithm finishes.

It's very simple and effective. It's also very dangerous pattern because it's rather hard to guarantee the first condition fulfillment in the production code. But this pattern is good for calculation concerned tasks (for example in the game development).

When using STL containers the concrete instance of the arena can be referred to the container in such a way that the definition and usage of the container will be almost the same to those of the original one, for example:

```
typedef utils::arena_custom_map<int,int,
                          utils::CGrowingArena<1024> >::result Map1_type;
Map1_type map1;

for(int i = 0;i<iNumberOfElements;++i)
```

```
{
   map1[i] = i+1;
}
```

In this example all memory for the object `map1` is allocated in the extendable buffer `CGrowingArena`, allocated memory will be disposed in the destructor when destroying the object.

All source, examples, performance and unit tests can be found in the lib **MakeItFaster** in the attachments. There are also projects for VC++ 7.1 and VC++ 8.0.

See: cmnArena.h/ win32_arena_tests, win32_arena_sample