Techniques for Scientific C++

Todd Veldhuizen
<tveldhui@acm.org>

Indiana University Computer Science Technical Report # 542
Version 0.4, August 2000

Abstract

This report summarizes useful techniques for implementing scientific programs in C++, with an emphasis on using templates to improve performance.

Contents

Chapter 1

Introduction

1.1 C++ Compilers

1.1.1 Placating sub-standard compilers

Every compiler supports a different subset of the ISO/ANSI C++ standard. Trying to support a C++ library or program on many platforms is guaranteed to be a major headache.

The two biggest areas of noncompliance are (1) templates; and (2) the standard library.

If you need to write code which will work over many platforms, you are in for a frustrating time. In the Blitz++ library, I have an extensive set of preprocessor kludges which handle the vagaries of various compilers. For example, consider this snippet of code:

```
#include <cmath>

namespace blitz {
    double pow(double x, double y)
    { return std::pow(x,y); }
};
```

This simply creates a function `pow` in the blitz namespace which calls the standard `pow` function. Some compilers don't have the <`cmath`> header, so it has to be replaced with <`math.h`>. Some compilers have the <`cmath`> header, but C math functions are not in the `std` namespace. Some compilers don't support namespaces at all. The solution I use is to write the code like this:

```
#ifdef BZ_MATH_FN_IN_NAMESPACE_STD
  #include <cmath>
#else
  #include <math.h>
#endif

#ifdef BZ_NAMESPACES
double pow(double x, double y)
{
  return BZ_MATHFN_SCOPE(pow)(x,y);
}
```

```
#endif
```

This is very nasty. Unfortunately, this situation is going to persist for a number of years, until all the compilers catch up with the standard.

Blitz++ comes with a script (bzconfig) which runs many small test programs and figures out which kludges are needed for a compiler. Luc Maisonobe (Luc.Maisonobe@cnes.fr) has folded these programs into the popular Autoconf utility. You are also welcome to hack out the bzconfig script from Blitz++ and use it.

Another way to avoid multi-platform headaches is to just support the gcc compiler. It is better than most compilers in standard-compliance, and has a rapidly improving implementation of the standard library. Unfortunately its ability to optimize is often not as good as KAI C++ or the vendor-specific compilers. But for large out-of-cache problems, it can do quite well. (Out-of-cache problems are much less sensitive to fine optimizations, since most of the CPU time is spent stalling for memory anyway).

1.1.2 The compiler landscape

Many C++ compilers use a front end made by Edison Design Group (EDG). This front end is in close conformance to the standard, is very robust and reliable, and issues good error messages. You can find EDG-based compilers for just about any platform. Some of them are: SGI C++, KAI C++, DEC cxx, Cray C++, Intel C++, Portland Group C++, Tera C++.

The major C++ compilers (Borland/Inprise, Microsoft, etc.) are surprisingly far behind the C++ standard. They all seem to be focused on things like COM, DCOM, WFC, etc. and not at all worried about standards compliance. Microsoft's 6.0 C++ compiler routinely crashed when asked to compile simple member-template code. In a conversation with the Microsoft developers, they stated that their customers are not interested in standards compliance, but rather things like WFC/COM etc. Perhaps this is true. But if you are frustrated by the lack of standards compliance in your favourite C++ compiler, complain – they won't make it a priority unless their customers want it enough to speak up.

If you are looking for a PC compiler with near-standard compliance, Intel C++ (available as a plug-in for Visual C++) and Metrowerk's CodeWarrior are good alternatives.

1.1.3 C++-specific optimization

Most compilers are disappointingly poor at optimizing C++. Traditional optimization techniques need to be modified to work with nested aggregates and temporary objects, which are very common in C++. An exception is KAI C++, which is a top-notch optimizing C++ compiler. Another exception is the SGI C++ compiler, which does a reasonably good job (although currently not as good as KAI C++ for things like temporary object elimination).

KAI has a US patent on some of its optimization techniques, which may discourage other compiler vendors from implementing similar optimizations. However, it has licensed its optimization engine to at least one other compiler vendor (EPC++).

1.2 Compile times

One of the most frequent complaints about Blitz++ (and other template-intensive C++ libraries) is compile times. The next few sections explain why C++ programs take so long to compile.

1.2.1 Headers

Headers are a large part of the problem. C++ does not have a very effective module system. As a result, vast amounts of header have to be parsed and analyzed for every compilation unit. For example, including `<iostream>` can pull in tens of thousands of lines of code for some compilers.

Template libraries are especially bad, because of the requirement that the full template implementation be available at compile time. So instead of function declarations, headers provide function definitions.

For example, compiling the `examples/array.cpp` example from the Blitz++ library with gcc on my desktop workstation takes 37 seconds. If you strip all the code out of this file, and just leave the `#include <blitz/array.h>` line at the top, then compiling takes 22 seconds. In other words, 60% of the compile time is taken just processing the headers!

Precompiled headers can sometimes help; but sometimes they can make the problem worse, too. Turning on precompiled headers with Blitz++ can result in ˜64 Mb dumps which take longer to read in than the headers they replace.

The lesson to be learned is to keep your headers as small as possible. When developing a library, provide many small headers which must be included separately, rather than one header which includes the whole library.

1.2.2 Prelinking

Templates cause a serious problem for compilers. When should they be instantiated? There are two common approaches:

- For each compilation unit, generate all template instances which are used. This leads to duplicate template instances – if files `foo1.cpp` and `foo2.cpp` both use `list<int>`, then you get two instantiations of `list<int>` – one in `foo1.o` and another in `foo2.o`. Some linkers will automatically throw out the duplicate instances.

- Don't do any instantiation. Instead, wait until link time to determine which template instances are required, and go back and generate them. This is called prelinking.

Unfortunately, prelinking can require recompiling your entire program several times.

It's typical to see modules recompiled three times. As you can imagine, for very large programs this can cause long compile times.

Compilers which use the EDG front end can be forced to not do prelinking by using the option `-tused` or `-ptused`.

1.2.3 The program database approach – Visual Age C++

IBM's Visual Age C++ compiler takes an interesting approach to the problem of separate compilation. It keeps all code in a large database. Since the complete source code for a program is available at compile time (instead of just one module at a time), VAC++ does not have to do prelinking or worry about separate compilation issues. The compiler is also incremental, meaning that it only recompiles functions which have changed, or rely on definitions which have changed. This approach has the potential to drastically reduce compile times for template-intensive C++ libraries.

1.2.4 Quadratic/Cubic template algorithms

Libraries such as Blitz++ make extensive use of templates. Some compilers have algorithms in their templates implementation which don't scale well. For example, consider this code snippet:

```
template<int N>
void foo()
{
    foo<N-1>();
}

template<>
void foo<0>()
{ }
```

If we instantiate `foo<N>`, then `foo<N-1>` is instantiated, as is `foo<N-2>`, ..., all the way down to `foo<0>`. How long should this take to compile? Well, the number of functions instantiated is N, so the compile time ought to be linear in N, right?

Well, unfortunately not, for some compilers. For simplicity, many compilers just keep a linked list of template instances. Determining whether a template has been instantiated requires a linear search through this list. So the compile time is $O(N^2)$. This is not a problem when there are a handful of instances for each template, but libraries such as Blitz++ routinely generate thousands of instances of template classes.

Similar problems can occur for optimization algorithms, such as aliasing analysis. I've personally found $O(N^2)$ and $O(N^3)$ algorithms in a major C++ compiler.

1.3 Static Polymorphism

1.3.1 Are virtual functions evil?

In the C++ world, polymorphism usually means virtual functions. This sort of polymorphism is a handy design tool. But virtual functions can have big performance penalties:

- Extra memory accesses
- Virtual function dispatch can cause an unpredictable branch– piplines can grind to a halt (note however, that some architectures have branch caches which can avoid this problem)
- Current C++ compilers (as far as I know) are unable to optimize around the virtual function call: it prevents instruction scheduling, data flow analysis, loop unrolling, etc.

There are several research projects which have demonstrated success at replacing virtual function calls with direct dispatches. But (AFAIK) this research has not made it into any C++ compilers yet. One of the snags is that removing virtual function dispatches requires "whole program analysis" – you must have the entire program available for analysis at compile time. Almost all C++ implementations compile each module (compilation unit) separately – the only exception I know of is Visual Age C++.

Note however, that virtual functions are not always a "kiss of death" when it comes to performance. The important thing is: how much code is inside the virtual function? If there is a lot of code, then the overhead of the virtual function will be insignificant. But if there is a small amount of code (for example, a virtual function which just returns a data member) then the overhead can be crucial.

When are virtual functions bad?

- When the amount of code inside the virtual function is small (e.g. fewer than 25 flops); and

- When the virtual function is used frequently (e.g. inside a loop). You can determine this with the help of a profiler.

Virtual functions are okay (and highly recommended!) if the function being called is big (does a lot of computations), or if it isn't called very often.

Unfortunately, in scientific codes some of the most useful places for virtual function calls are inside loops, and involve small routines. A classic example is:

```
class Matrix {
  public: virtual double operator()(int i, int j) = 0;
};

class SymmetricMatrix : public Matrix {
  public: virtual double operator()(int i, int j);
};

class UpperTriangularMatrix : public Matrix {
  public: virtual double operator()(int i, int j);
};
```

The virtual function dispatch to `operator()` to read elements from the matrix will ruin the performance of any matrix algorithm.

1.3.2 Solution A: simple engines

The "engines" term was coined by the POOMA team at LANL. The design pattern folks probably have a name for this too (if you know it, please tell me).

The idea is to replace dynamic polymorphism (virtual functions) with static polymorphism (template parameters). Here's a skeleton of the `Matrix` example using this technique:

```
class Symmetric {
 // Encapsulates storage info for symmetric matrices
};

class UpperTriangular {
 // Storage format info for upper tri matrices
};

template<class T_engine>
class Matrix {
private:
  T_engine engine;
};

// Example routine which takes any matrix structure
template<class T_engine>
double sum(Matrix<T_engine>& A);

// Example use ...
```

```
Matrix<Symmetric> A;
sum(A);
```

In this approach, the variable aspects of matrix structures are hidden inside the **engines** `Symmetric` and `UpperTriangular`. The `Matrix` class takes an engine as a template parameter, and contains an instance of that engine.

The notation is different than what users are used to: `Matrix<Symmetric>` rather than `SymmetricMatrix`. This is probably not a big deal. One could provide typedefs to hide this quirk. Also, in `Matrix` most interesting operations have to be delegated to the engines – again, not a big deal.

The biggest headache is that all `Matrix` subtypes must have the same member functions. For example, if we want `Matrix<Symmetric>` to have a method `isSymmetricPositiveDefinite()`, then a typical implementation would be

```
template<class T_engine>
class Matrix {
public:
    // Delegate to the engine
    bool isSymmetricPositiveDefinite()
    { return engine.isSymmetricPositiveDefinite(); }

private:
    T_engine engine;
};

class Symmetric {
public:
    bool isSymmetricPositiveDefinite()
    {
        ...
    }
};
```

But it doesn't make sense for `Matrix<UpperTriangular>` to have an `isSymmetricPositiveDefinite()` method, since upper triangular matrices cannot be symmetric! What you find with this approach is that the base type (in this example, `Matrix`) ends up having the union of all methods provided by the subtypes. So you end up having lots of methods in each engine which just throw exceptions:

```
class UpperTriangular {
public:
    bool isSymmetricPositiveDefinite()
    {
        throw makeError("Method isSymmetricPositiveDefinite() is "
              "not defined for UpperTriangular matrices");
        return false;
    }
};
```

Alternate (but not recommended) approach: just throw your users to the wolves by omitting these methods. Then users must cope with mysterious template instantiation errors which result when they try to use missing methods.

Fortunately, there is a better solution.

1.3.3 Solution B: the Barton and Nackman Trick

This trick is often called the "Barton and Nackman Trick", because they used it in their excellent book **Scientific and Engineering C++**. Geoff Furnish coined the term "Curiously Recursive Template Pattern", which is a good description.

If you haven't seen this trick before, it should have you scratching your head. Yes, it is legal.

```
// Base class takes a template parameter.  This parameter
// is the type of the class which derives from it.
template<class T_leaftype>
class Matrix {
public:
  T_leaftype& asLeaf()
  { return static_cast<T_leaftype&>(*this); }

  double operator()(int i, int j)
  { return asLeaf()(i,j); }     // delegate to leaf
};

class SymmetricMatrix : public Matrix<SymmetricMatrix> {
  ...
};

class UpperTriMatrix : public Matrix<UpperTriMatrix> {
  ...
};

// Example routine which takes any matrix structure
template<class T_leaftype>
double sum(Matrix<T_leaftype>& A);

// Example usage
  SymmetricMatrix A;
  sum(A);
```

This allows a more conventional inheritance-hierarchy approach. The trick is that the base class takes a template parameter, which is the type of the leaf class. This ensures that the complete type of an object is known at compile time – no need for virtual function dispatch.

Methods can be selectively specialized in the leaf classes (i.e. default implementations can reside in the base, overridden by the leaf classes when necessary).

Unlike the approach in the previous section, leaf classes can have methods which are unique to the leaf class.

In this approach, the base class still has to delegate everything to the leaf classes.

Making your inheritance tree more than one level deep requires some thought, but is possible.

1.4 Callback inlining techniques

Callbacks appear frequently in some problem domains. Some typical examples are:

- Numerical integration: a general-purpose integration routine uses callbacks to evaluate the function to be integrated (and often its derivatives). In C libraries, this is usually done with a pointer-to-function argument.
- Optimization of functions.
- Applying a function to every element in a container
- Sorting algorithms such as `qsort()` and `bsearch()` require callbacks which compare elements.

The typical C approach to callbacks is to provide a pointer-to-function:

```
double integrate(double a, double b, int numSamplePoints,
    double (*f)(double) )
{
    double delta = (b-a) / (numSamplePoints - 1);
    double sum = 0.0;

    for (int i=0; i < numSamplePoints; ++i)
        sum += f(a + i*delta);

    return sum * (b-a) / numSamplePoints;
}
```

This approach works fine, except for performance: the function calls in the inner loop cause poor performance. Some optimizing compilers may perform inter-procedural and points-to analysis to determine the exact function which `f` points to, but most compilers do not.

1.4.1 Callbacks: the typical C++ approach

In C++ libraries, callbacks were mostly replaced by virtual functions. The newer style is to have a base class, such as `Integrator`, with the function to be integrated supplied by subclassing and defining a virtual function:

```
class Integrator {
public:

    double integrate(double a, double b, int numSamplePoints)
    {
      double delta = (b-a) / (numSamplePoints-1);
      double sum = 0.;
      for (int i=0; i < numSamplePoints; ++i)
        sum += functionToIntegrate(a + i*delta);
      return sum * (b-a) / numSamplePoints;
    }

    virtual double functionToIntegrate(double x) = 0;
```

```
};

class IntegrateMyFunction : Integrator {
public:
    virtual double functionToIntegrate(double x)
    {
        return cos(x);
    }
};
```

As with C, the virtual function dispatch in the inner loop will cause poor performance. (An exception is if evaluating `functionToIntegrate()` is very expensive – in this case, the overhead of the virtual function dispatch is negligible. But for cheap functions, it is a big concern). For good performance, it is necessary to inline the callback function into the integration routine.

Here are three approaches to doing inlined callbacks.

Expression templates

This is a very elaborate, complex solution which is difficult to extend and maintain. It is probably not worth your effort to take this approach.

STL-style function objects

The callback function can be wrapped in a class, creating a function object or functor:

```
class MyFunction {
public:
    double operator()(double x)
    { return 1.0 / (1.0 + x); }
};

template<class T_function>
double integrate(double a, double b, int numSamplePoints,
    T_function f)
{
    double delta = (b-a) / (numSamplePoints-1);
    double sum = 0.;

    for (int i=0; i < numSamplePoints; ++i)
        sum += f(a + i*delta);

    return sum * (b-a) / numSamplePoints;
}

// Example use
integrate(0.3, 0.5, 30, MyFunction());
```

An instance of the `MyFunction` object is passed to the `integrate` function as an argument. The C++ compiler will instantiate `integrate`, substituting `MyFunction` for `T_function`; this will result in `MyFunction::operator()` being inlined into the inner loop of the `integrate()` routine.

A down side of this approach is that users have to wrap all the functions they want to use in classes. Also, if there are parameters to the callback function, these have to be provided as constructor arguments to the `MyFunction` object and stored as data members.

Note that you can also pass function pointers to the `integrate()` function in the above example. This is equivalent to C-style callback functions. No inlining will occur if you pass function pointers.

Pointer-to-function as a template parameter

There is a third approach which is quite neat:

```
double function1(double x)
{
    return 1.0 / (1.0 + x);
}

template<double T_function(double)>
double integrate(double a, double b, int numSamplePoints)
{
    double delta = (b-a) / (numSamplePoints - 1);
    double sum = 0.0;

    for (int i=0; i < numSamplePoints; ++i)
        sum += T_function(a + i*delta);

    return sum * (b-a) / numSamplePoints;
}

// ...
integrate<function1>(1.0, 2.0, 100);
```

The `integrate()` function takes a template parameter which is a pointer to the callback function. Since this pointer is known at compile-time, `function1` is inlined into the inner loop! See also the thread "Subclassing algorithms" in the oon-list archive.

1.5 Managing code bloat

Code bloat can be a problem for template-based libraries. For example, if you use these instances of `queue`:

```
    queue<float>
    queue<int>
    queue<Event>
```

Your program will have 3 slightly different versions of the `queue<>` code. This wasn't a problem with older-style C++ libraries which used polymorphic collections based on virtual functions.

However, the problem has been worsened because some compilers create absurd amounts of unnecessary code:

- Some compilers instantiate templates in every module, then fail to throw out the duplicate instances at link time.

- Some (older and non-ISO compliant) compilers instantiate every member function of a class template. The ISO/ANSI C++ standard forbids this: compilers are supposed to instantiate only members which are actually used.

Part of the problem is our fault: templates make it so easy to generate code that programmers do it without thinking about the consequences. I tend to use templates everywhere, and usually don't even consider virtual functions. I personally don't care about code bloat because the applications I work with are small, but people who work on big applications have to be conscious of the templates vs. virtual function decision:

- Templates may give you faster code. They almost always give you **more** code. (They don't give you more code if you only use one instance of the template).
- Virtual functions give you less code, but slower code. The slow-down is negligible if the body of the function is big, or the function isn't called often.

Of course, virtual functions and templates are not always interchangeable. If you need a polymorphic collection, then templates are not an option.

Here are some suggestions for reducing code bloat.

1.5.1 Avoid kitchen-sink template parameters

It's tempting to make every property of a class a template parameter. Here's an exaggerated example:

```
template<typename number_type, int N, int M,
  bool sparse, typename allocator, bool symmetric,
  bool banded, int bandwidthIfBanded>
class Matrix;
```

Every extra template parameter is another opportunity for code bloat. If you use many different instantiations of a "kitchen-sink" template class, you should consider eliminating some of the template parameters in favour of virtual functions, member templates, etc.

For example, suppose we had a class `list<>`. We want users to be able to provide their own allocator, so we could make the allocator a template parameter:

```
list<double,allocator<double> >  A;
list<double,my_allocator<double> > B;
```

Separate versions of the list code will be generated for both A and B, even though most of it will be identical (for example, when you are traversing a list you usually don't care about how the data was allocated). Is anything gained by making the allocator a template parameter? Probably not– memory allocation is typically slow, so the extra overhead of a virtual function call would be miniscule. The same flexibility could be accomplished with:

```
list<double>  A(allocator<double>());
list<double>  B(my_allocator<double>());
```

where `allocator` and `my_allocator` are polymorphic.

1.5.2 Put function bodies outside template classes

You can avoid creating unnecessary duplicates of member functions by moving the bodies outside the class. For example, here is a skeletal `Array3D` class:

```
template<class T_numtype>
class Array3D {
public:
    // Make sure this array is the same size
    // as some other array
    template<class T_numtype2>
    bool shapeCheck(Array3D<T_numtype2>&);

private:
    int base[3];      // starting index values
    int extent[3];    // length in each dimension
};

template<class T_numtype> template<class T_numtype2>
bool Array3D<T_numtype>::shapeCheck(Array3D<T_numtype2>& B)
{
    for (int i=0; i < 3; ++i)
      if ((base[i] != B.base(i)) ||
          (extent[i] != B.extent(i)))
        return false;

    return true;
}
```

The routine `shapeCheck()` verifies that two arrays have the same starting indices and lengths. If you have many different kinds of arrays (`Array3D<float>`, `Array3D<int>`, ...), you will wind up with many instances of the `shapeCheck()` method. But clearly all of these instances are identical, because `shapeCheck()` doesn't use the template parameters `T_numtype` and `T_numtype2`.

To avoid having duplicate instances, you can move `shapeCheck()` into a base class which doesn't have a template parameter, or into a global function. For example:

```
class Array3D_base {

public:
    bool shapeCheck(Array3D_base&);

private:
    int base[3];      // starting index values
    int extent[3];    // length in each dimension
};

template<class T_numtype>
class Array3D : public Array3D_base {
   ...
};
```

Since `shapeCheck()` is now in a non-templated class, there will only be one instance of it.

If you happen to be a C++ compiler writer and are reading this, note: this problem could be solved automatically by doing a liveness analysis on template parameters.

1.5.3 Inlining levels

By default, I tend to make lots of things `inline` in Blitz++, because this aids compiler optimizations (for the compilers which don't do interprocedural analysis or aggressive inlining). If someday I start getting complaints about excessive inlining in Blitz++, here is a possible strategy:

```
#if BZ_INLINING_LEVEL==0
  // No inlining
  #define _bz_inline1
  #define _bz_inline2
#elif BZ_INLINING_LEVEL==1
  #define _bz_inline1  inline
  #define _bz_inline2
#elif BZ_INLINING_LEVEL==2
  #define _bz_inline1  inline
  #define _bz_inline2  inline
#endif
```

Then users can compile with `-DBZ_INLINING_LEVEL=0` if they don't want any inlining, and `-DBZ_INLINING_LEVEL=2` if they want maximum inlining. Inlinable routines in Blitz++ would be marked with `_bz_inline1` or `_bz_inline2`:

```
// Only inline this when BZ_INLINING_LEVEL is 1 or higher
_bz_inline1 void foo(Array<int>& A)
{
   // ...
}
```

1.6 Containers

Container classes can be roughly divided into two groups: STL-style (iterator-based) and Data/View.

1.6.1 STL-style containers

- There is one container object which owns all the data.
- Subcontainers are specified by iterator ranges, usually a closed-open interval: `[iter1,iter2)`
- This is the approach taken by STL; if you write your containers this way, you can take advantage of STL's many algorithms.
- Requires there be a sensible 1-D ordering of your container. All subcontainers must be intervals in that 1-D ordering. This creates obvious problems for multidimensional arrays (for example, `A(2:6:2,2:6:2)` is clearly not a range of some sensible 1-D ordering of the array elements)

- Iterators are based conceptually on **pointers**, which are the most error-prone aspect of C++. There are many ways to go wrong with iterators.
- It can be difficult for methods and functions to return containers as values. Instead, a container to receive the result can be passed as an argument.

1.6.2 Data/View containers

- The container contents ("Data") exist outside the container objects. The containers are lightweight handles ("Views") of the Data.
- Multiple containers can refer to the same data, but provide different views
- Reference counting is used: the container contents disappear when the last handle disappears. This provides a primitive (but effective) form of garbage collection.
- It is sometimes easier to specify subcontainers (for example, subarrays of multidimensional arrays)
- Subcontainers can be first-class containers (e.g. when you take a subarray of an `Array<N>`, the result is an `Array<N>`, not a `Subarray<N>`)
- There are some subtleties associated with designing reference-counted containers; a cautious design is needed.
- It is easy to return containers from methods, and pass containers by value.

1.7 Aliasing and restrict

Aliasing is a major performance headache for scientific C++ libraries. As an example, consider calculating the rank-1 matrix update $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{x}\mathbf{x}^T$:

```
void rank1Update(Matrix& A, const Vector& x)
{
    for (int i=0; i < A.rows(); ++i)
      for (int j=0; j < A.cols(); ++j)
        A(i,j) += x(i) * x(j);
}
```

For good performance, the compiler needs to keep `x(i)` in a register instead of reloading it constantly in the inner loop. However, the compiler can't be sure that the data in the matrix `A` doesn't overlap the data in the vector `x` – there is a possibility that writing to `A(i,j)` could change the value of `x(i)` partway through. This might be impossible given the design of the `Matrix` and `Vector` classes, but the compiler can't know that. The remote possibility of aliasing forces the compiler to generate inefficient code.

Another effect of aliasing is inhibiting software pipelining of loops. For example, a DAXPY operation $\mathbf{y} \leftarrow \mathbf{y} + a{*}\mathbf{x}$:

```
void daxpy(int n, double a, double* x, double* y)
{
```

```
        for (int i=0; i < n; ++i)
          y[i] = y[i] + a * x[i];
    }
```

To get good performance, an optimizing compiler will partially unroll the loop, and then do software pipelining to find an efficient schedule for the inner loop. Designing this schedule often requires changing the order of loads and stores in the loop, for example:

```
    // fake pipelined DAXPY loop
    double* xt = x;
    double* yt = y;
    for (; i < n; i += 4)
    {
        double t0 = yt[0] + a * xt[0];
        double t1 = yt[1] + a * xt[1];
        yt[0] = t0;
        double t2 = yt[2] + a * xt[2];
        yt[1] = t1;
        double t3 = yt[3] + a * xt[3];
        i += 4;
        xt += 4;
        yt[2] = t2;
        yt += 4;
        yt[3] = t3;
    }
```

Unfortunately, the compiler can't change the order of loads and stores because there is a possibility that the `x` and `y` arrays might overlap in memory.

In Fortran 77/90, the compiler is able to assume that there is never any aliasing between arrays. This can make a difference of 20-50% on performance for in-cache arrays; it depends on the particular loop/architecture. Aliasing doesn't seem to be as large a problem for out-of-cache arrays.

Good C and C++ compilers do alias analysis, the aim of which is to determine whether pointers can safely be assumed to point to non-overlapping data. Alias analysis can eliminate many aliasing possibilities, but it is usually not perfect. For example, ruling out aliasing for a routine like `daxpy` (above) requires a whole-program analysis – the optimizer has to consider every function in your executable at the same time. Unless your your optimizer does a whole-program alias analysis (the SGI compiler will, with -IPA) then you will still have problems with aliasing.

NCEG (Numerical C Extensions Group) designed a keyword `restrict` which is recognized by some C++ compilers (KAI, Cray, SGi, Intel, gcc). `restrict` is a modifier which tells the compiler there are no aliases for a value. Examples:

```
double* restrict a;    // No aliases for a[0], a[1], ...
int& restrict b;       // No aliases for b
```

The exact meaning of `restrict` in C++ is not quite nailed down. But the ghist is that when the `restrict` keyword is used, the compiler is free to optimize merrily away without worrying about load/store orderings.

Unfortunately, a proposal to officially include the `restrict` keyword in the ISO/ANSI C++ standard was turned down. What we ended up with instead was a note about `valarray` which is of questionable usefulness:

```
2 The valarray array classes are defined to be free of
  certain forms  of aliasing, thus allowing operations
  on these classes to be optimized.
```

However, `restrict` has been adopted for the new ANSI/ISO C standard, which improves its chances for inclusion in the next C++ standard.

To develop code which uses restrict when available, you can use preprocessor kludges. For example, Blitz++ defines these macros:

```
#ifdef BZ_HAS_RESTRICT
 #define _bz_restrict restrict
#else
 #define _bz_restrict
#endif

// Example use of _bz_restrict

class Vector {
  double* _bz_restrict data_;
};
```

On some platforms, there are compilation flags which allow the compiler to make strong assumptions about aliasing. These can have a positive effect, but caution is required– these flags change the semantics of pointers throughout your application. If you do anything tricky with pointers, you may create problems.

1.8 Traits

The traits technique was pioneered by Nathan Myers: "Traits: a new and useful template technique," C++ Report, June 1995. You can read this online at http://www.cantrip.org/traits.html.

Trait classes are maps, in the sense that they map one kind of thing onto another. You can map from

- Types
- Compile-time known values (e.g. constants)
- Tuples of the above
- Tuples of the above

onto pretty much anything you want (types, constants, run-time variables, functions).

1.8.1 An example: `average()`

Consider this bit of code, which calculates the average of an array:

```
template<class T>
T average(const T* data, int numElements)
{
    T sum = 0;
    for (int i=0; i < numElements; ++i)
        sum += data[i];

    return sum / numElements;
}
```

This will work properly if `T` is a floating-point type. But what if:

- `T` is an integral type (`int`, `long`): then the average will be truncated and returned as an integer.
- `T` is a char: the sum will overflow after a few elements.

For integral types, we would like a floating-point representation for the sum and result. But note that we cannot just always use `double` – what if someone called `average()` for an array of `complex<float>`?

What we need is a mapping (or compile-time function) which will produce `double` if `T` is an integral type, and just `T` otherwise.

Here is how this can be done with traits:

```
template<class T>
struct float_trait {
    typedef T      T_float;
};

template<>
struct float_trait<int> {
    typedef double T_float;
};

template<>
struct float_trait<char> {
    typedef double T_float;
};

// etc.
```

The things you want to map **from** are template parameters of the class (or `struct`). The things you want to map **to** sit inside the class or `struct`.

In the above example, `float_trait<T>::T_float` gives an appropriate floating-point type for `T`.

Here is a revised version of `average()` which uses this traits class:

```
template<class T>
typename float_trait<T>::T_float average(const T* data,
    int numElements)
{
    typename float_trait<T>::T_float sum = 0;

    for (int i=0; i < numElements; ++i)
        sum += data[i];

    return sum / numElements;
}
```

This works for `char`, `int`, `float`, `double`, `complex<double>`, etc. By using traits, we've avoided having to specialize this function template for the special cases.

1.8.2 Type promotion example

Consider this operator, which adds two vector objects:

```
template<class T1, class T2>
Vector<???> operator+(Vector<T1>&, Vector<T2>&);
```

What should the return type be? Ideally, it would follow C-style type promotion rules, so that

```
Vector<int> + Vector<char>    ->  Vector<int>
Vector<int> + Vector<float>   ->  Vector<float>
// etc.
```

plus maybe some extra rules:

```
Vector<float> + Vector<complex<float>>
     -> Vector<complex<float>>
```

In this example, we want to map from pairs of types to another type. This is a perfect application for traits.

Here is a straightforward implementation:

```
template<class T1, class T2>
struct promote_trait {
};

#define DECLARE_PROMOTE(A,B,C)                  \
    template<> struct promote_trait<A,B> {      \
        typedef C T_promote;                    \
    };
```

```
DECLARE_PROMOTE(int,char,int);
DECLARE_PROMOTE(int,float,float);
DECLARE_PROMOTE(float,complex<float>,complex<float>);
// etc. for all possible combinations
```

Now we can write `operator+` like this:

```
template<class T1, class T2>
Vector<typename promote_trait<T1,T2>::T_promote>
operator+(Vector<T1>&, Vector<T2>&);
```

The down side of this approach is that we have to write a program to generate all the specializations of `promote_trait` – and there are tonnes.

If you are willing to get a bit dirty, you can do better. This next version is based on an idea due to Jean-Louis Leroy. First, a conceptual overview of how this will work:

- `bool`, `char`, `unsigned char`, `short int`, etc. will autopromote to `int`, as in C and C++.
- Assign each type a "precision rank". For example, `bool`=1, `int`=2, `float`=3, `double`=4, `complex<float>`=5, etc. We can use a traits class to map from a type such as `float` onto its "precision rank".
- We will promote to whichever type has a higher "precision rank".
- If there is no "precision rank" for a type, we'll promote to whichever type requires more storage space (and hopefully more precision).

Here is the implementation, which is excerpted with minor changes from the Blitz++ library:

```
template<class T>
struct precision_trait {
    enum { precisionRank = 0,
           knowPrecisionRank = 0 };
};

#define DECLARE_PRECISION(T,rank)              \
    template<>                                 \
    struct precision_trait< T > {              \
        enum { precisionRank = rank,           \
           knowPrecisionRank = 1 };            \
    };

DECLARE_PRECISION(int,100)
DECLARE_PRECISION(unsigned int,200)
DECLARE_PRECISION(long,300)
DECLARE_PRECISION(unsigned long,400)
DECLARE_PRECISION(float,500)
DECLARE_PRECISION(double,600)
```

```
DECLARE_PRECISION(long double,700)
DECLARE_PRECISION(complex<float>,800)
DECLARE_PRECISION(complex<double>,900)
DECLARE_PRECISION(complex<long double>,1000)

template<class T>
struct autopromote_trait {
    typedef T T_numtype;
};

#define DECLARE_AUTOPROMOTE(T1,T2)        \
    template<>                            \
    struct autopromote_trait<T1> {        \
      typedef T2 T_numtype;               \
    };

// These are the odd cases where small integer types
// are automatically promoted to int or unsigned int for
// arithmetic.
DECLARE_AUTOPROMOTE(bool, int)
DECLARE_AUTOPROMOTE(char, int)
DECLARE_AUTOPROMOTE(unsigned char, int)
DECLARE_AUTOPROMOTE(short int, int)
DECLARE_AUTOPROMOTE(short unsigned int, unsigned int)

template<class T1, class T2, int promoteToT1>
struct promote2 {
    typedef T1 T_promote;
};

template<class T1, class T2>
struct promote2<T1,T2,0> {
    typedef T2 T_promote;
};

template<class T1_orig, class T2_orig>
struct promote_trait {
    // Handle promotion of small integers to int/unsigned int
    typedef _bz_typename autopromote_trait<T1_orig>::T_numtype T1;
    typedef _bz_typename autopromote_trait<T2_orig>::T_numtype T2;

    // True if T1 is higher ranked
    enum {
      T1IsBetter =
        precision_trait<T1>::precisionRank >
          precision_trait<T2>::precisionRank,

    // True if we know ranks for both T1 and T2
      knowBothRanks =
        precision_trait<T1>::knowPrecisionRank
      && precision_trait<T2>::knowPrecisionRank,
```

```
    // True if we know T1 but not T2
      knowT1butNotT2 =  precision_trait<T1>::knowPrecisionRank
        && !(precision_trait<T2>::knowPrecisionRank),

    // True if we know T2 but not T1
      knowT2butNotT1 =  precision_trait<T2>::knowPrecisionRank
        && !(precision_trait<T1>::knowPrecisionRank),

    // True if T1 is bigger than T2
      T1IsLarger = sizeof(T1) >= sizeof(T2),

    // We know T1 but not T2: true
    // We know T2 but not T1: false
    // Otherwise, if T1 is bigger than T2: true
      defaultPromotion = knowT1butNotT2 ? _bz_false :
         (knowT2butNotT1 ? _bz_true : T1IsLarger)
    };

    // If we have both ranks, then use them.
    // If we have only one rank, then use the unknown type.
    // If we have neither rank, then promote to the larger type.

    enum {
      promoteToT1 = (knowBothRanks ? T1IsBetter : defaultPromotion)
                      ? 1 : 0
    };

    typedef typename promote2<T1,T2,promoteToT1>::T_promote T_promote;
};
```

1.9 Expression templates

Expression templates solve the pairwise evaluation problem associated with operator-overloaded array expressions in C++.

The problem with pairwise evaluation is that expressions such as

```
  Vector<double> a, b, c, d;
  a = b + c + d;
```

generate code like this:

```
  double* _t1 = new double[N];
  for (int i=0; i < N; ++i)
    _t1[i] = b[i] + c[i];
  double* _t2 = new double[N];
  for (int i=0; i < N; ++i)
```

```
    _t2[i] = _t1[i] + d[i];
  for (int i=0; i < N; ++i)
    a[i] = _t2[i];
  delete [] _t2;
  delete [] _t1;
```

1.9.1 Performance implications of pairwise evaluation

For small arrays, the overhead of `new` and `delete` result in poor performance – about 1/10th that of C. For medium (in-cache) arrays, the overhead of extra loops and cache memory accesses hurts (by about 30-50% for small expressions). The extra data required by temporaries also cause the problem to go out-of-cache sooner.

For large arrays, the cost is in the temporaries– all that extra data has to be shipped between main memory and cache. Typically scientific codes are limited by memory bandwidth (rather than flops), so this really hurts. For M distinct array operands and N operators, the performance is about $\frac{M}{2N}$ that of C or Fortran.

This is particularly bad for stencils, which have M=1 (or otherwise very small) and N very large. It is not unusual to get 1/7, 1/27 the performance of C or Fortran for big stencils.

1.9.2 Recursive templates

To understand expression templates, it helps a lot to first understand recursive templates.

Consider this class:

```
template<class Left, class Right>
class X { };
```

A class can take itself as a template parameter. This makes it possible for the type `X<>` to represent a linear list of types:

```
X<A, X<B, X<C, X<D, END> > > > a;
```

This represents the list `[A, B, C, D]`. More usefully, such template classes can represent trees of types:

```
X<X<A,B>,X<C,D> > a;
```

1.9.3 Expression templates: building parse trees

The basic idea behind expression templates is to use operator overloading to build parse trees. For example:

```
Array A, B, C, D;
D = A + B + C;
```

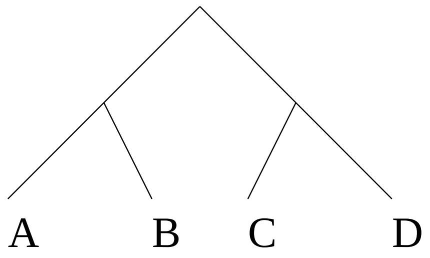

Figure 1.1: The tree represented by `X<X<A,B>,X<C,D> >`

The expression `A+B+C` could be represented by a type such as

```
X<Array, plus, X<Array, plus, Array> >
```

Building such types is not hard. Here is the overloaded operator to do it:

```
struct plus { };   // Represents addition
class Array { };   // some array class

// X represents a node in the parse tree
template<typename Left, typename Op, typename Right>
class X { };

// The overloaded operator which does parsing for
// expressions of the form "A+B+C+D+..."
template<class T>
X<T, plus, Array> operator+(T, Array)
{
    return X<T, plus, Array>();
}
```

With the above code, here is how "A+B+C" is parsed:

```
Array A, B, C, D;
D = A + B + C;
  = X<Array,plus,Array>() + C;
  = X<X<Array,plus,Array>,plus,Array>();
```

So there you have it! That's the hardest part to understand about expression templates.

1.9.4 A minimal implementation

Of course, to be useful you need to store data in the parse tree: at minimum, pointers to the arrays.

Here is a minimal expression templates implementation for 1-D arrays. First, the `plus` function object:

```
// This class encapsulates the "+" operation.
struct plus {
public:
    static double apply(double a, double b) {
        return a+b;
    };
};
```

The parse tree node:

```
template<typename Left, typename Op, typename Right>
struct X {
    Left leftNode_;
    Right rightNode_;

    X(Left t1, Right t2)
        : leftNode_(t1), rightNode_(t2)
    { }

    double operator[](int i)
    { return Op::apply(leftNode_[i],rightNode_[i]); }
};
```

Here is a simple array class:

```
struct Array {
    Array(double* data, int N)
        : data_(data), N_(N)
    { }

    // Assign an expression to the array
    template<typename Left, typename Op, typename Right>
    void operator=(X<Left,Op,Right> expression)
    {
        for (int i=0; i < N_; ++i)
            data_[i] = expression[i];
    }

    double operator[](int i)
    { return data_[i]; }

    double* data_;
    int N_;
};
```

And here is the `operator+`:

```
template<typename Left>
X<Left, plus, Array> operator+(Left a, Array b)
{
    return X<Left, plus, Array>(a,b);
}
```

Now see it in action:

```
int main()
{
    double a_data[] = { 2, 3, 5, 9 },
           b_data[] = { 1, 0, 0, 1 },
           c_data[] = { 3, 0, 2, 5 },
           d_data[4];

    Array A(a_data,4), B(b_data,4), C(c_data,4), D(d_data,4);
    D = A + B + C;

    for (int i=0; i < 4; ++i)
        cout << D[i] << "   ";
    cout << endl;

    return 0;
}
```

The output of this program is:

```
6       3       7       15
```

Here is a simulated run of the compiler, showing each method and constructor being invoked. First the parsing:

```
    D = A + B + C;
      = X<Array,plus,Array>(A,B) + C;
      = X<X<Array,plus,Array>,plus,Array>(X<Array,
            plus,Array>(A,B),C);
```

Then it matches `template Array::operator=`:

```
    D.operator=(X<X<Array,plus,Array>,plus,
        Array>(X<Array,plus,Array>(A,B),C) expression)
    {
       for (int i=0; i < N_; ++i)
         data_[i] = expression[i];
    }
```

Then `expression[i]` is expanded by inlining the `operator[]`'s from each node in the parse tree:

```
data_[i] = plus::apply(X<Array,plus,
             Array>(A,B)[i], C[i]);
         = plus::apply(A[i],B[i]) + C[i];
         = A[i] + B[i] + C[i];
```

So the end result is

```
for (int i=0; i < D.N_; ++i)
  D.data_[i] = A.data_[i] + B.data_[i] + C.data_[i];
```

Presto – no temporaries, and a single loop.

1.9.5 Refinements

Serious implementations often require iterators in the parse tree (instead of the arrays themselves).

There are many more refinements and extensions to the basic model shown here. You are limited only by the extent of your desire to inflict pain and suffering on your compiler.

```
``My desire to inflict pain on the compilers is large.
They've been tormenting me for the last 8 years.  Now's my
chance to strike back!'' --Scott Haney
```

Here's a partial list of some of the refinements people have implemented: type promotion, glommable expression templates, loop nest optimizations (tiling, loop interchange, unrolling, ...) stack iterators for multidimensional arrays, updaters, pretty-printing, unit stride optimizations, shape conformance checking, loop collapsing, pattern matching, index placeholders, tensor notation, reductions, etc. etc.

1.9.6 Pointers to more information

Don't do an expression templates implementation yourself, except for fun. It's a waste of time, and nobody will be able to understand your code! There are several good implementations available (Blitz++, PETE). PETE can be grafted onto your own array classes. Use it instead. It can be found on the web at http://www.acl.lanl.gov/pete/

PETE is the expression template engine for POOMA, which is a sequential and parallel framework for scientific computing in C++. POOMA II includes arrays, fields, meshes, particles, etc. You can find them on the web at http://www.acl.lanl.gov/pooma/

Blitz++ is an array library for C++ which offers performance competitive with Fortran, and lots of useful notations for array expressions (tensor notation, reductions, index placeholders, etc.) You can find Blitz++ on the web at http://oonumerics.org/blitz/

1.9.7 References

Expression templates were invented independently and concurrently by David Vandevoorde.

- Todd Veldhuizen, Expression Templates, C++ Report 7(5), June 1995, 26–31. Reprinted in C++ Gems, ed. Stanley Lippman. http://oonumerics.org/blitz/papers/
- Todd Veldhuizen, The Blitz++ Array Model. Proceedings of ISCOPE'98. http://oonumerics.org/blitz/papers/
- Geoffrey Furnish, Disambiguated Glommable Expression Templates, Computers in Physics 11(3), May/June 1997, 263–269.
- Scott W. Haney, Beating the Abstraction Penalty in C++ Using Expression Templates, Computers in Physics 10(6), Nov/Dec 1996, 552-557.

1.10 Template metaprograms

Template metaprograms are useful for generating specialized algorithms for small value classes. Value classes contain all their data, rather than containing pointers to their data. Typical examples are complex numbers, quaternions, intervals, tensors, and small fixed-size vectors and matrices. Algorithm specialization means replacing a general-purpose algorithm with a faster, more restrictive version.

Some example operations on small objects ($N < 10$):

- Dot products
- Matrix/vector multiply
- Matrix/matrix multiply
- Vector expressions, matrix expressions, tensor expressions

To get good performance out of small objects, one needs to unroll all the loops to produce a mess of floating-point code. This is just the sort of thing template metaprograms are good at. Some compilers will automatically unroll such loops, but not all. Template metaprograms let you do optimizations which are beyond the ability of the best optimizing compilers, as will be demonstrated later with an FFT example.

1.10.1 Template metaprograms: some history

When templates were added to C++, we got much more than intended. Erwin Unruh brought this program to one of the ISO/ANSI C++ committee meetings in 1994:

```
// Erwin Unruh, untitled program, ANSI X3J16-94-0075/ISO WG21-462, 1994.
template<int i> struct D { D(void*); operator int(); };

template<int p, int i> struct is_prime {
  enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i-1>
     :: prim };
};

template<int i> struct Prime_print {
  Prime_print<i-1> a;
```

```
  enum { prim = is_prime<i,i-1>::prim };
  void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum { prim = 1 }; };
struct is_prime<0,1> { enum { prim = 1 }; };
struct Prime_print<2> { enum { prim = 1 };
    void f() { D<2> d = prim; } };

void foo()
{
  Prime_print<10> a;
}
```

In some sense, this is not a very useful program, since it won't compile. The compiler spits out errors:

```
unruh.cpp 15: Cannot convert 'enum' to 'D<2>' in function Prime_print
unruh.cpp 15: Cannot convert 'enum' to 'D<3>' in function Prime_print
unruh.cpp 15: Cannot convert 'enum' to 'D<5>' in function Prime_print
unruh.cpp 15: Cannot convert 'enum' to 'D<7>' in function Prime_print
unruh.cpp 15: Cannot convert 'enum' to 'D<11>' in function Prime_print
unruh.cpp 15: Cannot convert 'enum' to 'D<13>' in function Prime_print
unruh.cpp 15: Cannot convert 'enum' to 'D<17>' in function Prime_print
unruh.cpp 15: Cannot convert 'enum' to 'D<19>' in function Prime_print
```

Unruh's program tricks the compiler into printing out a list of prime numbers at compile time. It turns out that C++ templates are an interpreted programming "language" all on their own. There are analogues for the common control flow structures: if/else/else if, for, do..while, switch, and subroutine calls.

Here is a simpler example which is easier to grasp than Unruh's program:

```
template<int N>
struct Factorial {
    enum { value = N * Factorial<N-1>::value };
};

template<>
struct Factorial<1> {
    enum { value = 1 };
};

// Example use
const int fact5 = Factorial<5>::value;
```

In the above example, `Factorial<5>::value` is evaluated to $5! = 120$ at compile time. The recursive template acts like a `while` loop, and the specialization of `Factorial<1>` stops the loop.

Are there any limits to what computations one can do with templates at compile time? In theory, no– it's possible to implement a Turing machine using template instantiation.

The implications are that (1) any arbitrary computation can be carried out by a C++ compiler; (2) whether a C++ compiler will ever halt when processing a given program is undecidable.

In practice, there are very real resource limitations which constrain what one can do with template metaprograms. Most compilers place a limit on recursive instantiation of templates; but there are still useful applications.

1.10.2 The need for specialized algorithms

Many algorithms can be sped up through specialization. Here's an example:

```
double dot(const double* a, const double* b, int N)
{
    double result = 0.0;
    for (int i=0; i < N; ++i)
      result += a[i] * b[i];
    return result;
}
```

Figure 1.2 shows the performance of the {\tt dot()} routine.

For small vectors, only a fraction of peak performance is obtained. To boost performance, `dot()` can be specialized for a particular N, say N=3:

```
inline double dot3(const double* a, const double* b)
{
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

This version will run at peak speed: unrolling exposes low-level parallelism and removes loop overhead; and inlining eliminates function call overhead, permits data to be registerized, and permits floating-point operations to be scheduled around adjacent computations.

Specialized algorithms achieve $n_{\frac{1}{2}} \rightarrow 0$.In some circumstances, `dot3` can be faster than $R_{\rm peak}$.

1.10.3 Using template metaprograms to specialize algorithms

By combining the "C++ compiler as interpreter" trick with normal C++ code, one can generate specialized algorithms. Here is a template metaprogram which generates specialized dot products: first, the vector class:

```
template<class T, int N>
class TinyVector {
public:
  T operator[](int i) const
  { return data[i]; }

private:
```

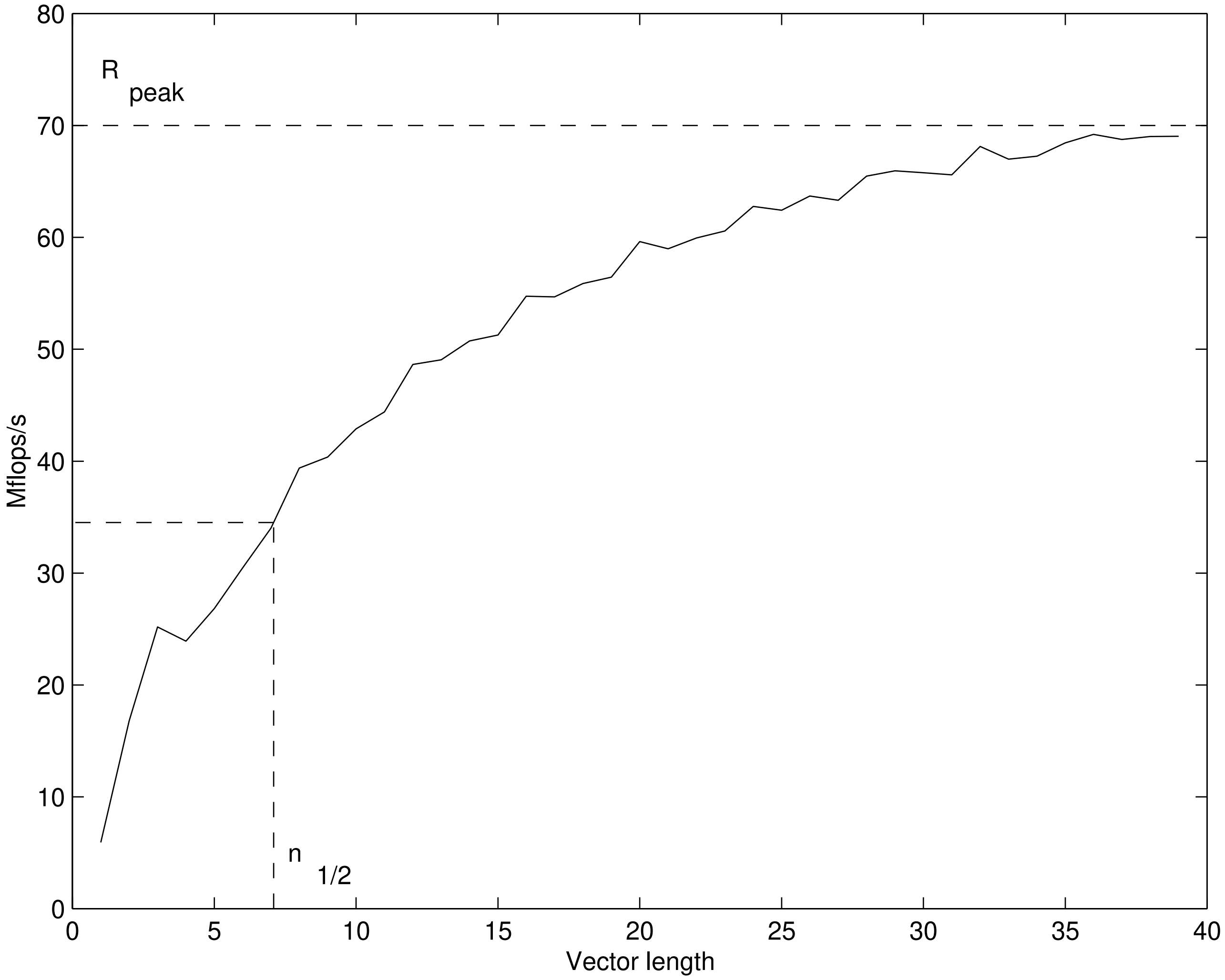

Figure 1.2: Performance of the `dot()` routine.

```
  T data[N];
};

// Example use
TinyVector<float,4> x;    // A length 4 vector of float
float x3 = x[3];          // The third element of x
```

Here is the template metaprogram:

```
// The dot() function invokes meta_dot -- the metaprogram
template<class T, int N>
inline float dot(TinyVector<T,N>& a,
  TinyVector<T,N>& b)
{ return meta_dot<N-1>::f(a,b); }

// The metaprogram
template<int I>
struct meta_dot {
  template<class T, int N>
  static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
  {
    return a[I]*b[I] + meta_dot<I-1>::f(a,b);
  }
};

template<>
struct meta_dot<0> {
  template<class T, int N>
  static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
  {
    return a[0]*b[0];
  }
};
```

Here is an illustration of what happens during compilation:

```
TinyVector<double,4> a, b;

double r = dot(a,b);
         = meta_dot<3>(a,b);
         = a[3]*b[3] + meta_dot<2>(a,b);
         = a[3]*b[3] + a[2]*b[2] + meta_dot<1>(a,b);
         = a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + meta_dot<0>(a,b);
         = a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + a[0]*b[0];
```

Note that some compilers can do this kind of unrolling automatically. But here is a more sophisticated example: a Fast-Fourier Transform metaprogram which calculates the roots of 1 and array reordering at compile time, and spits out a solid block of floating-point code.

```
// This code was written in 1994, so it contains a few
// anachronisms.

#include <iostream.h>
#include <iomanip.h>
#include <math.h>
#include <float.h>

/****************************************************************************
 * four1() implements a Cooley-Tukey FFT on N points, where N is a power of 2.
 * "Numerical Recipes in C", 2nd Edition (Teukolsky,Press,Vetterling,Flannery).
 *
 * It serves as the basis for the inline version found below.  All of the
 * variable names have been preserved in the inline version for comparison
 * purposes.  Also, the line numbers are referred to in the corresponding
 * template classes.
 ****************************************************************************/

#define SWAP(a,b) tempr=(a); (a)=(b); (b) = tempr

// Replaces data[1..2*nn] by its discrete Fourier transform.  data[] is a
// complex array
void four1(float data[], unsigned long nn, int isign)
{
    unsigned long n, mmax, m, j, istep, i;
    double wtemp, wr, wpr, wpi, wi, theta;
    float tempr, tempi;

    n = nn << 1;
    j=1;

    for (i=1; i < n; i+= 2) {                                     //   1
        if (j > i)                                                //   2
        {                                                         //   3
            SWAP(data[j], data[i]);                               //   4
            SWAP(data[j+1], data[i+1]);                           //   5
        }                                                         //   6
        m = n >> 1;                                               //   7
        while (m >= 2 && j > m) {                                 //   8
            j -= m;                                               //   9
            m >>= 1;                                              //  10
        }                                                         //  11
        j += m;                                                   //  12
    };                                                            //  13
                                                                  //  14
    mmax=2;                                                       //  15
                                                                  //  16
    while (n > mmax) {                                            //  17
        istep = mmax << 1;                                        //  18
        theta = isign * (6.28318530717959/mmax);                  //  19
```

```
        wtemp = sin(0.5*theta);                                          //  20
        wpr = -2.0*wtemp*wtemp;                                          //  21
        wpi = sin(theta);                                                //  22
        wr = 1.0;                                                        //  23
        wi = 0.0;                                                        //  24
                                                                         //  25
        for (m=1; m < mmax; m += 2) {                                    //  26
                                                                         //  27
            for (i=m; i <= n; i += istep) {                              //  28
                                                                         //  29
                j=i+mmax;                                                //  30
                tempr = wr*data[j] - wi*data[j+1];                       //  31
                tempi = wr * data[j+1] + wi*data[j];                     //  32
                                                                         //  33
                                                                         //  34
                data[j] = data[i] - tempr;                               //  35
                data[j+1] = data[i+1] - tempi;                           //  36
                data[i] += tempr;                                        //  37
                data[i+1] += tempi;                                      //  38
            }                                                            //  39
                                                                         //  40
              wr = (wtemp=wr)*wpr-wi*wpi+wr;                             //  41
              wi=wi*wpr+wtemp*wpi+wi;                                    //  42
        }                                                                //  43
        mmax=istep;                                                      //  44
    }                                                                    //  45
}

/****************************************************************************
 * Now begins the inline FFT implementation.
 *
 * The way to read this code is from the very bottom (starting with FFTServer)
 * upwards.  This is because the classes are declared in order from the
 * optimizations in the innermost loops, to the outer loops, and finally
 * FFTServer itself.
 *
 * Why are floats passed as const float&?  This eliminates some unnecessary
 * instructions under Borland C++ 4.02 compiler for an 80486.  It may be
 * better to pass floats as 'float' for other compilers.
 *
 * Here is a summary of the classes involved:
 *
 * Sine<N,I>         Calculates sin(2*Pi*I/N)
 * Cos<N,I>          Calculates cos(2*Pi*I/N)
 * bitNum<N>         Calculates 16-log2(N), where N is a power of 2
 * reverseBits<N,I>  Reverses the order of the bits in I
 * FFTSwap2<I,R>     Implements swapping of elements, for array reordering
 * FFTSwap<I,R>      Ditto
 * FFTReorder<N,I>   Implements array reordering
 * FFTCalcTempR<M>   Squeezes out an optimization when the weights are (1,0)
 * FFTCalcTempI<M>   Ditto
```

```
 * FFTInnerLoop<N,MMAX,M,I> Implements the inner loop (lines 28-39 in four1)
 * FFTMLoop<N,MMAX,M> Implements the middle loop (lines 26, 41-42 in four1)
 * FFTOuterLoop<N,MMAX> Implements the outer loop (line 17 in four1)
 * FFTServer<N>      The FFT Server class
 ***************************************************************************/


/*
 * First, compile-time trig functions sin(x) and cos(x).
 */

template<unsigned N, unsigned I>
class Sine {
public:
  static inline float sin()
  {
    // This is a series expansion for sin(I*2*M_PI/N)
    // Since all of these quantities are known at compile time, it gets
    // simplified to a single constant, which can be included in the code:
    // mov dword ptr es:[bx],large 03F3504F3h

    return (I*2*M_PI/N)*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/2/3*(1-(I*2*M_PI/N)*
           (I*2*M_PI/N)/4/5*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/6/7*(1-(I*2*M_PI/N)*
           (I*2*M_PI/N)/8/9*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/10/11*(1-(I*2*M_PI/N)*
           (I*2*M_PI/N)/12/13*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/14/15*
           (1-(I*2*M_PI/N)*(I*2*M_PI/N)/16/17*
           (1-(I*2*M_PI/N)*(I*2*M_PI/N)/18/19*(1-(I*2*M_PI/N)*
           (I*2*M_PI/N)/20/21))))))))));
  }
};


template<unsigned N, unsigned I>
class Cosine {
public:
  static inline float cos()
  {
    // This is a series expansion for cos(I*2*M_PI/N)
    // Since all of these quantities are known at compile time, it gets
    // simplified to a single number.
    return 1-(I*2*M_PI/N)*(I*2*M_PI/N)/2*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/3/4*
        (1-(I*2*M_PI/N)*(I*2*M_PI/N)/5/6*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/7/8*
        (1-(I*2*M_PI/N)*(I*2*M_PI/N)/9/10*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/11/12*
        (1-(I*2*M_PI/N)*(I*2*M_PI/N)/13/14*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/15/16*
        (1-(I*2*M_PI/N)*(I*2*M_PI/N)/17/18*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/19/20*
        (1-(I*2*M_PI/N)*(I*2*M_PI/N)/21/22*(1-(I*2*M_PI/N)*(I*2*M_PI/N)/23/24
        )))))))))));
  }

};
```

```
/*
 * The next few classes (bitNum, reverseBits, FFTSwap, FFTReorder) implement
 * the bit-reversal reordering of the input data set (lines 1-14).  However,
 * the algorithm used here bears no resemblence to that in the routine above.
 */

/*
 * bitNum<N> returns 16-log2(N); this is used to implement the bit-reversal
 * swapping portion of the routine.  There's a nice recursive
 * implementation for this, but Borland C++ 4.0 gives a 'Too many template
 * instances' warning. So use specialization instead.  Note that N must be
 * a power of 2.
 */

template<unsigned N>
class bitNum {
public:
    enum _b { nbits = 0 };
};

class bitNum<1U>     { public: enum _b { nbits = 16    }; };
class bitNum<2U>     { public: enum _b { nbits = 15    }; };
class bitNum<4U>     { public: enum _b { nbits = 14    }; };
class bitNum<8U>     { public: enum _b { nbits = 13    }; };
class bitNum<16U>    { public: enum _b { nbits = 12    }; };
class bitNum<32U>    { public: enum _b { nbits = 11    }; };
class bitNum<64U>    { public: enum _b { nbits = 10    }; };
class bitNum<128U>   { public: enum _b { nbits = 9     }; };
class bitNum<256U>   { public: enum _b { nbits = 8     }; };
class bitNum<512U>   { public: enum _b { nbits = 7     }; };
class bitNum<1024U>  { public: enum _b { nbits = 6     }; };
class bitNum<2048U>  { public: enum _b { nbits = 5     }; };
class bitNum<4096U>  { public: enum _b { nbits = 4     }; };
class bitNum<8192U>  { public: enum _b { nbits = 3     }; };
class bitNum<16384U> { public: enum _b { nbits = 2     }; };
class bitNum<32768U> { public: enum _b { nbits = 1     }; };

/*
 * reverseBits<N,I>::reverse is the number I with the order of its bits
 * reversed.  For example,
 *
 * <8,3>   3 is 011, so reverse is 110 or 6
 * <8,1>   1 is 001, so reverse is 100 or 8
 *
 * The enum types are used as temporary variables.
 */

template<unsigned N, unsigned I>
class reverseBits {
public:
    enum _r4 { I4 = (((unsigned)I & 0xaaaa) >> 1)
```

```
                  | (((unsigned)I & 0x5555) << 1) };
    enum _r3 { I3 = (((unsigned)I4 & 0xcccc) >> 2)
                  | (((unsigned)I4 & 0x3333) << 2) };
    enum _r2 { I2 = (((unsigned)I3 & 0xf0f0) >> 4)
                  | (((unsigned)I3 & 0x0f0f) << 4) };
    enum _r1 { I1 = (((unsigned)I2 & 0xff00) >> 8)
                  | (((unsigned)I2 & 0x00ff) << 8) };
    enum _r { reverse = ((unsigned)I1 >> bitNum<N>::nbits) };
};

/*
 * FFTSwap<I,R>::swap(float* array) swaps the (complex) elements I and R of
 * array.  It does this only if I < R, to avoid swapping a pair twice or
 * an element with itself.
 */

template<unsigned I, unsigned R>
class FFTSwap2 {
public:
    static inline void swap(float* array)
    {
        register float temp = array[2*I];
        array[2*I] = array[2*R];
        array[2*R] = temp;
        temp = array[2*I+1];
        array[2*I+1] = array[2*R+1];
        array[2*R+1] = temp;
    }
};

class FFTSwap2<0U,0U> {
public:
    static inline void swap(float* array)
    { }
};

template<unsigned I, unsigned R>
class FFTSwap {
public:
    enum _go { go = (I < R) };

    static inline void swap(float* array)
    {
        // Only swap if I < R
        FFTSwap2<go * I, go * R>::swap(array);
    }
};

/*
 * FFTReorder<N,0>::reorder(float* array) reorders the complex elements
 * of array[] using bit reversal.
```

```
 */
template<unsigned N, unsigned I>
class FFTReorder {
public:
    enum _go { go = (I+1 < N) };    // Loop from I=0 to N.

    static inline void reorder(float* array)
    {
        // Swap the I'th element if necessary
        FFTSwap<I,(unsigned)reverseBits<N,I>::reverse>::swap(array);

        // Loop
        FFTReorder<go * N, go * (I+1)>::reorder(array);
    }
};

// Base case to terminate the recursive loop
class FFTReorder<0U,0U> {
public:
    static inline void reorder(float* array)
    { }
};


/*
 * FFTCalcTempR and FFTCalcTempI are provided to squeeze out an optimization
 * for the case when the weights are (1,0), on lines 31-32 above.
 */
template<unsigned M>
class FFTCalcTempR {
public:
    static inline float tempr(const float& wr, const float& wi, float* array,
        const int j)
    {
        return wr * array[j] - wi * array[j+1];
    }
};

class FFTCalcTempR<1U> {
public:
    static inline float tempr(const float& wr, const float& wi, float* array,
        const int j)
    {
        // For M == 1, (wr,wi) = (1,0), so specialization leads to a nice
        // optimization:
        return array[j];
    }
};

template<unsigned M>
class FFTCalcTempI {
```

```
public:
    static inline float tempi(const float& wr, const float& wi, float* array,
        const int j)
    {
        return wr * array[j+1] + wi * array[j];
    }
};

class FFTCalcTempI<1U> {
public:
    static inline float tempi(const float& wr, const float& wi, float* array,
        const int j)
    {
        return array[j+1];
    }
};

/*
 * FFTInnerLoop implements lines 28-39.
 */

template<unsigned N, unsigned MMAX, unsigned M, unsigned I>
class FFTInnerLoop {
public:
    // Loop break condition
    enum _go { go = (I+2*MMAX <= 2*N) };

    static inline void loop(const float& wr, const float& wi, float* array)
    {
        const int j = I+MMAX;       // Known at compile time

        float tempr = FFTCalcTempR<M>::tempr(wr,wi,array,j);
        float tempi = FFTCalcTempI<M>::tempi(wr,wi,array,j);

        array[j] = array[I] - tempr;
        array[j+1] = array[I+1] - tempi;
        array[I] += tempr;
        array[I+1] += tempi;

        // Loop
        FFTInnerLoop<go * N,
                     go * MMAX,
                     go * M,
                     go * (unsigned)(I+2*MMAX)>::loop(wr,wi,array);
    }
};

// Specialization to provide base case for loop recursion
class FFTInnerLoop<0U,0U,0U,0U> {
public:
    static inline void loop(const float& wr, const float& wi, float* array)
```

```
    { }
};

/*
 * FFTMLoop implements the middle loop (lines 26, 41-42).  However, instead
 * of using a trigonometric recurrence to compute the weights wi and wr, they
 * are computed at compile time using the Sine and Cosine classes.  This way,
 * the weights are inserted into the code, eg.
 *   mov dword ptr [bp-360],large 03F6C835Eh   // Cosine of something or other
 */
template<unsigned N, unsigned MMAX, unsigned M>
class FFTMLoop {
public:
    enum _go { go = (M+2 < MMAX) };

    static inline void loop(float* array)
    {
        float wr = Cosine<MMAX,(M-1)/2>::cos();
        float wi = Sine<MMAX,(M-1)/2>::sin();

        // Call the inner loop
        FFTInnerLoop<N,MMAX,M,M>::loop(wr,wi,array);

        // Loop
        FFTMLoop<go * N,
                 go * MMAX,
                 go * (unsigned)(M+2)>::loop(array);
    }
};

// Specialization for base case to terminate recursive loop
class FFTMLoop<0U,0U,0U> {
public:
    static inline void loop(float* array)
    { }
};

/*
 * FFTOuterLoop implements the outer loop (Line 17).
 */
template<unsigned N, unsigned MMAX>
class FFTOuterLoop {
public:
    // Loop break condition
    enum _go { go = ((MMAX << 1) < (2*N)) };

    static inline void loop(float* array)
    {
        // Invoke the middle loop
        FFTMLoop<N, MMAX, 1U>::loop(array);
```

```
        // Loop
        FFTOuterLoop<go * N,
                     go * (unsigned)(MMAX << 1)>::loop(array);
    }
};

// Specialization for base case to terminate recursive loop
class FFTOuterLoop<0U,0U> {
public:
    static inline void loop(float* array)
    { }
};


/*
 * And finally, the FFT server itself.
 *
 * Input is an array of float precision complex numbers, stored in (real,imag)
 * pairs, starting at array[1].  ie. for an N point FFT, need to use arrays of
 * float[2*N+1].  The element array[0] is not used, to be compatible with
 * the Numerical Recipes implementation.  N MUST be a power of 2.
 */

template<unsigned N>
class FFTServer {

public:
    static void FFT(float* array);
};

template<unsigned N>
void FFTServer<N>::FFT(float* array)
{
    // Reorder the array, using bit-reversal.  Confusingly, the
    // reorder portion expects the data to start at 0, rather than 1.
    FFTReorder<N,0U>::reorder(array+1);

    // Now do the FFT
    FFTOuterLoop<N,2U>::loop(array);
}

/*
 * Test program to take an N-point FFT.
 */

int main()
{
    const unsigned N = 8;

    float data[2*N];
```

```
    int i;

    for (i=0; i < 2*N; ++i)
        data[i] = i;

    FFTServer<N>::FFT(data-1);       // Original four1() used arrays starting
                                     // at 1 instead of 0

    cout << "Transformed data:" << endl;

    for (i=0; i < N; ++i)
        cout << setw(10) << setprecision(5) << data[2*i] << "\t"
             << data[2*i+1] << "I" << endl;

    return 0;
}
```

To see the corresponding assembly-language output of this FFT template metaprogram, see http://oonumerics.org/blitz/examples/fft.html.

For more information about template metaprograms, see these articles:

- Using C++ template metaprograms, C++ Report Vol. 7 No. 4 (May 1995), pp. 36–43. Reprinted in C++ Gems, ed. Stanley Lippman. Available online at http://extreme.indiana.edu/~tveldhui/papers/Temp
  Metaprograms/meta-art.html.
- Linear Algebra with C++ Template Metaprograms, Dr. Dobb's Journal Vol. 21 No. 8 (August 1996), pp. 38–44. Available online at http://www.ddj.com/articles/1996/9608/9608d/9608d.htm.

1.11 Comma overloading

It turns out that you can overload comma in C++. This can provide a nice notation for initializing data structures:

```
Array A(5);
A = 0.1, 0.5, 0.3, 0.8, 1.5;
```

The implementation requires moderate trickery. The above statement associates this way:

```
(((((A = 0.1) , 0.5) , 0.3) , 0.8) , 1.5)
```

So the assignment `A = 0.1` must return a special type, which can then be matched to an overloaded comma operator. Here's a simple implementation:

```
class Array {
public:
  ListInitializer<double,double*> operator=(double x)
```

```
  {
      data_[0] = x;
      return ListInitializer<double,double*>(data_ + 1);
  }

  ...

private:
  double* data_;
};

template<class T_numtype, class T_iterator>
class ListInitializer {
public:  // ...
  ListInitializer<T_numtype, T_iterator>
    operator,(T_numtype x)
  {
    *iter_ = x;
    return ListInitializer<T_numtype,
        T_iterator>(iter_ + 1);
  }
};
```

In this example, the assignment `A = 0.1` returns an instance of `ListInitializer`; this class has an overloaded operator comma. Note that there is no check here that the list of initializers is the same length as the array, but this is simple to add.

There is a more complicated trick to disambiguate between comma initialization lists and a scalar initialization:

```
A = 0.1, 0.2, 0.3, 0.4, 0.5;   // comma initialization list
A = 1.0;                       // scalar initialization
```

We'd like the second version to initialize all elements to be `1.0`.

Here is a more complicated example which handles both these cases:

```
template<class T_numtype, class T_iterator>
class ListInitializer {

public:
    ListInitializer(T_iterator iter)
        : iter_(iter)
    {
    }

    ListInitializer<T_numtype, T_iterator> operator,(T_numtype x)
    {
        *iter_ = x;
        return ListInitializer<T_numtype, T_iterator>(iter_ + 1);
```

```
    }

private:
    ListInitializer();

protected:
    T_iterator iter_;
};

template<class T_array, class T_iterator = typename T_array::T_numtype*>
class ListInitializationSwitch {

public:
    typedef typename T_array::T_numtype T_numtype;

    ListInitializationSwitch(const ListInitializationSwitch<T_array>& lis)
        : array_(lis.array_), value_(lis.value_),
          wipeOnDestruct_(true)
    {
        lis.disable();
    }

    ListInitializationSwitch(T_array& array, T_numtype value)
        : array_(array), value_(value), wipeOnDestruct_(true)
    { }

    ~ListInitializationSwitch()
    {
        if (wipeOnDestruct_)
            array_.initialize(value_);
    }

    ListInitializer<T_numtype, T_iterator> operator,(T_numtype x)
    {
        wipeOnDestruct_ = false;
        T_iterator iter = array_.getInitializationIterator();
        *iter = value_;
        T_iterator iter2 = iter + 1;
        *iter2 = x;
        return ListInitializer<T_numtype, T_iterator>(iter2 + 1);
    }

    void disable() const
    {
        wipeOnDestruct_ = false;
    }

private:
    ListInitializationSwitch();

protected:
```

```
    T_array& array_;
    T_numtype value_;
    mutable bool wipeOnDestruct_;
};

class Array {
public:
  ListInitializer<double,double*> operator=(double x)
  {
      data_[0] = x;
      return ListInitializer<double,double*>(data_ + 1);
  }

  ...

private:
  double* data_;
};
```

1.11.1 An example

Here is a neat list initialization example from Blitz++, which implements this rotation of a 3D point about the z-axis:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

The Blitz++ version is:

```
    TinyMatrix<double,3,3> C;

    double cosa = cos(alpha),
           sina = sin(alpha);

    C = cosa,  -sina,   0.0,
        sina,   cosa,   0.0,
         0.0,    0.0,   1.0;

    x = product(C, y);
```

The matrix-vector product is unrolled using a template metaprogram. KAI C++ is able do copy propagation to eliminate the list initialization stuff, and optimize away the multiplications by 0 and 1. The code which it produces differs by 1 instruction from:

```
    double cosa = cos(alpha),
           sina = sin(alpha);

    x[0] = cosa*y[0] - sina*y[1];
```

```
    x[1] = sina*y[0] + cosa*y[1];
    x[2] = y[2];
```

1.12 Overloading on return type

Suppose we want to overload functions based on return type. For example:

```
class A { };
class B { };

A foo(int x);
B foo(int x);

A a = foo(3);   // Calls A foo(int)
B b = foo(3);   // Calls B foo(int)
```

Unfortunately, you can't do this in C++.

At least... not directly. But you can fake it. The trick is to return a proxy object which will invoke the appropriate foo() via a type conversion operator.

Here's an implementation:

```
#include <iostream.h>

class A { };
class B { };

A foo1(int x)
{
    cout << "foo1 called" << endl;
    return A();
}

B foo2(int x)
{
    cout << "foo2 called" << endl;
    return B();
}

class C {
public:
    C(int _x)
      : x(_x)
    { }

    operator A()
    { return foo1(x); }
```

```
    operator B()
    { return foo2(x); }

private:
    int x;
};

C foo(int x)
{
    return C(x);
}

int main()
{
    A a;
    B b;
    a = foo(3);   // Will call foo1
    b = foo(4);   // Will call foo2
}
```

This can create problems if the return value is never used, because then neither `foo()` is ever called! Here is another version which handles this problem.

```
#include <iostream.h>

class A { };
class B { };

A foo1(int x)
{
    cout << "foo1 called" << endl;
    return A();
}

B foo2(int x)
{
    cout << "foo2 called" << endl;
    return B();
}

class C {
public:
    C(int _x)
      : x(_x)
    { used = false;}

    ~C()
    {
      if (!used)
        cerr << "error: ambiguous use of function "
```

```
              << "overloaded on return type" << endl;
    }

    operator A()
    { used = true; return foo1(x); }

    operator B()
    { used = true; return foo2(x); }

private:
    int x;
    bool used;
};

C foo(int x)
{
    return C(x);
}

int main()
{
    A a;
    B b;
    a = foo(3);
    b = foo(4);
    foo(5);
}
```

The output from this code is

```
foo1 called
foo2 called
error: ambiguous use of function overloaded on return type
```

1.13 Dynamic scoping in C++

Dynamic scoping is almost unheard of these days, but occasionally it can be useful. With dynamic scoping, what a variable refers to is determined at run time (i.e. dynamically). A variable reference is bound to the most recent instance of that variable on the call stack. For example, in this code:

```
void a()
{
   int x = 7;
   int y = 6;

   b();
}
```

```
void b()
{
   int y = 9;
   c();
}

void c()
{
   int z = x + y;
}
```

Inside `c()`, `x` is bound to the instance of `x` in `a()`, and `y` is bound to the instance of `y` in `b()`.

Some library APIs require "context parameters" which are almost always the same, and clutter up the code. Take these X11 functions as an example:

```
int XDrawLine(Display* display, Drawable d, GC gc, int x1, int y1,
   int x2, int y2);
int XDrawArcs(Display* display, Drawable d, GC gc, XArc* arcs, int narcs);
int XDrawImageString(Display* display, Drawable d, GC gc, int x, int y,
   const char* string, int length);
```

See how the first three parameters are always Display, Drawable, GC? This makes X11 code very wordy. If we had dynamic scoping, we could just write code like this:

```
   // These are dynamically scoped variables.
   Display display = ...;
   Drawable d = ...;
   GC gc = ...;

   // Do some drawing operations
   XDrawLine(32,50,100,50);
   XDrawLine(32,0,100,0);
   XDrawImageString(30, 22, "hello", 5);
```

And the X11 library would read the display, d, and gc variables from our stack frame. If we called some function which drew to a different window, its `d` and `gc` variables would temporarily hide ours from view.

Here is some code which provides macros for dynamically scoped variables in C++.

```
#include <iostream.h>
#include <assert.h>

// Dynamically scoped variables for C++
//
// DECLARE_DYNAMIC_SCOPE_VAR(type,name) declares a new dynamically
```

```
// scoped variable.  This declaration must appear in the global
// scope before any uses of the variable.
//
// Inside a function, DYNAMIC_SCOPE_VAR(name) defines a new instance
// of the variable.  From now on, any references to ‘name’ (either in
// this function, or descendents in the call graph) will bind to
// the instance defined in this function.  Unless "shadowing" occurs
// (i.e. a function deeper in the call graph declares an instance
// of ‘name’).
//
// DYNAMIC_SCOPE_REF(name) makes ‘name’ a reference to the
// closest defined instance of the dynamic variable ‘name’
// on the call graph. The binding of name is done at runtime, but
// only involves a couple of pointer dereferences, so is very fast.
//
// See below for examples.

#define DECLARE_DYNAMIC_SCOPE_VAR(type,name)                     \
    struct _dynamic_scope_link_ ## name {                        \
        typedef type T_type;                                     \
        type instance;                                           \
        _dynamic_scope_link_ ## name * previous_instance;        \
        static _dynamic_scope_link_ ## name * current_instance;  \
                                                                 \
        _dynamic_scope_link_ ## name() {                         \
            previous_instance = current_instance;                \
            current_instance = this;                             \
        }                                                        \
                                                                 \
        ~_dynamic_scope_link_ ## name() {                        \
            current_instance = previous_instance;                \
        }                                                        \
    };                                                           \
                                                                 \
    _dynamic_scope_link_ ## name*                                \
         _dynamic_scope_link_ ## name::current_instance = 0;

#define DYNAMIC_SCOPE_VAR(name)                                  \
    _dynamic_scope_link_ ## name _ ## name ## _instance;         \
    _dynamic_scope_link_ ## name::T_type& name                   \
          = _ ## name ## _instance.instance;

#define DYNAMIC_SCOPE_REF(name)                                  \
    assert(_dynamic_scope_link_ ## name::current_instance != 0); \
    _dynamic_scope_link_ ## name::T_type& name                   \
        = _dynamic_scope_link_ ## name::current_instance->instance;
```

And here are some example uses:

```
/*
```

```
 * Examples
 */

void foo();
void bar();

// Simple example

DECLARE_DYNAMIC_SCOPE_VAR(int,x)

int main()
{
    DYNAMIC_SCOPE_VAR(x)

    x = 5;
    foo();      // x = 5;  foo() sees instance of x in this function
    bar();      // x = 10; foo() sees instance of x in bar()
    foo();      // x = 5;  foo() sees instance of x in this function
}

void foo()
{
    DYNAMIC_SCOPE_REF(x)

    cout << "x = " << x << endl;
}

void bar()
{
    DYNAMIC_SCOPE_VAR(x)
    x = 10;
    cout << "x = " << x << endl;
}
```

When this example is compiled, the output is:

```
x = 5
x = 10
x = 5
```

1.13.1 Some caution

There are reasons why dynamic scoping has disappeared from programming languages. Bruno Haible comments:

```
Dynamic scoping is almost always a bad advice:

  1. As long as you have to pass the same arguments over and over again, it
```

```
   makes the code superficially look simpler. But when you start needing
   to pass different values, the code becomes harder to understand.

2. It is less efficient, because access to global variables is slower
   than access to stack-allocated variables.

3. It is not multithread-safe.

Dynamic scoping has proven to be worse than lexical scoping.

If you have to pass the same values as arguments to multiple function calls,
it might be a good idea (in the sense of object-orientedness) to group them
in a single structure/class and pass that structure instead.
```

1.14 Interfacing with Fortran codes

Interfacing to Fortran can be a major inconvenience. One of the problems is that every Fortran compiler has different ideas about how symbol names should look: a routine such as

```
      subroutine daxpy(n,da,dx,incx,dy,incy)
         .
         .
      return
      end
```

Might be compiled to a symbol name `daxpy`, `daxpy_`, `daxpy__`, or `DAXPY`, depending on the compiler. Declaring Fortran routines in C++ is therefore highly platform-dependent.

Fortran 2000 is standardizing interoperability with C. I'm not sure if this will solve the symbol name difference headaches or not.

A potentially useful package is `cppf77` by Carsten Arnholm, which can be found on the web at http://home.sol.no/~arnholm/. It provides extensive support for portable, mixed-language programming (e.g. calling Fortran from C++).

1.15 Some thoughts on performance tuning

1.15.1 General suggestions

Here are some general suggestions for tuning C++ applications.

- Build your application on top of libraries which offer high performance. If you need an array class, use Blitz++, POOMA, A++/P++, etc. Writing an efficient array class is a lot of work; trust me, you don't want to do it yourself.
- Be aware that what is good for one architecture may be bad for another. If you have to support many platforms, you should code in a way that guarantees good (not necessarily stellar) performance on all platforms. Note that some libraries (Blitz++, FFTW, PhiPAC) are able to tune themselves for particular architectures; if you build on top of them, problem solved.

- Make use of the large base of existing C, C++, F77/F90 libraries (there's no shame in mixed-language applications, but there may be portability issues). "Buy [or download!], don't build."
- It helps to be aware of language-specific performance issues (especially C++). These issues need to be understood and dealt with in early design stages, because they can affect the design in fundamental ways (e.g. virtual functions being replaced by static polymorphism).
- Use a profiler before, during, and after you optimize to gauge improvements and ensure you aren't wasting your time trying to tune a routine that accounts for only 2% of the runtime.
- Cost-benefit analysis: Does it really matter if a part of your code is slow? This depends on usage patterns and user expectations.
- Look for algorithm and data structure improvements first. These will offer the largest performance savings, and will outlive platform-specific tuning.
- Experiment with optimization flags. Sometimes higher levels of optimization produce slower code. Measure compile time/execution time tradeoffs: who cares if a 3% performance improvement requires two more hours of compilation? I will sometimes hack up a script to run through the relevant combinations of optimization flags and measure compile time vs. run time.
- Try using the "assume no aliasing" flag on your compiler to see if there is a big performance improvement. This is a dangerous flag to use, but it provides a quick way to determine if aliasing is an issue. If it is, you might be able to use the `restrict` keyword to fix the problem.
- Understand which optimizations the compiler is competent to perform, and let it do them. You should be optimizing at a level above that of the compiler, or you are wasting your time. (See the section below).
- Be aware of platform tools: profilers, hardware counters, assembly annotators, etc. Some compilers (e.g. Intel, SGI) have full-blown performance tuning environments.
- You can commit efficiency sins of the grossest sort as long as your inner loops are tight. You don't have to avoid slow language features such as virtual functions, `new` and `delete`; or even scripting (e.g. driving your C++ program from Python/Perl/Eiffel etc.) Just keep these features away from your inner loops. It's all about amortization.

1.15.2 Know what your compiler can do

There are many optimizations which are usually useless and only serve to obscure the code. These optimizations make no difference because of modern architectures and compilers:

- Replacing floating-point math with integer arithmetic. On some platforms, integer math is slower than floating-point math. Floating-point has been optimized to death by the hardware gurus, and it's amazingly efficient.
- Deciding what data should be in registers. The `register` keyword is obsolete. Compilers do a much better job of register assignment than you will.
- Loop unrolling. The compiler almost always does a better job. When some compilers encounter manually unrolled loops, they will de-unroll it, then unroll it properly.
- Reversing loops: `for (i=N-1; i >= 0; --i)`. On some platforms this can increase performance by 3%; on others it can decrease performance by 10%.

- Replacing multiplications with additions. Modern CPUs can issue one multiplication each clock cycle. Many of the "strength reduction" tricks from the 70s and 80s will now result in slower code. Replacing floating-point division with multiplication is still relevant; some compilers will do this automatically if a certain option is specified.

Know what optimizations your compiler can perform competently. Let it do its job. Know what optimizations are relevant for modern architectures. Don't clutter up your code with irrelevant tricks from the 1970/80s.

1.15.3 Data structures and algorithms

Data structures and algorithms (trees, hash tables, spatial data structures) are the single most important issue in performance. Computational scientists have often have a weak background in this area– sometimes bringing in a data structures expert is the best way to solve your performance problems. Maybe you have $O(N)$ algorithms which could be replaced by $O(logN)$ ones.

Algorithm improvements will offer the biggest savings in performance, and will remain relevant for many generations of hardware.

1.15.4 Efficient use of the memory hierarchy

Typical workstations these days have main memory that runs at 1/10 the speed of the CPU. Efficient use of the memory hierarchy (registers, cache, memory, disk) is necessary to get good performance. But for many applications, good performance is simply not achievable. If your problem is large and doesn't fit in cache, and there isn't a clever way to rearrange your memory access patterns (by doing blocking or tiling), then your code will perform at about 10% of the CPU peak flop rate. Don't feel badly about this – the peak flop rate is only achievable on highly artificial problems. At one of the US supercomputing centres, every major code on their T3E runs at 8-15% of the peak flop rate. And that's after performance tuning by experts.

Don't sweat too much about the memory hierarchy. The tuning issues associated with memory hierarchies change yearly. Fifteen years ago, avoiding floating point multiplies was a cottage industry for computer scientists. Now multiplies take one clock cycle (pipelined). Currently people fret about managing the memory hierarchy. A few years from now, this too will be irrelevant. The reason main memory is slow is economics, not because of some fundamental limit – DRAM is much cheaper than SRAM. Someday soon, some company will figure out a cheap way to manufacture SRAM, and the thousands of person-years spent finding clever ways to massage the memory hierarchy will be suddenly irrelevant.

If you do want to worry about memory hierarchies, understand your hardware and read about blocking/tiling techniques. Be aware that any tuning you do may be obsolete with the next machine. Make use of hardware counters if available– they can give you a breakdown of cache/main memory data movement.

A great reference on performance tuning for modern architectures is: "High Performance Computing" by Kevin Dowd (O'Reilly & Associates). A new edition came out in August 1998.

Compiler manuals often have good advice for platform-specific tuning.

Some other starting points are these papers:

- Monica Lam et al, Cache performance and optimizations of blocked algorithms, APLOS IV, 63–74 (online)
- Larry Carter et al, Hierarchical tiling for improved superscalar performance, IPPS'95, 239–245 (online)
- Michael Wolfe, Iteration space tiling for memory hierarchies, Proc. 3rd Conf. on Parallel Processing for Scientific Computing, 1989, 357–361

Index