Static Polymorphism vs Dynamic Brief Note

Andreas Bærentzen

November 26, 2001

1 Introduction

In C++ polymorphism is (mainly) implemented through the notion of virtual functions. While virtual functions are extremely useful, one should be aware that the call to a virtual function has a bit of overhead compared to a call to a non–virtual function. Moreover, virtual functions are rarely inlined by compilers. These two issues are unimportant if the virtual function is a large function (in which case the overhead is dwarfed by the execution time of the function) or infrequently called.

However, small frequently called functions that could be inlined should only be virtual if we **absolutely** cannot resolve the function call at compile time. Avoiding virtual functions without letting go of (all of) the advantages of polymorphism is a bit tricky, but it can be accomplished using templates. This leads to static polymorphism, and the aim of this note is to describe a method for implementing that construct.

To test the method, an example has been coded using both virtual functions (dynamic polymorphism) and static polymorphism. These two examples have been compared speedwise on several platforms, and – as we shall see – static polymorphism is much faster given the conditions above.

2 Polymorphism

I'll warm up to the subject a bit before explaining what static polymorphism really is, but, first of all, I should point you to the only reference in this note, namely Todd Veldhuizen's excellent paper "Techniques for Scientific C++" available from his homepage

http://osl.iu.edu/~tveldhui

which is the place where I learned about static polymorphism. In fact, all methods described below are also described in Veldhuizen's paper, but in this note I have coded a complete example using both normal (dynamic) polymorphism and static polymorphism and timed both examples on several platforms. In

other words, there is nothing novel here, I just test the method and provide a relatively comprehensive discussion.

Let us consider the notion of ordinary (or dynamic) polymorphism. Polymorphism is closely related to virtual functions, but what does it mean? Well, the word means something like "having many shapes", I guess, and in programming this translates into classes that have the same interface but work in different ways. The classical example seems to be the shape library where Circle, Oval, Square etc. are all derived from the same abstract ancestor Shape and

```
class Shape {
public:
  virtual void draw_yourself() = 0;
};
class Circle: public Shape {
public:
  virtual void draw_yourself();
};
class Square: public Shape { ... };
class Oval: public Shape { ... };
...
Shape* shape_ptr = new Circle ;
...
shape_ptr->draw_yourself();
```

results in a function being called which depends on the *dynamic type* of shape_ptr – i.e. the actual type of the class instance pointed to by shape_ptr. For instance, if shape_ptr points to an instance of class Circle (class Circle being, of course, derived from class Shape) then the actual called function is, in fact, the draw_yourself function belonging to class Circle which (we assume) overrides the default function in the Shape class (which is abstract virtue of the fact that draw_yourself() is pure virtual (that is what "=0" means ;-) – so Circle must override).

That is pretty much the story. The advantage lies in the fact that we don't have to do an explicit case analysis in order to call the correct function. Let us look at another example which will be used throughout the rest of this note. Below is a class template for a generic class representing (mathematical) functions (aka mappings):

```
template<class DomT, class ImgT>
class DMap
{
public:
    virtual ImgT operator()(const DomT& x) const = 0;
};
```

where the template arguments represent the domain and the image of the function, respectively. To use DMap we derive a class from the instantiation of its template, like this:

```
class DCosine: public DMap<float,float>
{
public:
    float operator()(const float& x) const {return cos(x);}
};
```

DCosine is a concrete class that represents ... cosine. Notice that it is derived from DMap and (as it must) overrides the function call operator. This means that if the function call operator is invoked through a reference of type DMap& or a pointer of type DMap*, respectively, the DCosine::operator() function is still called. Let us exploit this to create a new class for computing, numerically, derivatives of arbitrary functions. It looks as follows ...

```
class DDerivative: public DMap<float,float>
{
    typedef DMap<float,float> DMapT;

    static const float dt_2 = 1e-4;
    const DMapT& fun;
public:

    DDerivative(const DMapT& _fun): fun(_fun) {}

    float operator()(const float& x) const
    {
        return (fun(x+dt_2)-fun(x-dt_2))/(2*dt_2);
    }
};
```

Class DDerivative is constructed with an instance of a class derived from a DMap – e.g. DCosine. It is also itself a descendant of DMap and overrides its function call operator. If DDerivative is constructed using the DMap fun then given an argument x it will return a numerical estimate of the derivative of fun at x.

We can use DDerivative to generate a plot of cosine and its derivative

```
DCosine cosine;
DDerivative dcos_dt(cosine);

const int N = 1000000;
for(int i=0;i<N;++i)
{
    float t = M_PI*2*i/N;
    cout << t << " " << cosine(t) << " " << dcos_dt(t) << endl;
```

```
}
```

To give a slightly simpler example, we can also use a DMap in a function. Below is a function that integrates a DMap

```
float integrate(float a, float b, const DMap<float,float>& f)
{
    int N=10000;
    float d =(b-a)/N;
    float s =0;
    for(int i=0;i<N;++i)
        s += f(a+d*i);
    return s*d;
}
```

Again, a simple example of the use of virtual functions. Any descendant of DMap is a valid argument to integrate.

3 Static Polymorphism

You frown, and I understand why you are worried. I just called a virtual function ONE MILLION times. It is very fast, but mother always told us that virtual functions in tight loops are a *no no*. The problem is that virtual functions are rarely inlined and, furthermore, have more function call overhead than normal functions. Of course, this overhead is negligible for functions if (a) the function itself is large or (b) it is not called very often, but for small functions (especially one liners) it is a problem, and for this reason, small, frequently called functions should not be virtual. Now what can we do about it? We do NOT want to sacrifice the elegance of object oriented programming (or we wouldn't be here) but we also want to write insaaanely fast code.

Enter: static polymorphism (but we still need to warm up a little). The advantage of virtual functions is that it is only at run time that we decide (based on the true type of the pointed to object) what function to call. However, very often, it is, in fact, known at compile time what function we want to call. Take the code snippet

```
DCosine cosine;
integrate(cosine);
```

It is quite clear that the argument to integrate must be of type DCosine above no matter how the rest of the program looks. In other words, the virtual function calls are unnecessary. How do we go about removing them?

Todd Veldhuizen outlines some of the options in his scientific computing note referenced earlier. One option is to use a callback function, i.e. to pass a function pointer to integrate. That was good C, but it is not good C++ and it certainly ensures that the function call is not inlineable. A far better option is to use templates. Below is a new version of Cosine and integrate.

```
class Cosine {
public:
    float operator()(const float& x) const {return cos(x);}
};
template<class T> float integrate(float a, float b, const T& f)
{
    int N=1000000;
    float d =(b-a)/N;
    float s =0;
    for(int i=0;i<N;++i)
        s += f(a+d*i);
    return s*d;
}
```

The function template above, assumes that the class represented by type T has a function call operator. If it doesn't we get an error. In any case, there are no virtual functions, so the call is resolved statically and probably inlined which is what we wanted. Cosine is not (unlike DCosine) derived from any Map class. This is no longer necessary, because integrate will now accept *any* class with a function call operator taking one float argument. Is that good? If you think so - go ahead use this style, but note that it may be static but it is not really polymorphism. We do not constrain the argument of the function to be of a type derived from DMap. Potentially, this can lead to nasty errors. We need more type rigour.

Todd Veldhuizen outlines several solutions. One of these is a trick where Map is a template and Cosine (or whatever) is used as a template argument for Map. I'll skip this technique since there is a more elegant solution (also due to Mr. Veldhuizen) which involves something known as the Barton–Nackman trick. Look at this example:

```
template<class DomT, class ImgT, class ChildT>
class SMap
{
public:

 ImgT operator()(const DomT& x) const
 {
  return static_cast<const ChildT&>(*this)(x);
 }
};


class SCosine: public SMap<float,float,SCosine>
{
public:

 float operator()(const float& x) const
```

```
 {
  return cos(x);
 }
};
```

Above is a class SMap and SCosine. They obviously correspond to DMap and DCosine which we have seen earlier. However, note that SMap has an extra template argument, ChildT. Now, the weird thing is that when SCosine is declared, it inherits from SMap using ... itself as the final template argument. This construct is called the Barton and Nackman trick after the inventors, and it seems to compile everywhere so it is certainly legal. The advantage of passing the derived class as a template argument to the ancestor is that we can use the type of the derived class in the ancestor.

This leads to a beautiful new possibility! Since the ancestor (e.g. SMap) knows the derived class (e.g. SCosine), it can call a function in the derived class (e.g. the function call operator):

```
  return static_cast<const ChildT&>(*this)(x);
```

We can exploit this to create an integrate function which is passed a range a DMap and computes the integral:

```
template<class FunT>
float integrate(float a, float b, const SMap<float,float,FunT>& f)
{
    int N=10000;
    float d =(b-a)/N;
    float s =0;
    for(int i=0;i<N;++i)
        s += f(a+d*i);
    return s*d;
}
```

In the code above, f(a+d*i) must call SMap::operator() which in turn calls FunT::operator (where FunT might, for instance, be SCosine). With a little luck and optimization flags set right, both calls are inlined since they are known at compile time. Hence, the code above should run just as fast as the previous static example, but now it is also type checked since every argument to integrate must be derived from SMap. We can also use the trick to implement a new class SDerivative which is just like DDerivative

```
template<class OrigFunT>
class SDerivative: public SMap<float,float,SDerivative<OrigFunT> >
{
    typedef SMap<float,float,OrigFunT> SMapT;

    static const float dt_2 = 1e-4;
    const SMapT& fun;
```

```
public:

    SDerivative(const SMapT& _fun): fun(_fun) {}

    float operator()(const float& x) const
    {
        return (fun(x+dt_2)-fun(x-dt_2))/(2*dt_2);
    }
};
```

Now, because SMap must know its descendant, we must also give SDerivative a template argument – the class of the function whose derivative we are computing. This may not be intuitive, but look at the full example in Appendix B and it becomes pretty clear. SDerivative is used like this

```
SCosine cosine;
SDerivative<SCosine> dcos_dt(cosine);

const int N = 1000000;
for(int i=0;i<N;++i)
{
    float t = M_PI*2*i/N;
    cout << t << "  " << cosine(t) << "  " << dcos_dt(t) << endl;
}
```

and this concludes the examples. We have seen how the BN trick can be used to implement static polymorphism. In other words, we have seen how to create an inheritance hierarchy of classes with what might be called *static virtual* functions. The important point about the technique is that the complete type of an object must be known at compile time. This is enforced by the fact that the SMap template takes the descendant type as a template argument.

4 Results

Two almost identical programs – one written using dynamic polymorphism and one written using static were timed on a number of platforms: PIII/Linux, Athlon/Linux, MIPS/IRIX and Sparc/SunOS. On each platform two experiment were conducted, one using dynamic polymorphism and one using static polymorphism. The programs used for the experiments are in Appendices A and B to this note. Note that tiny changes (of no practical consequnce) were required to make the code compile on the IRIX and Sun platforms. On each platform each program was tested using

```
time ./a.out > /dev/null
```

The program was timed three times and timings are best out of three. The results (in the order of decreasing speed of the static implementation) are summarized in the table below:

Platform	Compiler	Optimization	Static (sec)	Dynamic (sec)
PIII 800	icc 5.0.1	-O3	0.369	0.466
Athlon 900	gcc 2.96	-O3	0.443	0.593
PIII 800	gcc 2.96	-O3	0.537	0.618
PIII 800	gcc 3.0.2	-O3	0.542	0.608
MIPS R5000	MIPSPro 7.30	-O3	0.91	1.69
sparc	Sun C++ 5.3	-xO5	5.7	6.1

The only flags (except –DSPEEDRUN) used during compilation were optimization flags which are also noted in the table.

Two things are noticeable. Static polymorphism is, never, slower than dynamic. That is comforting, and if the result had been otherwise, I probably hadn't written this note.

However, also comforting, but in a different way (we don't have to rewrite all our code) is the fact that the difference is not enormous. Only in the case of the MIPSPro 7.30 compiler (which I believe is rather good) is the speed–up dramatic (close to a doubling). It is a bit surprising that the Sparc machine is so slow – it is a big 20 processor computer and it was heavily loaded so I believe the program did not have a CPU to itself. Another interesting result is that the Intel compiler (icc 5.0.1) is somewhat faster than either of the two versions of gcc also used on the PIII 800 platform. In fact, the dynamic version compiled using icc is faster than the static version using gcc. However, this single example is, of course, quite insufficient as a test of whether icc is, in general, faster than gcc.

Clearly, if the innermost function in the loop did more work than simply compute the cosine of the argument (see appendices A and B), the difference between the schemes would be smaller. However, cosine involves a function call and many reasonable functions do much less work. To see what would be the most one could gain by using static polymorphism, the experiment was changed slightly. The loop was changed from one million to ten million iterations and instead of computing the cosine of the argument, SCosine::operator() and DCosine::operator() were changed to simply return 1.0. This new program was tested on the Athlon/gcc-2.96 platform. The result was that the static program ran more than twenty times as fast as the dynamic (0.048 seconds vs 1.143 seconds).

Finally, both programs were compiled to assembler. Analysis of the output reveals that all function calls (except calls to the math library cos() function and calls pertaining to output) in the static program are inlined. On the other hand, in the dynamic program, the virtual function calls are not inlined.

5 Conclusions

Static polymorphism using the Barton–Nackman trick is not too hard to implement, and for our little example it seems to speed up programs consistently

on all tested platforms. The advantage of using the BN trick over just using templates is that the BN trick preserves type safety: For instance, the static version of integrate will only integrate a mapping class derived from SMap.

The speed–up is only important in very tight loops, but in some cases the improvement can be vast. This is probably almost exclusively when the use of dynamic linking would cause a very small function to be non–inlineable.

Note, also, that you cannot use static polymorphism everywhere as a replacement for virtual functions. Let us take an example: Say we have a vector of DMap* pointers. We iterate through the list and call a virtual function using each pointer. We cannot replace the list with a list of SMap* pointers (except if we fill in the last template argument which indicates the derived type) because the pointers types must all be the same. More precisely, we can have an SMap<float,float,Cosine>* and an SMap<float,float,Sine>* but we cannot have a pointer type that can point to either. In fact, that is the whole point - I apologize for threading water.

Finally, I would like to mention that the BN trick has other applications than the one discussed here. When the ancestor knows the type of the derived class, we can let a function in the ancestor class return a value of derived type. I used this technique in a library of vector and matrix class intended for use in computer graphics called CGLA

http://www.imm.dtu.dk/~jab/software.html#cgla

A Dynamic Example

```
#include <iostream>
#include <cmath>

using namespace std;

template<class DomT, class ImgT>
class DMap
{
public:
    virtual ImgT operator()(const DomT& x) const = 0;
};


class DCosine: public DMap<float,float>
{
public:
    float operator()(const float& x) const {return cos(x);}
};

class DDerivative: public DMap<float,float>
{
    typedef DMap<float,float> DMapT;

    static const float dt_2 = 1e-4;
    const DMapT& fun;
public:

    DDerivative(const DMapT& _fun): fun(_fun) {}

    float operator()(const float& x) const
    {
        return (fun(x+dt_2)-fun(x-dt_2))/(2*dt_2);
    }
};


main()
{
    DCosine cosine;
    DDerivative dcos_dt(cosine);

    const int N = 1000000;
    float dummy;
    for(int i=0;i<N;++i)
```

```
        {
            float t = M_PI*2*i/N;
#ifdef SPEEDRUN
            dummy += cosine(t) - dcos_dt(t);
#else
            cout << t << "  " << cosine(t) << "  " << dcos_dt(t) << endl;
#endif
        }
#ifdef SPEEDRUN
    cout << "Dummy:   " << dummy;
#endif
}
```

B Static Example

```
#include <iostream>
#include <cmath>

using namespace std;

template<class DomT, class ImgT, class ChildT>
class SMap
{
public:

    ImgT operator()(const DomT& x) const
    {
        return static_cast<const ChildT&>(*this)(x);
    }
};


class SCosine: public SMap<float,float,SCosine>
{
public:

    float operator()(const float& x) const
    {
        return cos(x);
    }
};

template<class OrigFunT>
class SDerivative: public SMap<float,float,SDerivative<OrigFunT> >
{
    typedef SMap<float,float,OrigFunT> SMapT;

    static const float dt_2 = 1e-4;
    const SMapT& fun;
public:

    SDerivative(const SMapT& _fun): fun(_fun) {}

    float operator()(const float& x) const
    {
        return (fun(x+dt_2)-fun(x-dt_2))/(2*dt_2);
    }
};
```

```
main()
{
    SCosine cosine;
    SDerivative<SCosine> dcos_dt(cosine);

    const int N = 1000000;
    float dummy;
    for(int i=0;i<N;++i)
        {
            float t = M_PI*2*i/N;
#ifdef SPEEDRUN
            dummy += cosine(t) - dcos_dt(t);
#else
            cout << t << "  " << cosine(t) << "  " << dcos_dt(t) << endl;
#endif
        }
#ifdef SPEEDRUN
    cout << "Dummy:   " << dummy;
#endif

}
```